Kiss

A Complete Guide to the
Web Application Framework

Blake McBride

November 7, 2026

Copyright © 2026 Blake McBride
All rights reserved.

ISBN 979-8-9902780-7-3

November 7, 2026

Acknowledgments

Thanks to Melaine Sarbey for the creation of the Kiss logo.

Contents

I Getting Started **1**

1 Introduction **3**
1.1 Design Philosophy 3
1.2 What Kiss Provides 4
1.3 What Kiss Is Not 5
1.4 Who This Book Is For 5
1.5 How to Read This Book 6
1.6 Conventions Used 6

2 Setup and Installation **7**
2.1 Prerequisites 7
2.2 Downloading Kiss 8
2.3 Directory Structure 8
2.4 First Run 9
2.5 Verifying the Installation 10
2.6 The Example Application 10
2.7 Development URLs 11
2.8 Server Logs 11

3 Architecture Overview **13**
3.1 High-Level View 13
3.2 Request Lifecycle 14
3.3 Service Discovery and Language Support 14
3.4 Hot Reload 15
3.5 Thread Model 16
3.6 The JSON-RPC Protocol 16
3.6.1 REST vs. JSON-RPC 17
3.7 Authentication Flow 18

3.8 Database Connection Lifecycle 19
3.9 Frontend Architecture 19
3.10 Putting It All Together 20

4 The Build System **21**
4.1 Why `bld` Instead of Maven or Gradle? 21
4.2 The `bld` Script 22
4.3 IDE Integration via Ant (`build.xml`) 23
4.4 Build Commands 23
4.4.1 Development 23
4.4.2 Build and Package 24
4.4.3 Cleanup 24
4.4.4 Testing 24
4.5 Development Ports: the Port Block 24
4.5.1 Port Overrides 25
4.6 Compilation Order 26
4.7 Incremental Compilation 27
4.8 The `precompiled` Directory 27
4.9 WAR Generation 28
4.10 JavaDoc Generation 28
4.11 Helper Scripts 28
4.12 Dependency Management 29
4.12.1 Foreign Dependencies and Superseded Versions 29
4.12.2 Where `pom.xml` Comes From 31

5 Configuration **33**
5.1 `application.ini` 33
5.1.1 Database Settings 33
5.1.2 When the Configured Database Cannot Be Reached 34
5.1.3 Server Settings 35
5.1.4 Connection Pool Settings 36
5.1.5 Accessing Configuration from Code 37
5.1.6 Secrets and External Configuration 38
5.2 `web.xml` 39
5.3 `log4j2.xml` 40
5.3.1 Changing Log Levels 40
5.3.2 Using Logging in Services 40
5.4 `context.xml` 41
5.5 Development vs. Production Mode 41

II Backend Development **43**

6 Writing Backend Services **45**
6.1 The Service Pattern 45
6.2 Service Discovery 46
6.3 Groovy Services 47
6.4 Java Services 47
6.5 Lisp Services 48
6.6 Reading Input and Writing Output 48
6.6.1 Returning Record Lists 48
6.6.2 Returning a Single Record 49
6.7 Error Handling 49
6.8 A Complete CRUD Service 50
6.9 File Upload Handling 52
6.10 Interfacing with External Systems 53
6.11 Calling Other Dynamically Compiled Services 53
6.12 Reusable Scripts 54
6.13 Services Without a Database 55
6.14 Long-Running Requests 55
6.15 Service Best Practices 56

7 Groovy for Kiss Developers **57**
7.1 Why Groovy 57
7.2 Key Differences from Java 58
7.3 Variables and Types 58
7.4 String Interpolation 59
7.5 Safe Navigation and Elvis Operator 59
7.6 Closures 60
7.7 Groovy Idioms in Kiss Services 60
7.7.1 Concise Service Methods 60
7.7.2 Null-Safe Input Handling 61
7.7.3 File Operations 61
7.8 Java Interoperability 61
7.9 When to Use Groovy vs. Java 62
7.10 Common Pitfalls 62

8 Database Connections and Transactions **65**
8.1 The Connection Class 65
8.1.1 Creating Standalone Connections 65
8.2 Supported Databases 66
8.3 Executing SQL 66

8.3.1 Existence Checks . . . 67
8.3.2 Row Counting . . . 67
8.4 Fetching Data . . . 67
8.4.1 Single Record . . . 67
8.4.2 All Records . . . 67
8.4.3 Limited Records . . . 68
8.4.4 Paginated Records . . . 68
8.4.5 Fetching as JSON . . . 68
8.5 Cursors for Large Result Sets . . . 69
8.6 The Command Class . . . 70
8.7 Transactions . . . 71
8.8 Parameterized Queries . . . 72
8.9 Database-Specific SQL . . . 72
8.10 Connection Metadata . . . 73
8.11 Connection Pooling . . . 74
8.12 Per-Request Connection Preparation . . . 74

9 Database CRUD with Records **77**
9.1 The Record Class . . . 77
9.2 Creating and Inserting Records . . . 77
9.2.1 Auto-Increment Keys . . . 78
9.2.2 Method Chaining . . . 78
9.3 Fetching Records . . . 78
9.4 Reading Field Values . . . 79
9.4.1 Date and Time Fields . . . 80
9.4.2 Setting Date and Time Fields . . . 80
9.5 Updating Records . . . 81
9.6 Deleting Records . . . 81
9.6.1 Delete Callbacks . . . 81
9.7 Checking Column Existence . . . 82
9.8 Copying Records . . . 82
9.9 Converting to JSON . . . 82
9.10 A Complete CRUD Service . . . 83
9.10.1 Backend Service . . . 83
9.10.2 Frontend Screen (HTML) . . . 84
9.10.3 Frontend Screen (JavaScript) . . . 85

10 The Query Builder **87**
10.1 Creating a QueryBuilder . . . 87
10.2 Table Aliases . . . 88
10.3 SELECT . . . 89

10.3.1 DISTINCT 89
10.3.2 Aggregate Functions 89
10.4 WHERE Conditions 89
10.4.1 Dynamic WHERE Clauses 90
10.5 OR / AND Grouping 90
10.6 ORDER BY 91
10.7 GROUP BY and HAVING 91
10.8 LIMIT 92
10.9 Automatic Join Resolution 92
10.9.1 Foreign Keys vs. Join Types 92
10.10 Explicit Joins 93
10.10.1 Overriding Auto-Join Types 93
10.10.2 Fully Explicit Joins 94
10.10.3 Self-Joins 95
10.10.4 Composite Foreign Keys 95
10.11 Subqueries 95
10.11.1 WHERE IN / NOT IN 95
10.11.2 WHERE EXISTS / NOT EXISTS 96
10.12 Common Table Expressions (CTEs) 96
10.13 UNION and UNION ALL 97
10.14 Execution Methods 97
10.14.1 Formatted Debug Output 98
10.15 Using QueryBuilder with Commands 99
10.16 Complete Example: Flexible Employee Search 100
10.17 Client-Driven Sorting with SortBuilder 101
10.17.1 Registering Keys 101
10.17.2 Joins 102
10.17.3 Building the ORDER BY 103
10.17.4 Blanks Last and Database Dialects 104
10.17.5 The Service Method 105

11 The Schema Graph **107**
11.1 Graph Model 107
11.2 Building a Schema Graph 108
11.2.1 From Database Metadata 108
11.2.2 Cached per Schema (Thread-Safe) 108
11.2.3 Programmatic Declaration 108
11.2.4 Composite Foreign Keys 108
11.2.5 Hybrid Approach 109
11.3 File Caching 109

11.4 In-Memory Cache Management 109
11.5 Querying the Graph 110
11.6 Finding Join Paths 110
11.7 The Edge Inner Class 111
11.8 Integration with Connection 111
11.9 Integration with Command 112
11.10Example: Inspecting a Schema 112

III Frontend Development 113

12 Frontend Overview 115
12.1 Architecture 115
12.2 Script Loading Order 116
12.3 Component Registration 118
12.4 The `$$()` Function 119
12.5 Screen Loading 120
12.6 Screen JavaScript Pattern 121
12.7 The Login Flow 121
12.8 The Framework Screen 123
12.9 Backend URL Configuration 123
12.10Mobile Support 124
12.11Cache Control 125

13 Client-Side Routing and State 127
13.1 AppState: The Client-Side State Store 128
13.1.1 The Storage Layering Rule 128
13.1.2 Backing Stores 128
13.1.3 The Per-Tab Default and Its Consequences . 129
13.1.4 Namespacing and Isolation 130
13.1.5 Serialization Caveats 130
13.1.6 Graceful Fallback 131
13.1.7 The API 131
13.1.8 The Session Token Is Just `_uuid` 132
13.1.9 Cross-Screen Data Helpers 133
13.2 Router: Hash-Based Client-Side Routing 133
13.2.1 Why Hash Routing 134
13.2.2 Two Navigation Levels 134
13.2.3 Route Definitions 135
13.2.4 The Route Table (`routes.js`) 135
13.2.5 Path Parameters 136

13.2.6 File-Based Fallback Routing 137
13.2.7 The `?tag` Override 138
13.2.8 The API 138
13.2.9 Navigation: `go` vs. `replace` 139
13.3 The Dispatch Algorithm 139
13.4 Session Resume and the Auth Gate 140
13.4.1 Login and Logout Integration 141
13.4.2 Browser Back/Forward Is Now In-App 141
13.5 Putting It Together 142

14 Cache Busting and Browser Security **145**
14.1 Cache Busting 145
14.1.1 The Problem 145
14.1.2 The Version and the Application Mode . . . 146
14.1.3 The Byte-Stable Inline Kernel 147
14.1.4 Busting Everything Else: `bootstrap.js` . . . 149
14.1.5 What Is *Not* Busted 150
14.1.6 Why It Is Rock-Solid 151
14.1.7 Development Mode: Nothing Is Ever Cached 151
14.1.8 How an Open Tab Picks Up a New Release . 151
14.1.9 Production Stamping by `./bld war` 152
14.1.10 Restoring the Staged Files After the WAR Is Built 153
14.2 Browser Security 154
14.2.1 A Self-Registering Filter, No Configuration Needed 154
14.2.2 The Content-Security-Policy 155
14.2.3 Report-Only Versus Enforcing 157
14.2.4 Always-On Header Protections 158
14.2.5 The Master Switch 160

15 UI Components **161**
15.1 The Component System 161
15.2 Common API 161
15.3 Text Input 163
15.4 Numeric Input 163
15.5 Date Input 164
15.6 Time Input 164
15.7 Duration Input 165
15.8 Drop-Down 166
15.9 Combo Box 167

15.10 Smart Chooser . . . 170
15.11 List Box . . . 172
15.12 Checkbox . . . 173
15.13 Radio Button . . . 174
15.14 Push Button . . . 174
15.15 Text Label . . . 175
15.16 Textbox Input . . . 175
15.17 File Upload . . . 175
15.18 Picture . . . 176
15.19 Additional UI Components . . . 176
15.19.1 Search Input . . . 176
15.19.2 Segmented Control . . . 178
15.19.3 Menu Button . . . 179
15.19.4 Accordion . . . 181
15.19.5 Panel Card . . . 182
15.19.6 Section Title . . . 183
15.19.7 Badge and Chip . . . 183
15.19.8 Avatar . . . 184
15.19.9 Toast Notifications . . . 185
15.20 Stylesheets and Design Tokens . . . 186
15.20.1 Design Tokens . . . 187
15.21 Styling Guidelines . . . 189
15.22 Validation Pattern . . . 189

16 Data Grids **191**
16.1 Creating a Grid . . . 191
16.2 Column Configuration . . . 192
16.2.1 Custom Cell Renderers . . . 192
16.2.2 Built-in Formatters . . . 192
16.3 Loading Data . . . 193
16.4 Row Selection . . . 193
16.4.1 Selection Events . . . 193
16.4.2 Reading Selection . . . 194
16.4.3 Programmatic Selection . . . 194
16.4.4 Multi-Select . . . 194
16.4.5 Disabling Selection . . . 194
16.5 Querying Grid Data . . . 194
16.6 Row Styling . . . 195
16.6.1 Row Highlighting . . . 195
16.7 Column Visibility . . . 195

16.8 Drag and Drop . . . 195
16.9 Server-Side Sorting . . . 196
16.9.1 Example: Primary and Secondary Sort . . . 196
16.9.2 setServerSort(fn) . . . 197
16.9.3 setSortIndicators(list) . . . 198
16.10Enable/Disable . . . 199
16.11Cleanup . . . 199
16.12Complete Grid Example . . . 199

17 Popups and Dialogs **201**
17.1 The Popup Component . . . 201
17.2 Opening and Closing Popups . . . 202
17.2.1 Popup Stacking . . . 202
17.2.2 DOM Preservation After Close . . . 203
17.3 Popup Sizing . . . 203
17.4 The Standard Edit Pattern . . . 203
17.5 Message Dialogs . . . 205
17.6 Confirmation Dialogs . . . 205
17.7 Wait Messages . . . 206

18 Frontend Utility Classes **207**
18.1 DateUtils . . . 207
18.1.1 Conversion . . . 207
18.1.2 Date Arithmetic . . . 208
18.1.3 Date Components . . . 208
18.1.4 Validation . . . 208
18.2 TimeUtils . . . 208
18.2.1 Conversion and Parsing . . . 209
18.2.2 Time Arithmetic . . . 209
18.2.3 Time Components . . . 209
18.3 DateTimeUtils . . . 209
18.3.1 Epoch Millisecond Handling . . . 210
18.3.2 Component Extraction . . . 210
18.3.3 DateTime Arithmetic (DST-Aware) . . . 210
18.3.4 Timezone Support . . . 210
18.4 NumberUtils . . . 211
18.5 DOMUtils . . . 211
18.5.1 Browser Back Button Prevention . . . 212
18.6 Utils . . . 213
18.7 MutableString . . . 214
18.8 Editor . . . 214

19 Server Communication **217**
19.1 Server.call() . . . 217
19.2 Request/Response Format . . . 218
19.3 Error Handling . . . 219
19.4 Quiet Calls . . . 219
19.5 Parallel Calls . . . 220
19.6 Session Management . . . 221
19.6.1 Inactivity Timeout (Frontend) . . . 221
19.7 Logging Out . . . 221
19.8 File Uploads . . . 222
19.9 File Downloads . . . 223
19.10Binary Data . . . 223
19.11A Complete Screen Example . . . 224

IV Application Features **227**

20 Authentication and Sessions **229**
20.1 The Login Flow . . . 229
20.2 The Login Service . . . 230
20.3 KissInit.groovy . . . 231
20.4 Allowing Unauthenticated Access . . . 232
20.5 Session Management . . . 233
20.5.1 UserData . . . 233
20.5.2 UserCache . . . 233
20.5.3 Session Expiration . . . 233
20.5.4 Logout Handler . . . 234
20.6 Automatic Authentication in Services . . . 234
20.7 Service-Decides Authentication . . . 234
20.7.1 Deciding Inside the Service . . . 235
20.7.2 How `requireLogin()` Works Across Runners 236
20.7.3 Closing the Loop on the Front End . . . 237
20.8 Server-Restart Detection (Boot ID) . . . 237
20.8.1 The Front-End Check . . . 238
20.8.2 Restart Versus Session Expiry . . . 239
20.9 Login Form Best Practices . . . 240
20.10Browser Back Button Prevention . . . 240
20.11Frontend Inactivity Timeout . . . 241

21 Reports and Exports **243**
21.1 PDF Reports with Groff 243
21.1.1 Table Column Formatting 244
21.1.2 Numeric Columns 244
21.1.3 Date Columns 245
21.1.4 Report Configuration 245
21.1.5 Generating the PDF 245
21.2 CSV Export 245
21.2.1 Field Writing Methods 246
21.3 NumberFormat for Reports 246
21.4 Temporary File Management 247
21.5 The PDF Class 248

22 Scheduled Tasks **249**
22.1 The Cron System 249
22.2 The Crontab File 249
22.3 Writing a Cron Task 250
22.4 Example: Nightly Data Cleanup 251
22.5 Example: Periodic Report Generation 252
22.6 Dynamic Crontab Reloading 253

23 File Handling **255**
23.1 File Uploads 255
23.1.1 Frontend: The file-upload Component 255
23.1.2 Backend: Receiving Uploads 256
23.1.3 Complete Upload Example 257
23.2 Parsing CSV Files 258
23.2.1 Basic Usage 258
23.2.2 Custom Delimiters 258
23.2.3 Header-Based Access 258
23.2.4 Field Access Methods 259
23.2.5 Tolerant Header Matching 259
23.2.6 Structured Imports with DelimitedFileImport 261
23.3 Writing CSV Files 275
23.4 FTP Transfers 275
23.4.1 Connecting and Transferring 275
23.4.2 FTP Methods 276
23.4.3 Example: Automated Export via FTP 276
23.5 PDF Generation 277
23.6 Temporary File Management 278
23.6.1 Creating and Delivering Files 279

23.6.2 How It Works . . . 279
23.6.3 Additional FileUtils Methods . . . 279
23.7 External File Storage . . . 280
23.7.1 Configuration . . . 280
23.7.2 Virtual Columns . . . 281
23.7.3 Cascade Delete . . . 283
23.7.4 Hybrid Columns: A Real Column That Overflows to a File . . . 283

24 LLM Integration **287**
24.1 Overview . . . 287
24.1.1 Error Handling . . . 288
24.1.2 Web Search . . . 288
24.2 OpenAI . . . 289
24.2.1 Construction . . . 289
24.2.2 Chat Completions . . . 290
24.2.3 Streaming Responses . . . 290
24.2.4 The Two Endpoints: Chat Completions and Responses . . . 290
24.2.5 Configuration . . . 293
24.2.6 Embeddings . . . 294
24.2.7 Model Listing . . . 294
24.3 Anthropic (Claude) . . . 294
24.3.1 Construction and Usage . . . 295
24.3.2 Streaming . . . 295
24.3.3 Endpoint and Version Configuration . . . 295
24.3.4 Anthropic Methods . . . 296
24.4 OpenRouter (Multi-Provider Gateway) . . . 296
24.4.1 Construction and Usage . . . 297
24.4.2 Streaming . . . 297
24.4.3 Model Fallback Routing . . . 297
24.4.4 Reasoning Effort . . . 298
24.4.5 Attribution Headers . . . 298
24.4.6 OpenRouter Methods . . . 298
24.5 Ollama (Local Models) . . . 299
24.5.1 Construction . . . 299
24.5.2 Sending a Prompt . . . 299
24.5.3 Multi-Turn Conversations . . . 300
24.5.4 Streaming Responses . . . 301
24.5.5 Timeouts and Long Generations . . . 304

24.5.6 Embeddings 305
24.5.7 Model Management 305
24.5.8 HTML Conversion 306
24.6 QdrantClient (Vector Database) 306
24.6.1 Collections 306
24.6.2 Inserting Vectors 307
24.6.3 Searching 307
24.6.4 Record Management 308
24.6.5 QdrantClient Methods 308
24.7 RAG Pattern 309
24.8 Example: A Chatbot Service 310
24.8.1 Backend Service 310
24.8.2 Frontend Screen 310

25 Interfacing with External Systems 313
25.1 RestClient — Calling External APIs 314
25.1.1 Advanced Configuration 315
25.2 RestServerBase — Receiving External REST Calls . 316
25.2.1 What `RestServerBase` Provides 316
25.2.2 Configuration — No `web.xml` Required . . . 317
25.2.3 Why It Must Live Under `precompiled/` . . . 317
25.3 RestServerExample — The Template 318
25.4 MCPServerBase — Exposing Tools to AI Assistants 318
25.4.1 The Wire Protocol 319
25.4.2 The Three Methods That Matter 319
25.4.3 Subclass Hooks 320
25.4.4 Helpers 320
25.4.5 Tool Errors vs. Protocol Errors 321
25.4.6 What MCPServerBase Does Not Implement . 321
25.5 MCPServerExample — The Template 322
25.5.1 Build and Authentication 323
25.6 MCPClientBase — Consuming a Remote MCP Server 324
25.6.1 The Lazy Handshake 324
25.6.2 Subclass Hooks 325
25.6.3 Operations 326
25.6.4 Reading Tool Results 326
25.6.5 Tool Errors vs. Protocol Errors 327
25.6.6 Authentication 327
25.6.7 What MCPClientBase Does Not Implement . 328
25.7 MCPClientExample — The Template 328

26 OAuth 2.1 Resource Server **331**
26.1 What the Resource Server Does 331
26.2 Configuration 332
26.3 Automatic Enforcement for MCP Servers 334
26.4 Accessing the Validated Token 335
26.4.1 ValidatedToken Methods 335
26.5 Discovery Endpoint 336
26.6 Customizing Authentication 337
26.6.1 An Explicitly Public MCP Server 337
26.6.2 A Different Authentication Scheme 337
26.6.3 OAuth Plus Extra Checks 337
26.7 Using OAuth from Non-MCP Servlets 338
26.8 JWKS Caching and Key Rotation 339
26.8.1 Resolving the JWKS URI 339
26.9 Security Considerations 341
26.10Components 342
26.11A Worked Example 343
26.12Troubleshooting 344
26.13What's Not Implemented 346
26.14Further Reading 347

27 OAuth 2.1 Authorization Server **349**
27.1 Configuration 350
27.2 Required App Code: UserAuthenticator 351
27.3 Optional App Code: ConsentProvider and HTML Overrides 353
27.3.1 ConsentProvider 353
27.3.2 HTML Overrides 354
27.4 The Five Endpoints 355
27.5 The Authorization Code Flow Step by Step 356
27.6 Dynamic Client Registration 358
27.7 Refresh-Token Rotation and Reuse Detection 359
27.8 Persistence: The oauth.db Database 360
27.8.1 Transactions 361
27.8.2 Lazy Pruning 361
27.8.3 Concurrency Model 361
27.8.4 Scale 362
27.8.5 Inspection 362
27.8.6 Build-Time Safety 362
27.9 Security Considerations 362

27.10 Components . . . 364
27.11 A Worked Example . . . 365
27.11.1 Step 1: Turn on the AS in application.ini . . . 365
27.11.2 Step 2: Provide a UserAuthenticator . . . 366
27.11.3 Step 3: Register the authenticator and consent provider . . . 366
27.11.4 That's It . . . 367
27.12 Troubleshooting . . . 367
27.13 What's Not Implemented . . . 369
27.14 Further Reading . . . 370

28 OAuth 2.1 Client **371**
28.1 What the Client Does . . . 372
28.2 Configuration . . . 372
28.3 Using the Client from App Code . . . 374
28.4 The Authorization-Code Flow Step by Step . . . 376
28.5 Token Refresh and Rotation . . . 377
28.6 Discovery and Registration in Detail . . . 378
28.6.1 Discovery . . . 378
28.6.2 Registration . . . 379
28.7 Persistence: The Shared oauth.db . . . 380
28.8 Security Considerations . . . 381
28.9 Components . . . 382
28.10 A Worked Example . . . 383
28.10.1 Step 1: Declare the provider in application.ini 383
28.10.2 Step 2: Drive the flow from a service . . . 383
28.10.3 What Happens at Runtime . . . 384
28.11 Troubleshooting . . . 385
28.12 What's Not Implemented . . . 386
28.13 Further Reading . . . 387

V Utilities and Advanced Topics **389**

29 Utility Classes Reference **391**
29.1 StringUtils . . . 391
29.1.1 Substring Operations . . . 391
29.1.2 Trimming . . . 391
29.1.3 Joining and Splitting . . . 392
29.1.4 Testing and Conversion . . . 392
29.1.5 Key StringUtils Methods . . . 392

29.2 DateUtils . . . 393
29.3 TimeUtils . . . 394
29.4 DateTime . . . 394
29.5 NumberUtils and NumberFormat . . . 395
29.5.1 NumberUtils . . . 395
29.5.2 NumberFormat . . . 395
29.6 Choosing a Cryptography or Hashing Utility . . . 396
29.7 Crypto . . . 396
29.7.1 Two Ways to Supply the Key . . . 397
29.7.2 Explicit and Random Salts . . . 398
29.7.3 Versioned Container and Backward Compatibility . . . 399
29.8 PasswordHash . . . 401
29.8.1 Upgrading the Work Factor: `needsRehash` . . . 401
29.9 Hash . . . 402
29.10 Hmac . . . 403
29.10.1 The Blind-Index Pattern . . . 404
29.11 ConstantTime and RandomUtil . . . 405
29.11.1 ConstantTime . . . 405
29.11.2 RandomUtil . . . 406
29.12 Image . . . 406
29.12.1 Compression . . . 407
29.12.2 File I/O . . . 407
29.12.3 Resizing . . . 407
29.12.4 Rotation . . . 407
29.12.5 EXIF Orientation . . . 408
29.12.6 HEIC to PNG Conversion . . . 409
29.13 IniFile . . . 411
29.14 JsonPath . . . 412
29.15 Base64 . . . 413
29.16 URLBuilder . . . 413
29.17 XML . . . 414
29.18 LRUCache and LRUSet . . . 414
29.18.1 LRUCache . . . 414
29.18.2 LRUSet . . . 415
29.19 ThreadLevelCache . . . 415
29.20 ArrayUtils . . . 415
29.21 BuildUtils . . . 416
29.22 JsonTemplate . . . 416
29.23 GroovyUtils . . . 417

29.24Summary 417

30 Desktop Applications **419**
30.1 Overview 419
30.2 Prerequisites 419
30.3 How It Works 420
30.4 Launching the Desktop Application 420
30.5 Backend URL Detection 420
30.6 Development Mode 421
30.7 Packaging and Distribution 421
30.8 Differences from Web Deployment 422

31 Testing **423**
31.1 Running Tests 423
31.2 The Existing Test Suite 423
31.3 Test Dependencies 424
31.4 Writing Tests 425
31.4.1 Common Assertions 425
31.4.2 Setup and Teardown 426
31.4.3 Parameterized Tests 426
31.5 Testing Database Operations 427
31.6 Testing Services 428
31.7 Testing Strategies 429

32 Deployment **431**
32.1 Building a WAR File 431
32.2 Web Configuration Variants 432
32.3 Deploying to Tomcat 433
32.3.1 Embedded Tomcat 433
32.3.2 External Tomcat 433
32.4 Other Servlet Containers 434
32.5 Split-System Deployment 434
32.5.1 Frontend Server 435
32.5.2 Backend Server 435
32.5.3 Connecting Them 435
32.6 Production Configuration Checklist 435
32.7 Logging in Production 436
32.8 Performance Tuning 437

33 Building a Complete Application **439**
33.1 Requirements . . . 439
33.2 Database Schema . . . 440
33.3 Backend Services . . . 441
33.3.1 DepartmentService . . . 441
33.3.2 EmployeeService . . . 442
33.3.3 ProjectService . . . 444
33.3.4 AssignmentService . . . 445
33.4 Frontend Screens . . . 447
33.4.1 Employee List Screen . . . 447
33.4.2 Project Dashboard . . . 451
33.5 Reports . . . 453
33.5.1 PDF Status Report . . . 453
33.5.2 CSV Export . . . 454
33.6 Navigation . . . 455
33.7 Lessons Learned . . . 456

VI Appendices **459**

A API Quick Reference **461**
A.1 Connection . . . 461
A.2 Record . . . 462
A.3 Command . . . 462
A.4 Cursor . . . 463
A.5 QueryBuilder . . . 463
A.6 SortBuilder . . . 464
A.7 SchemaGraph . . . 464
A.8 Frontend: Server.js . . . 465
A.9 Frontend: Component Methods . . . 465
A.10 Frontend: Utils.js . . . 466
A.11 Frontend: DateUtils.js . . . 466
A.12 Frontend: AGGrid . . . 467

B Groovy Primer **469**
B.1 Why Groovy? . . . 469
B.2 Basic Syntax . . . 469
B.3 Differences from Java . . . 470
B.4 Variables and Types . . . 470
B.5 Closures . . . 471
B.6 String Features . . . 471

B.7 Collections 472
B.8 Control Structures 473
B.9 Null Safety 473
B.10 Common Patterns in Kiss Services 473
B.11 Working with Java 474
B.12 Common Pitfalls 474

C IDE Setup **477**
C.1 General Approach 477
C.2 IntelliJ IDEA 478
C.2.1 Community Edition 478
C.2.2 Ultimate Edition 478
C.3 Eclipse 479
C.4 NetBeans 480
C.5 VS Code 480
C.6 General Tips 481

Part I

Getting Started

Chapter 1

Introduction

The *Kiss Framework* is an application development framework for building web-based business applications, portable desktop applications, and command-line utilities. Kiss focuses on simplicity and development speed, abstracting away infrastructure so that developer effort is maximally applied to the business problem.

Note

The Kiss framework has undergone a security audit spanning authentication and session management, the OAuth authorization server, database access, request and file handling, the cryptographic utilities, and the frontend. Issues identified during the audit have been resolved.

1.1 Design Philosophy

Kiss is built on several core principles:

Simplicity
: Minimal configuration, no annotations, no XML wiring. A service is a class with methods; nothing more is required. There are no dependency injection containers, no aspect-oriented programming, and no code generators.

Hot Reload
: Backend and frontend code can be changed while the system is running. There is no manual build step, no restart, and no

redeploy during development. Save a file, reload the browser, and the new code runs immediately. Despite being compiled on the fly, every backend service — whether written in Groovy, Java, or Common Lisp — always executes at full compiled bytecode speed. The hot-reload mechanism never falls back to interpretation; you pay no runtime performance penalty for the development convenience.

Clean Separation
: The backend never generates HTML or JavaScript. Communication between frontend and backend is strictly JSON-RPC. This means the backend and frontend can be developed, tested, and deployed independently.

Multi-Language Services
: Backend services can be written in Java, Groovy, or Common Lisp (via ABCL). Groovy is the most common choice because its concise syntax reduces boilerplate and its on-the-fly compile phase is faster than Java's, but Java is equally supported for teams that prefer static typing.

Completeness
: Kiss ships with everything needed for a business application: database access, authentication, session management, PDF reports, CSV export, file uploads, scheduled tasks, and LLM integration. There is no need to assemble dozens of third-party libraries.

1.2 What Kiss Provides

A Kiss application comes with the following out of the box:

- A backend server (Tomcat 11) with JSON-RPC services that auto-compile on change.
- A frontend component library with custom HTML tags for forms, grids, popups, and file uploads.
- A database abstraction layer supporting PostgreSQL, MySQL, SQLite, SQL Server, and Oracle.
- An automatic join-resolution query builder based on foreign key graph analysis.

- Built-in authentication and session management with configurable inactivity timeouts.

- PDF report generation via Groff (which must be installed on the host system; see Chapter 21), CSV export, and temporary file management.

- A build system that requires no Maven, Gradle, or other external build tools.

- Electron compatibility for packaging as a desktop application.

1.3 What Kiss Is Not

Kiss is not a microservices framework, a REST API toolkit, or a single-page application library. It is an integrated full-stack platform where the frontend and backend are designed to work together out of the box.

Kiss does not use REST conventions (resources, HTTP verbs, status codes). Despite the `/rest` endpoint name, all communication uses JSON-RPC: every request is a POST with a service name, method name, and a JSON payload.

Kiss does not generate HTML on the server. There are no template engines, no server-side rendering, and no view models. The backend returns data; the frontend decides how to display it.

1.4 Who This Book Is For

This book is for developers with working knowledge of Java, SQL, HTML, CSS, and JavaScript who want to build business applications efficiently. No prior experience with Groovy, Lisp, or Kiss is assumed.

If you are an experienced Java developer, you will find that Kiss lets you build a working application in a fraction of the code and time that a Spring Boot or Jakarta EE project would require. If you are a full-stack developer accustomed to Node.js or Python, you will appreciate the zero-configuration service model and the hot-reload development cycle.

1.5 How to Read This Book

The chapters are ordered as a tutorial: each builds on the ones before it. However, each chapter is also designed to stand alone as a reference. If you already know the basics, skip to the chapter you need and follow the cross-references backward as necessary.

Tutorial track
: Read Parts I through IV in order. By Chapter 17, you will have built screens that talk to backend services, query a database, and display results in a grid.

Reference track
: Jump to the chapter you need. Appendix A provides quick-lookup tables for every major API.

Capstone
: Chapter 33 walks through a complete application from schema design to deployment.

1.6 Conventions Used

- `Monospace text` denotes class names, method names, file paths, and configuration keys.
- Code examples are shown in shaded listings with syntax highlighting.

Three types of callout box appear throughout the book:

Tip

A shortcut, best practice, or productivity hint.

Warning

A common mistake, pitfall, or behavior that surprises newcomers.

Note

Supplementary information that provides context or background.

Chapter 2

Setup and Installation

This chapter walks through downloading, installing, and running Kiss for the first time. By the end you will have a working development environment with both the frontend and backend servers running.

2.1 Prerequisites

Kiss requires:

- **Java 17 or later** (JDK, not just JRE). Kiss compiles and runs Java and Groovy code at runtime, so a full JDK is required. Verify with `java -version` and `javac -version`.
- **A modern web browser** — Chrome, Firefox, Edge, or Safari.
- **Optionally, an IDE** for code editing. See Appendix C for setup instructions. Any text editor works; Kiss does not depend on IDE tooling.

Note

Kiss includes its own build system, an embedded Tomcat server, and all required JAR dependencies. No Maven, Gradle, or Node.js installation is necessary.

Warning

Windows requires several additional system-level dependencies (such as a package manager and supporting tools) that Linux and macOS typically provide out of the box. These are not duplicated here — consult the *Kiss Getting Started Guide* for the complete Windows setup procedure before proceeding.

2.2 Downloading Kiss

Kiss is distributed as a self-contained directory. Clone the repository or download and extract the archive:

```
git clone https://github.com/blakemcbride/Kiss.git
cd Kiss
```

Alternatively, download a release archive from the project's GitHub page, extract it, and `cd` into the resulting directory.

2.3 Directory Structure

The top-level directory contains everything needed to build and run the application:

```
Kiss/
  src/
    main/
      core/         # Core framework (do not modify)
      backend/      # Your application services
      frontend/     # Your application screens
        kiss/       # Framework components (do not modify)
      precompiled/  # Shared Java utilities
    test/           # Unit tests
  libs/             # Third-party JAR dependencies
  manual/           # Documentation source
  tomcat/           # Embedded Tomcat server
    logs/           # Server log files
  work/             # Build output
```

Warning

Two directories must not be modified:

- `src/main/core/` — the Kiss framework's Java classes.
- `src/main/frontend/kiss/` — the Kiss frontend component library.

Your application code goes in `src/main/backend/` (services) and `src/main/frontend/` (screens, excluding the `kiss/` subdirectory).

2.4 First Run

Start both the frontend and backend development servers with a single command:

```
./bld -v develop          [Linux, macOS, BSD, etc.]
     -or-
bld -v develop            [Windows]
```

Warning

On Linux, macOS, BSD, and other Unix-like systems, run the script as `./bld` (the leading `./` is required). On Windows, run it as `bld` (without the `./` prefix), which invokes the `bld.cmd` batch file. All examples in this book use the `./bld` form; Windows users should drop the `./` prefix in every command.

This command:

1. Compiles the core framework and any precompiled utilities.
2. Starts the backend server on port 8001.
3. Starts a frontend development server on port 8000.
4. Watches for file changes and recompiles automatically.

Open `http://localhost:8000` in a browser. The login page should appear.

Tip

Keep the terminal running. You can edit backend and frontend files while the system is running, and changes will take effect without a restart.

2.5 Verifying the Installation

Log in with the default credentials: username `kiss`, password `password`. These credentials are valid when using the bundled SQLite demo database.

After logging in, the example application's navigation menu appears. Click through the screens to confirm that the database, frontend, and backend are communicating correctly:

Controls
: A showcase of every Kiss UI component: text inputs, dropdowns, date pickers, checkboxes, radio buttons, and more.

CRUD
: A phone book application demonstrating create, read, update, and delete operations with a data grid, popup editor, PDF report, and CSV export.

REST Services
: Examples of calling backend service methods and displaying results.

Users
: User management for the built-in authentication system.

File Upload
: Demonstrates file upload handling.

2.6 The Example Application

Kiss ships with a working example application that serves two purposes: it verifies that everything is installed correctly, and it provides a template for your own code. The example includes:

- A login page with session management.

- A navigation framework that loads screens dynamically.
- A CRUD screen (phone book) with a grid, popup editor, PDF report, and CSV export.
- A controls showcase demonstrating every UI component.
- A file upload screen.
- An AI/LLM integration screen (requires Ollama).

All example services are in `src/main/backend/services/` and all example screens are in `src/main/frontend/screens/`. Study these files as you read through the book — they are the simplest working examples of each feature.

2.7 Development URLs

During development, two servers run simultaneously:

URL	Purpose
`http://localhost:8000`	Frontend (HTML/CSS/JS)
`http://localhost:8001`	Backend (JSON-RPC services)

Both ports come from a single `portBase` variable in `Tasks.java` (front end = `portBase`, back end = `portBase` + 1; default 8000), so several Kiss applications can run in development at once by giving each a different base — see Section 4.5. Pages served by the frontend dev server call the backend at their own port + 1 automatically. In production, both are served from the same Tomcat instance (see Chapter 32).

2.8 Server Logs

Backend log output is written to `tomcat/logs/catalina.out`. This file contains startup messages, service compilation events, request logs, and any errors thrown by backend code.

On Linux, macOS, BSD, and other Unix-like systems, use the bundled `view-log` script to tail the log:

```
./view-log
```

On Windows, the log window opens automatically when the backend starts — no separate command is needed.

This is the first place to look when something goes wrong.

Chapter 3

Architecture Overview

This chapter describes how the major pieces of Kiss fit together: the frontend, the backend, the database layer, and the communication protocol that connects them. Understanding this architecture will make every subsequent chapter easier to follow.

3.1 High-Level View

A Kiss application has three tiers:

1. **Browser** — HTML, CSS, and JavaScript served as static files. All UI rendering happens here. The browser never receives generated HTML from the server.

2. **Backend server** — A Java servlet running inside an embedded Tomcat instance. It receives JSON-RPC requests, executes service methods, and returns JSON responses.

3. **Database** — Any of the five supported engines: PostgreSQL, MySQL, SQLite, SQL Server, or Oracle.

Note

In development, the frontend runs on a separate lightweight server and the backend runs on Tomcat — by default ports 8000 and 8001, both derived from the application's `portBase` (Section 4.5). In production, both are served from the same Tomcat instance as a single WAR file.

3.2 Request Lifecycle

Every user action that involves the server follows the same path:

1. The browser calls `Server.call("ServiceName", "methodName", data)`.
2. `Server.js` sends a JSON-RPC POST request to the `/rest` endpoint with the service name, method name, input data, and the session UUID.
3. `MainServlet` receives the HTTP request and starts an async context so the Tomcat connector thread is not blocked.
4. The request is submitted to the `QueueManager`, which manages a fixed-size thread pool.
5. A worker thread picks up the request. `ProcessServlet` obtains a database connection from the C3P0 connection pool, validates the session UUID, and locates the service class.
6. The service method executes. It receives four parameters: the input JSON, an output JSON object, a database `Connection`, and the `ProcessServlet` instance.
7. On success, the framework commits the transaction and sends the output JSON back to the browser with `_Success: true`.
8. On failure, the framework rolls back the transaction and sends an error response with `_Success: false`.

Tip

Because the framework handles connection management, transaction commit/rollback, authentication, and JSON serialization, a typical service method contains only business logic — no boilerplate.

3.3 Service Discovery and Language Support

When a request arrives, `ProcessServlet` searches for the service class in the following order:

1. **Groovy** (`.groovy`) — looks in `src/main/backend/services/` for a matching file.

2. **Java** (`.java`) — same directory, Java source file.

3. **Common Lisp** (`.lisp`) — same directory, ABCL Lisp source.

The first match wins. All three languages are auto-compiled when the source file changes. No build step is required during development.

Note

Groovy is the most common choice for application services. It offers concise syntax, optional typing, closures, and seamless Java interoperability, and the on-the-fly compile phase for Groovy services is faster than for Java services — which makes the edit-save-reload cycle noticeably snappier in development. Chapter 7 covers Groovy in detail.

3.4 Hot Reload

Kiss detects file changes and recompiles automatically:

Backend services
: When you save a Groovy or Java file in `services/`, Kiss detects the change on the next request, recompiles the file, and loads the new class. There is no restart and no delay beyond the compilation itself.

Frontend files
: HTML, CSS, and JavaScript are served directly. Save the file, reload the browser, and the new code runs.

Precompiled classes
: Files in `src/main/precompiled/` are compiled by `./bld build`, not on-the-fly. After changing a precompiled class, run `./bld -v build` to recompile.

Note

On-the-fly compilation does *not* mean interpreted execution. Every Kiss web service — Groovy, Java, or Common Lisp — is compiled to JVM bytecode and runs at full compiled speed. The framework recompiles only when source changes; subsequent requests reuse the cached bytecode. There is no runtime performance penalty for the hot-reload convenience.

3.5 Thread Model

The `QueueManager` maintains a fixed-size pool of worker threads. The pool size is set by `MaxWorkerThreads` in `application.ini` (see Chapter 5). If the setting is absent, the framework uses a default based on the number of CPU cores.

Each request runs on its own thread with its own database `Connection`. This model:

- Is simple — no shared mutable state between requests.
- Scales naturally — the thread pool limits concurrency to a safe level.
- Avoids deadlocks — each thread owns exactly one database connection.

3.6 The JSON-RPC Protocol

Despite the `/rest` endpoint name, Kiss uses JSON-RPC, not REST. Every request is an HTTP POST containing a JSON object with at least three fields:

```
{
    "_class": "ServiceName",
    "_method": "methodName",
    "_uuid": "session-token",
    // ... application data ...
}
```

Every response is a JSON object containing:

```
{
```

```
    "_Success": true,     // or false
    "_ErrorCode": 0,      // 0 = success, 2 = auth failure
    "_ErrorMessage": "",  // present on error
    // ... application data ...
}
```

This uniform protocol means there is no need to design URL schemes, choose HTTP verbs, or interpret status codes. Every call is the same shape.

3.6.1 REST vs. JSON-RPC

The two styles solve the same problem — letting a client invoke server-side behavior over HTTP — but they take very different shapes.

REST models the server as a collection of *resources*. Each resource has a URL (e.g., `/employees/42`), and the operation is encoded in the HTTP verb (`GET` to read, `POST` to create, `PUT`/`PATCH` to update, `DELETE` to remove). Outcomes are signaled through HTTP status codes (`200`, `201`, `404`, `409`, ...). REST works well when the API really is resource-oriented and when third-party clients benefit from a uniform, cacheable, hyperlinked interface.

JSON-RPC models the server as a collection of *procedures*. Every call is a POST of a JSON object that names a class and a method and carries the arguments; every response is a JSON object that carries the return value (or an error). There is one URL, one verb, and one response shape.

For the kind of business applications Kiss is designed to build, JSON-RPC is the better fit:

- **Operations rarely map cleanly to CRUD on a resource.** "Approve this invoice and notify the requester," "run the month-end close," or "transfer funds between accounts" are procedures, not nouns. Forcing them into REST verbs and URLs adds modeling work that produces no real benefit.

- **Uniform shape, uniform tooling.** Because every call has the same shape, the framework can handle authentication, transaction management, JSON parsing, error reporting, and logging in one place. A service method receives `injson`, `outjson`, `db`, and `servlet` — nothing more — and the rest is

automatic. REST's verb/URL/status-code matrix would have to be re-implemented per endpoint.

- **Easier evolution.** Adding a parameter, returning an extra field, or deprecating a method is just a code change. There are no URL schemes to redesign and no HTTP semantics to renegotiate.
- **Errors are first-class data.** Application-level failures (validation errors, business-rule violations) come back as fields in the response body, not as overloaded HTTP status codes that must be reverse-engineered by the client.
- **Simpler client code.** The frontend uses a single `Server.call(class, method, data)` function for every backend interaction. There is no client-side router, no per-endpoint wrapper, and no debate about whether something should be a `PUT` or a `POST`.

REST remains the right choice when you are publishing a public, resource-oriented API for arbitrary third-party consumers. Kiss assumes a different scenario: an application's own frontend talking to its own backend, where shared, uniform plumbing pays for itself many times over. The `/rest` endpoint name is preserved purely for historical reasons; the protocol on the wire is JSON-RPC.

3.7 Authentication Flow

Authentication is built into the request lifecycle:

1. The login screen sends the username and password to the `Login` service.
2. `Login` validates the credentials and creates a `UserData` object in the `UserCache`. A UUID is generated and returned to the browser.
3. `Server.js` stores the UUID and includes it in every subsequent request.
4. On each request, `ProcessServlet` looks up the UUID in the `UserCache`. If the UUID is missing or the session has expired, the request is rejected with error code 2.

5. The browser detects error code 2 and redirects to the login screen.

Services can be exempted from authentication by calling `MainServlet.allowWithoutAuthentication()` in `KissInit.groovy` (see Chapter 20).

3.8 Database Connection Lifecycle

Each request receives a database `Connection` from a C3P0 connection pool. The connection's auto-commit is disabled. The lifecycle is:

1. `ProcessServlet` obtains a connection from the pool before invoking the service method.
2. The service method uses the connection for queries, inserts, updates, and deletes.
3. If the service method returns normally, the framework calls `commit()`.
4. If the service method throws an exception, the framework calls `rollback()`.
5. The connection is returned to the pool.

Warning

Because auto-commit is off, changes are not visible to other connections until the service completes successfully. If you need to commit partway through a long operation, call `db.commit()` explicitly.

3.9 Frontend Architecture

The Kiss frontend is a single-page application built with plain HTML, CSS, and JavaScript. There is no build step, no transpiler, and no framework beyond Kiss's own component library. All DOM manipulation is performed through `DOMUtils.js`, a thin wrapper over the native browser DOM API; jQuery is not used.

Screens are loaded dynamically by `Utils.loadPage()`, which injects HTML into the page and executes the associated JavaScript file. Each screen is a pair of files: `ScreenName.html` and `ScreenName.js`.

Custom HTML tags (`<text-input>`, `<drop-down>`, `<push-button>`, etc.) are registered by the Kiss component system and provide a uniform API for getting/setting values, validation, and event handling. Chapter 15 covers every component in detail.

3.10 Putting It All Together

A typical user interaction:

1. The user clicks a "Save" button on a screen.
2. The button's `onclick` handler calls `Server.call("EmployeeService", "save", data)`.
3. The request travels through MainServlet → QueueManager → ProcessServlet.
4. `ProcessServlet` verifies the session, obtains a database connection, and invokes `EmployeeService.save()`.
5. The service method reads input from `injson`, updates the database, and writes a response to `outjson`.
6. The framework commits the transaction and sends the JSON response.
7. The browser callback receives the response and updates the screen.

Every interaction follows this pattern. Once you understand it, you understand Kiss.

Chapter 4

The Build System

Kiss uses a custom build system centered on the `bld` script. This chapter covers the build commands, how incremental compilation works, and how to generate deployable artifacts. No Maven, Gradle, or other external build tool is required.

4.1 Why `bld` Instead of Maven or Gradle?

A reasonable first question is why Kiss ships its own build system at all when Maven and Gradle are widely used and well understood.

The short answer: Maven and Gradle work very well for applications that conform to their standards — a fixed source layout, a fixed lifecycle, dependency resolution from public repositories, artifacts compiled ahead of time and packaged into a JAR or WAR. Kiss does not fit cleanly into that mould. Its defining feature is that backend service code is compiled *at runtime* when source files change, not as a separate build step, and that the running server itself participates in the development loop. Wedging that workflow into a Maven lifecycle or a Gradle task graph requires fighting the tool at every turn.

`bld` is far simpler and more flexible. It is a small Java program wrapped by a one-page shell script. It performs smart, timestamp-based incremental builds; it knows how to start and stop the embedded Tomcat, run tests, generate JavaDoc, and package a WAR; and because it is just Java code, customizing it is the same as editing any other class in the project — no plugins, no DSL, no third-party lifecycle to learn. The result is a build system that supports the dynamic nature of Kiss directly instead of working around it.

The build system is organized as three Java source files, layered by how widely each is shared:

`BuildUtils.java` (in `src/main/core/org/kissweb/`) The fully generic build utilities: compiling Java, creating and unpacking JARs, downloading dependencies, copying file trees, and running external commands. It contains nothing specific to Kiss or Tomcat, so it can also serve as the engine of a generic build system in projects unrelated to Kiss.

`KissBuildUtils.java` (in `src/main/core/org/kissweb/`) Build procedures shared by every Kiss application: the development port block and the port-override option parsing (Section 4.5), Tomcat installation and control (`installTomcat`, `shutdownTomcat`, `rewriteServerXmlPorts`), and the version stamping applied to production WARs for cache busting.

`Tasks.java` (in `src/main/precompiled/`) The application-specific layer: this application's tasks, dependency lists, third-party versions, and its `portBase` (Section 4.5). This is the only build file an application normally edits.

When you add build functionality of your own, application-specific work goes in `Tasks.java`. Something any Kiss application could use belongs in `KissBuildUtils.java`; something universally generic belongs in `BuildUtils.java`.

4.2 The `bld` Script

The `bld` script (with a `bld.cmd` counterpart for Windows) is the single entry point for all build operations. It is a thin shell wrapper that:

1. Checks whether the build engine source files (`Tasks.java`, `BuildUtils.java`, and `KissBuildUtils.java`) are newer than their compiled classes.
2. If a rebuild is needed, compiles them with `javac`.
3. Invokes the compiled build system, passing along any command-line arguments.

This two-stage design means the build engine itself rebuilds only when its source changes.

Note

On Linux, macOS, BSD, and other Unix-like systems, invoke the script as `./bld` (the leading `./` is required because the current directory is not on `PATH`). On Windows, invoke it without the prefix as `bld` — this runs the `bld.cmd` batch file. Throughout this book, examples use the `./bld` form; Windows users should drop the `./` prefix.

4.3 IDE Integration via Ant (`build.xml`)

Kiss includes an Ant build file, `build.xml`, in the project root. Its sole purpose is to act as a bridge between IDEs and `bld`: most IDEs ship with first-class Ant integration, so an Ant script gives them a familiar way to invoke Kiss's native build system.

The targets in `build.xml` do not reimplement the build — each one simply shells out to `bld` (or directly invokes the compiled `Tasks` class) for the corresponding action. IDE users get the standard Ant tool window with build, clean, test, and run targets, and those targets drive exactly the same code paths that running `bld` on the command line would.

If you build only from the terminal you can ignore `build.xml` entirely; it exists for the convenience of IDE users.

4.4 Build Commands

4.4.1 Development

```
# Build and start frontend + backend servers
./bld develop
# Build and start backend only (background)
./bld start-backend
./bld stop-backend  # Stop background backend server
./bld start-frontend # Start frontend dev server only
./bld stop-frontend # Stop frontend dev server
```

`./bld develop` is the primary development command. It compiles everything, starts Tomcat on the back-end port (8001 by default),

starts the frontend server on the front-end port (8000 by default — the URL you browse to), and watches for changes. See Section 4.5 for how the ports are assigned.

If the backend is not running when `stop-backend` is invoked (for example, Tomcat exited after a startup error), the command prints one clear line pointing at `tomcat/logs/catalina.out` rather than a stack trace.

4.4.2 Build and Package

```
./bld -v build      # Compile all source code (verbose)
./bld war           # Create deployable WAR file
# Build Kiss.jar (reusable utility library)
./bld jar
./bld javadoc       # Generate JavaDoc documentation
```

4.4.3 Cleanup

```
# Remove compiled files (keep JARs and Tomcat)
./bld clean
# Remove build artifacts, JARs, and Tomcat
./bld realclean
./bld ideclean      # Total cleanup including IDE files
```

4.4.4 Testing

```
./bld unit-tests    # Build and run unit tests
```

4.5 Development Ports: the Port Block

The embedded Tomcat and the frontend development server bind to four ports. All four come from a *single variable* at the top of `Tasks.java`:

```
private static int portBase = 8000;
```

`Tasks.main()` passes it to `KissBuildUtils.setPortBase()`, which assigns the four ports as one consecutive block:

Port	Default	Purpose
`portBase`	8000	front-end dev server (the URL you browse to)
`portBase + 1`	8001	back-end (Tomcat HTTP)
`portBase + 2`	8002	Tomcat shutdown
`portBase + 3`	8003	JDWP debug

The point of the block is running *several unrelated Kiss applications in development mode at the same time*: give each application its own `portBase` (say 8000, 8100, 8200, ...) and they never collide. Because the base lives in `Tasks.java`, the choice is made once per application and is checked into that application's repository.

The front end automatically finds the back end under this convention — a page served by the front-end dev server calls the back end at its own port + 1, and pages served by Tomcat itself use their own origin — so changing `portBase` requires no front-end configuration. Section 12.9 describes the mechanism.

4.5.1 Port Overrides

Each of the four ports can additionally be overridden for a single run from the `bld` command line. The overrides are parsed *after* `setPortBase` runs, so they always win over the block.

Short	Long	Default	Purpose
`-dp PORT`	`--debug-port=PORT`	8003	JDWP debug port
`-bp PORT`	`--backend-port=PORT`	8001	development back-end port
`-sp PORT`	`--shutdown-port=PORT`	8002	Tomcat shutdown port
`-fp PORT`	`--frontend-port=PORT`	8000	development front-end port

Options may appear in any position on the command line and are consumed before the task name is dispatched. Both short and long

forms are accepted; the long form may use either a separate argument or an `=` suffix. `./bld help` always shows the live defaults — the values produced by this application's `portBase`.

```
./bld -bp 8090 -fp 8010 develop
./bld --backend-port=8090 --frontend-port=8010 develop
```

Warning

An override that breaks the consecutive-block relationship (for example `-fp` alone) also breaks the front end's port + 1 discovery of the back end. For a one-off unusual topology, set `SystemInfo.backendUrl` explicitly (Section 12.9).

The HTTP and shutdown ports are written into `tomcat/conf/server.xml` on every `setup-tomcat` run; the debug port is written into `tomcat/bin/debug`; the frontend port is passed on the `SimpleWebServer` command line. All four values are re-applied on every `bld` invocation, so different runs can pick different ports with no manual cleanup.

Non-numeric or out-of-range values are reported on `stderr` and the default is retained for that port.

4.6 Compilation Order

The build compiles in this order:

1. **Core framework files** (`src/main/core/`) — the Kiss framework itself. You do not modify these.

2. **Precompiled utilities** (`src/main/precompiled/`) — shared Java classes for your application.

3. **Test files** (`src/test/core/`) — JUnit test classes.

All compiled classes go to `work/exploded/WEB-INF/classes/`.

Note

Backend service files (`.groovy`, `.java`, `.lisp`) in `src/main/backend/services/` are *not* compiled by the build system. They are compiled on-the-fly by the framework when first accessed or when changed.

4.7 Incremental Compilation

The build system tracks file timestamps and only recompiles files that have changed. `BuildUtils.java` provides the infrastructure for this, including methods for checking file modification times and running shell commands.

4.8 The `precompiled` Directory

Place shared Java utility classes in `src/main/precompiled/` when they need to be accessible from Groovy services at compile time. These classes are compiled by `./bld build` and placed on the classpath.

Warning

Classes in the `precompiled` directory **must have a package declaration** to be importable from Groovy services. Classes in the default package (no package declaration) cannot be accessed from packaged Groovy code — this is a Java/Groovy language restriction, not a Kiss limitation.

```java
// src/main/precompiled/com/myapp/utils/IdGenerator.java
package com.myapp.utils;

public class IdGenerator {
    public static String generate() {
        return java.util.UUID.randomUUID().toString();
    }
}
```

The directory structure must match the package declaration. After adding or modifying precompiled classes, run `./bld -v build`.

4.9 WAR Generation

```
./bld war
```

This produces `work/Kiss.war`, a standard WAR file deployable to any Jakarta EE 11 (Servlet 6.1) container.

The WAR build process:

1. Compiles all source code.
2. Copies `web-secure.xml` (restrictive CORS) into `web.xml` for production packaging.
3. Packages everything into a standard WAR file.
4. Copies `web-unsafe.xml` back into `web.xml` so local development continues with permissive CORS.

4.10 JavaDoc Generation

```
./bld javadoc
```

Generates JavaDoc for the backend API in `work/javadoc/`. This documents the core framework classes (Connection, Record, QueryBuilder, SchemaGraph, etc.) and any precompiled application classes.

4.11 Helper Scripts

Kiss includes several convenience scripts in the project root:

Script	Purpose
`serve`	Start the frontend development server on port 8000.
`is-tomcat-running`	Check if the backend server is running.
`kill-tomcat`	Force-stop the Tomcat process.
`view-log`	Tail the server log (`tomcat/logs/catalina.out`).

Script	Purpose
`upgrade-kiss`	Update the framework from a fresh Kiss clone using `rsync`.
`new-app`	Remove git history and documentation to start a new application from the Kiss template.
`build-dist`	Package the application with Tomcat into a distributable zip.
`shrink`	Minify JavaScript files using UglifyJS.
`expand`	Restore original (unminified) JavaScript files.

4.12 Dependency Management

Third-party JAR files are stored in the `libs/` directory. The `./bld libs` command downloads all required dependencies. Key libraries include:

- **Groovy** — dynamic language support.
- **C3P0** — database connection pooling.
- **Log4j** — logging framework.
- **PDFBox** — PDF generation and manipulation.
- Database drivers for PostgreSQL, MySQL, SQLite, SQL Server, and Oracle.
- **JUnit 5** — unit testing framework.

Tip

To add a new dependency, place the JAR in `libs/` and it will automatically be included on the classpath during builds and at runtime. No configuration file changes are needed.

4.12.1 Foreign Dependencies and Superseded Versions

Two kinds of JAR live in `libs/`. *Local* dependencies are simply dropped into the directory and committed with the project,

as the tip above describes. *Foreign* dependencies are declared in `Tasks.java`, in `buildForeignDependencies()`, as a repository URL each; `BuildUtils.downloadAll()` fetches them on demand and they are normally excluded from version control. Upgrading one is a matter of editing its version in that URL.

That upgrade has a subtlety worth understanding. Downloading is keyed on the file name, and the new version has a new name, so the file holding the *previous* version is not overwritten — it simply stays where it is. Both versions then sit in `libs/` and both land on the classpath, at which point which one the class loader actually uses is arbitrary. The consequences are unpleasant precisely because they are so far removed from the cause: a library may load but be missing a class its companion library expects, and the resulting `ClassNotFoundException` arrives during startup, in code nowhere near the dependency you edited.

`downloadAll()` therefore prunes superseded versions after it finishes downloading, reporting each file it removes. The rule it applies is deliberately narrow. A file is deleted only when it sits in the same target directory as a current dependency, its artifact name is *exactly* that dependency's artifact name, it carries a version — meaning the artifact name is followed by a hyphen and a digit — it shares the same extension, and it is not itself one of the wanted files.

Insisting on an exact artifact-name match is what makes this safe, because a great many real artifacts are prefixes of their siblings. Pruning `pdfbox` must not disturb `pdfbox-io-3.0.5.jar`, and pruning `junit-jupiter` must not disturb `junit-jupiter-api-5.11.0.jar`. In both cases the text following the prefix is another name rather than a version, so neither file is a candidate. The same reasoning keeps `log4j-api` and `log4j-core` independent of one another. The artifact name itself is taken to be everything ahead of the *last* hyphen followed by a digit, which is what allows an artifact whose own name ends in a digit, such as `commons-lang3`, to be parsed correctly.

Note

Pruning only ever considers artifacts that are actually declared as foreign dependencies. A JAR you placed in `libs/` yourself is never touched, even when it shares a prefix with a managed artifact. Nor is the download cache pruned: reverting to an earlier version restores it from the cache rather than downloading it again.

4.12.2 Where `pom.xml` Comes From

Kiss ships a `pom.xml`, which is easy to misread as a sign that Maven builds the project. It does not. The file exists so that an IDE can understand the project layout, and so that repository scanners — the automated services that watch a project's declared dependencies and report known vulnerabilities in them — can see what the project uses.

That combination is a trap. Because no build ever reads the dependency list in `pom.xml`, nothing fails when it falls behind the list the build actually downloads. A hand-maintained copy drifts quietly, and the drift is invisible until a scanner reports a vulnerability against a version the project abandoned long ago — or, worse, stays silent about one it still uses. A second copy of a list that nothing validates is a liability rather than documentation.

The `pom` task removes the second copy. Everything between the two marker comments in `pom.xml`,

```
<!-- BEGIN GENERATED DEPENDENCIES -->
<!-- END GENERATED DEPENDENCIES -->
```

is written from the same `ForeignDependencies` the build downloads, with coordinates recovered from each dependency's repository URL. The task runs as part of `libs`, so an ordinary build keeps the file current, and it rewrites the file only when the generated text actually differs, leaving the timestamp alone otherwise. **Dependencies are therefore edited in `Tasks.java` and never in `pom.xml`.**

Only the marked region is generated. The `build` element — the source roots, the enforcer configuration, and the `antrun` executions that let `mvn antrun:run@`*`task`* invoke `bld` tasks — is hand-maintained and left untouched, as are the project coordinates above it.

The division of labor follows the usual rule. The generic machinery lives in `BuildUtils`: `mavenCoordinatesFromUrl()` recovers a group, artifact, and version from a repository URL, and the `MavenDependencies` builder assembles the elements and splices them into the file. `Tasks.java` contributes only what is particular to the application: which artifacts are needed solely for tests, and which JARs are supplied locally rather than downloaded and so must be emitted with system scope.

Note

A URL that does not follow the Maven layout of `group/artifact/version/artifact-version.ext` yields no coordinates and is passed over rather than guessed at, so a dependency fetched from a CDN or any other non-repository source simply does not appear in the generated list.

Chapter 5

Configuration

Kiss is configured through a small number of files. This chapter covers each one and explains every setting.

5.1 `application.ini`

The primary configuration file, located at `src/main/backend/application.ini`. It uses a simple `key = value` format with `#` for comments. Settings are organized under an `[main]` section header.

5.1.1 Database Settings

```
[main]
DatabaseType = PostgreSQL
DatabaseHost = localhost
# DatabasePort = 5432
DatabaseName = myapp
# DatabaseSchema = public
DatabaseUser = appuser
DatabasePassword = secret
```

`DatabaseType`
: One of: `PostgreSQL`, `MySQL`, `SQLite`, `MicrosoftServer`, `Oracle`.

`DatabaseHost`
: Hostname or IP address (default: `localhost`).

`DatabasePort`
: Port number. Uses the database's default port if omitted.

`DatabaseName`
: Database name or file path (for SQLite, e.g., `DB.sqlite`).

`DatabaseSchema`
: Default schema (e.g., `public` for PostgreSQL).

`DatabaseUser`
: Database username.

`DatabasePassword`
: Database password.

`DatabaseConnectionParameters`
: Additional JDBC connection string parameters.

Note

The default `application.ini` ships configured for SQLite with no authentication, which is convenient for initial development and testing.

5.1.2 When the Configured Database Cannot Be Reached

If a database *is* configured but cannot be reached at startup — the database server is down, the database was dropped or never created, or the credentials are wrong — Kiss logs an unmistakable multi-line `FATAL` banner to `tomcat/logs/catalina.out` naming the database and the underlying error, and then fails the web application's deployment. Nothing in the application can work without its configured database, so the application is disabled rather than left half-alive.

The servlet container (Tomcat) itself is deliberately left running: a bad database must never take down the container or any other applications it hosts on a production system.

Be aware of what the disabled state looks like from the outside. With the application context down, every request receives a bare 404/503 carrying none of the application's response headers — and browsers frequently misreport that as something else entirely, most commonly a CORS policy failure on the front end. If you see

unexplained CORS errors or bare 404s immediately after a restart, check `catalina.out` for the banner before chasing anything else:

```
************************************************************
*  FATAL: cannot connect to the configured database.
*
*      <the underlying error>
*
*      Database: localhost:myapp
*
*  The application cannot run without its configured database.
*  Verify that the database server is running, that the database
*  exists, and that the Database* settings in
*  application.ini are correct; then restart.
*
*  THIS APPLICATION HAS BEEN DISABLED. The servlet container
*  itself has been left running.
************************************************************
```

Running with *no* database configured is a supported mode and is unaffected by any of this.

5.1.3 Server Settings

```
MaxWorkerThreads = 30
UserInactiveSeconds = 900
```

`MaxWorkerThreads`
: Size of the request-handling thread pool. Each concurrent request uses one thread. Default: 30.

`UserInactiveSeconds`
: Seconds of user inactivity before a session expires and the user is logged out. Default: 900 (15 minutes). Set to 0 for the framework default of 3600 seconds (1 hour).

`RequireAuthentication`
: Set to `true` to require authentication even when no SQL database is configured (useful when using an alternative data store).

`SecurityHeaders`
: Set to `false` to disable the browser security-header filter (CSP and friends) for this deployment — chiefly for an application

whose front end predates the bootstrap kernel whose hash the CSP pins. Default: enabled. See Chapter 14.

`CspReportOnly`
: Set to `true` to deliver the Content-Security-Policy in report-only form (violations logged, nothing blocked) for a validation pass. Default: enforcing. See Chapter 14.

5.1.4 Connection Pool Settings

Kiss uses C3P0 for connection pooling. The framework provides sensible defaults based on CPU core count, but all settings can be overridden:

```
# DatabaseMinPoolSize = 10
# DatabaseInitialPoolSize = 10
# DatabaseMaxPoolSize = 100
# DatabaseAcquireIncrement = 5
# DatabaseCheckoutTimeout = 10000
# DatabaseMaxStatements = 500
# DatabaseMaxStatementsPerConnection = 25
# DatabaseIdleTestPeriod = 300
# DatabaseMaxIdleTime = 1800
# DatabaseTestOnCheckout = false
# DatabaseTestOnCheckin = true
# DatabaseHelperThreads = 10
# DatabaseUnreturnedTimeout = 60
# DatabaseDebugStackTraces = false
```

`DatabaseMinPoolSize`
: Minimum connections to maintain in pool.

`DatabaseMaxPoolSize`
: Maximum connections allowed. For high-load production, set to at least CPU cores × 4.

`DatabaseCheckoutTimeout`
: Milliseconds to wait for a connection before throwing an error.

`DatabaseMaxStatements`
: Total prepared statements to cache across all connections.

`DatabaseIdleTestPeriod`
: Seconds between testing idle connections.

`DatabaseMaxIdleTime`
: Seconds before removing idle connections.

`DatabaseDebugStackTraces`
: Enable stack traces for debugging unreturned connections. Disable in production.

Tip

For most applications, the framework defaults work well. Only tune connection pool settings when you observe connection-related bottlenecks in production.

5.1.5 Accessing Configuration from Code

Configuration values can be read programmatically:

```
import org.kissweb.restServer.MainServlet

String dbHost = MainServlet.getEnvironment(
    "DatabaseHost") ?: "localhost"
String dbName = MainServlet.getEnvironment("DatabaseName")
String custom = MainServlet.getEnvironment("MyCustomSetting")
```

You can add your own application-specific settings to `application.ini` and read them the same way. Note that `getEnvironment` is declared to return `Object` (the stored values are strings); Groovy assigns the result to a `String` directly, but a Java caller must cast — `(String) MainServlet.getEnvironment("DatabaseHost")`.

5.1.6 Secrets and External Configuration

Warning

All URLs, passwords, security keys, API keys, tokens, and user names must be kept in `src/main/backend/application.ini`. Nothing of this kind should be hard-coded in source files (Java, Groovy, Lisp, JavaScript, HTML, or other configuration files). Whenever such a value is found outside `application.ini`, it must be moved into `application.ini` and accessed via `MainServlet.getEnvironment()`.

The retrieval API is the same `getEnvironment` method shown above, defined in `src/main/core/org/kissweb/restServer/MainServlet.java`:

```
import org.kissweb.restServer.MainServlet

String apiUrl  = MainServlet.getEnvironment("ExternalApiUrl")
String apiKey  = MainServlet.getEnvironment("ExternalApiKey")
String smtpUser = MainServlet.getEnvironment("SmtpUser")
String smtpPass = MainServlet.getEnvironment("SmtpPassword")
```

Centralizing these values in `application.ini` provides several benefits:

- Secrets can be rotated without code changes.
- Credentials are kept out of version control when `application.ini` is excluded from the repository.
- Values are not duplicated across multiple files.
- Per-environment configuration (development, staging, production) is straightforward — only `application.ini` differs between environments.

Categories of values that must live in `application.ini`:

URLs of external services
: REST endpoints, SMTP servers, LDAP servers, third-party APIs, webhooks.

Passwords
: Database, SMTP, service accounts, third-party APIs.

Security keys
: API keys, tokens, client secrets, encryption keys, signing keys.

User names
: Account identifiers used by the application to authenticate to external systems.

5.2 web.xml

The servlet configuration lives in `src/main/core/WEB-INF/`. Kiss ships two source variants of this file:

`web-unsafe.xml`
: Development configuration. CORS filter allows any origin (`*`), enabling rapid frontend development from any port or device.

`web-secure.xml`
: Production configuration. CORS restricted to specific hosts only (e.g., `localhost:8000`).

Both register the `StartupListener` for REST server initialization and configure the CORS filter with allowed methods: GET, POST, HEAD, OPTIONS, PUT.

The deployed `web.xml` is not edited by hand — the `bld` script copies the appropriate variant into place: `web-unsafe.xml` for development builds and `web-secure.xml` when packaging a WAR for production. Any `web.xml` present in the source tree is simply a working copy left over from the most recent build and will be overwritten on the next `bld` run.

Warning

Never deploy with the permissive CORS configuration (`web-unsafe.xml`) in production. The WAR build automatically substitutes `web-secure.xml`, but verify this if you have a custom deployment process.

5.3 log4j2.xml

Kiss uses Log4j for logging. The configuration file `src/main/core/log4j2.xml` defines the logging behavior:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="INFO">
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern=
    "%d{yyyy-MM-dd HH:mm:ss} [%-5p] [%-25c{2}] - %m%n" />
        </Console>
    </Appenders>
    <Loggers>
        <Root level="error">
            <AppenderRef ref="Console" />
        </Root>
    </Loggers>
</Configuration>
```

The default configuration logs only errors to the console (which is captured in `tomcat/logs/catalina.out`).

5.3.1 Changing Log Levels

To see more detail during development, change the root level:

```
<Root level="info">   <!-- or "debug" for maximum detail -->
```

5.3.2 Using Logging in Services

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

private static final Logger logger =
    LogManager.getLogger(MyService.class);

logger.info("Processing employee " + empId);
logger.error("Failed to save record", exception);
```

For production logging best practices, see Chapter 32.

5.4 context.xml

The Tomcat context configuration at `src/main/core/META-INF/context.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<Context path="/Kiss"/>
```

This sets the application context path. Typically requires no modification.

5.5 Development vs. Production Mode

Kiss automatically adapts its behavior based on the deployment context:

Feature	Development	Production
Service compilation	Auto-compile on change	Auto-compile on change
Frontend serving	Separate server (port 8000)	Same Tomcat instance
CORS	Permissive (any origin)	Restricted
Logging	Configurable	Typically ERROR/WARN
`web.xml`	`web-unsafe.xml`	`web-secure.xml`

Tip

Service auto-compilation works in both modes. This means you can fix a production bug by editing a service file directly on the server — the change takes effect on the next request without a restart.

Part II

Backend Development

Chapter 6

Writing Backend Services

Services are the heart of a Kiss backend. A service is simply a class with methods — no configuration files, no annotations, no registration. This chapter covers how to write, organize, and debug services.

6.1 The Service Pattern

Every service method has the same four-parameter signature:

```
class MyService {
    void myMethod(JSONObject injson, JSONObject outjson,
                  Connection db, ProcessServlet servlet) {
        // Read input from injson
        // Do work (database queries, business logic)
        // Write output to outjson
    }
}
```

The four parameters are:

`injson`
: A `JSONObject` containing the data sent from the frontend.

`outjson`
: A `JSONObject` that the method populates with the response.

`db`
: A database `Connection` with auto-commit disabled. The framework commits on success and rolls back on exception.

`servlet`
: The `ProcessServlet` instance, used for file uploads and advanced features.

The framework handles everything else: authentication, JSON parsing, database connection management, transaction control, and error responses.

Tip

A service method's only job is business logic. It reads from `injson`, does work, and writes to `outjson`.

Note

Although services are compiled on the fly when their source changes, they always run at full compiled bytecode speed — never interpreted. This applies equally to Groovy, Java, and Common Lisp services. The compile step happens once per change; every subsequent request executes the cached bytecode at full JVM performance. Hot reload is a development-time convenience with no runtime cost.

6.2 Service Discovery

Service files live in `src/main/backend/services/`. The framework finds them by class name: a call to `Server.call("services/MyService", "myMethod", ...)` maps to `services/MyService.groovy` (or `.java` or `.lisp`).

The resolution order is:

1. Groovy (`.groovy`)
2. Java (`.java`)
3. Common Lisp (`.lisp`)

Subdirectories are supported. A call to `Server.call("services/admin/UserService", ...)` maps to `services/admin/UserService.groovy`.

6.3 Groovy Services

Groovy is the most common language for Kiss services. Files are automatically compiled on change — no build step required. The on-the-fly compile phase for Groovy is also faster than for Java, so the edit-save-reload cycle is noticeably quicker during development.

```
class Crud {
    void getRecords(JSONObject injson, JSONObject outjson,
                    Connection db, ProcessServlet servlet) {
        List<Record> recs = db.fetchAll(
            "select * from phone order by last_name, first_name")
        JSONArray rows = new JSONArray()
        for (Record rec : recs) {
            JSONObject row = new JSONObject()
            row.put("id", rec.getInt("rec_id"))
            row.put("firstName", rec.getString("first_name"))
            row.put("lastName", rec.getString("last_name"))
            row.put("phoneNumber", rec.getString(
                "phone_number"))
            rows.put(row)
        }
        outjson.put("rows", rows)
    }
}
```

This service fetches all phone records, builds a JSON array, and returns it. The frontend receives the `rows` array in the callback.

Note

Groovy's concise syntax means that semicolons are optional, `return` is optional for the last expression, and type declarations can use `def` instead of explicit types. See Chapter 7 for a full primer.

6.4 Java Services

Java services work identically but use `.java` files. They are also auto-compiled on change during development.

```
import org.kissweb.json.JSONObject;
import org.kissweb.database.Connection;
import org.kissweb.restServer.ProcessServlet;
```

```
public class MyJavaService {
    public void addNumbers(JSONObject injson,
        JSONObject outjson,
        Connection db, ProcessServlet servlet) {
        Integer num1 = injson.getInt("num1");
        Integer num2 = injson.getInt("num2");
        if (num1 == null || num2 == null)
            outjson.put("num3", 0);
        else
            outjson.put("num3", num1 + num2);
    }
}
```

6.5 Lisp Services

Services can be written in Common Lisp using ABCL (Armed Bear Common Lisp). This option is available for teams that prefer Lisp but is less commonly used.

6.6 Reading Input and Writing Output

Input arrives in the `injson` parameter. Output is written to `outjson`.

```
// Reading input
String name = injson.getString("name")
int id = injson.getInt("id")
double amount = injson.getDouble("amount")
boolean active = injson.getBoolean("active")

// Writing output
outjson.put("status", "success")
outjson.put("count", 42)
outjson.put("total", 1234.56)
```

Input values come directly from the JavaScript object passed to `Server.call()`. Output values become properties of the response object in the JavaScript callback.

6.6.1 Returning Record Lists

The most common pattern is to return a list of database records as a JSON array:

```
// Option 1: Build JSON manually (full control over
// field names)
List<Record> recs = db.fetchAll("select * from employee")
JSONArray rows = new JSONArray()
for (Record rec : recs) {
    JSONObject row = new JSONObject()
    row.put("id", rec.getInt("employee_id"))
    row.put("name", rec.getString("last_name"))
    rows.put(row)
}
outjson.put("rows", rows)

// Option 2: Use fetchAllJSON (column names become JSON keys)
outjson.put("rows", db.fetchAllJSON(
    "select employee_id, last_name, first_name from employee"))
```

Tip

Use `fetchAllJSON()` when the database column names match what the frontend expects. Use manual JSON building when you need to rename columns or compute derived values.

6.6.2 Returning a Single Record

```
Record rec = db.fetchOne(
    "select * from employee where employee_id = ?", id)
if (rec == null)
    throw new UserException("Employee not found")
outjson.put("firstName", rec.getString("first_name"))
outjson.put("lastName", rec.getString("last_name"))
outjson.put("salary", rec.getDouble("salary"))
```

6.7 Error Handling

Kiss provides three exception types for different error scenarios:

`UserException`
: Sends a message to the frontend for the user to see. Does *not* log a backend stack trace. Use this for validation errors and expected conditions.

```
    if (name == null || name.isEmpty())
        throw new UserException("Name is required")
```

`ServerException`
: Logs the full stack trace in the backend log and sends a generic error code to the frontend. Use this for unexpected errors where the user should not see implementation details.

`LogException`
: Logs the full stack trace *and* sends the error message to the frontend. Use this when both debugging information and a user-visible message are needed.

If a service method throws any other exception, the framework catches it, logs the stack trace, rolls back the transaction, and sends a generic error response.

6.8 A Complete CRUD Service

The following service demonstrates all four CRUD operations, a PDF report, and a CSV export:

```
package services

import org.kissweb.json.*
import org.kissweb.database.*
import org.kissweb.*
import org.kissweb.restServer.ProcessServlet

class Crud {

    void getRecords(JSONObject injson, JSONObject outjson,
                    Connection db, ProcessServlet servlet) {
        List<Record> recs = db.fetchAll(
            "select * from phone order by last_name, first_name")
        JSONArray rows = new JSONArray()
        for (Record rec : recs) {
            JSONObject row = new JSONObject()
            row.put("id", rec.getInt("rec_id"))
            row.put("firstName", rec.getString("first_name"))
            row.put("lastName", rec.getString("last_name"))
            row.put("phoneNumber", rec.getString(
                "phone_number"))
            rows.put(row)
        }
```

```
        outjson.put("rows", rows)
    }

    void addRecord(JSONObject injson, JSONObject outjson,
                Connection db, ProcessServlet servlet) {
        Record rec = db.newRecord("phone")
        rec.set("first_name", injson.getString("firstName"))
        rec.set("last_name", injson.getString("lastName"))
        rec.set("phone_number", injson.getString(
            "phoneNumber"))
        rec.addRecord()
    }

    void updateRecord(JSONObject injson, JSONObject outjson,
                    Connection db, ProcessServlet servlet) {
        Record rec = db.fetchOne(
            "select * from phone where rec_id=?",
            injson.getInt("id"))
        rec.set("first_name", injson.getString("firstName"))
        rec.set("last_name", injson.getString("lastName"))
        rec.set("phone_number", injson.getString(
            "phoneNumber"))
        rec.update()
    }

    void deleteRecord(JSONObject injson, JSONObject outjson,
                    Connection db, ProcessServlet servlet) {
        db.execute("delete from phone where rec_id=?",
                injson.getInt("id"))
    }

    void runReport(JSONObject injson, JSONObject outjson,
                Connection db, ProcessServlet servlet) {
        Groff rpt = new Groff("PhoneList-", "Phone List",
            false)
        rpt.startTable("Lw(1i) Lw(1i) Lw(1i)")
        rpt.column("Last Name")
        rpt.column("First Name")
        rpt.column("Phone Number")
        rpt.endTitle()

        List<Record> recs = db.fetchAll(
            "select * from phone order by last_name, first_name")
        for (Record rec : recs) {
            rpt.column(rec.getString("last_name"))
            rpt.column(rec.getString("first_name"))
            rpt.column(rec.getString("phone_number"))
        }
        outjson.put("reportUrl", rpt.process())
```

```
    }

    void runExport(JSONObject injson, JSONObject outjson,
                   Connection db, ProcessServlet servlet) {
        File f = FileUtils.createReportFile("PhoneList-",
            ".csv")
        DelimitedFileWriter dfw =
            new DelimitedFileWriter(f.getAbsolutePath())

        dfw.writeField("Last Name")
        dfw.writeField("First Name")
        dfw.writeField("Phone Number")
        dfw.endRecord()

        List<Record> recs = db.fetchAll(
            "select * from phone order by last_name, first_name")
        for (Record rec : recs) {
            dfw.writeField(rec.getString("last_name"))
            dfw.writeField(rec.getString("first_name"))
            dfw.writeField(rec.getString("phone_number"))
            dfw.endRecord()
        }
        dfw.close()
        outjson.put("exportUrl", FileUtils.getHTTPPath(f))
    }
}
```

Note

This is the actual `Crud.groovy` from the Kiss example application. Notice:

- No configuration files or annotations.
- No explicit transaction management (the framework handles it).
- No authentication code (the framework checks the session automatically).
- Six methods, each a complete JSON-RPC endpoint.

6.9 File Upload Handling

File uploads use the `servlet` parameter to access uploaded files:

```
void upload(JSONObject injson, JSONObject outjson,
            Connection db, ProcessServlet servlet) {
    int numberOfFiles = servlet.getUploadFileCount()
    String fileName = servlet.getUploadFileName(0)
    BufferedInputStream is
        = servlet.getUploadBufferedInputStream(0)
    // Process the input stream (save to disk, parse, etc.)
    is.close()
}
```

See Chapter 23 for complete file upload examples.

6.10 Interfacing with External Systems

Services frequently need to call out to third-party APIs or accept inbound calls from external systems that don't speak Kiss's JSON-RPC. Both directions, plus exposing the application as an MCP server to AI assistants, are covered in Chapter 25.

6.11 Calling Other Dynamically Compiled Services

Dynamically compiled services (Groovy, Java, or Common Lisp) can freely call any precompiled code in the system — framework classes such as `Connection`, `Record`, `RestClient`, the Kiss utility classes, and anything you place in `src/main/precompiled/` — using ordinary imports and method calls. Within a single source file, methods, inner classes, and helpers can also call one another directly with no special handling.

What you *cannot* do is have one dynamically compiled service hold a direct, compile-time reference to a class defined in a *different* dynamically compiled source file. Each on-the-fly compile produces a fresh class identity, so any direct reference would become stale the moment the target file is edited and recompiled — defeating hot reload. The framework therefore requires cross-file calls to go through an indirection that always resolves to the most recently compiled version of the target.

For Groovy services, that indirection is `GroovyService.run` (with `GroovyService.getMethod` as a companion for repeated calls):

```
import org.kissweb.restServer.GroovyService

// Single call: latest version is loaded, compiled if needed,
// and the method is invoked.
Object result = GroovyService.run(
    "services",       // path relative to src/main/backend/
    "OtherService",   // class/file name (no .groovy)
    "doWork",         // method name
    null,             // instance, or null for static
    arg1, arg2)       // boxed or unboxed arguments
```

When the same method will be called repeatedly, fetch the `Method` handle once and reuse it:

```
import java.lang.reflect.Method
import org.kissweb.restServer.GroovyService

Method m = GroovyService.getMethod(
    "services", "OtherService", "doWork",
    String.class, Integer.class)

Object r1 = GroovyService.run(m, null, "alpha", 1)
Object r2 = GroovyService.run(m, null, "beta", 2)
```

Note

`GroovyService.run` verifies that the target source file is loaded and up-to-date before dispatching the call, so hot reload continues to work seamlessly across service boundaries. Direct references would not.

Tip

This restriction is *only* for crossing source-file boundaries between dynamically compiled units. Methods within the same source file call each other normally, and any dynamically compiled service can call precompiled (framework or `precompiled/`) code directly.

6.12 Reusable Scripts

The same cross-file mechanism is the standard way to share logic between services via reusable scripts. Groovy scripts in

`src/main/backend/scripts/` can be called from any service:

```
// In scripts/MyScript.groovy
public static Object myMethod(Integer a, Integer b) {
    return a + b + 100
}
```

```
// In a service method
import org.kissweb.restServer.GroovyService

Object result = GroovyService.run(false, "~/scripts",
    "MyScript", "myMethod", null, num1, num2)
```

6.13 Services Without a Database

If the application has no database configured, the `db` parameter will be `null`. Services should check for this:

```
void myMethod(JSONObject injson, JSONObject outjson,
              Connection db, ProcessServlet servlet) {
    if (db == null) {
        outjson.put("nodb", true)
        return
    }
    // ... normal database operations ...
}
```

6.14 Long-Running Requests

Most service methods return quickly. A few — a slow LLM generation, a large report, a bulk export — can run for many seconds or minutes. Kiss processes every request asynchronously, and the servlet container imposes a default timeout (roughly thirty seconds with the bundled Tomcat) on each asynchronous request. When a method exceeds it, the container aborts the request in flight: the front-end's call fails, and the user sees "Error communicating with the server."

A method that may legitimately run long should lift that limit by calling `setTimeout` on its `ProcessServlet` argument before the slow work begins:

```
void bigReport(JSONObject injson, JSONObject outjson,
               Connection db, ProcessServlet servlet) {
    // no async timeout for this request
    servlet.setTimeout(0)
    // ... work that may take minutes ...
}
```

A value of zero (or any value ≤ 0) means no timeout at all; a positive value sets the timeout, in milliseconds, for this request only. Other requests are unaffected and keep the default as a safety net, so one stuck handler cannot quietly tie up a worker forever.

Note

Lifting the servlet timeout governs only how long Kiss itself will wait for the method to finish. Work that calls out to something else — a long database query, an external HTTP request, an LLM server — has its *own* timeout that must also be generous enough. For the Ollama client specifically, see "Timeouts and Long Generations" in the LLM Integration chapter.

6.15 Service Best Practices

- Keep services focused. One service class per entity or functional area.
- Use `UserException` for validation errors the user should see.
- Use parameterized queries (`?` placeholders) for all SQL to prevent injection.
- Let the framework handle transactions. Only call `db.commit()` explicitly for long-running batch operations.
- Do not set database columns that have defaults (e.g., `CURRENT_TIMESTAMP`).
- Use Kiss utility classes (`DateTime`, `NumberUtils`, `FileUtils`) instead of standard Java/Groovy alternatives for consistent behavior.

Chapter 7

Groovy for Kiss Developers

Groovy is the primary language for Kiss backend services. This chapter covers the Groovy features most relevant to Kiss development. For a more complete Groovy primer, see Appendix B. A standalone overview of the language — what it is, why it was chosen, and how it is used in Kiss — is also distributed with the project as `manual/GroovyOverview.html`; readers new to Groovy may find it useful to skim before continuing.

7.1 Why Groovy

Groovy compiles to JVM bytecode and interoperates seamlessly with Java. Every Java library works unchanged. Groovy adds:

- Dynamic and optional typing — less boilerplate.
- Closures — anonymous functions for concise iteration and callbacks.
- String interpolation — embed expressions directly in strings.
- Safe navigation — avoid null pointer exceptions with `?.`.
- Simplified syntax — optional semicolons, optional parentheses, optional `return`.

Kiss services run at compiled speed, not interpreted speed. Groovy source files are compiled to bytecode on first access and recompiled automatically when the file changes.

7.2 Key Differences from Java

Java	Groovy
`String[] arr = new String[]{"a"};`	`def arr = ["a"]`
`map.get("key")`	`map.key` or `map["key"]`
`for (String s : list)`	`for (s in list)` or `list.each { }`
`obj != null ? obj.method() : null`	`obj?.method()`
`value != null ? value : def`	`value ?: defaultValue`
`String.format("%s is %d", n, a)`	`"${n} is ${a}"`
Semicolons required	Semicolons optional
`return` required	Last expression is implicit return
Types required	`def` for dynamic typing

7.3 Variables and Types

Groovy supports both explicit types and dynamic typing with `def`:

```
// Explicit types (same as Java)
String name = "Alice"
int count = 42
double salary = 75000.50

// Dynamic typing with def
def name = "Alice"   // inferred as String
def count = 42       // inferred as Integer
def list = [1, 2, 3] // inferred as ArrayList
// inferred as LinkedHashMap
def map = [name: "Alice", age: 30]
```

Tip

In Kiss services, use explicit types for method parameters and return values (clarity for other developers), and `def` for local variables where the type is obvious from context.

7.4 String Interpolation

Groovy distinguishes between single-quoted strings (plain) and double-quoted strings (interpolated):

```
String name = "Alice"
int age = 30

// GStrings (interpolated) - double quotes
println "Hello, ${name}! Age: ${age}"
println "Next year: ${age + 1}"

// Plain strings - single quotes
println 'No interpolation: ${name}' // prints literally

// Multi-line strings - triple quotes
def sql = """
    SELECT *
    FROM employee
    WHERE active = 'Y'
"""
```

7.5 Safe Navigation and Elvis Operator

These two operators eliminate most null-checking boilerplate:

```
// Safe navigation: returns null if employee is null
String dept = employee?.department?.name

// Elvis operator: returns right side if left is null
String dept = employee?.department?.name ?: "Unknown"

// Combined pattern in Kiss services
String email = injson.getString("email")?.toLowerCase() ?: ""
```

7.6 Closures

Closures are anonymous functions enclosed in `{ }`. They are used heavily in Groovy for iteration, filtering, and callbacks:

```
// Implicit 'it' parameter (single parameter closures)
[1, 2, 3].each { println it }

// Named parameters
[1, 2, 3].collect { n -> n * 2 } // [2, 4, 6]

// Multi-line with typed parameters
files.sort { File a, File b ->
    Long.compare(b.lastModified(), a.lastModified())
}

// Common collection operations
def active = employees.findAll { it.active == "Y" }
def names = employees.collect { it.lastName }
def senior = employees.find { it.yearsOfService > 10 }
def totalSalary = employees.sum { it.salary }
```

7.7 Groovy Idioms in Kiss Services

7.7.1 Concise Service Methods

Groovy's concise syntax makes service methods shorter than equivalent Java:

```
class EmployeeService {
    void list(JSONObject injson, JSONObject outjson,
              Connection db, ProcessServlet servlet) {
        outjson.put("employees", db.fetchAllJSON(
            "select employee_id, first_name, last_name " +
            "from employee order by last_name"))
    }

    void save(JSONObject injson, JSONObject outjson,
              Connection db, ProcessServlet servlet) {
        Record rec = db.fetchOne(
            "select * from employee where employee_id = ?",
            injson.getInt("id"))
        rec.set("first_name", injson.getString("firstName"))
        rec.set("last_name", injson.getString("lastName"))
        rec.update()
    }
```

```
}
```

7.7.2 Null-Safe Input Handling

```
// Safe extraction with defaults
String name = injson.getString("name") ?: ""
int page = injson.getInt("page") ?: 0
```

7.7.3 File Operations

```
// Reading a file
String content = new File("data.txt").text

// Listing and filtering files
File dir = new File("/path/to/files")
File[] allFiles = dir.listFiles()
List<File> sqlFiles = []
for (File file : allFiles) {
    if (file.isFile() && file.getName().endsWith(".sql"))
        sqlFiles.add(file)
}

// Sorting files by modification time (newest first)
sqlFiles.sort { File a, File b ->
    Long.compare(b.lastModified(), a.lastModified())
}
```

Warning

Use simple `for` loops instead of closure-based filters (`.findAll{}`) when filtering files. Groovy's type inference can cause `ClassCastException` in closure-based file operations.

7.8 Java Interoperability

Groovy can call any Java class directly and vice versa. All Java libraries in `libs/` are available without configuration. This is why Kiss services can seamlessly use the Java-based `Connection`, `Record`, and `QueryBuilder` classes.

```
import org.kissweb.DateUtils
import org.kissweb.NumberFormat
import java.util.UUID

// Kiss utility classes
String formatted = NumberFormat.Format(12345.67, "CD", 12, 2)
int today = DateUtils.today()     // YYYYMMDD

// Standard Java
String uuid = UUID.randomUUID().toString()
```

7.9 When to Use Groovy vs. Java

Use Groovy for
: Application services, business logic, scripts, cron tasks — anything where conciseness and rapid development matter.

Use Java for
: Performance-critical code, shared library classes in `precompiled/`, and when your team prefers static typing and IDE support.

Both languages produce the same bytecode and run at the same speed. The choice is primarily about developer preference and tooling.

7.10 Common Pitfalls

Default package
: Classes without a package declaration cannot be imported from packaged Groovy services. Always use a package declaration in `precompiled/` classes.

Closure types
: Use explicit `for` loops when Groovy's type inference causes `ClassCastException` in closure-based operations, especially with file filtering.

Optional syntax can hurt readability
: Optional semicolons and optional parentheses on method calls are convenient but can reduce readability, especially in chained

or nested expressions. Prefer parentheses on method calls when the intent would otherwise be unclear.

Parentheses ambiguity
While parentheses are optional in many cases, omitting them can cause ambiguity with property access. When in doubt, include them.

No `char` type
Groovy has no distinct character literal — `'x'` is a `String`, not a `char`. When a character is needed (for example, calling a Java API that takes a `char`), use an explicit cast:

```
char c1 = (char) 'x'
char c2 = 'x' as char
```

Floating-point literals are `BigDecimal`
A literal like `3.14` is a `BigDecimal`, not a `double`. This has caused many subtle bugs — for example, when passing a literal to an overloaded method, the `BigDecimal` overload is selected instead of the `double` overload, or arithmetic produces unexpected scale or rounding behavior. When a `double` is required, use the `d` suffix:

```
def a = 3.14    // BigDecimal
double b = 3.14d // double
```

Framework utilities
Always use Kiss framework utilities (`DateTime`, `NumberUtils`, `FileUtils`) instead of standard Java alternatives (`SimpleDateFormat`, `DecimalFormat`). The Kiss utilities are designed to work consistently with the framework.

Chapter 8

Database Connections and Transactions

Kiss provides a database abstraction layer that supports five database engines through a unified API. This chapter covers the `Connection`, `Command`, and `Cursor` classes that form the foundation of all database access.

8.1 The Connection Class

`Connection` wraps a JDBC connection with convenience methods for queries, transactions, and metadata access. In a service method, the framework provides a `Connection` as the `db` parameter — you rarely need to create one yourself.

```
void myMethod(JSONObject injson, JSONObject outjson,
              Connection db, ProcessServlet servlet) {
    // db is ready to use --- connection pooling, auto-commit
    // off, and transaction management are all handled by
    // the framework
    Record emp = db.fetchOne(
        "select * from employee where employee_id = ?", empId)
}
```

8.1.1 Creating Standalone Connections

For batch jobs, utilities, or code outside the service framework, you can create a connection directly:

```
try (Connection db = new Connection(
        ConnectionType.PostgreSQL,
        "localhost", null, "mydb", "user", "password")) {
    // use db ...
    db.commit();
} // connection auto-closes via try-with-resources
```

Available `ConnectionType` values: `PostgreSQL`, `MySQL`, `SQLite`, `MicrosoftServer`, `Oracle`.

8.2 Supported Databases

Kiss supports five database engines. The database type is specified in `application.ini` and affects SQL generation for features like `LIMIT` and pagination:

Database	ConnectionType	LIMIT Syntax
PostgreSQL	`PostgreSQL`	`LIMIT n`
MySQL	`MySQL`	`LIMIT n`
SQLite	`SQLite`	`LIMIT n`
SQL Server	`MicrosoftServer`	`TOP n`
Oracle	`Oracle`	`OFFSET ... FETCH`

8.3 Executing SQL

The `execute()` method runs INSERT, UPDATE, and DELETE statements:

```
// Update
db.execute(
    "UPDATE employee SET active = ? WHERE employee_id = ?",
    "N", empId)

// Insert
db.execute("INSERT INTO log (message, created) VALUES (?, ?)",
          msg, new Date())

// Delete
db.execute("DELETE FROM temp_data WHERE created < ?",
    cutoffDate)
```

Warning

Always use `?` parameter placeholders instead of string concatenation. This prevents SQL injection and handles type conversion automatically.

8.3.1 Existence Checks

```
if (db.exists("SELECT * FROM employee WHERE email = ?", email))
    throw new UserException("Email already in use")
```

8.3.2 Row Counting

```
long total = db.fetchCount(
    "SELECT * FROM employee WHERE active = 'Y'")
```

`fetchCount()` wraps the query in `SELECT COUNT(*) FROM (...)` — use it primarily for pagination display, as it executes the full query internally.

8.4 Fetching Data

Kiss provides several methods for retrieving data, each suited to a different use case.

8.4.1 Single Record

```
Record emp = db.fetchOne(
    "SELECT * FROM employee WHERE employee_id = ?", id)
if (emp == null)
    throw new UserException("Employee not found")
String name = emp.getString("first_name")
```

`fetchOne()` returns the first record or `null` if no rows match. The returned `Record` can be updated or deleted (see Chapter 9).

8.4.2 All Records

```
List<Record> emps = db.fetchAll(
    "SELECT * FROM employee WHERE active = 'Y'")
for (Record emp : emps) {
    String name = emp.getString("last_name")
    // process each record
}
```

Returns an empty list (not `null`) if no rows match.

8.4.3 Limited Records

```
// Fetch at most 100 records
List<Record> page = db.fetchAll(100,
    "SELECT * FROM employee ORDER BY last_name")
```

8.4.4 Paginated Records

```
// Fetch page 2 (zero-based), 25 records per page
List<Record> page = db.fetchAll(2, 25,
    "SELECT * FROM employee ORDER BY last_name")
```

8.4.5 Fetching as JSON

For the common case of returning query results directly to the frontend:

```
// All records as JSON array
outjson.put("employees", db.fetchAllJSON(
    "SELECT employee_id, first_name, last_name FROM employee"))

// Single record as JSON object
outjson.put("employee", db.fetchOneJSON(
    "SELECT * FROM employee WHERE employee_id = ?", id))
```

`fetchAllJSON()` also supports `max` and `page/max` variants.

Tip

Use `fetchAllJSON()` and `fetchOneJSON()` when the database column names match what the frontend expects. This eliminates the need to manually build JSON objects from Records.

8.5 Cursors for Large Result Sets

When a result set is too large to hold in memory, use a `Cursor` to iterate row by row:

```
try (Cursor cursor = db.query(
        "SELECT * FROM large_table ORDER BY id")) {
    while (cursor.isNext()) {
        Record r = cursor.getRecord()
        String name = r.getString("name")
        // process one row at a time
    }
}
```

Despite the name, Kiss cursors are *not* database cursors. The framework downloads the entire result set into a local cache (in memory for small sets, on disk for large ones) and then closes the database cursor. This means:

- Multiple cursors can be open simultaneously without database performance issues.
- Nested cursor operations work correctly.
- Memory usage is controlled by the caching strategy, not the result set size.

The `query()` method also supports `max` and `page/max` parameters:

```
// At most 1000 rows
Cursor cursor = db.query(1000, "SELECT * FROM audit_log")

// Page 3, 50 rows per page
Cursor cursor = db.query(3, 50, "SELECT * FROM audit_log")
```

8.6 The Command Class

`Command` provides fine-grained control over prepared statements. Use it when you need to execute the same SQL repeatedly or need a separate statement context:

```
try (Command cmd = db.newCommand()) {
    cmd.execute("INSERT INTO log (message) VALUES (?)", msg)
}
```

`Command` supports all the same methods as `Connection` — `execute()`, `fetchOne()`, `fetchAll()`, `fetchAllJSON()`, `query()`, `exists()` — but against its own prepared statement. This is useful when you need multiple active statements simultaneously.

```
try (Command selectCmd = db.newCommand()
     Command updateCmd = db.newCommand()) {
    Cursor cursor = selectCmd.query(
        "SELECT * FROM employee WHERE needs_update = 'Y'")
    while (cursor.isNext()) {
        Record r = cursor.getRecord()
        updateCmd.execute(
            "UPDATE employee SET processed = 'Y' " +
            "WHERE employee_id = ?",
            r.getInt("employee_id"))
    }
    db.commit()
}
```

`Command` also provides `newQueryBuilder()` for using the `QueryBuilder` (Chapter 10) with a specific command instance. This is essential when iterating over one query's results while executing another query on the same connection:

```
try (Command cmd1 = db.newCommand()
     Command cmd2 = db.newCommand()) {
    Cursor cursor = cmd1.newQueryBuilder()
        .select("employee.first_name", "employee.last_name")
        .where("employee.active = ?", "Y")
        .query()
    while (cursor.isNext()) {
        Record emp = cursor.getRecord()
        List<Record> tasks = cmd2.newQueryBuilder()
            .select("task.description", "task.due_date")
            .where("task.employee_id = ?",
                   emp.getString("employee_id"))
```

```
            .fetchAll()
        // process tasks...
    }
}
```

8.7 Transactions

Kiss connections always have auto-commit disabled. This means:

- Changes are not visible to other connections until `commit()` is called.
- The framework calls `commit()` automatically when a service method returns successfully.
- The framework calls `rollback()` automatically when a service method throws an exception.
- After each `commit()` or `rollback()`, a new transaction begins automatically.

```
// Usually you don't need to call commit() explicitly. The
// framework does it for you at the end of each service call.

// Exception: long-running batch operations
int count = 0
for (Record rec : records) {
    db.execute("UPDATE ...", rec.getInt("id"))
    if (++count % 100 == 0)
        db.commit() // commit every 100 rows
}
db.commit() // commit remaining rows
```

Warning

If you call `db.commit()` partway through a service method and then throw an exception, the committed changes cannot be rolled back. Use mid-method commits only for batch operations where partial progress is acceptable.

8.8 Parameterized Queries

All query and execute methods accept `?` parameter placeholders:

```
db.fetchAll(
    "SELECT * FROM employee WHERE dept = ? AND active = ?",
    deptId, "Y")
```

Parameters are bound using JDBC prepared statements. This:

- Prevents SQL injection attacks.
- Handles type conversion automatically (strings, numbers, dates).
- Allows the database to cache query plans.

Date parameters are handled automatically. You can pass `java.util.Date`, `java.util.Calendar`, `ZonedDateTime`, `LocalDateTime`, `LocalDate`, or `LocalTime` objects, and Kiss will convert them to the appropriate JDBC type.

8.9 Database-Specific SQL

When you need database-portable row limiting, pass a maximum row count to the fetch methods instead of writing `LIMIT` or `TOP` directly. The `max`-argument overloads (and `QueryBuilder.limit(int)`) generate the correct syntax for the connected database:

```
// Limit to max rows (database-portable)
List<Record> recs = db.fetchAll(100,
    "SELECT * FROM employee ORDER BY last_name")
```

Internally Kiss produces the correct syntax for the connected database:

- PostgreSQL/MySQL/SQLite: appends `LIMIT 100`
- SQL Server: inserts `TOP 100` after `SELECT`
- Oracle: appends `OFFSET 0 ROWS FETCH NEXT 100 ROWS ONLY`

8.10 Connection Metadata

`Connection` provides several metadata methods:

```
// Check if a table exists
if (db.tableExists("employee")) { ... }

// Get the width of a VARCHAR column (cached, exact)
int maxLen = db.getColumnWidth("employee", "last_name")

// Get primary key column names
List<String> pkCols = db.getPrimaryColumns("employee")

// Get single primary key column name
String pkCol = db.getPrimaryColumnName("employee")

// Get the database type
ConnectionType type = db.getDBType()
```

Column widths. `getColumnWidth(table, column)` returns the declared width of a column in characters, read from the database metadata, so a value can be checked against the schema instead of against a constant repeated in Java: `value.length() > db.getColumnWidth("person", "fname")`. Each answer is read once and then remembered for the life of the JVM, per schema (or catalog), table and column; the cache is thread-safe and shared by all connections. Names are case-insensitive, and the match is exact — a `_` in a table or column name is an ordinary character, not the single-character wildcard that `DatabaseMetaData` patterns use. A column with no limit (text, or a character type declared without a length) and a column that does not exist both answer `Connection.UNLIMITED_COLUMN_WIDTH` (`Integer.MAX_VALUE`), so the comparison above never rejects anything for them; a missing column never throws and is not remembered, so a column added later is picked up. An `SQLException` raised by the database while reading the metadata is not hidden: it propagates, because treating an outage as "unlimited" would silently switch the check off. `getColumnSize()` is the older, uncached call: it treats the names as patterns and answers −1 for a missing column.

Metadata is scoped to the current schema. Every metadata read — column definitions, column sizes, and primary keys — is

scoped to the connection's current schema (the current catalog on MySQL). In a multi-schema database, such as a schema-per-tenant layout, a table of the same name in another schema never influences the result. Anything derived from metadata follows the same rule; for example, the inline threshold of a hybrid column (Section 23.7.4) always reflects the current schema's column definition.

8.11 Connection Pooling

Kiss uses C3P0 for connection pooling. Pool parameters are configured automatically based on CPU core count:

- **Minimum pool size**: max(2, number of cores)
- **Initial pool size**: max(minimum, cores × 2)
- **Maximum pool size**: cores × 4, minimum 20
- **Idle timeout**: 30 minutes
- **Test period**: Every 5 minutes on idle connections
- **Checkout timeout**: 10 seconds

These defaults work well for most applications. For high-load production deployments, see Chapter 32 for tuning guidance.

8.12 Per-Request Connection Preparation

Every REST request that carries a database connection gets that connection handed to the service method fully ready to use. Sometimes "ready to use" involves per-request state — the canonical example is a **multi-tenant** application that stores each tenant in its own database schema and must point every request's connection at the right one (for example, the PostgreSQL `search_path`) before any service code runs.

`RequestConnectionPreparer` is the framework hook for exactly this. An application registers one implementation, typically from `KissInit.groovy`:

```
MainServlet.setRequestConnectionPreparer(
    new MyConnectionPreparer())
```

The interface has two methods:

```
public interface RequestConnectionPreparer {
    /** Called after authentication, before the web service
     *  method runs. */
    void prepare(Connection db,
        UserData userData) throws Exception;

    /** Called when a request connection is closed, before the
     *  underlying connection returns to the pool. Default:
     *  no action. */
    default void release(Connection db) {
    }
}
```

The contract:

- `prepare` runs for every REST request that carries a database connection, *after* authentication has completed and *before* the service method executes. `userData` is the authenticated user — or `null` when the method is allowed without authentication (Chapter 20) and no valid session accompanies the request.
- **Fail-closed:** if `prepare` throws, the request is aborted with an error response and the service method never runs. An authenticated session that cannot be mapped to a valid tenant therefore reaches no business data.
- `release` runs when the request's connection is closed, before the underlying pooled connection is returned — use it to clear whatever `prepare` set (reset the search path, clear thread-local state), so pooled connections never leak state from one consumer to the next.
- Connections obtained *outside* the request path — cron tasks, `MainServlet.openNewConnection()` — are **not** passed to `prepare`; such consumers must establish any needed connection state themselves.

A multi-tenant implementation, condensed:

```
public class MyConnectionPreparer
        implements RequestConnectionPreparer {
    @Override
    public void prepare(Connection db,
```

```
            UserData userData) throws Exception {
            if (userData == null) {
                // Allowed-without-authentication: public
                // schema only.
                setSearchPath(db, "public");
                return;
            }
            // set at login
            String tenantId = (String) userData.getUserData(
                "tenantId");
            if (tenantId == null)
                throw new Exception(
                    "No tenant is associated with this session");
            // every service is now tenant-neutral
            setSearchPath(db, tenantId);
        }

        @Override
        public void release(Connection db) {
            // never return a tenant-pointed
            setSearchPath(db, "public");
        }
    }
```

The payoff of this design is that *no web service ever mentions a tenant*: services read and write tables normally, and the schema selection happens once, in one place, for every request. When no preparer is registered the hook costs nothing.

Chapter 9

Database CRUD with Records

The `Record` class represents a single database row. This chapter covers creating, reading, updating, and deleting records using the Kiss pattern, which replaces raw SQL for most single-table operations.

9.1 The Record Class

A `Record` holds column name/value pairs for one row. Records come from two sources:

1. **Query results** — `fetchOne()`, `fetchAll()`, and cursor operations return Records populated with data from the database. These records know their table name and primary key, so they can be updated or deleted.

2. **New records** — `db.newRecord("table")` creates an empty Record for a specified table. You populate it with `set()` calls and then insert it with `addRecord()`.

9.2 Creating and Inserting Records

The Kiss pattern for inserting a row:

```
Record rec = db.newRecord("phone")
rec.set("first_name", "Alice")
rec.set("last_name", "Smith")
```

```
rec.set("phone_number", "555-1234")
rec.addRecord()
```

Only columns that are explicitly set are included in the INSERT statement. Columns with database defaults (auto-increment, `CURRENT_TIMESTAMP`, etc.) are left out and receive their default values.

Note

Always use the `newRecord`/`set`/`addRecord` pattern rather than raw SQL `INSERT` statements. It is cleaner, handles column types automatically, and only includes non-null columns in the INSERT.

9.2.1 Auto-Increment Keys

If the table has an auto-increment primary key and you need the generated value:

```
Record rec = db.newRecord("employee")
rec.set("first_name", "Bob")
rec.set("last_name", "Jones")
Object generatedId = rec.addRecordAutoInc()
int employeeId = (int) generatedId
```

`addRecordAutoInc()` inserts the record and returns the generated primary key value. Cast it to the appropriate type (`int`, `long`, or `short`).

9.2.2 Method Chaining

The `set()` methods return the Record, allowing method chaining:

```
db.newRecord("phone")
    .set("first_name", "Alice")
    .set("last_name", "Smith")
    .set("phone_number", "555-1234")
    .addRecord()
```

9.3 Fetching Records

```
// Single record by primary key
Record emp = db.fetchOne(
    "SELECT * FROM employee WHERE employee_id = ?", empId)

// Multiple records
List<Record> active = db.fetchAll(
    "SELECT * FROM employee WHERE active = 'Y' " +
    "ORDER BY last_name")
```

`fetchOne()` returns `null` if no rows match. `fetchAll()` returns an empty list if no rows match.

Warning

For a Record to be updatable or deletable, the SELECT must:

- Query a single table (no joins).
- Include the primary key column(s) in the result.

9.4 Reading Field Values

Records provide typed getter methods for each data type:

```
String name     = rec.getString("first_name")
int id          = rec.getInt("employee_id")
double salary   = rec.getDouble("salary")
long bigNum     = rec.getLong("big_number")
short small     = rec.getShort("small_value")
float pct       = rec.getFloat("percentage")
BigDecimal amount = rec.getBigDecimal("invoice_total")
boolean flag    = rec.getBoolean("active_flag")
char code       = rec.getChar("status_code")
byte[] data     = rec.getByteArray("binary_data")
```

All getters return `null` if the column value is NULL. Column names are case-insensitive.
`getBigDecimal()` is the right choice for `NUMERIC` and `DECIMAL` columns where exact precision matters (money, tax rates, quantities with fixed scale). Many JDBC drivers return these columns as `BigDecimal` natively, so `getInt()`, `getLong()`, `getFloat()`, and `getDouble()` also accept a `BigDecimal`-backed column and convert it — exactly for the integer variants (throwing if the value does not fit), and with possible loss of precision for `float`/`double`.

9.4.1 Date and Time Fields

Kiss provides several methods for reading date and time values:

```
// Date with time information
Date hired = rec.getDateTime("hire_date")

// Date portion only (time zeroed out)
Date dateOnly = rec.getDateOnly("birth_date")

// Date as YYYYMMDD integer
int dateInt = rec.getDateAsInt("birth_date") // e.g. 19950315

// Date/time as milliseconds since epoch
long ms = rec.getDateTimeMS("created_at")

// Time portion only (milliseconds)
long time = rec.getTime("start_time")

// ZonedDateTime (preserves timezone)
ZonedDateTime zdt = rec.getZonedDateTime("event_time")
```

Warning

`getDateTime()` returns `java.util.Date`, *not* `java.sql.Timestamp`. Never cast the result to `Timestamp`. This is a common mistake that causes `ClassCastException` at runtime.

9.4.2 Setting Date and Time Fields

```
// Date and time
rec.setDateTime("created_at", new Date())

// Date portion only (no time)
rec.setDateOnly("birth_date", new Date())

// Date from YYYYMMDD integer
rec.setDateOnly("birth_date", 19950315)

// Date/time from milliseconds since epoch
rec.setDateTime("event_time", System.currentTimeMillis())

// Time from milliseconds
rec.setTime("start_time", System.currentTimeMillis())
```

9.5 Updating Records

Fetch the record, change the fields, and call `update()`:

```
Record emp = db.fetchOne(
    "SELECT * FROM employee WHERE employee_id = ?", id)
emp.set("salary", newSalary)
emp.set("active", "Y")
emp.update()
```

`update()` is smart: it compares the current column values against the original values fetched from the database and only includes changed columns in the UPDATE statement. If nothing has changed, no SQL is executed.

Tip

Because `update()` only sends changed fields, it is safe to fetch a record, modify one column, and call `update()` without worrying about overwriting concurrent changes to other columns.

9.6 Deleting Records

```
Record emp = db.fetchOne(
    "SELECT * FROM employee WHERE employee_id = ?", id)
emp.delete()
```

Alternatively, use `Connection.execute()` for a direct delete without fetching:

```
db.execute("DELETE FROM employee WHERE employee_id = ?", id)
```

9.6.1 Delete Callbacks

You can register a connection-wide callback that fires whenever a record is deleted:

```
db.setDeleteCallback({ String table, Object pkValue ->
    logger.info("Deleted from ${table}: ${pkValue}")
})
```

`ExternalFile` uses this exact mechanism to remove a deleted row's associated file(s) automatically, opted in one table at a time — see Section 23.7.3.

9.7 Checking Column Existence

```
if (rec.columnExists("middle_name")) {
    String middle = rec.getString("middle_name")
}
```

9.8 Copying Records

```
// Copy all columns from one record to another
newRec.copy(existingRec)

// Copy only columns that exist in both records (by name
// and type)
newRec.copyCorresponding(existingRec)
```

`copyCorresponding()` is useful when copying between records from different tables that share some column names.

9.9 Converting to JSON

Records can be converted to JSON for direct frontend communication:

```
// Single record to JSON object
JSONObject json = emp.toJSON()
outjson.put("employee", json)

// Add record columns to an existing JSON object
emp.addToJSON(outjson)

// Convert a list of records to a JSON array
List<Record> recs = db.fetchAll("SELECT * FROM employee")
JSONArray arr = Record.toJSONArray(recs)
outjson.put("employees", arr)
```

For the common case, use `fetchAllJSON()` or `fetchOneJSON()` directly on the Connection — these skip the intermediate Record objects:

```
// Query directly to JSON array
outjson.put("records", db.fetchAllJSON(
    "SELECT employee_id, first_name, last_name FROM employee"))

// Query directly to JSON object
outjson.put("employee", db.fetchOneJSON(
    "SELECT * FROM employee WHERE employee_id = ?", id))
```

9.10 A Complete CRUD Service

The following example shows a complete service with list, add, edit, and delete methods, along with the corresponding frontend screen.

9.10.1 Backend Service

```
package services

class EmployeeService {

    void list(JSONObject injson, JSONObject outjson,
              Connection db, ProcessServlet servlet) {
        outjson.put("employees", db.fetchAllJSON(
            "SELECT employee_id, first_name, last_name, " +
            "active FROM employee " +
            "ORDER BY last_name, first_name"))
    }

    void get(JSONObject injson, JSONObject outjson,
             Connection db, ProcessServlet servlet) {
        Record emp = db.fetchOne(
            "SELECT * FROM employee WHERE employee_id = ?",
            injson.getInt("employeeId"))
        if (emp == null)
            throw new UserException("Employee not found")
        emp.addToJSON(outjson)
    }

    void create(JSONObject injson, JSONObject outjson,
                Connection db, ProcessServlet servlet) {
        Record rec = db.newRecord("employee")
        rec.set("first_name", injson.getString("firstName"))
```

```
        rec.set("last_name", injson.getString("lastName"))
        rec.set("active", "Y")
        rec.addRecord()
    }

    void save(JSONObject injson, JSONObject outjson,
              Connection db, ProcessServlet servlet) {
        Record rec = db.fetchOne(
            "SELECT * FROM employee WHERE employee_id = ?",
            injson.getInt("employeeId"))
        rec.set("first_name", injson.getString("firstName"))
        rec.set("last_name", injson.getString("lastName"))
        rec.set("active", injson.getString("active"))
        rec.update()
    }

    void delete(JSONObject injson, JSONObject outjson,
                Connection db, ProcessServlet servlet) {
        db.execute(
            "DELETE FROM employee WHERE employee_id = ?",
            injson.getInt("employeeId"))
    }
}
```

9.10.2 Frontend Screen (HTML)

```
<div>
  <h3>Employees</h3>
  <div id="emp-grid" style="width: 100%; height: 300px;"></div>
  <div style="margin-top: 10px;">
    <push-button id="emp-new">New</push-button>
    <push-button id="emp-edit" disabled>Edit</push-button>
    <push-button id="emp-delete" disabled>Delete</push-button>
  </div>

  <popup id="emp-popup" height="160px" width="400px">
    <popup-title>Employee</popup-title>
    <popup-body>
      <div style="display: grid;
                  grid-template-columns: auto 1fr;
                  gap: 8px; padding: 10px;">
        <span>First Name:</span>
        <text-input id="ep-first" required
                    maxlength="50"></text-input>
        <span>Last Name:</span>
        <text-input id="ep-last" required
                    maxlength="50"></text-input>
```

```
      </div>
      <div style="text-align: right; margin-top: 10px;">
        <push-button id="ep-ok">OK</push-button>
        <push-button id="ep-cancel">Cancel</push-button>
      </div>
    </popup-body>
  </popup>
</div>
```

9.10.3 Frontend Screen (JavaScript)

```
'use strict';

(async function () {
    const WS = 'services/EmployeeService';

    const grid = new AGGrid('emp-grid', [
        {headerName: 'Last Name', field: 'last_name',
            width: 40},
        {headerName: 'First Name', field: 'first_name',
            width: 40},
        {headerName: 'Active', field: 'active', width: 20}
    ], 'employee_id');
    grid.show();

    async function loadGrid() {
        grid.clear();
        $$('emp-edit').disable();
        $$('emp-delete').disable();
        const res = await Server.call(WS, 'list');
        if (res._Success)
            grid.addRecords(res.employees);
    }

    loadGrid();

    grid.setOnSelectionChanged((rows) => {
        $$('emp-edit').enable(rows);
        $$('emp-delete').enable(rows);
    });

    $$('emp-new').onclick(() => {
        $$('ep-first').clear();
        $$('ep-last').clear();
        Utils.popup_open('emp-popup', 'ep-first');

        $$('ep-ok').onclick(async () => {
```

```
            if ($$('ep-first').isError('First Name')) return;
            if ($$('ep-last').isError('Last Name')) return;
            const res = await Server.call(WS, 'create', {
                firstName: $$('ep-first').getValue(),
                lastName: $$('ep-last').getValue()
            });
            if (res._Success) {
                Utils.popup_close();
                loadGrid();
            }
        });

        $$('ep-cancel').onclick(() => Utils.popup_close());
    });

    $$('emp-delete').onclick(() => {
        Utils.yesNo('Confirm', 'Delete this employee?',
                    async () => {
            const row = grid.getSelectedRow();
            const res = await Server.call(WS, 'delete', {
                employeeId: row.employee_id
            });
            if (res._Success) loadGrid();
        });
    });
})();
```

This example demonstrates the standard Kiss full-stack pattern: a backend service with CRUD methods, a frontend screen with a grid and popup editor, and `Server.call()` connecting the two.

Chapter 10

The Query Builder

In systems with many interrelated tables, writing queries with correct joins is complex and error-prone. The `QueryBuilder` solves this by automatically determining the join path from the database schema. You specify what to select, filter, and order — the builder figures out the joins.

10.1 Creating a QueryBuilder

The recommended approach uses the `Connection` factory method:

```
List<Record> records = db.newQueryBuilder()
    .select("employee.first_name", "employee.last_name")
    .select("department.name AS dept_name")
    .where("employee.active = ?", "Y")
    .orderBy("employee.last_name")
    .fetchAll()
```

The builder automatically joins `employee` to `department` using the foreign key relationship discovered from the database schema.

Four construction options are available:

`db.newQueryBuilder()`
: Creates a builder with the connection's schema graph and stores the connection for no-argument execution methods. This is the preferred approach.

`new QueryBuilder(conn)`
: Equivalent to the above.

`cmd.newQueryBuilder()`
: Creates a builder bound to a specific `Command` instance. Use this when multiple queries must be active simultaneously on the same connection (see Section 10.15).

`new QueryBuilder(cmd)`
: Equivalent to the above.

`new QueryBuilder(schemaGraph)`
: Uses a provided `SchemaGraph`. Execution methods require an explicit `Connection` or `Command` parameter.

10.2 Table Aliases

Table names in complex schemas can be verbose. The `alias()` method lets you define short names for tables that can be used throughout the query:

```
db.newQueryBuilder()
    .alias("employee", "e")
    .alias("department", "d")
    .select("e.first_name", "e.last_name", "d.name")
    .where("e.active = ?", "Y")
    .orderBy("e.last_name")
    .fetchAll()
```

The builder resolves aliases to real table names internally for join path resolution, and emits `employee e` and `department d` in the `FROM` and `JOIN` clauses so the database recognizes the short names. Aliases can be used everywhere: `select()`, `where()`, `orderBy()`, `groupBy()`, `having()`, and even in single-argument `leftJoin()`/`rightJoin()` calls.

```
db.newQueryBuilder()
    .alias("employee", "e")
    .alias("department", "d")
    .select("e.first_name", "d.name")
    .leftJoin("d")
    .fetchAll()
```

10.3 SELECT

Columns are specified as `table.column`, with optional aliases:

```
db.newQueryBuilder()
    .select("employee.first_name", "employee.last_name")
    .select("department.name AS dept")
    .fetchAll()
```

10.3.1 DISTINCT

```
db.newQueryBuilder()
    .select("department.name")
    .distinct()
    .fetchAll()
```

10.3.2 Aggregate Functions

Helper methods are provided for common aggregates:

```
db.newQueryBuilder()
    .select("department.name")
    .selectCount("employee.employee_id", "emp_count")
    .selectSum("employee.salary", "total_salary")
    .selectAvg("employee.salary", "avg_salary")
    .selectMin("employee.salary", "min_salary")
    .selectMax("employee.salary", "max_salary")
    .groupBy("department.name")
    .fetchAll()
```

Each aggregate method accepts a `table.column` and an optional alias. Without the alias parameter, the raw aggregate expression is used.

10.4 WHERE Conditions

Conditions use parameterized `?` placeholders. Multiple `where()` calls at the same level are combined with AND:

```
db.newQueryBuilder()
    .select("employee.first_name", "employee.last_name")
    .where("employee.active = ?", "Y")
```

```
    .where("employee.salary > ?", 50000)
    .where("employee.hire_date >= ?", 20200101)
    .fetchAll()
```

Conditions without parameters are also supported:

```
.where("employee.active = 'Y'")
```

Tip

Always use `?` placeholders for user-supplied values. This prevents SQL injection and handles type conversion automatically.

10.4.1 Dynamic WHERE Clauses

Because `where()` calls are additive, you can build queries dynamically:

```
def qb = db.newQueryBuilder()
    .select("employee.first_name", "employee.last_name")

String name = injson.getString("name")
if (name != null && !name.isEmpty())
    qb.where("employee.last_name LIKE ?", name + "%")

String dept = injson.getString("department")
if (dept != null && !dept.isEmpty())
    qb.where("department.name = ?", dept)

qb.orderBy("employee.last_name")
outjson.put("results", qb.fetchAllJSON())
```

10.5 OR / AND Grouping

Use `startOr()` and `endOr()` to group conditions with OR:

```
db.newQueryBuilder()
    .select("employee.first_name")
    .startOr()
        .where("employee.department_id = ?", 10)
        .where("employee.department_id = ?", 20)
    .endOr()
```

```
    .where("employee.active = 'Y'")
    .fetchAll()
```

This generates: `WHERE (dept_id = 10 OR dept_id = 20) AND active = 'Y'`.

Groups can be nested:

```
db.newQueryBuilder()
    .select("employee.first_name")
    .startOr()
        .startAnd()
            .where("employee.department_id = ?", 10)
            .where("employee.salary > ?", 50000)
        .endAnd()
        .startAnd()
            .where("employee.department_id = ?", 20)
            .where("employee.salary > ?", 40000)
        .endAnd()
    .endOr()
    .fetchAll()
```

10.6 ORDER BY

```
db.newQueryBuilder()
    .select("employee.first_name", "employee.last_name")
    .orderBy("employee.last_name")
    .orderByDesc("employee.hire_date")
    .fetchAll()
```

10.7 GROUP BY and HAVING

```
db.newQueryBuilder()
    .select("department.name")
    .selectCount("employee.employee_id", "emp_count")
    .selectAvg("employee.salary", "avg_salary")
    .groupBy("department.name")
    .having("COUNT(employee.employee_id) > ?", 5)
    .orderByDesc("avg_salary")
    .fetchAll()
```

10.8 LIMIT

```
db.newQueryBuilder()
    .select("employee.first_name", "employee.last_name")
    .orderBy("employee.hire_date")
    .limit(10)
    .fetchAll()
```

The builder generates database-appropriate syntax (`LIMIT`, `TOP`, or `FETCH`).

10.9 Automatic Join Resolution

The key feature of `QueryBuilder` is automatic join resolution. When you reference columns from multiple tables, the builder:

1. Collects all referenced table names from SELECT, WHERE, ORDER BY, GROUP BY, and HAVING clauses.
2. Selects a root table (preferring the first table mentioned in WHERE, then SELECT).
3. Calls `SchemaGraph.findJoinPath()` to find the shortest path connecting all tables via foreign key relationships.
4. Generates INNER JOIN clauses in the correct order.

This means you never need to write JOIN clauses manually for tables connected by foreign keys.

Note

Auto-join resolution uses a BFS (breadth-first search) algorithm, which always finds the shortest path. It runs in sub-millisecond time even for schemas with hundreds of tables.

10.9.1 Foreign Keys vs. Join Types

It is important to understand that foreign keys and join types are orthogonal concepts:

- **Foreign keys** define the *path* between tables — which columns to use in the `ON` clause. For example, the foreign key `employee.department_id → department.department_id` tells the builder *how* to connect the two tables.
- **Join types** (INNER, LEFT, RIGHT) define the *behavior* — what happens when a row in one table has no matching row in the other.

The same foreign key relationship can be traversed with any join type. Using `employee.department_id → department.department_id` as the example:

INNER JOIN
: Returns only employees that have a matching department.

LEFT JOIN
: Returns all employees, with `NULL` department columns when no match exists.

RIGHT JOIN
: Returns all departments, with `NULL` employee columns when no match exists.

The automatic join resolution always uses INNER JOIN because it is the safe default — it assumes every foreign key relationship has a matching row on both sides. When you need outer join behavior, use the `leftJoin()` or `rightJoin()` methods described in the next section. The `ON` condition comes from the same foreign key columns either way.

10.10 Explicit Joins

For self-joins, ambiguous paths, or outer joins, specify the join explicitly. When you specify an explicit join to a table, it fully replaces any automatic join that would otherwise be generated for that table — there is no duplication or conflict.

10.10.1 Overriding Auto-Join Types

The simplest way to change an auto-discovered join from INNER to LEFT or RIGHT is with the single-argument form of `leftJoin()`

or `rightJoin()`. You name only the table — the foreign key path is still resolved automatically from the schema graph:

```
// Include all employees, even those without a department
db.newQueryBuilder()
    .select("employee.first_name", "department.name")
    .leftJoin("department")
    .fetchAll()
```

This produces the same `ON` condition that the auto-join would generate, but uses `LEFT JOIN` instead of `INNER JOIN`. Multiple overrides can be combined:

```
db.newQueryBuilder()
    .select("employee.first_name", "department.name",
            "building.address")
    .leftJoin("department")
    .leftJoin("building")
    .fetchAll()
```

Note

Naming just the table is always unambiguous because the auto-join resolution brings each table into the query exactly once. Even if a table has foreign keys to several other tables, only one join clause targets it. When the same table must appear multiple times in a query (e.g., joining `t2` twice with different aliases), auto-join cannot express that — use the multi-argument explicit join form described below.

10.10.2 Fully Explicit Joins

When you need full control over the join columns — for example, self-joins, composite foreign keys, or choosing a specific path when multiple foreign keys exist between two tables — use the multi-argument forms. These specify the exact columns and completely replace any auto-join for the target table.

```
// Explicit LEFT JOIN specifying columns
db.newQueryBuilder()
    .select("employee.first_name", "department.name")
    .leftJoin("employee", "department_id",
              "department", "department_id")
```

```
    .fetchAll()
```

`leftJoin()` and `rightJoin()` multi-argument forms have the same signatures as `join()`.

10.10.3 Self-Joins

```
db.newQueryBuilder()
    .select("employee.first_name")
    .select("mgr.first_name AS manager_name")
    .join("employee", "manager_id",
         "employee", "employee_id", "mgr")
    .fetchAll()
```

The fifth parameter is an alias for the joined table, required for self-joins.

10.10.4 Composite Foreign Keys

```
db.newQueryBuilder()
    .select("order_line.quantity")
    .join("order_line",
         new String[]{"order_id", "product_id"},
         "order_product",
         new String[]{"order_id", "product_id"},
         null)
    .fetchAll()
```

10.11 Subqueries

10.11.1 WHERE IN / NOT IN

```
// Using a QueryBuilder subquery
def sub = db.newQueryBuilder()
    .select("assignment.employee_id")
    .where("assignment.project_id = ?", projId)

db.newQueryBuilder()
    .select("employee.first_name", "employee.last_name")
    .whereIn("employee.employee_id", sub)
    .fetchAll()
```

```
// Using raw SQL
db.newQueryBuilder()
    .select("employee.first_name")
    .whereNotIn("employee.department_id",
       "SELECT department_id FROM closed_departments")
    .fetchAll()
```

10.11.2 WHERE EXISTS / NOT EXISTS

```
def sub = db.newQueryBuilder()
    .select("assignment.assignment_id")
    .where("assignment.employee_id = employee.employee_id")
    .where("assignment.project_id = ?", projId)

db.newQueryBuilder()
    .select("employee.first_name")
    .whereExists(sub)
    .fetchAll()
```

10.12 Common Table Expressions (CTEs)

CTEs are defined with `with()` and can use raw SQL or another `QueryBuilder`:

```
// CTE with raw SQL
db.newQueryBuilder()
    .with("active_emps",
       "SELECT * FROM employee WHERE active = 'Y'")
    .select("active_emps.first_name")
    .select("department.name")
    .fetchAll()

// CTE with QueryBuilder
def cte = db.newQueryBuilder()
    .select("employee.employee_id", "employee.first_name")
    .where("employee.active = ?", "Y")

db.newQueryBuilder()
    .with("active_emps", cte)
    .select("active_emps.first_name")
    .fetchAll()
```

10.13 UNION and UNION ALL

```
def current = db.newQueryBuilder()
    .select("employee.first_name", "employee.last_name")
    .where("employee.active = 'Y'")

def former = db.newQueryBuilder()
    .select("employee.first_name", "employee.last_name")
    .where("employee.active = 'N'")

current.unionAll(former)
    .orderBy("employee.last_name")
    .fetchAll()
```

10.14 Execution Methods

All execution methods come in three forms:

- **No argument** — uses the stored connection or command from the constructor.
- **With a `Connection`** — uses an explicit connection (creates a temporary command internally).
- **With a `Command`** — uses an explicit command, for concurrent query scenarios.

When a `QueryBuilder` is created from a `Command`, the no-argument methods execute through that command. When created from a `Connection`, they execute through the connection.

Method	Description
`fetchAll()`	Execute and return `List<Record>`.
`fetchAll(conn)`	Execute via explicit connection.
`fetchAll(cmd)`	Execute via explicit command.
`fetchOne()`	Execute and return a single `Record` or `null`.
`fetchOne(conn)`	Execute via explicit connection.
`fetchOne(cmd)`	Execute via explicit command.
`fetchAllJSON()`	Execute and return a `JSONArray`.
`fetchAllJSON(conn)`	Execute via explicit connection.

Method	Description
`fetchAllJSON(cmd)`	Execute via explicit command.
`query()`	Execute and return a `Cursor` for row-by-row iteration.
`query(conn)`	Execute via explicit connection.
`query(cmd)`	Execute via explicit command.
`build()`	Generate the SQL string without executing. Useful for debugging or logging.
`getParameters()`	Return the list of parameter values (call after `build()`).
`toString()`	Return a formatted, human-readable SQL string with newlines between clauses and parameter values appended. Calls `build()` automatically if needed.

```
// Inspect generated SQL
def qb = db.newQueryBuilder()
    .select("employee.first_name", "department.name")
    .where("employee.active = ?", "Y")
String sql = qb.build()
println sql
println qb.getParameters()
```

10.14.1 Formatted Debug Output

The `toString()` method returns a formatted version of the built SQL that is easier to read during debugging. Major clauses (`SELECT`, `FROM`, `WHERE`, `ORDER BY`, etc.) appear on separate lines, joins are indented, and parameter values are appended as a comment at the end.

```
def qb = db.newQueryBuilder()
    .select("employee.first_name", "employee.last_name")
    .select("department.name AS dept_name")
    .select("building.address")
    .where("project.project_id = ?", projId)
    .where("employee.active = 'Y'")
    .orderBy("employee.last_name")
println qb.toString()
```

This produces output like:

```
SELECT employee.first_name, employee.last_name,
       department.name AS dept_name, building.address
FROM project
    JOIN project_assignment ON project_assignment.project_id
        = project.project_id
    JOIN employee ON project_assignment.employee_id
        = employee.employee_id
    JOIN department ON employee.department_id
        = department.department_id
    JOIN building ON department.building_id
        = building.building_id
WHERE project.project_id = ?
  AND employee.active = 'Y'
ORDER BY employee.last_name
-- Parameters: [42]
```

Because `toString()` calls `build()` automatically if it has not already been called, it can be used directly in log statements or print calls without any prior setup.

10.15 Using QueryBuilder with Commands

The `Connection` convenience methods (`db.fetchAll()`, `db.fetchOne()`, etc.) each create a temporary `Command` internally and close it immediately. This works well for isolated queries, but when you need multiple queries active simultaneously — for example, iterating over one result set while running another query for each row — you must use separate `Command` instances to avoid interference.

`Command.newQueryBuilder()` creates a `QueryBuilder` bound to that specific command, giving you the full fluent API without query interference:

```
try (Command cmd1 = db.newCommand()
     Command cmd2 = db.newCommand()) {
    Cursor cursor = cmd1.newQueryBuilder()
        .select("employee.first_name", "employee.last_name")
        .select("department.name AS dept")
        .where("employee.active = ?", "Y")
        .query()
    while (cursor.isNext()) {
        Record emp = cursor.getRecord()
        List<Record> tasks = cmd2.newQueryBuilder()
```

```
            .select("task.description", "task.due_date")
            .where("task.employee_id = ?",
                   emp.getString("employee_id"))
            .fetchAll()
        // process tasks for this employee
    }
}
```

Without this integration, the developer would be forced to write raw SQL in the very scenarios where queries are most complex (multiple related tables queried simultaneously).

Tip

Use `db.newQueryBuilder()` for the common case of isolated, one-shot queries. Use `cmd.newQueryBuilder()` only when you need multiple simultaneously active queries on the same connection.

10.16 Complete Example: Flexible Employee Search

```
void search(JSONObject injson, JSONObject outjson,
            Connection db, ProcessServlet servlet) {
    def qb = db.newQueryBuilder()
        .select("employee.employee_id")
        .select("employee.first_name", "employee.last_name")
        .select("department.name AS department")

    // Dynamic filters
    String name = injson.getString("lastName")
    if (name)
        qb.where("employee.last_name LIKE ?", name + "%")

    String dept = injson.getString("department")
    if (dept)
        qb.where("department.name = ?", dept)

    Double minSalary = injson.getDouble("minSalary")
    if (minSalary)
        qb.where("employee.salary >= ?", minSalary)

    String active = injson.getString("activeOnly")
    if (active == "Y")
```

```
            qb.where("employee.active = 'Y'")

        qb.orderBy("employee.last_name")
          .orderBy("employee.first_name")
          .limit(100)

        outjson.put("results", qb.fetchAllJSON())
    }
```

This single method handles any combination of search filters. The QueryBuilder automatically joins `employee` to `department` only when the department filter or column is used.

10.17 Client-Driven Sorting with SortBuilder

A screen that lets the user click grid headers to sort needs an `ORDER BY` that depends on client input. The sort must happen in SQL: when a result is capped or paged, sorting in the browser reorders only the rows on screen (see Section 16.9). But a column name sent by the client must never be pasted into SQL.

`org.kissweb.database.SortBuilder` solves this. The application registers, once, every sort key it accepts together with the fixed SQL that produces the value. For each request the client sends only keys; `SortBuilder` looks them up and emits the registered SQL. Unknown keys are ignored, never executed.

Security

Client text never reaches the SQL. A key is used only as a lookup into the registered table; the `ORDER BY` text comes entirely from your program constants. An unknown key is dropped and reported through `rejectedKeys()`. Expressions passed to `raw()`, join clauses and tiebreak items are emitted verbatim, so they must be fixed program constants, never client text.

10.17.1 Registering Keys

Build the `SortBuilder` once, typically in a `static` initializer, then only read it. `build()` is thread-safe once configuration is complete.

```
import org.kissweb.database.SortBuilder
```

```
import static org.kissweb.database.SortBuilder.*

static final String LAST_PAY_JOIN =
    "left join lateral (select max(paid_on) " +
    "as paid_on from payment pay " +
    "where pay.order_id = o.order_id) lp on true"

static final SortBuilder SORT = new SortBuilder()
    .key("customer", text("c.name"))
    .key("contact", text("c.last_name"),
                    text("c.first_name"))
    .key("placed", zeroBlank("o.placed_date"))
    .key("status", raw("st.seqno"))
    .key("paid",   raw("lp.paid_on"))
    .join(LAST_PAY_JOIN, "paid")
```

Each `key()` call names a key and gives one or more expressions. A multi-part key such as `contact` sorts by last name, then first name; every expression receives the requested direction. The three expression helpers are:

Helper	Meaning
`text(col)`	Text, compared case-insensitively. Empty and whitespace-only values count as blank.
`zeroBlank(col)`	Number or integer date where 0 means blank (rendered as `nullif(col, 0)`).
`raw(sql)`	The expression verbatim. It must yield `NULL` when the value is blank.

For a raw expression that must differ by database, pass a lambda instead. An `Expr` receives the `ConnectionType`, and `SortBuilder.textSql(type, col)` gives the dialect's text rule for use inside it:

```
.key("fullName", { type ->
    "coalesce(" + SortBuilder.textSql(type, "c.name")
        + ", '')" } as SortBuilder.Expr)
```

10.17.2 Joins

Some keys sort on a value that needs an extra join (above, the latest payment date). `join(joinSql, keys...)` declares a join clause

and the keys that need it. The clause is emitted, once, only when at least one of those keys is actually used, so unsorted requests do not pay for the join. The keys must already be registered.

10.17.3 Building the ORDER BY

```
SortBuilder.Result r = SORT.build(
    db.getDBType(),
    [SortBuilder.Sort.of(sort1, asc1),
     SortBuilder.Sort.of(sort2, asc2)],
    SortBuilder.Sort.of("customer", true),
    "o.order_id asc")
```

The arguments are:

- the database type (so the SQL is rendered for the right dialect);
- the requested sorts in priority order (may be `null`);
- the *default* sort, used only when no usable key was sent (may be `null`; must be a registered key);
- zero or more *tiebreak* items: complete `ORDER BY` items, including their own direction, appended last so the order is deterministic.

The `Result` provides:

Method	Description
`orderBy()`	The complete clause, such as `order by a asc nulls last, b asc`. No leading or trailing space. Empty if there is nothing to sort on.
`joins()`	The needed join clauses, each once, in first-use order, with a trailing space. Empty string if none.
`joinList()`	The same joins as a `List<String>`.
`rejectedKeys()`	Non-blank keys that were not registered: trimmed, each once, in request order.

Behavior to rely on:

- **Blank keys** (null, empty, whitespace) are silently ignored.
- **Unknown keys** are never an exception. They are ignored and listed in `rejectedKeys()`. Because that is client text, strip line breaks before logging it.
- **The default** applies only when no requested key was usable.
- **Programmer errors** throw `IllegalArgumentException`: a default that is not registered, a `join()` naming an unregistered key, or a `key()` with a blank name or no expressions.
- **Re-registering** a key replaces it.

10.17.4 Blanks Last and Database Dialects

Every expression gets the requested direction, and blank values (`NULL`) sort *last in both directions*. Databases differ in how they express this, so the SQL is rendered for the connection type at `build()` time:

The `text()` blank rule, with `x` the column:

Database	`text()` SQL
PostgreSQL	`lower(nullif(btrim(x), ''))`
SQLite	`lower(nullif(trim(x), ''))`
Oracle	`lower(trim(x))`
MySQL	`lower(nullif(trim(x), ''))`
SQL Server	`lower(nullif(ltrim(rtrim(x)), ''))`

How each database orders blanks last, for an ascending key:

Databases	Emitted ORDER BY
PostgreSQL, Oracle, SQLite	`x asc nulls last`
MySQL, SQL Server	`case when x is null then 1 else 0 end, x asc`

Oracle already treats an empty string as `NULL`, so it needs no `nullif`. Raw expressions, joins and tiebreak items are not translated; they are emitted as written.

10.17.5 The Service Method

This is the back end of the grid example in Section 16.9. The client sends up to two keys and directions; the method builds the query around the `SortBuilder` result:

```
void search(JSONObject injson, JSONObject outjson,
            Connection db, ProcessServlet servlet) {
    def r = SORT.build(db.getDBType(),
        [SortBuilder.Sort.of(
            injson.getString("sort1"),
            injson.getBoolean("asc1", true)),
         SortBuilder.Sort.of(
            injson.getString("sort2"),
            injson.getBoolean("asc2", true))],
        SortBuilder.Sort.of("customer", true),
        "o.order_id asc")

    for (String bad : r.rejectedKeys())
        logger.warn("Rejected sort key: " +
            bad.replaceAll("[\\r\\n]", " "))

    String sql = "select o.order_id, c.name as customer, " +
        "o.placed_date as placed, st.name as status " +
        "from orders o " +
        "join customer c on c.customer_id = o.customer_id " +
        "join status st on st.status_id = o.status_id " +
        r.joins() + r.orderBy()
    outjson.put("orders", db.fetchAllJSON(200, sql))
}
```

Because the `ORDER BY` is part of the SQL, the 200-row cap applies to the correctly sorted result. Clicking *Placed* then *Status* on the grid sends `sort1=status`, `sort2=placed`, and the query orders by status first, then date, then `order_id`.

Tip

Keep the client's grid column `field` names identical to the `SortBuilder` keys. The key the grid reports is then the key you register, with no translation table to maintain.

Chapter 11

The Schema Graph

The `SchemaGraph` models a database schema as a graph where tables are nodes and foreign key relationships are edges. It powers the `QueryBuilder`'s automatic join resolution (Chapter 10). Most applications use it indirectly through `db.newQueryBuilder()`, but understanding it helps when debugging join paths or working with schemas that lack declared foreign keys.

11.1 Graph Model

A `SchemaGraph` is an undirected graph:

- **Nodes** are database table names.
- **Edges** are foreign key relationships, each connecting two tables via their key columns.
- Edges are traversable in both directions: knowing that `employee.department_id` references `department.department_id` lets the builder join from either table to the other.
- Both single-column and composite (multi-column) foreign keys are supported.

11.2 Building a Schema Graph

11.2.1 From Database Metadata

The simplest approach reads the schema directly from the live database connection using JDBC `DatabaseMetaData`:

```
SchemaGraph graph = SchemaGraph.fromDatabase(conn);
```

This discovers all tables and their foreign key relationships. It works well for databases with properly declared foreign keys.

11.2.2 Cached per Schema (Thread-Safe)

For applications with multiple schemas or concurrent requests, use the cached variant:

```
SchemaGraph graph = SchemaGraph.fromDatabase(conn, "mySchema");
```

This uses a thread-safe cache (double-checked locking with `ConcurrentHashMap`) so the schema graph is built only once per schema name, regardless of how many threads request it simultaneously.

11.2.3 Programmatic Declaration

For databases without declared foreign keys (common in legacy systems), build the graph manually:

```
SchemaGraph graph = new SchemaGraph();
graph.addForeignKey("employee", "department_id",
                    "department", "department_id");
graph.addForeignKey("assignment", "employee_id",
                    "employee", "employee_id");
graph.addForeignKey("assignment", "project_id",
                    "project", "project_id");
```

11.2.4 Composite Foreign Keys

```
graph.addForeignKey("order_line",
    new String[]{"order_id", "product_id"},
    "order_product",
    new String[]{"order_id", "product_id"});
```

11.2.5 Hybrid Approach

Start from database metadata and add undeclared relationships:

```
SchemaGraph graph = SchemaGraph.fromDatabase(conn);
// Add a relationship for a legacy table without declared FKs
graph.addForeignKey("audit_log", "user_id",
                    "employee", "employee_id");
```

Tip

The hybrid approach is the best option for databases that have some declared foreign keys but also contain legacy tables without formal constraints.

11.3 File Caching

Building a schema graph from database metadata takes a few hundred milliseconds for large schemas. For faster startup, save the graph to a file and load it later:

```
// Save (once, during setup or schema migration)
graph.saveToFile("schema-cache.txt");

// Load (sub-millisecond)
SchemaGraph graph = SchemaGraph.loadFromFile(
    "schema-cache.txt");
```

The file format is a simple text format with `TABLE` and `FK` lines:

```
TABLE employee
TABLE department
FK employee department_id department department_id
FK assignment employee_id employee employee_id
```

11.4 In-Memory Cache Management

The per-schema cache can be cleared when the database schema changes:

```
// Clear all cached schema graphs
SchemaGraph.clearSchemaCache();
```

```
// Clear a specific schema
SchemaGraph.clearSchemaCache("mySchema");
```

11.5 Querying the Graph

```
// Check if a table exists in the graph
boolean exists = graph.hasTable("employee");

// Get all table names
Set<String> tables = graph.getTables();

// Get all edges (FK relationships) for a table
List<SchemaGraph.Edge> edges = graph.getEdges("employee");
for (SchemaGraph.Edge e : edges) {
    System.out.println(e.getFromTable() + "." +
        e.getFromColumn() + " -> " +
        e.getToTable() + "." + e.getToColumn());
}
```

11.6 Finding Join Paths

The core algorithm finds the shortest join path connecting a set of tables:

```
Set<String> tables = Set.of("employee", "department",
    "building");
List<SchemaGraph.Edge> path =
    graph.findJoinPath(tables, "employee");
```

The second parameter is the root table (the FROM table in the generated SQL). The method returns an ordered list of edges representing the required JOINs.

The algorithm uses BFS (breadth-first search), which:

- Always finds the shortest path (fewest joins).
- Runs in sub-millisecond time even for schemas with 300+ tables.
- Uses a greedy Steiner tree approximation for connecting multiple tables.

- Throws `SQLException` if no path exists between some tables.

11.7 The Edge Inner Class

`SchemaGraph.Edge` represents a foreign key relationship. Key methods:

Method	Description
`getFromTable()`	Child table containing the FK column(s).
`getToTable()`	Parent/referenced table.
`getFromColumn()`	FK column name (first column if composite).
`getToColumn()`	Referenced column name (first if composite).
`getFromColumns()`	All FK column names (unmodifiable list).
`getToColumns()`	All referenced column names (unmodifiable list).
`isComposite()`	`true` if multi-column foreign key.
`buildOnCondition(from, to)`	Builds the ON clause string for a JOIN.

11.8 Integration with Connection

A `Connection` lazily builds and caches its schema graph:

```
// Get or build the schema graph for this connection
SchemaGraph graph = db.getSchemaGraph();

// Set the default schema
db.setSchema("myschema");
```

When you call `db.newQueryBuilder()`, the builder automatically uses the connection's schema graph. Most applications never interact with `SchemaGraph` directly.

11.9 Integration with Command

The `Command` class also provides `newQueryBuilder()`, which creates a `QueryBuilder` bound to that specific command. The schema graph is obtained from the command's parent connection.

This is essential when multiple queries must be active simultaneously on the same connection. Each `Command` wraps its own `PreparedStatement`, avoiding interference between concurrent result sets. See Section 10.15 for usage examples.

11.10 Example: Inspecting a Schema

```
void inspectSchema(JSONObject injson, JSONObject outjson,
                   Connection db, ProcessServlet servlet) {
    SchemaGraph graph = db.getSchemaGraph()

    // List all tables
    Set<String> tables = graph.getTables()
    outjson.put("tableCount", tables.size())

    // For each table, list its relationships
    JSONArray tableInfo = new JSONArray()
    for (String table : tables) {
        JSONObject info = new JSONObject()
        info.put("name", table)
        List<SchemaGraph.Edge> edges = graph.getEdges(table)
        JSONArray rels = new JSONArray()
        for (SchemaGraph.Edge e : edges) {
            rels.put(e.getFromTable() + "." +
                e.getFromColumn() + " -> " +
                e.getToTable() + "." + e.getToColumn())
        }
        info.put("relationships", rels)
        tableInfo.put(info)
    }
    outjson.put("tables", tableInfo)
}
```

Part III

Frontend Development

Chapter 12

Frontend Overview

The Kiss frontend is a single-page application built with plain HTML, CSS, and JavaScript — no build step, no transpiler, and no framework dependencies beyond Kiss's own component library. This chapter covers the architecture, loading mechanism, and conventions that every Kiss frontend screen follows.

12.1 Architecture

The frontend is served as static files. There is no server-side rendering, no template engine, and no build pipeline. You write HTML, CSS, and JavaScript directly, and the browser executes them as-is.

Navigation is handled by dynamically loading screen HTML and JavaScript into the page via `Utils.loadPage()`. There is no client-side routing library; the framework manages screen transitions directly.

The frontend depends on:

- **DOMUtils** — Kiss's native DOM API wrapper, used for any direct DOM manipulation that falls outside the component layer. jQuery is *not* used.
- **AG-Grid Community** — for data grids (Chapter 16).
- **CKEditor** — for rich text editing.
- **Kiss component library** — custom HTML tags for forms and dialogs.

12.2 Script Loading Order

The `index.html` file is the entry point, but it does very little itself: it contains a small inline bootstrap kernel whose only job is to keep `index.html` fresh and then hand off to `kiss/bootstrap.js`, which performs the actual ordered loading. `bootstrap.js` defines the loader functions used throughout the framework:

`getScript(url)`
: Loads a single script file. Returns a Promise.

`getScripts(urls)`
: Loads multiple scripts in parallel using `Promise.all()`.

`addStylesheet(url)`
: Appends a stylesheet `<link>` to the page head.

Every one of these helpers routes its URL through `window.cacheBust()` before fetching it, so every downloaded resource — scripts and stylesheets alike — is version-busted. The mechanics of that busting (and of the kernel that precedes `bootstrap.js`) are covered in Chapter 14; this section is concerned only with *what loads, in what order, and why.*
`bootstrap.js`'s `loadUtils()` function runs the following sequence:

1. **Stylesheets** are added first and do not block anything that follows: `normalize.css`, `normalize2.css`, `kiss/starter-theme.css`, `kiss/Utils.css`, and the AG-Grid theme (`kiss/ag-grid-custom.css`).

2. **Third-party libraries** load next, in parallel, and *are* awaited: the AG-Grid data-grid engine and CKEditor. They must finish before the Kiss wrapper scripts that reference their globals (`AGGrid.js`, `Editor.js`) load later in the sequence.

3. **`SystemInfo.js`** loads next, by itself. It carries the per-deployment configuration — including `SystemInfo.stateStore`, needed by `AppState.init()` in the next step. Once it has loaded, `bootstrap.js` mirrors the cache-control values from the `index.html` `<meta>` tags onto `SystemInfo.softwareVersion` / `SystemInfo.controlCache`, so existing references to those properties keep working.

4. **`DOMUtils.js`** loads next, by itself. Every other Kiss script that touches the DOM depends on it, and `Utils.js` in particular checks for the `DOMUtils` object at load time.

5. **`AppState.js`** loads next, by itself, and `AppState.init()` runs immediately afterward — before `Server.js` loads, since `Server`'s `uuid` accessor reads through `AppState`. Chapter 13 covers what `AppState` does.

6. **The remaining framework scripts** load in parallel: `Utils.js`, `DateUtils.js`, `DateTimeUtils.js`, `TimeUtils.js`, `NumberUtils.js`, `Server.js`, `Router.js`, `AGGrid.js`, `Editor.js`, `MutableString.js`.

7. **`routes.js`**, the application-owned route table, loads next, by itself — it must be in place before `index.js` calls `Router.start()`. See Section 13.2.4.

8. **`index.js`** loads last, and is *not* awaited — nothing later in `loadUtils()` depends on it finishing.

```
async function loadUtils() {
    // Stylesheets
    addStylesheet("normalize.css");
    addStylesheet("normalize2.css");
    addStylesheet("kiss/starter-theme.css");
    addStylesheet("kiss/Utils.css");
    addStylesheet("kiss/ag-grid-custom.css");

    // Third-party libraries -- must load before the kiss
    // AGGrid/Editor wrappers that use them
    await getScripts([
        "lib/ag-grid-community.min.noStyle.js",
        "lib/ckeditor.js"
    ]);

    // Per-deployment configuration (needed by AppState.init)
    await getScript("SystemInfo.js");
    SystemInfo.softwareVersion = window.KissVersion;
    SystemInfo.controlCache = window.KissCacheOn;

    // DOMUtils first (must complete before Utils.js); AppState
    // before Server.js uses it
    await getScript("kiss/DOMUtils.js");
    await getScript("kiss/AppState.js");
    AppState.init();
```

```
    await getScripts([
        "kiss/Utils.js",
        "kiss/DateUtils.js",
        "kiss/DateTimeUtils.js",
        "kiss/TimeUtils.js",
        "kiss/NumberUtils.js",
        "kiss/Server.js",
        "kiss/Router.js",
        "kiss/AGGrid.js",
        "kiss/Editor.js",
        "kiss/MutableString.js"
    ]);

    // routes.js (application-owned) registers routes and must
    // load before index.js calls Router.start()
    await getScript("routes.js");
    getScript("index.js");
}
```

Warning

If `DOMUtils.js` has not finished loading before `Utils.js` runs, you will see: "DOMUtils object not found — DOMUtils.js should be loaded before Utils.js." This is a race condition caused by loading them in parallel. The sequence above avoids this, and the other ordering dependencies (third-party libraries before `AGGrid.js`/`Editor.js`, `SystemInfo.js` before `AppState.init()`, `AppState.js` before `Server.js`, `routes.js` before `index.js`), by `await`-ing each dependency before the scripts that need it are loaded.

12.3 Component Registration

Before the login page can load, all UI components must be registered. This happens in `index.js`:

```
(function () {
    Utils.useComponent('Popup');
    Utils.useComponent('CheckBox');
    Utils.useComponent('DateInput');
    Utils.useComponent('DropDown');
    Utils.useComponent('ListBox');
    Utils.useComponent('NumericInput');
```

```
    Utils.useComponent('PushButton');
    Utils.useComponent('RadioButton');
    Utils.useComponent('TextboxInput');
    Utils.useComponent('TextInput');
    Utils.useComponent('TextLabel');
    Utils.useComponent('TimeInput');
    Utils.useComponent('FileUpload');
    Utils.useComponent('NativeDateInput');
    Utils.useComponent('Picture');
})();
```

Each `Utils.useComponent()` call loads a component's JavaScript from `kiss/component/<name>/`. After registration, the corresponding custom HTML tags (e.g., `<text-input>`, `<push-button>`) become functional.

12.4 The $$() Function

The `$$()` function is the primary way to access Kiss components. Pass it the HTML `id` of a component and it returns the component's Kiss object with methods like `getValue()`, `setValue()`, `clear()`, `enable()`, `disable()`, and event handlers:

```
// Set a value
$$('employee-name').setValue("Alice");

// Get a value
let name = $$('employee-name').getValue();

// Clear a component
$$('employee-name').clear();

// Enable/disable
$$('save-btn').enable();
$$('save-btn').disable();

// Validate (shows error message if invalid, returns true
// on error)
if ($$('employee-name').isError('Employee Name'))
    return;

// Set focus
$$('employee-name').focus();

// Event handling
$$('save-btn').onclick(() => saveRecord());
```

```
$$('department').onChange(() => filterByDept());
```

Warning

Always use `$$()` and Kiss component methods for events — not native DOM methods like `addEventListener()`. The Kiss methods handle event wiring consistently across all components.

12.5 Screen Loading

Screens are pairs of `.html` and `.js` files in the `screens/` directory. Each screen lives in its own subdirectory:

```
src/main/frontend/screens/
  CRUD/
    CRUD.html
    CRUD.js
  Users/
    Users.html
    Users.js
  Framework/
    Framework.html
    Framework.js
```

Load a screen with `Utils.loadPage()`:

```
Utils.loadPage("screens/CRUD/CRUD", "app-screen-area",
    "first-name");
```

The three parameters are:

1. **Path** — the path to the screen files (without extension). Kiss appends `.html` and `.js` automatically.
2. **Target** — the ID of the HTML element where the screen's HTML will be injected. If omitted, the body is used.
3. **Focus** — the ID of a component that should receive focus after loading.

When a screen loads:

1. The HTML file is fetched and injected into the target element.

2. `Utils.rescan()` processes the injected HTML to activate any custom components.
3. The JavaScript file is loaded and executed.
4. Focus is set to the specified component.

12.6 Screen JavaScript Pattern

Every screen JavaScript file follows the same pattern — an immediately-invoked async function expression (IIFE) that provides variable scope isolation:

```
'use strict';

(async function () {
    const WS = 'services/MyService'; // backend service path

    // Initialize grids, load data, set up event handlers
    const grid = new AGGrid('my-grid', columnDefs, 'id');
    grid.show();

    // Load initial data
    const res = await Server.call(WS, 'list');
    if (res._Success)
        grid.addRecords(res.records);

    // Button handlers
    $$('save-btn').onclick(async () => {
        // validate, call backend, refresh
    });
})();
```

Tip

The IIFE pattern ensures that each screen's variables are scoped to that screen. When a new screen loads, the previous screen's variables are garbage-collected.

12.7 The Login Flow

The application starts at `login.html`, which presents a login form:

```
<form onsubmit="return false;" autocomplete="on">
    <text-input id="username" required
                autocomplete="username"></text-input>
    <text-input id="password" required password
                autocomplete="current-password"></text-input>
    <push-button id="login">Login</push-button>
</form>
```

The `<form>` tag with `onsubmit="return false;"` prevents actual form submission while enabling browser password managers via `autocomplete` attributes.

The `login.js` file:

1. Validates the username and password fields.
2. Calls `Server.call(", 'Login', data)` to authenticate.
3. On success, stores the session UUID with `Server.setUUID(res.uuid)`.
4. Enables back-button protection with `DOMUtils.preventNavigation()`.
5. Loads the main application shell via `Utils.loadPage()`.

```
const data = {
    username: $$('username').getValue().toLowerCase(),
    password: $$('password').getValue()
};
const res = await Server.call('', 'Login', data);
if (res._Success) {
    Server.setUUID(res.uuid);
    DOMUtils.preventNavigation(
        function () { return true; },
        function () {
            Utils.yesNo('Confirm',
                'Are you sure you want to logout?',
                function () { Server.logout(); });
        }
    );
    Utils.loadPage('screens/Framework/Framework');
}
```

12.8 The Framework Screen

After login, the `Framework` screen provides the application shell: a navigation menu and a content area where individual screens are loaded.

```
// Framework.js - navigation button handlers
$$('crud-btn').onclick(function () {
    Utils.cleanup();
    Utils.loadPage('screens/CRUD/CRUD', 'app-screen-area');
});

$$('users-btn').onclick(function () {
    Utils.cleanup();
    Utils.loadPage('screens/Users/Users', 'app-screen-area');
});

$$('logout').onclick(function () {
    Server.logout();
});
```

`Utils.cleanup()` removes event handlers and state from the current screen before loading a new one, preventing memory leaks.

12.9 Backend URL Configuration

The `index.js` initialization code auto-detects the backend URL based on the deployment mode:

```
Utils.afterComponentsLoaded(function () {
    if (SystemInfo.backendUrl) {
        // explicit config
        Server.setURL(SystemInfo.backendUrl);
    } else if (SystemInfo.sameOriginBackend) {
        // served by the backend
        Server.setURL(Utils.getAppUrl());
    } else if (window.location.protocol === "file:") {
        // Electron desktop
        Server.setURL('http://localhost:8001');
    } else if (window.location.port >= 8000) {
        // development: backend is one port above
        Server.setURL('http://' +
            window.location.hostname + ':' +
            (Number(window.location.port) + 1));
    } else {
        // production
```

```
        Server.setURL(Utils.getAppUrl());
    }
});
```

The resolution order is:

1. **Explicit configuration.** `SystemInfo.backendUrl`, when set, always wins. Use it for split deployments, or whenever the conventions below do not apply.

2. **Served by the backend itself.** `SystemInfo.sameOriginBackend` is `false` in the source tree, but the build system stamps it `true` into every copy of the front end that Tomcat serves — the production WAR (during `./bld war`) and the development `tomcat/webapps/ROOT` (during `develop/start-backend`). A stamped page calls the backend at its own origin, which makes browsing the Tomcat origin directly, and any production port, work with no configuration. Never set this by hand.

3. **Electron.** Pages loaded from `file://` default to `http://localhost:8001`, the backend of the default port block; an application with a different `portBase` sets `SystemInfo.backendUrl` instead.

4. **Development.** An unstamped page on a local dev port came from the front-end static server, and under the Kiss port-block convention (Section 4.5) the backend runs one port above it. This derivation works for any `portBase` with no front-end changes.

5. **Production fallback.** Same origin as the page.

12.10 Mobile Support

Kiss includes a separate `mobile/` directory with mobile-optimized login and screen templates. The `index.js` initialization detects screen size and loads the appropriate login page:

```
const screenPixels = screen.height * screen.width;
if (screenPixels < 600000)
    Utils.loadPage("mobile/login");
else
```

```
        Utils.loadPage('login');
```

Mobile screens use the same component library and `Server.call()` communication pattern as desktop screens, but with layouts optimized for smaller displays.

12.11 Cache Control

In development, you may want to disable browser caching to ensure that file changes are picked up immediately. The `index.html` file supports cache busting through the `SystemInfo.controlCache` flag:

```
const SystemInfo = {
    version: "2026.01.01.1",
    // append ?ver=VERSION to script URLs
    controlCache: true
};
```

When `controlCache` is `true`, the `getScript()` function appends a version query parameter to each script URL, forcing the browser to re-fetch after version changes.

Chapter 13

Client-Side Routing and State

Two tightly-coupled front-end modules give a Kiss application durable client-side state and real, deep-linkable URLs:

`AppState`
: The single client-side state store. It holds arbitrary front-end state that must survive screen navigation and page reloads — including the session token.

`Router`
: A hash-based client-side router. It gives each screen its own URL, makes the browser Back and Forward buttons navigate within the application, and resumes the user's location across a reload.

They are coupled because the router's authentication gate and session-resume behavior are both expressed in terms of a single value held by `AppState`: the persisted session token `_uuid`. This chapter documents `AppState` first, then `Router`, and finally how they cooperate to resume a session from a deep link.

Both modules live under `src/main/frontend/kiss/` (framework code). The one application-owned piece is the route table, `routes.js`, described in Section 13.2.4.

13.1 AppState: The Client-Side State Store

`AppState` (`src/main/frontend/kiss/AppState.js`) is the framework's single place to keep front-end state that needs to outlive a screen transition or a page reload. Everything the application wants to remember between navigations goes through it.

A central design decision is that the session token is *not* special. It is simply the reserved entry `_uuid` stored alongside any other application state. Because everything lives in one store, logout is a single `AppState.clear()` call — there is no separate "session" concept to tear down.

13.1.1 The Storage Layering Rule

`AppState` is the *only* module permitted to touch `window.localStorage` or `window.sessionStorage` directly. This mirrors the rule that `DOMUtils` is the only module permitted to touch the DOM directly (Chapter 12). All other code — framework or application — reaches Web Storage through the `AppState` API, never through the browser objects.

Note

The layering rule is what makes the per-tab/per-origin storage choice (below) a one-line configuration change rather than a code change scattered across the application. Because all persistence flows through one module, the backing store is selected in exactly one place.

13.1.2 Backing Stores

The backing store is chosen *once* at startup from `SystemInfo.stateStore` (in `src/main/frontend/SystemInfo.js`). Three values are recognized:

Value	Backing store	Behavior
`'session'`	`sessionStorage`	Survives reload; **per-tab** (each tab is its own isolated session, so two tabs never clobber each other's state); cleared when the tab closes. *This is the default.*
`'local'`	`localStorage`	Survives reload, new tabs, and Electron restarts; **shared** across all tabs of the same origin.
`'memory'`	in-memory only	Lost on reload — the framework's classic single-page behavior.

If `SystemInfo.stateStore` is unset or holds an unrecognized value, `AppState` defaults to `'session'`.

13.1.3 The Per-Tab Default and Its Consequences

The default, `'session'`, makes state *per-tab*. Each browser tab gets an isolated `sessionStorage`, so two tabs of the same application never overwrite each other's state, and closing a tab discards that tab's state.

This has one important consequence to understand:

- **A new tab starts with no token, so it requires its own login.** Opening the application in a second tab does *not* carry the first tab's session across — the new tab has an empty `sessionStorage` and therefore no `_uuid`.

- **A reload within a tab resumes.** Because `sessionStorage` survives a reload, refreshing the page (or following a deep link in the same tab) keeps the `_uuid` and resumes the session.

Tip

If your deployment wants one login shared across every tab of the origin (for example, a kiosk or an Electron desktop application that should stay logged in across windows), set `SystemInfo.stateStore = 'local'`. The token then lives in `localStorage`, which is shared across tabs and survives an Electron restart. There is no cross-tab logout coordination, but under the per-tab default that coordination is unnecessary.

13.1.4 Namespacing and Isolation

Every stored key is namespaced with a `kiss.` prefix before it reaches the backing store. Two things follow from this:

- Kiss state never collides with anything else stored on the same origin (other libraries, other applications served from the same host).
- `AppState.clear()` removes *only* the `kiss.`-prefixed keys. Unrelated keys on the same origin are left untouched, so logging out of the Kiss application does not wipe out a co-resident application's storage.

The namespace is applied and stripped internally; callers always use the bare key (`'_uuid'`, not `'kiss._uuid'`).

13.1.5 Serialization Caveats

Values are stored by JSON serialization (`JSON.stringify` on write, `JSON.parse` on read). Only JSON-serializable data survives a round trip. In particular:

Warning

A `Date`, a `Map`, a `Set`, or a function does *not* survive being stored in `AppState`. A `Date` comes back as a string; a `Map`/`Set` comes back as an empty object; a function is dropped entirely. Store primitives, plain objects, and arrays. If you need a date, store it as a YYYYMMDD integer (Chapter 18) or an ISO string and reconstruct it on read.

Two robustness details are worth knowing:

- A corrupt stored entry (one that fails `JSON.parse`) is discarded rather than thrown — `get()` returns `undefined` and removes the bad entry.
- If a single write fails (for example, a quota error on one large value), `AppState` holds that one key in memory for the rest of the session and keeps the application running, rather than failing the write.

13.1.6 Graceful Fallback

When Web Storage is unavailable — private-browsing mode in some browsers, storage disabled by policy, or an exhausted quota — `AppState.init()` probes the requested store at startup and silently falls back to the in-memory backend. The application still runs; it simply behaves as if `stateStore` were `'memory'` (state is lost on reload). Call `AppState.backend()` to learn which backend is actually in use.

13.1.7 The API

All methods are static.

Method	Description
`init()`	Select the backing store from `SystemInfo.stateStore` and probe it. Called once by the bootstrap before any other `AppState` use.
`set(key, value)`	Store a JSON-serializable value. Passing `undefined` removes the key.
`get(key)`	Retrieve a value, or `undefined` if absent.
`has(key)`	`true` if the key is present.
`remove(key)`	Remove a single key.
`keys()`	All application keys currently held (namespace stripped), as a `string[]`.
`clear()`	Remove all Kiss state (used by logout). Only `kiss.`-namespaced keys are removed.
`backend()`	The resolved backend in use: `'memory'`, `'session'`, or `'local'`.

Typical usage from application code:

```
// Remember the user's grid filter so it survives a reload.
AppState.set('orderGridFilter', { status: 'open',
    region: 'NE' });

// ... after a reload, on the next visit to the screen:
// {status:'open', region:'NE'}
const filter = AppState.get('orderGridFilter');
if (filter)
    grid.applyFilter(filter);

// Remove it when no longer wanted.
// same as AppState.remove('orderGridFilter')
AppState.set('orderGridFilter', undefined);
```

13.1.8 The Session Token Is Just `_uuid`

`Server.js` (Chapter 19) consumes `AppState` for its session token. There is no separate session variable:

```
// Server.js (excerpt)
static setUUID(uuid) {
    // login stores the token here
    AppState.set('_uuid', uuid);
}

static get uuid() {
    // every Server.call() reads it here
    return AppState.get('_uuid') || '';
}

static async logout(captureReturn = false) {
    // ... call the backend Logout service ...
    // logout is a single clear()
    AppState.clear();
    // ... then reload a fresh index.html onto /login
    // (see below) ...
}
```

Because the token persists through the configured backing store, a reload within a tab keeps the user logged in, and the router can resume the user's location (Section 13.4). The boot id used for server-restart detection (`_bootId`) is stored the same way.

13.1.9 Cross-Screen Data Helpers

The "pass data between screens" helpers on `Utils` are backed by `AppState`:

Method	Description
`Utils.saveData(key, val)`	Store a value for another screen to read.
`Utils.getData(key)`	Read a previously stored value (leaves it in place).
`Utils.getAndEraseData(key)`	Read a value and remove it in one step (the common "hand off once" pattern).

These are stored under a `data.` sub-namespace inside `AppState`, so application data passed between screens can never collide with framework entries such as `_uuid` or `_bootId`:

```
// Screen A: stash the selected customer id, then navigate.
Utils.saveData('selectedCustomerId', 12345);
Router.go('/customer-detail');

// Screen B: pick it up once.
// 12345, then erased
const id = Utils.getAndEraseData('selectedCustomerId');
```

Note

Because these helpers ride on `AppState`, data passed between screens survives a reload and is per-tab (under the default backend). An in-memory variable could not provide this — a router-driven reload would wipe it — which is why the helpers use persistent, namespaced storage.

13.2 Router: Hash-Based Client-Side Routing

`Router` (`src/main/frontend/kiss/Router.js`) gives each screen its own URL. Instead of a single opaque page that loads screens

invisibly, the application now has addressable locations such as `#/controls` or `#/customer/123`.

13.2.1 Why Hash Routing

The router encodes the current screen in the URL *hash* (the part after `#`). Hash routing needs no server-side URL rewriting, because the server never sees the hash — it is processed entirely in the browser. As a result the router works identically across every Kiss deployment target:

- the development static server,
- `file://` (opening the files directly),
- Electron desktop, and
- a deployed WAR served by Tomcat.

The benefits the router brings:

- **Deep links and bookmarks** — the hash changes per screen, so any screen can be linked to or bookmarked directly.
- **Back/Forward become in-app navigation** — pressing the browser Back button moves between the screens the user visited, rather than leaving the application.
- **Session resume** — a reload (or a followed deep link) lands on the same screen, validated lazily by the first backend call (Section 13.4).

13.2.2 Two Navigation Levels

The router maps the hash onto Kiss's two existing navigation levels:

Full-body screens
: A "shell" or the login page. These are loaded by replacing `document.body`. A route with no `tag` is a full-body screen.

Sub-screens
: Screens loaded into a region *within* a shell — for example the home shell's content area, `app-screen-area`. A route with a `tag` is a sub-screen; the router first ensures the sub-screen's shell occupies the body, then loads the sub-screen into the named region.

13.2.3 Route Definitions

A route is a path plus a definition object. The definition fields are:

Field	Description
`page`	The `loadPage` path (without extension), *or* a function returning one. A function lets a route choose its page at navigation time — for example, a device-aware mobile/desktop selection.
`tag`	The id of the container element the screen loads into. Omit it for a full-body screen (shell or login).
`shell`	For a sub-screen (a route with a `tag`), the route whose page hosts it and must occupy the body first. **Defaults to the default route** (the application's main shell), so it is usually omitted.
`focus`	The id of the control to focus after the screen loads.
`auth`	Defaults to `true` (the route requires a session). Set it to `false` for public routes such as login.

13.2.4 The Route Table (`routes.js`)

Routes are registered in the application-owned file `src/main/frontend/routes.js` using `Router.add(path, def)`. This is the one piece of the routing system that belongs to the application rather than the framework; add a route here for each addressable screen. `index.js` calls `Router.start()` once the components are loaded.

```
// routes.js -- the application's route table
(function () {

    // A route's page may be a function for
    // device-aware selection.
    function loginPage() {
        return (screen.height * screen.width < 600000)
            ? 'mobile/login' : 'login';
    }
    function homePage() {
        return (screen.height * screen.width < 600000)
            ? 'mobile/page1' : 'screens/Framework/Framework';
    }

    // Full-body screens (no tag). /login is public; / is the
```

```
    // default shell.
    Router.add('/login', { page: loginPage, auth: false });
    Router.add('/',    { page: homePage });

    // Sub-screens loaded into the home shell's content region.
    // shell defaults to '/', so it is omitted.
    Router.add('/controls', {
        page: 'screens/Controls/Controls',
        tag: 'app-screen-area', focus: 'ctl-text' });
    Router.add('/crud', {
        page: 'screens/CRUD/CRUD',
        tag: 'app-screen-area' });
    Router.add('/users', {
        page: 'screens/Users/Users',
        tag: 'app-screen-area' });
    Router.add('/customer/:id', {
        page: 'screens/Customer/Customer',
        tag: 'app-screen-area' });

    // File-based fallback (see below): any unmatched URL is
    // loaded as a screen under 'screens', into the home
    // shell's content region.
    Router.setScreenRoot('screens', { shell: '/',
        tag: 'app-screen-area' });

})();
```

`Router.setDefault(path)` changes the route used when the hash is empty or unmatched; it defaults to `'/'`.

13.2.5 Path Parameters

A path segment of the form `:name` is a path parameter. When the router matches such a route, it:

1. parses the parameter values out of the URL,
2. passes them to the screen through the existing `loadPage` `argv` mechanism, and
3. makes them readable from anywhere via `Router.params()`.

So a deep-linked screen can rebuild its state purely from the URL:

```
// Route: Router.add('/customer/:id', {
//           page: 'screens/Customer/Customer',
//           tag: 'app-screen-area' });
```

```
// URL:   #/customer/123

// Inside screens/Customer/Customer.js:
const id = Router.params().id;      // '123'
const res = await Server.call(WS, 'getCustomer', { id });
```

13.2.6 File-Based Fallback Routing

Registering a route for every screen is unnecessary. `Router.setScreenRoot(root, {shell, tag})` enables *file-based fallback*: a URL that matches no registered route is treated as a screen path under `root`.

With root `'screens'`, the URL `#/Reports/Daily` loads `screens/Reports/Daily` into the configured shell and region — no `Router.add()` required. Register a route explicitly only for screens that need a clean or stable URL, path parameters, or special metadata (such as `auth: false`); every other screen resolves by convention.

The options control where fallback screens render: `shell` defaults to the default route, and without a `tag` they load full-body. Three rules keep fallback routing safe:

- **Fallback screens always require authentication.** There is no way to expose a public screen by convention — a public screen must be registered explicitly with `auth: false`.

- **No path traversal.** Empty segments and `.` / `..` segments are rejected, so a URL cannot escape the screen root.

- **Every screen file under the root becomes URL-addressable.** This is the point of the feature, but it is also a security consideration: with fallback enabled, the route table is no longer an allowlist. Anything reachable as a file under `root` can be navigated to (subject to the auth gate).

Warning

Enabling `setScreenRoot` makes the route table no longer an allowlist of reachable screens. If a screen must never be directly addressable, either keep it outside the screen root or do not enable fallback routing. All fallback screens still pass through the authentication gate, so an unauthenticated user cannot reach them.

13.2.7 The `?tag` Override

A `?tag=` query parameter on the URL — or the second argument to `Router.go(path, tag)` — overrides a screen's region at navigation time. This lets the same screen be directed into a different container without a separate route. For example, `Router.go('/controls', 'side-panel')` loads the Controls screen into `side-panel` instead of its route's default `app-screen-area`.

13.2.8 The API

All methods are static.

Method	Description
`add(path, def)`	Register a route; `path` may contain `:name` parameters.
`setDefault(path)`	Set the route used when the hash is empty or unmatched (default `'/'`).
`setScreenRoot(root, opts)`	Enable file-based fallback routing under `root`; `opts` is `{shell, tag}`.
`start()`	Begin routing: listen for hash changes and dispatch the current hash. Call once at startup.
`isStarted()`	`true` once `start()` has been called.
`go(path, tag?)`	Navigate to a path, pushing a Back-button history entry. The optional `tag` overrides the route's region.
`replace(path)`	Navigate to a path, replacing the current history entry (no new Back step).
`params()`	The current route's path parameters, e.g. `{id: '123'}`.

Method	Description
`query()`	The current hash query string parsed to an object, e.g. `{return: '#/controls'}`.
`returnTarget()`	The post-login destination: the `return` query parameter when present, otherwise `'/'`.
`current()`	The currently matched route path.

13.2.9 Navigation: `go` vs. `replace`

The two navigation methods differ only in their effect on browser history:

`Router.go(path, tag?)`
: Pushes a new history entry, so the previous screen becomes the Back-button target. Use this for ordinary navigation.

`Router.replace(path)`
: Replaces the current history entry, leaving no Back step. Use this when going Back to the current location would be wrong — most notably after login, so that pressing Back does not return to the login screen.

13.3 The Dispatch Algorithm

The same algorithm runs when `Router.start()` is called and on every `hashchange` (which fires whenever the hash changes, including Back/Forward). Conceptually:

1. **Match.** Parse the hash into a path plus a query object. Find the registered route whose pattern matches. If none matches and file-based fallback is enabled, build a fallback route under the screen root. If there is still no match, fall back to the default route.

2. **Auth gate.** If the route requires authentication (`auth !== false`) and `AppState.get('_uuid')` is absent, `replace('/login?return=<hash>')` — redirecting to login while remembering where the user was headed.

3. **Ensure the shell.** For a sub-screen (a route with a `tag`), make sure the sub-screen's shell page occupies the body first; for a full-body screen, load it directly into the body.

4. **Load.** `loadPage` the screen — into the named region for a sub-screen, or into the body for a full-body screen — passing the path parameters and the focus id.

The shell is reloaded only when the body is not already showing it, so navigating between sub-screens of the same shell does not rebuild the shell each time.

13.4 Session Resume and the Auth Gate

The auth gate (step 2 above) plus persistent `AppState` produce *session resume* for free. Consider a user who has logged in and is sitting on `#/customer/123` when the tab is reloaded:

1. On reload, the bootstrap re-initializes `AppState`. Because the backend is `sessionStorage` (the default), the persisted `_uuid` is still present.

2. `Router.start()` dispatches the current hash, `#/customer/123`. The route requires auth; `AppState.get('_uuid')` is present, so the gate passes and the screen loads straight through.

3. The session is validated *lazily* by the first `/rest` call the screen makes. If the session is still valid, nothing more happens — the user simply resumes.

4. If the session has expired, that first call returns `_ErrorCode 2`, which triggers `Server.logout(true)` — clearing `AppState` and reloading a fresh `index.html` onto `/login` *carrying the current location as the* `return`. After re-authenticating, the user lands back on `#/customer/123`.

The deep-link case is identical: opening `#/customer/123` in a tab that already has a persisted `_uuid` loads straight through and is validated lazily by the first backend call.

13.4.1 Login and Logout Integration

The login and logout paths close the loop:

Login
: On a successful login, `login.js` calls `Router.replace(Router.query().return || '/')` — equivalently `Router.replace(Router.returnTarget())`. It uses `replace` (not `go`) so that pressing Back does not return to the login screen, and it honors the `return` the auth gate stashed, so the user lands back where they were headed.

Logout
: `Server.logout()` clears `AppState` and performs a *full page reload* of a fresh `index.html`, landing on the login route. The login route's URL form (`#/login`, `#/login?return=...`) is defined in exactly one place — `Router.loginHash(captureReturn)`, also used by `Router.gotoLogin()` — so an intentional logout (`Server.logout()`) lands on `/login` with no `return`, while a session-expiry logout (`Server.logout(true)`) captures the current location as the `return` for resume. The reload re-boots the tab on the latest cache-busted code — which is how an open tab picks up a new release after a redeploy (Chapter 14).

```
// login.js (on success)
const res = await Server.call('', 'Login', data);
if (res._Success) {
    // persisted through AppState
    Server.setUUID(res.uuid);
    // back to where the user was headed, or '/'
    Router.replace(Router.returnTarget());
}
```

13.4.2 Browser Back/Forward Is Now In-App

Because each screen has its own hash, the browser Back and Forward buttons move *between screens* rather than leaving the application. This is the intended behavior. The per-screen unsaved-changes guard (`DOMUtils.preventNavigation`, Chapter 20) is available as an opt-in for screens that must warn before navigating away.

Note

Session expiry vs. server restart are detected separately. The resume path above handles an expired *session* — the back end is still the same process, but the session timed out. A back-end *restart* (the process itself was bounced, invalidating every in-memory session) is detected by a distinct boot-id mechanism that runs once at startup, before routing begins. See the "Server-Restart Detection" material in Chapter 20.

13.5 Putting It Together

A minimal but complete setup looks like this:

```
// routes.js -- application-owned
(function () {
    Router.add('/login', { page: 'login', auth: false });
    Router.add('/',
        { page: 'screens/Framework/Framework' });
    Router.add('/customer/:id', {
        page: 'screens/Customer/Customer',
        tag: 'app-screen-area'
    });
    Router.setScreenRoot('screens', { shell: '/',
        tag: 'app-screen-area' });
})();

// index.js -- start routing once components are ready
Utils.afterComponentsLoaded(function () {
    Router.start();
});

// Any screen -- navigate by URL, pass data, read params
// survives reload, per-tab
Utils.saveData('lastViewed', Date.now());
// pushes history; #/customer/123
Router.go('/customer/123');
// ... inside Customer.js:
const id = Router.params().id;          // '123'
```

With this in place: every screen is addressable and bookmarkable, the Back button navigates within the application, a reload resumes the user's location, a brand-new tab requires its own login (under the per-tab default), and arbitrary ap-

plication state survives navigation through `AppState` and the `Utils.saveData`/`getData`/`getAndEraseData` helpers.

Chapter 14

Cache Busting and Browser Security

This chapter covers two cross-cutting front-end concerns that the Kiss framework solves with zero servlet-container configuration: forcing every browser to download the new code on each release (*cache busting*), and hardening the application against cross-site scripting, clickjacking, and related browser-level attacks (the *Content-Security-Policy* and the accompanying security headers).

Both mechanisms are deliberately deployment-agnostic. They behave identically on the development static server, on a `file://` (Electron) origin, and on a deployed WAR served by Tomcat, and neither requires you to touch a web server, a CDN, or a servlet container's configuration files.

14.1 Cache Busting

14.1.1 The Problem

When you ship a new release, every client that has the old front-end cached must re-download the changed files — HTML, CSS, JavaScript, third-party libraries, and images. The conventional technique is to append a version query parameter (`?ver=...`) to each URL so that a version bump produces new, uncached URLs.

This works for every file *except one*: the file that carries the version number itself. That file cannot also be version-stamped, because the browser would have to already know the new version in order to request the fresh copy — a chicken-and-egg problem. If the

version-carrying file is ever served stale, the whole scheme pins itself to an old version and never updates.

Note

The entire cache-busting design hinges on one invariant: **the file that carries the version number must never be served stale.** Everything else follows from keeping that one file perpetually fresh.

Kiss solves this by putting the version in the only file it can keep perpetually fresh with no server cooperation: `index.html`.

14.1.2 The Version and the Application Mode

Two `<meta>` tags in `index.html` are the *only* per-deployment cache values in the entire system:

```
<meta name="app-version" content="EDIT-1">
<meta name="app-mode" content="development">
```

`app-version`
: The version of the *application* built on Kiss (not of the Kiss framework itself). In production it is the bust token appended as `?ver=<version>` to every other downloaded file: bumping it invalidates every cached file at once.

`app-mode`
: Either `development` or `production` (any other value behaves as `development`). The mode selects the bust token. In `production` the token is the stable `app-version`, so the browser caches everything until the version changes. In `development` the token is a fresh per-page-load timestamp, so nothing is ever served from cache and a plain reload always fetches fresh files.

The source ships with `EDIT-1` and `development`. Production values are stamped automatically by `./bld war` (Section 14.1.9); you do not bump the version or switch the mode by hand.

14.1.3 The Byte-Stable Inline Kernel

`index.html` contains one small inline `<script>`, the *bootstrap kernel*. It is intentionally tiny and byte-stable — it carries no per-deployment values (those live in the `<meta>` tags) so its bytes never change from one deployment to the next. Its job is fourfold: keep `index.html` itself uncached, read the version from the meta, expose a global `window.cacheBust()` helper, and load `kiss/bootstrap.js` version-busted.

```
<script>
    (function () {
        function meta(n) {
            var m = document.querySelector(
                'meta[name="' + n + '"]');
            return m ? m.getAttribute('content') : '';
        }
        var ver = meta('app-version');
        var cacheOn = meta('app-mode') === 'production';
        window.KissVersion = ver;
        window.KissCacheOn = cacheOn;
        //  Bust token: the release version in production
        //  (stable across a release), or a fresh
        //  per-page-load timestamp in development so
        //  nothing is ever served from cache.
        var token = cacheOn ? ver : String(Date.now());
        window.KissBustToken = token;
        window.cacheBust = function (url) {
            return url + (url.indexOf('?') >= 0 ? '&' : '?')
                + 'ver=' + token;
        };
        //  Keep index.html itself uncached via a ?now
        //  redirect (both modes) so the version/meta and
        //  this kernel are always fresh on reload.
        var now = null, s = window.location.search;
        if (s) {
            var parts = s.substring(1).split('&');
            for (var i = 0; i < parts.length; i++) {
                var kv = parts[i].split('=');
                if (kv[0] === 'now')
                    now = kv[1];
            }
        }
        if (!(now &&
              Math.abs((Date.now() - Number(now)) / 1000)
                  <= 30)) {
            window.onbeforeunload = null;
            var args = s ?
```

```
                s.substring(1).replace(/now=[^&]*&*/g, '') : '';
            window.location.href = 'index.html?now=' +
                Date.now() +
                (args ? '&' + args : '') +
                window.location.hash;
            return;
        }
        var b = document.createElement('script');
        b.src = window.cacheBust('kiss/bootstrap.js');
        document.head.appendChild(b);
    })();
</script>
```

The `?now` redirect

The key trick is the `?now` redirect. On every page load — in both modes — the kernel looks for a `now=` query parameter carrying a timestamp. If that timestamp is missing or older than 30 seconds, the kernel redirects the browser to `index.html?now=<current-timestamp>` and stops. Because the timestamp is freshly generated by `Date.now()`, the redirect target is a URL the browser has *never* seen, so it cannot be served from cache — it forces a fresh fetch from the server. On that second load the `now` value is fresh (within 30 seconds), so the kernel skips the redirect and proceeds to load `kiss/bootstrap.js`.

The consequence is that **`index.html` loads twice** on a cold navigation (once to bounce through the `?now` redirect, once to actually run). The 30-second tolerance is the redirect's loop terminator — the freshly redirected load must recognize itself as fresh and stop — and it also skips the extra hop on rapid successive reloads. The extra round trip is the price of guaranteeing freshness with no server configuration.

Note

This is the deployment-agnostic way to keep a file fresh on *any* host. It needs no `Cache-Control` header, no CDN purge, no `.htaccess` rule, and no servlet-container tuning — it works on the dev static server, on `file://`, on Tomcat, and on Electron, because the freshness is enforced entirely by the unique `?now` query string the browser cannot have cached.

The redirect preserves the URL's hash fragment (so a deep-linked screen survives the bounce) and any other query parameters (stripping only the previous `now=`). It also clears `window.onbeforeunload` first, so an unsaved-changes guard cannot block the redirect.

`window.cacheBust()`

The kernel exposes `window.cacheBust(url)`, which appends `?ver=<token>` (or `&ver=<token>` when the URL already has a query string), where the token is the application version in production and the per-page-load timestamp in development. It loads `kiss/bootstrap.js` through this helper, so even the bootstrap loader is busted.

14.1.4 Busting Everything Else: `bootstrap.js`

`kiss/bootstrap.js` takes over from the kernel and loads *every* remaining downloaded resource through the buster. Its `getScript()` / `getScripts()` loaders and its `addStylesheet()` helper all route their URLs through `window.cacheBust()`:

```
function cacheBustUrl(url) {
    return (typeof window.cacheBust === 'function')
        ? window.cacheBust(url) : url;
}

function getScript(url) {
    return new Promise(function (resolve, reject) {
        const script = document.createElement("script");
        script.src = cacheBustUrl(url); // version-busted
        // ...
    });
}
```

Through these helpers, `bootstrap.js` busts:

- the stylesheets (`normalize.css`, `kiss/starter-theme.css`, `kiss/Utils.css`, the AG-Grid theme, ...);
- the third-party libraries (the data grid and the rich-text editor);
- `SystemInfo.js` (the per-deployment configuration);

- all framework JavaScript (`DOMUtils.js`, `AppState.js`, `Utils.js`, `Server.js`, `Router.js`, ...) and the application's `routes.js` and `index.js`.

Screen HTML loaded later through `Utils.getHTML()` is busted too. After fetching a screen's HTML, `Utils.cacheBustHtmlResources()` rewrites relative `<img src>` URLs in the HTML *text*, before it is inserted into the DOM, so the browser never even issues a request for the un-busted image URL. Absolute, protocol-relative, `data:`, and already-busted URLs are left alone.

`DOMUtils.fetchHTML()` — the generic `$.get`-style HTML fetch available to application code — participates as well. A relative URL is fetched with the standard bust token, conservatively: pre-existing query arguments are preserved (the token is appended), a `#fragment` stays in place, and URLs that are absolute (`http:`, `https:`, `//`), `data:`, `blob:`, or that already carry a `ver=` argument are left untouched.

After loading `SystemInfo.js`, `bootstrap.js` also mirrors the meta values onto `SystemInfo`, so application code may read them from either place:

```
await getScript("SystemInfo.js");
// from <meta name="app-version">
SystemInfo.softwareVersion = window.KissVersion;
// true in production mode
SystemInfo.controlCache = window.KissCacheOn;
```

14.1.5 What Is *Not* Busted

One category falls outside the busting path:

`background-image:url()` inside CSS
: The browser resolves these static URLs itself; they are not rewritten. They refresh only when their containing stylesheet is re-fetched (and the stylesheet *is* busted, so a version bump still pulls a fresh stylesheet, which then re-resolves its background images). When an image referenced from CSS is changed in place, rename the asset.

Tip

Version busting is global: because a single version token drives every URL, each release re-downloads *everything*, including unchanged third-party libraries. This is a deliberate trade — absolute correctness (no client ever runs a stale mix of files) in exchange for re-downloading some bytes that did not change.

14.1.6 Why It Is Rock-Solid

The design is self-healing. The version lives in always-fresh `index.html`, and the kernel is byte-stable. Suppose a browser *does* serve a stale, cached copy of `index.html`: that stale copy still contains the identical byte-stable kernel, which immediately redirects to `index.html?now=<fresh>` — a never-before-seen URL that forces a fresh fetch from the server. The fresh copy then carries the current version, and busting proceeds correctly. No cacheable file anywhere in the chain can pin the application to a stale version, because the one file that determines the version refuses to be served stale.

14.1.7 Development Mode: Nothing Is Ever Cached

In development mode the bust token is a fresh timestamp generated on every page load, so no two loads ever request the same URLs. The practical effect is that **a plain browser reload always fetches every file fresh** — edited framework files, application screens, stylesheets, and loaded HTML alike. There is never a need for a hard reload or for clearing the browser cache during development.

The one thing cache busting cannot do — in either mode — is update a page that is already *running*: an open tab keeps executing the code it booted with until a page load happens. In development that page load is your reload after an edit. In production it happens automatically, as the next section explains.

14.1.8 How an Open Tab Picks Up a New Release

Cache busting guarantees that every *page load* gets the current release. But a browser tab that stays open across a redeploy performs no page load — it keeps running the JavaScript it booted with, against a back end that has since changed.

Kiss closes this gap through the logout path. `Server.logout()` (Chapter 19) performs a **full page reload** of a fresh `index.html` (using the same always-fresh `?now` URL form the kernel uses), landing on the login route. A redeploy restarts the back end, which invalidates every session; the next service call from an open tab therefore receives the standard session-expired response, which funnels through `Server.logout()` — and the reload re-boots the tab on the new release's cache-busted files.

Reloading on every logout costs nothing when there is no new release: the bust token is the unchanged version, so the re-booted page loads from the browser cache. When the version *did* change, the reload is exactly what re-downloads the new code. Any session-expiry resume target is carried into the reloaded login route, so the user still lands back where they were after re-authenticating (Chapter 13).

14.1.9 Production Stamping by `./bld war`

You never edit the version by hand for a release. `./bld war` stamps production cache-busting values automatically, via `KissBuildUtils.stampVersion(explodedDir)`:

```
./bld war
```

`stampVersion()` edits *only the meta values* in the staged `work/exploded` copies of the front-end files:

- `app-version` is set to a fresh random UUID (so every WAR forces every client to re-download);
- `app-mode` is set to `production`;
- `SystemInfo.releaseDate` (in `SystemInfo.js`) is set to the build date;
- `SystemInfo.sameOriginBackend` is set to `true`, because a WAR is always served by the back end itself (Section 12.9).

```
public static void stampVersion(String explodedDir) {
    final String uuid = java.util.UUID.randomUUID().toString();
    final String date = java.time.LocalDate.now().toString();
    final java.nio.file.Path idx = java.nio.file.Paths.get(
```

```
            explodedDir, "index.html");
        String html = java.nio.file.Files.readString(idx);
        html = html.replaceAll(
            "(name=\"app-version\"\\s+content=\")[^\"]*(\")",
            "$1" + java.util.regex.Matcher.quoteReplacement(uuid)
                + "$2");
        html = html.replaceAll(
            "(name=\"app-mode\"\\s+content=\")[^\"]*(\")",
            "$1production$2");
        java.nio.file.Files.writeString(idx, html);
        // ... SystemInfo.releaseDate replaced with the build
        // date ...
        // ... then stampSameOriginBackend(explodedDir) ...
    }
```

Because it touches only the staged copies, the *source* files keep their `EDIT-1` / `false` placeholders. And because it rewrites only the meta values — never the bootstrap kernel — the kernel's bytes are unchanged, so the CSP hash that pins the kernel (Section 14.2.2) is unaffected by stamping.

14.1.10 Restoring the Staged Files After the WAR Is Built

There is a subtle build-ordering hazard. `stampVersion()` rewrites the staged `index.html` and `SystemInfo.js` *in place*, giving them production busting and a newer modification time. Once the WAR is jarred, those stamped files have already been captured inside the archive — but they are still sitting in `work/exploded` with production values and fresh mtimes.

The problem is that the routine which stages the front-end (`copyTree`) is *mtime-incremental*: it copies a source file over the staged copy only when the source is newer. After stamping, the staged copies are newer than the source, so a later `develop` build would *not* overwrite them — and the developer would silently end up running with production busting (stale `?ver`, the double `index.html` load) on their dev machine.

To prevent this "poisoning," `war()` explicitly restores the two staged files from source, using `copyForce` (which overwrites regardless of mtime), immediately after the WAR is jarred:

```
    public static void war() {
        buildSystem();
        stampVersion(explodedDir);
```

```
        copyForce("src/main/core/WEB-INF/web-secure.xml",
            explodedDir + "/WEB-INF/web.xml");
        createJar(explodedDir, BUILDDIR + "/Kiss.war");
        copyForce("src/main/core/WEB-INF/web-unsafe.xml",
            explodedDir + "/WEB-INF/web.xml");
        // Un-stamp the staged frontend files so a later 'develop'
        // isn't poisoned with production busting. copyTree is
        // mtime-incremental and would NOT overwrite the newer
        // stamped files; copyForce overwrites regardless.
        copyForce("src/main/frontend/index.html",
            explodedDir + "/index.html");
        copyForce("src/main/frontend/SystemInfo.js",
            explodedDir + "/SystemInfo.js");
    }
```

This mirrors the build's existing `web.xml` dance, where `web-secure.xml` is swapped in for the jar step and `web-unsafe.xml` is restored afterward (see Chapter 32). In both cases the WAR captures the secure/production form, and the staged tree is restored to its safe development form so the next local build behaves correctly.

Warning

The restore after jarring is not optional. Without it, the mtime-incremental `copyTree` leaves the production-stamped `index.html` and `SystemInfo.js` in `work/exploded`, and a subsequent `./bld develop` runs with production cache busting — meaning your local edits may be served stale behind a frozen `?ver`.

14.2 Browser Security

Kiss ships a single servlet filter, `SecurityHeadersFilter` (in `src/main/core/org/kissweb/restServer/`), that adds a baseline set of browser-security response headers — most importantly the Content-Security-Policy — to every response.

14.2.1 A Self-Registering Filter, No Configuration Needed

The entire CSP is delivered as an **HTTP response header**, not as an `<meta>` tag. There are two reasons. First, the Report-Only form of the policy (used during rollout) is *invalid* in a meta tag —

browsers ignore it — so a meta tag could never support the report-only-then-enforce workflow. Second, a response header is the single source of truth for every deployment (production and Electron are all served by the same Tomcat), whereas a meta tag would have to be duplicated and kept in sync.

The filter self-registers through annotation scanning with `@WebFilter("/*")`, so it needs no `web.xml` entry and ships entirely inside the application WAR. Like the framework's `@WebServlet`-annotated endpoints, it requires no servlet-container or Tomcat configuration — dropping the WAR in is enough.

```
@WebFilter(filterName = "SecurityHeadersFilter",
          urlPatterns = "/*", asyncSupported = true)
public class SecurityHeadersFilter implements Filter {
    // ...
}
```

14.2.2 The Content-Security-Policy

The XSS-relevant policy is the constant `CONTENT_SECURITY_POLICY`:

```
public static final String CONTENT_SECURITY_POLICY =
        "default-src 'self'; " +
        // The one allowed inline script is the byte-stable
        // bootstrap kernel in index.html. If that kernel is
        // ever edited, recompute this hash.
        "script-src 'self' " +
        "'sha256-JaGBSNrOPztIc2kS" +
        "DoiyDCHVqjcvVQSMYv5X2fO3RCE='; " +
        "style-src 'self' 'unsafe-inline'; " +
        "img-src 'self' data: blob:; " +
        "font-src 'self' data:; " +
        "connect-src 'self'; " +
        "object-src 'none'; " +
        "base-uri 'self'; " +
        "form-action 'self'";
```

Directive by directive:

`default-src 'self'`
: The fallback for any fetch directive not otherwise listed: only same-origin resources are allowed.

`script-src 'self' 'sha256-...'`
: Scripts must come from the same origin — with exactly one

exception: the inline bootstrap kernel in `index.html`, allowed by its SHA-256 hash. Every other script in the application is external and served under `'self'`. No `'unsafe-inline'` and no `'unsafe-eval'` are permitted for scripts.

`style-src 'self' 'unsafe-inline'`
: Inline styles are allowed because the bundled rich-text editor and data grid (CKEditor / AG-Grid) inject inline `<style>`/`style=""` at runtime.

`img-src 'self' data:  blob:`
: Same-origin images, plus `data:` and `blob:` URLs, which are required by the binary-image fetch path (`Server.binaryCall`).

`font-src 'self' data:`
: Same-origin fonts plus inline `data:` fonts.

`connect-src 'self'`
: Restricts the origins `fetch`/XHR/WebSocket may reach. Same-origin suffices because every page this filter covers is served by the back end itself (the front-end dev server cannot set headers, so its pages never carry a CSP at all). **A separated production back-end must be added here** (see below).

`object-src 'none'`
: Forbids `<object>`, `<embed>`, and `<applet>` plugins entirely.

`base-uri 'self'`
: Prevents an injected `<base>` tag from rewriting relative URLs to an attacker's origin.

`form-action 'self'`
: Forms may only submit to the same origin.

Warning

`script-src` pins the inline kernel by its SHA-256 hash. If the bootstrap kernel in `index.html` is ever edited — even by a single byte — its hash changes and the browser will refuse to run it, breaking the entire application. After any kernel edit you must recompute the hash and update `SecurityHeadersFilter.CONTENT_SECURITY_POLICY`. This is precisely why the kernel is kept byte-stable and carries no per-deployment values (those live in the `<meta>` tags, which are *not* part of the hashed script).

Note

In a split deployment where the back-end lives on a different origin from the front-end, add that back-end's origin to `connect-src`. Otherwise the browser will block the application's `Server.call()` requests as CSP violations.

14.2.3 Report-Only Versus Enforcing

A Content-Security-Policy can be delivered in two forms. The `Content-Security-Policy` header *enforces* the policy: violations are blocked. The `Content-Security-Policy-Report-Only` header only *reports* violations (to the browser console / a report endpoint) without blocking anything. The report-only form lets you validate a policy against the real application — exercising every screen — and confirm there are no violations before you turn on enforcement, so you never break the app for users.

The filter chooses between the two with a single flag:

```
/** true => send as Content-Security-Policy-Report-Only
 *  (log, don't block); false => enforce. */
private static final boolean CSP_REPORT_ONLY = false;
```

The framework ships with `CSP_REPORT_ONLY = false`, so the XSS policy is **enforced**. To re-validate after changing the policy, exercise every screen (the rich-text editor, the data grid, file upload, binary images, login/logout) with the browser console open, confirm the console is free of CSP violations, then re-enable enforcement.

A validation pass does not require touching the compiled

flag: both switches can be overridden per deployment from `application.ini` (the `[main]` section), with the secure defaults applying when the keys are absent:

```
; report violations instead of blocking them
; (rollout / re-validation)
CspReportOnly = true

; emit no security headers at all -- see below before using
this
SecurityHeaders = false
```

`SecurityHeaders = false` exists chiefly for applications whose front end predates the byte-stable bootstrap kernel (Section 14.1.6): the CSP permits exactly one inline script, pinned by the kernel's hash, so a pre-kernel `index.html` with its own inline startup script would be blocked outright. Such an application disables the filter in its own configuration — keeping the framework source identical across applications — and re-enables it after its front end adopts the kernel design.

```
response.addHeader(CSP_REPORT_ONLY
    ? "Content-Security-Policy-Report-Only"
    : "Content-Security-Policy",
    CONTENT_SECURITY_POLICY);
```

Warning

The CSP must be validated by browsing the application through the **Tomcat origin** — the application's back-end port, `http://localhost:8001` with the default port block. The development static server is a prebuilt static file server and *cannot* set response headers, and a `file://` origin has no server at all — in both of those contexts the browser receives no CSP, so you would be testing nothing. Only the Tomcat-served origin exercises the filter.

14.2.4 Always-On Header Protections

Regardless of the report-only flag, and as long as the filter is enabled (the compiled master switch, which `SecurityHeaders = false` in `application.ini` can override per deployment), every response also

carries a set of header-only protections:

```
public void doFilter(ServletRequest req, ServletResponse res,
    FilterChain chain) {
    if (ENABLED && res instanceof HttpServletResponse) {
        final HttpServletResponse response =
            (HttpServletResponse) res;

        response.setHeader("X-Content-Type-Options",
            "nosniff");
        response.setHeader("Referrer-Policy", "no-referrer");

        // Clickjacking protection is always enforced.
        response.addHeader("Content-Security-Policy",
            "frame-ancestors 'none'");
        response.setHeader("X-Frame-Options", "DENY");

        // The XSS policy: report-only during rollout,
        // enforcing afterward.
        response.addHeader(CSP_REPORT_ONLY
            ? "Content-Security-Policy-Report-Only"
            : "Content-Security-Policy",
            CONTENT_SECURITY_POLICY);

        // HSTS only over HTTPS; never on plain-http dev.
        if (req instanceof HttpServletRequest
            && ((HttpServletRequest) req).isSecure())
            response.setHeader("Strict-Transport-Security",
                "max-age=31536000; includeSubDomains");
    }
    chain.doFilter(req, res);
}
```

Clickjacking protection
: `Content-Security-Policy: frame-ancestors 'none'` plus the legacy `X-Frame-Options: DENY`. These are sent *always*, independent of the report-only rollout — the application may never be embedded in a frame on any origin. (`frame-ancestors` must be a header; it has no meta-tag form, which is another reason CSP is delivered as a header.)

`X-Content-Type-Options: nosniff`
: Stops the browser from MIME-sniffing a response into a different content type than declared.

`Referrer-Policy: no-referrer`
: Prevents leaking URLs (which may carry application state) to

other origins via the `Referer` header.

`Strict-Transport-Security`
: Sent *only* when the request is secure (`request.isSecure()`), so it is never emitted on plain-HTTP development such as `http://localhost`. Over HTTPS it instructs the browser to use HTTPS for a year, including subdomains.

14.2.5 The Master Switch

The filter has its own master switch, separate from the CSP rollout flag:

```
/** Set to false to emit no security headers at all
 *  (pass-through). */
private static final boolean ENABLED = true;
```

Setting `ENABLED = false` turns the filter into a pass-through that emits no security headers whatsoever. This is useful only as a diagnostic — to A/B isolate whether these headers are responsible for some observed behavior. In normal operation it stays `true`.

Chapter 15

UI Components

Kiss provides a library of custom HTML components for building forms and screens. This chapter covers every component, its attributes, and its API. The complete component API is documented in `components.js` in the framework source.

15.1 The Component System

Components are custom HTML tags that Kiss registers at startup. Each component encapsulates rendering, validation, and event handling. When `Utils.loadPage()` injects screen HTML into the page, `Utils.rescan()` processes the HTML and activates any custom tags.

Every component instance is accessed through the `$$()` function:

```
$$('my-component-id').getValue();
$$('my-component-id').setValue("hello");
```

15.2 Common API

Most components share these methods (push-buttons, text-labels, and pictures support only a relevant subset — typically `onclick` rather than `onChange`):

Method	Description
`getValue()`	Get the component's current value.
`setValue(val)`	Set the component's value.
`clear()`	Clear the component to its default/empty state.
`enable()` / `disable()`	Enable or disable the component. Can pass a boolean: `enable(false)` disables.
`show()` / `hide()`	Show or hide the component. Can pass a boolean.
`isVisible()` / `isHidden()`	Check visibility state.
`isDisabled()` / `isReadOnly()`	Check enabled/readonly state.
`readOnly()` / `readWrite()`	Set read-only mode. Can pass a boolean.
`isDirty()`	Returns `true` if the user has changed the value.
`isError(desc)`	Validate; show error message if invalid; return `true` on error. The `desc` parameter is the field label shown in the error message.
`focus()`	Set keyboard focus to the component.
`onChange(fn)`	Set a change event handler (fires when user changes value and leaves the field).
`onCChange(fn)`	Immediate change handler on the text/numeric/date/time input components (fires as the value changes).

Warning

Always use Kiss component methods for events (`.onChange()`, `.onclick()`), not native DOM methods like `addEventListener()`. The Kiss methods handle event wiring consistently across all components.

15.3 Text Input

`<text-input>` is the most commonly used component. It renders a single-line text field.

```
<text-input id="first-name" required
            maxlength="50"></text-input>
<text-input id="password" password></text-input>
<text-input id="search" placeholder="Search..."></text-input>
```

Supported attributes:

Attribute	Description
`required`	Field must not be empty. `isError()` checks this.
`password`	Masks input characters.
`maxlength`	Maximum number of characters.
`minlength`	Minimum number of characters (validated by `isError()`).
`placeholder`	Placeholder text shown when empty.
`upcase`	Converts input to uppercase.
`fixcap`	Auto-capitalizes the first letter of each word.
`autocomplete`	Passed through to the underlying `<input>` element for browser autofill.

Additional methods:

```
$$('search').onEnter(() => performSearch()); // fires on Enter
key
```

15.4 Numeric Input

`<numeric-input>` accepts only numeric values with optional formatting.

```
<numeric-input id="salary" min="0" max="999999"
               decimal-places="2"></numeric-input>
<numeric-input id="quantity" min="1"></numeric-input>
<numeric-input id="price" money></numeric-input>
```

Supported attributes:

Attribute	Description
`min`	Minimum allowed value.
`max`	Maximum allowed value.
`decimal-places`	Number of decimal places (default 0).
`money`	Shorthand for 2 decimal places with dollar sign formatting.
`dollar-sign`	Display a dollar sign prefix.
`required`	Field must have a value.

`getValue()` returns a *number*, not a string. This eliminates the need for parsing on the frontend.

15.5 Date Input

`<date-input>` provides a date picker that works with Kiss's YYYYMMDD integer date format.

```
<date-input id="hire-date"></date-input>
```

Key methods:

Method	Description
`getDateValue()`	Returns a JavaScript Date object.
`getIntValue()`	Returns date as YYYYMMDD integer.
`getSQLValue()`	Returns date in SQL format.
`setValue(val)`	Accepts Date object or YYYYMMDD integer.
`setMinValue(val)`	Set minimum allowed date.
`setMaxValue(val)`	Set maximum allowed date.

Kiss also provides `<native-date-input>`, which uses the browser's built-in date picker instead of Kiss's custom picker.

15.6 Time Input

`<time-input>` accepts times in HHMM integer format.

```
<time-input id="start-time"></time-input>
```

15.7 Duration Input

`<duration-input>` accepts a duration (not a time of day) entered as two integers separated by a colon — hours and minutes. The minutes portion must be in the range 0–59. If the user omits the colon, the value is treated as hours only with an implied `:00` (so `3` means three hours, zero minutes). Input is restricted to digits and at most one colon.

```
<duration-input id="elapsed"></duration-input>
<duration-input id="shift" min="0:30" max="12:00"
                required></duration-input>
```

The control's value is the total number of minutes. `getValue()` returns the total minutes as an integer and `getHours()` returns the duration as a floating-point number of hours. Both return `0` when the control is blank or invalid.

```
$$('elapsed').setValue(135);      // displays as "2:15"
const minutes = $$('elapsed').getValue(); // 135
const hours  = $$('elapsed').getHours(); // 2.25
$$('elapsed').setHours(1.5);      // displays as "1:30"
```

On blur, the entered value is normalized to `H:MM` (for example, `5` becomes `5:00` and `5:7` becomes `5:07`). `min` and `max` attributes accept either an `"h:mm"` string or an integer number of total minutes.

If the `decimal-hours` attribute is set, the user enters and the control displays a decimal number of hours instead of `H:MM` (for example, the user types `1.25` for one hour and fifteen minutes). Only digits and at most one decimal point are accepted. The JavaScript API is unchanged: `getValue()` still returns total minutes (so `1.25` entered returns `75`) and `setValue(75)` displays `1.25`.

```
<duration-input id="elapsed-decimal"
                decimal-hours></duration-input>
```

15.8 Drop-Down

`<drop-down>` renders a select list. Items are added programmatically:

```
$$('department').clear();
$$('department').add('', '(select)'); // default prompt
for (let dept of departments)
    $$('department').add(dept.id, dept.name);
```

Key methods:

Method	Description
`add(value, label)`	Add an option with the given value and display label.
`addItems(items, valField, lblField)`	Add multiple items from an array of objects.
`fill(selectedItem, items, valField, labelField)`	Clear and fill the drop-down from an array of objects, selecting the given item.
`getValue()`	Returns the value of the selected option.
`setValue(val)`	Select the option with the given value.
`clear()`	Remove all options.
`onChange(fn)`	Handler called when selection changes.

Tip

When a drop-down requires a selection, use `add('', '(select)')` as the first item. This clearly indicates that a selection is needed, instead of defaulting to the first real option.

Opening a drop-down with the mouse or the keyboard (a click, `Space`, `Alt+Down`, or `F4`) shows a list drawn by Kiss itself, under the control or above it when there is more room there, rather than the browser's own pop-up. The reason is the filtering described next: while the browser's pop-up is open the browser delivers every keystroke to it and none to the page, so nothing on the page could filter the list. The Kiss list behaves as the native one does. The

current value is highlighted; `Up` and `Down` move it, firing `onChange` just as they do in a closed control; a click or `Enter` closes the list keeping the value; `Escape` closes it and returns to the value it opened with; and clicking elsewhere, leaving the control, or scrolling the page closes it. Touch input is left to the browser's native picker, which is the better control on a device with no keyboard to filter with. A read-only drop-down neither opens nor changes its value from the keyboard.

While a drop-down has focus, whether or not its list is open, typing narrows the list to the options whose label contains the typed characters. Matching is case-insensitive and the text may appear anywhere in the label, not only at its start, so typing `miss` in a list of states leaves only *Mississippi* and *Missouri*. With the list open, the typed text is shown at the top of the list and the rows narrow with each keystroke. As the user types, the current selection is kept when it still matches; otherwise the first matching option becomes selected and `onChange` fires, exactly as with the browser's own first-letter type-ahead. `Backspace` removes the last typed character. `Escape` erases the whole sequence, restores the full list, and returns to the selection that was current before typing began, so the user can start again; an open list stays open, and a second `Escape` closes it. Leaving the control also restores the full list, keeping whatever is then selected. A lone space opens and closes the list, as it does natively, but once a sequence has begun a space is part of it.

The filter changes only what the list displays and which option is selected. `size()`, `getValue(idx)`, `getLabel(idx)`, `getAllLabels()`, `selectedIndex()`, and the other index-based methods always refer to the complete list, and any method that modifies the list first restores it and closes the list.

15.9 Combo Box

`<combo-box>` is an editable drop-down — the equivalent of the Microsoft Windows combobox control. It behaves like a `<drop-down>` but also lets the user type a value that is not in the list (a *free entry*). As the user types, the list opens (if closed) and narrows to the items whose label contains the typed text anywhere, ignoring case; an empty field shows every item, and the list closes when nothing matches. Clicking the drop-down arrow, or clicking into the field, always reveals the complete list, and any text the user has typed

is preserved. The `Up` and `Down` arrow keys, whether the list is open or closed, show and move through only the items that match the current text and never widen it to the full list; `Enter` and `Escape` also act on the items shown. Setting the value programmatically never opens or filters the list.

A static list may be supplied as `<option>` content, exactly as with `<drop-down>`:

```
<combo-box id="color" placeholder="pick or type">
    <option value="Red">
    <option value="Green">
    <option value="Blue">
</combo-box>
```

Items may also be added programmatically. Each item carries a *value*, a *label* (the text shown and typed), and optional associated *data*:

```
$$('color').clearList(); // empty the suggestion list
$$('color').add('CA', 'California', {capital: 'Sacramento'});
$$('color').add('NY', 'New York');
// one argument: value and label are the same
$$('color').add('TX');

$$('color').onChange((value, text, data) => {
    // value is 'CA' when California is chosen, or the typed
    // text itself for a free entry that is not in the list
});
```

Note

`<combo-box>` is a custom drop-down — an input, an arrow, and a list — *not* the native HTML `<datalist>`. A `<datalist>` filters itself away once the typed text matches no option and will not reopen the full set. A Windows combobox, and therefore `<combo-box>`, also filters as the user types but always reopens the complete list from the arrow no matter what has been typed.

Key methods:

Method	Description
`add(value [, label, data])`	Add an item; `label` defaults to `value`, and `data` is arbitrary associated data.
`addItems(items, valField, lblField [, dataField])`	Add multiple items from an array of objects.
`fill(selected, items, valField, lblField [, dataField])`	Clear the list, fill it, and select the item whose value is `selected`.
`getValue()`	The underlying value of the chosen item, or the typed text for a free entry not in the list.
`getText()`	The text currently shown (the label), list item or free entry alike.
`setValue(val)`	Show the item whose value is `val`; if none matches, show `val` as free text.
`getData()`	Data associated with the shown item (`undefined` for a free entry).
`clear()`	Clear the entered text; the suggestion list is left intact.
`clearList()`	Remove all items from the suggestion list.
`onChange(fn)`	Commit handler; receives `(value, text, data)`.
`onCChange(fn)`	Per-keystroke handler; receives the current value.

Tip

Unlike `<drop-down>`, `clear()` here empties only the typed text and leaves the suggestion list intact; use `clearList()` to empty the list. Use `getValue()` when a coded value (such as `'CA'`) sits behind a friendly label (`'California'`): it returns the code for a chosen item, but the raw text for anything the user types that is not in the list. Add the `required` attribute to force a non-empty entry, then call `isError(desc)` during validation, exactly as for the other input controls.

15.10 Smart Chooser

`<smart-chooser>` is a `select`-like control that automatically avoids unusably long drop-down lists. Depending on how many items it is given, compared against a caller-supplied threshold, it renders as one of four presentations: a normal drop-down (few enough items); a disabled text field paired with a small chooser button that invokes an application-supplied `async` selection function — typically a search popup — when there are too many items for a drop-down; a disabled text field showing the single available choice (exactly one item and selection is not forced); or a disabled text field reading "(nothing to select)" (zero items and selection is not forced).

```
<smart-chooser id="employee" required></smart-chooser>
```

Supported attributes:

Attribute	Description
`required`	A selection is required.
`disabled`	Disables the control.
`default-option="label"`	Text shown before a selection is made (drop-down presentation only).
`inside-width="n"` / `outside-width="n"`	Sets the control's inner width, or outer width including padding, in pixels.

The control can be driven at either of two levels. The *low-level* API builds the list or the chooser wiring by hand, much like `<drop-down>`:

```
$$('employee').useDropdown();          // force the drop-doum
presentation
$$('employee').add('', '(choose)');
$$('employee').addItems(employees, 'id', 'name');
```

Key low-level methods:

Method	Description
`add(val, label, data)` / `addItems(items, valField, labelField, dataField)`	Populate the drop-down list; `labelField` can be a field name or a function that formats the label from the item.

Method	Description
`useDropdown([items, valField, labelField, dataField])`	Force the drop-down presentation, optionally filling it.
`forceSelect([title])`	Force the chooser-button presentation, showing `title` (default "(choose)") until a value is set.
`nothingToSelect()` / `singleValue(val, label, data)`	Force the "(nothing to select)" or single-value disabled presentations.
`setSelectFunction(fun)`	The `async` function run when the chooser button is clicked; its resolved item is handled by the caller (see the high-level API below for the common wiring).
`setForceSelection([flg])` / `setForceDropDown([flg])`	Override the automatic presentation choice in either direction.
`getValue()` / `setValue(val, label, data)`	Read or set the current selection.
`getLabel()` / `getData()`	Read the label text, or the associated data, of the current selection.
`clear()` / `reset()`	Empty the control, or restore it to its original underlying element (undoes a prior `forceSelect()`).
`disable([flg])` / `enable([flg])`	Enable or disable the control and its chooser button together.
`onChange(fn)`	Handler called with `(value, label, data)` when the drop-down selection changes.

The *high-level* API is the more common way to drive the control: supply the item source once, and `run()` picks the presentation automatically from the item count.

```
$$('employee').setup(15, true)              // threshold; add
a "(choose)" option
```

```
        .setupItems(employees, 'id', 'name')
        .setupSelectFunction(searchEmployees, 'id', 'name')
        .run();
```

Method	Description
`setup(maxNumberInDropDown, addChooseOption)`	First call; `maxNumberInDropDown` is the item-count threshold deciding drop-down vs. chooser button.
`setupItems(items, valField, labelField, dataField)`	The candidate item array and its field names.
`setupSelectFunction(selectFunction, valField, labelField, dataField)`	The `async` function used for the chooser-button presentation, plus the field names to read from its resolved item.
`setupDefault(defaultValue, defaultLabel, defaultData)`	The initially-selected value.
`run()`	Builds the control, automatically choosing the drop-down, chooser-button, single-value, or nothing-to-select presentation from the item count.

Warning

Kiss ships no CSS for the chooser button. The control emits it with class `btn-smart-chooser` (plus a `:disabled` state), but its appearance — including any icon — is entirely the hosting application's responsibility to style. A page that uses `<smart-chooser>` without ever styling `.btn-smart-chooser` shows a plain, unstyled button.

15.11 List Box

`<list-box>` is a multi-line selectable list. It supports single and multiple selection.

```
<list-box id="employee-list" style="height: 200px;"></list-box>
```

```
$$('employee-list').clear();
for (let emp of employees)
    $$('employee-list').add(emp.id, emp.name);

$$('employee-list').onChange(() => {
    const selected = $$('employee-list').getValue();
    loadEmployeeDetails(selected);
});
```

`getValue()` returns a single value for single-select lists, or an array for multi-select. `onDblClick(fn)` fires when a row is double-clicked. In a multi-select list, `setValue()` accepts an array of values and selects exactly the rows whose values (compared as strings) are in the array, deselecting all others. A single scalar selects just that row, and `null`, `undefined`, or `''` clears the selection.

```
$$('employee-list').setValue([3, 7, 12]); // multiple list
$$('employee-list').setValue(null);     // clear all
```

A list box filters as the user types, exactly as a drop-down does (see Section 15.8), and because the list is always open the rows visibly disappear with each keystroke. Matching is case-insensitive and anywhere in the label. In a single-select list the current selection is kept when it still matches and otherwise moves to the first match, firing `onChange`; a `multiple` list only narrows what is shown and leaves every selection alone, including selections on rows the filter has hidden. `Backspace` edits the sequence, `Escape` erases it and returns to the selection that was current before typing began, and leaving the control restores the full list keeping whatever is then selected. As with the drop-down, `getValue()`, `size()`, and the index-based methods always report the complete list. A read-only list (`readOnly()`) shows its selection but neither the keyboard nor a click on a row can change it; it can still be scrolled.

15.12 Checkbox

`<check-box>` renders a labeled checkbox.

```
<check-box id="active" label="Active"></check-box>
```

```
let isActive = $$('active').getValue(); // true or false
$$('active').setValue(true);
```

15.13 Radio Button

`<radio-button>` groups are managed by a shared `name` attribute. Access the group through the group name with `$$()`:

```
<radio-button name="emp-type" value="full">
    Full-time</radio-button>
<radio-button name="emp-type" value="part">
    Part-time</radio-button>
<radio-button name="emp-type" value="contract">
    Contract</radio-button>
```

```
// Access the group
let type = $$('emp-type').getValue(); // "full", "part", or "
contract"
$$('emp-type').setValue("part");
$$('emp-type').onChange(() => handleTypeChange());
$$('emp-type').getIntValue();        // numeric value if
applicable
```

15.14 Push Button

`<push-button>` renders a clickable button.

```
<push-button id="save">Save</push-button>
<push-button id="cancel">Cancel</push-button>
```

```
$$('save').onclick(async () => {
    if ($$('name').isError('Name'))
        return;
    const res = await Server.call(WS, 'save', data);
    if (res._Success)
        Utils.showMessage('Info', 'Saved successfully');
});
```

Buttons can be enabled and disabled:

```
$$('edit').disable();                // disable
$$('edit').enable();                 // enable
// enable only if row selected
$$('edit').enable(grid.getSelectedRow());
```

15.15 Text Label

`<text-label>` displays static text that can be updated programmatically.

```
<text-label id="status"></text-label>
```

```
$$('status').setValue("Ready");
$$('status').setHTMLValue("<b>Processing...</b>");
$$('status').setColor("red");
$$('status').onclick(() => showDetails());
```

15.16 Textbox Input

`<textbox-input>` provides a multi-line text area.

```
<textbox-input id="notes" style="width: 400px; height: 150px;">
</textbox-input>
```

```
$$('notes').getValue();              // plain text
$$('notes').setHtmlValue("<p>Hello</p>"); // set HTML content
```

15.17 File Upload

`<file-upload>` provides a file picker with upload capability.

```
<file-upload id="attachment"></file-upload>
```

```
// Check if files were selected
let count = $$('attachment').numberOfUploadFiles();

// Get FormData for upload
let fd = $$('attachment').getFormData();
```

```
// Get filename
let name = $$('attachment').uploadFilename(0);

// Upload to backend
Server.fileUploadSend(WS, 'upload', fd);
```

See Chapter 23 for complete file upload examples.

15.18 Picture

`<picture>` displays an image that can be loaded from the backend.

```
<picture id="photo"
        style="width: 200px; height: 200px;"></picture>
```

```
$$('photo').setValue(filename, imageData);
$$('photo').onclick(() => enlargePhoto());
```

15.19 Additional UI Components

The components in this section build page structure, present status, drive navigation, and give the user feedback. Unlike the form controls above, they do not participate in `isError()` validation, but they are declared as custom tags and accessed through `$$()` in exactly the same way as the rest of the library. The full API for each is also documented in `components.js`.

15.19.1 Search Input

`<search-input>` is a single-line search field with a built-in search icon, a clear (`x`) button that appears once text is entered, and an optional drop-down result list for type-ahead / autocomplete behavior.

```
<search-input id="customer-search"
             placeholder="Search customers..."></search-input>
```

Supported attributes:

Attribute	Description
`placeholder`	Placeholder text shown when empty.
`value`	Initial value.
`maxlength`	Maximum number of characters.
`disabled`	Disables the field.
`required`	Marks the underlying input as required.
`clear-label`	Accessible label and tooltip for the clear button (default `Clear search`).
`autocomplete`	Passed through to the input (default `off`).
`spellcheck`	Passed through to the input (default `false`).

Key methods:

Method	Description
`getValue()` / `setValue(val)`	Read or set the search text.
`clear()`	Empty the field and hide the result list.
`focus()`	Focus the search field.
`oninput(fn)`	Called on every keystroke with the current text (`onChange` is an alias).
`onsearch(fn)`	Called with the text when the user presses Enter.
`onclear(fn)`	Called when the clear button is pressed.
`setResults(items [, opts])`	Populate and show the result list.
`onselect(fn)`	Called with `(value, label, item, element)` when a result is chosen.
`showResults()` / `hideResults()`	Show or hide the result list.
`enable([flg])` / `disable([flg])`	Enable or disable the field.

`setResults()` accepts an array of strings or of objects. For

objects, the field names default to `value`, `label`, `description`, and `meta`, and can be overridden through the options object (`valueField`, `labelField`, `descriptionField`, `metaField`). An item with `disabled: true` is shown but not selectable.

```
$$('customer-search').oninput(async (text) => {
    if (text.length < 2) {
        $$('customer-search').hideResults();
        return;
    }
    const res = await Server.callQuiet(WS, 'searchCustomers',
                                       {q: text});
    if (res._Success)
        $$('customer-search').setResults(res.customers, {
            valueField: 'id',
            labelField: 'name',
            descriptionField: 'city'
        });
});

$$('customer-search').onselect((value, label) => {
    loadCustomer(value);
});
```

The list is keyboard-navigable: the down arrow moves from the field into the list, the up and down arrows move between results, Enter selects, and Escape (or a click outside the control) closes it.

Tip

Pair `<search-input>` with `Server.callQuiet()` (Chapter 19) for type-ahead lookups. Because `callQuiet()` does not raise the global wait cursor, rapid keystroke-driven searches do not make the whole screen flicker as busy.

15.19.2 Segmented Control

`<segmented-control>` is a horizontal group of mutually exclusive buttons — the equivalent of a compact tab bar, or of a radio group styled as a toolbar. Each choice is declared with a child `<segment>` tag.

```
<segmented-control id="view-mode" value="list" equal>
    <segment value="list">List</segment>
    <segment value="grid">Grid</segment>
```

```
    <segment value="map">Map</segment>
</segmented-control>
```

Supported attributes:

Attribute	Description
`value`	Value of the initially selected segment.
`equal`	All segments share equal width.
`disabled`	Disables the whole control.

Each child `<segment>` supports a `value`, a `selected` (or `active`) flag, and `disabled`. Key methods:

Method	Description
`getValue()`	Value of the selected segment.
`setValue(val [, silent])`	Select a segment.
`onchange(fn)`	Called with `(value, label, element)` when the selection changes.
`fill(items, sel [, valField, labelField])`	Rebuild the segments from an array of strings or objects, selecting `sel`.
`enable([flg])` / `disable([flg])`	Enable or disable all segments.

Note

`setValue()` is *silent* by default — it changes the selection without firing the `onchange` handler. Pass `false` as the second argument (`setValue(val, false)`) when you want programmatic selection to fire `onchange`, as it does for user clicks. The arrow keys, Home, and End move and select from the keyboard.

15.19.3 Menu Button

`<menu-button>` is a button that opens a small drop-down menu of actions. Menu contents are declared with child `<menu-item>`

and `<menu-separator>` tags; an optional `<menu-trigger>` supplies custom trigger content in place of a plain text label.

```
<menu-button id="row-actions" label="Actions">
    <menu-item value="edit">Edit</menu-item>
    <menu-item value="dup">Duplicate</menu-item>
    <menu-separator></menu-separator>
    <menu-item value="del" danger>Delete</menu-item>
</menu-button>
```

Supported attributes:

Attribute	Description
`label`	Trigger text when no `<menu-trigger>` is supplied (`text` is an alias).
`align`	Menu alignment: `start` or `end`.
`placement`	Menu placement: `bottom` or `top`.
`disabled`	Disables the trigger.
`close-on-select`	Default `true`; set `false` to leave the menu open after a choice.
`open`	Start with the menu open.

Each `<menu-item>` supports a `value` (or `id`), `disabled`, `danger` (destructive styling), and `href` / `target` (navigates when selected). Key methods:

Method	Description
`open()` / `close()` / `toggle()`	Control menu visibility.
`isOpen()`	Whether the menu is open.
`onselect(fn)`	Called with `(value, label, itemElement)` when an item is chosen.
`getValue()` / `setValue(val)`	Read or store the last selected value.
`enable([flg])` / `disable([flg])`	Enable or disable the trigger.
`focus()`	Focus the trigger.

```
$$('row-actions').onselect((value) => {
    switch (value) {
        case 'edit': editRow();   break;
        case 'dup': duplicateRow(); break;
        case 'del': deleteRow();  break;
    }
});
```

The menu is fully keyboard-operable: the trigger opens on Enter, Space, or the down arrow (focusing the first item); the up and down arrows, Home, and End move between items; and Escape closes the menu and returns focus to the trigger. Clicking outside the menu closes it.

15.19.4 Accordion

`<accordion>` is a group of collapsible disclosure sections. Each section is declared with a child `<accordion-item>` tag whose `title` attribute is the clickable header and whose content is the collapsible body.

```
<accordion id="settings" single>
    <accordion-item title="General" value="general" open>
        <!-- section content -->
    </accordion-item>
    <accordion-item title="Notifications" value="notify">
        <!-- section content -->
    </accordion-item>
</accordion>
```

Supported attributes:

Attribute	Description
`single`	Only one section may be open at a time (the default allows several).
`persist-key`	Remembers which sections are open in `localStorage` under this key, restoring them on the next load.

Each `<accordion-item>` supports `title`, `value` (or `id`), `note` (secondary header text), `open` (start expanded), and `disabled`. The header may alternatively be supplied as an `<accordion-title>` child element. Key methods:

Method	Description
`open(value)` / `close(value)` / `toggle(value)`	Control an individual section.
`getOpenValues()`	Array of the values of all open sections.
`setOpen(values [, silent])`	Open exactly the given sections (silent by default).
`onchange(fn)`	Called with `(value, open, openValues)` when a section is toggled.

15.19.5 Panel Card

`<panel-card>` is a generic card / panel surface — a bordered, padded container with a subtle background, used to group related content.

```
<panel-card id="summary" hover>
    <h3>Account summary</h3>
    <p>...</p>
</panel-card>
```

Supported attributes:

Attribute	Description
`as`	The element actually rendered: `div`, `section`, `article`, `aside`, `main`, `header`, or `footer` (default `section`).
`no-pad`	Remove the default interior padding.
`hover`	Raise the card with a shadow on hover.
`interactive`	Make the whole card behave like a button (adds keyboard focus and a button role).

Key methods:

Method	Description
`onclick(fn)`	Register a click handler (used with `interactive`).

Method	Description
`show([flg])` / `hide([flg])`	Show or hide the card.
`isVisible()` / `isHidden()`	Check visibility.

15.19.6 Section Title

`<section-title>` renders a consistent heading row for a region of a screen: an optional small "eyebrow" label, a heading, optional supporting subtitle text, and an action area on the right for buttons.

```
<section-title title="Team members"
               subtitle="People with access to this project"
               eyebrow="Settings">
    <push-button id="invite">Invite</push-button>
</section-title>
```

Supported attributes:

Attribute	Description
`title`	Heading text.
`subtitle`	Secondary text under the heading (`note` is an alias).
`eyebrow`	Small label above the heading.
`level`	Heading level 1–6 (default 2), controlling the underlying `<h1>`–`<h6>`.

Any child content is placed in the action area on the right when title attributes are supplied. If none of `title`, `subtitle`, or `eyebrow` are given, the tag's own text content becomes the title. The methods `setTitle(text)` and `setSubtitle(text)` update the heading and subtitle at runtime.

15.19.7 Badge and Chip

`<badge-chip>` is a small colored label for statuses, counts, categories, and tags.

```
<badge-chip tone="success">Active</badge-chip>
<badge-chip tone="warning" dot>Pending</badge-chip>
<badge-chip tone="info" icon="i">Note</badge-chip>
```

Supported attributes:

Attribute	Description
`tone`	Color: `neutral`, `accent`, `info`, `success`, `warning`, `error`, or `danger` (default `neutral`; `type` is an alias).
`size`	`sm`, `md`, or `lg` (default `md`).
`icon`	Optional leading icon text.
`dot`	Show a small leading status dot.

The label text is the tag's content (or a `value` / `text` attribute). `getValue()` and `setValue(text)` read or replace the label, and `setTone(tone)` changes the color at runtime.

15.19.8 Avatar

`<avatar-badge>` renders a circular avatar from a name — showing derived initials — or from an image, with an optional presence dot.

```
<avatar-badge name="Jane Doe" status="online"></avatar-badge>
<avatar-badge src="/photos/jane.png" name="Jane Doe"
              size="48px"></avatar-badge>
```

Supported attributes:

Attribute	Description
`name`	Used to derive the initials and the accessible label.
`src`	Image URL; when present, the image is shown instead of initials.
`size`	CSS size of the avatar (e.g., `40px`).
`tone`	Background color when showing initials: `neutral`, `accent`, `success`, `warning`, `error`, or `danger`.
`status`	Presence dot: `online`, `away`, `busy`, or `error`.

The initials are derived from the name — the first letters of the first two words, or the first two characters of a single word. `setName(name)` updates the initials and label (clearing

any image), `setSrc(url [, alt])` sets or replaces the image, and `setStatus(status)` updates the presence dot.

15.19.9 Toast Notifications

A toast is a small, non-modal notification that appears briefly and then dismisses itself. Toasts are the lightweight alternative to `Utils.showMessage()` (Chapter 17) for transient feedback such as "Saved" or "Copied to clipboard".

The simplest way to raise a toast needs no markup at all — call `Utils.toast()` (or one of its typed helpers) from anywhere:

```
Utils.toast('Settings saved');
Utils.toast.success('Record created');
Utils.toast.error('Could not reach the server');
Utils.toast.warning('Session about to expire',
                    {duration: 6000});
Utils.toast.info('3 new messages',
                 {position: 'bottom-right'});

const t = Utils.toast.manual('Uploading...'); // stays until
closed
// ... later ...
t.close();

Utils.toast.clear(); // dismiss all visible toasts
```

Each call returns a handle with a `close()` method. The options object accepts:

Option	Description
`type`	`info`, `success`, `warning`, or `error`.
`position`	`top-left`, `top-center`, `top-right`, `bottom-left`, `bottom-center`, or `bottom-right` (default `top-center`). Top positions automatically sit below an `.app-nav` bar when one is present.
`duration`	Milliseconds before auto-dismiss; `0` keeps the toast open until dismissed (default 3000). `time`, `timeout`, and `ms` are accepted as aliases.
`title`	Optional bold title line above the message.

Option	Description
`maxVisible`	Maximum number of toasts shown at once in a position (default 3).
`dismissible`	Whether the close (`x`) button is shown (default `true`).

For a dedicated notification region — to scope toasts to a particular area, or to preconfigure defaults — declare a `<toast>` tag and drive it through `$$()`:

```
<toast id="notices" position="bottom-center"
      type="info" duration="4000" max-visible="2"></toast>
```

```
$$('notices').success('Profile updated');
$$('notices').show('Custom message',
                {type: 'warning', duration: 0});
$$('notices').configure({position: 'top-right'});
$$('notices').clear();
```

The tag supports the same values as attributes: `position`, `type`, `duration` (or `time`), `auto-dismiss` (`false` to keep toasts open), `manual` (the same), `max-visible`, and `no-dismiss` (hide the close button). If the tag has text content, that content is shown as an initial toast. A region exposes `show(message [, options])` (returning a handle with `close()`), the typed helpers `success`, `info`, `warning`, `error`, and `manual`, plus `clear()` and `configure(options)`.

15.20 Stylesheets and Design Tokens

Every component in this chapter is styled by two framework stylesheets, loaded automatically for every Kiss application, in this order (Section 14.1.4):

```
addStylesheet("kiss/starter-theme.css");
addStylesheet("kiss/Utils.css");
```

`kiss/starter-theme.css`
: Opinionated default styling for *bare, unnamespaced* HTML elements — plain `input[type=text]`, `input[type=password]`, `input[type=number]`, `input[type=date]`, `textarea`, and

`label` — so a stock application with no CSS of its own still gets reasonable-looking plain form controls. Because it restyles bare element selectors rather than anything `kiss-`-prefixed, it necessarily also restyles whatever plain markup the hosting application writes itself. An application that owns its own input styling may simply stop linking this file (or override it with a later-loaded application stylesheet) without losing anything else Kiss provides.

`kiss/Utils.css`
: Styling for the component library itself — the form controls earlier in this chapter and the Additional UI Components in Section 15.19. Every selector it declares is `kiss-`-prefixed (e.g. `.kiss-popup-content`, `.kiss-toast`) or scoped under a `kiss-`-prefixed class, and every custom property it declares is `--kiss-`-prefixed. Unlike `starter-theme.css`, it never restyles a bare application element and never squats on an unprefixed custom-property name, so it is namespace-clean — and it is the file an application should *not* stop linking, since it is what makes the component library legible in the first place.

Note

The `<popup>`, `<popup-title>`, and `<popup-body>` tags (Chapter 17) are markup vocabulary, not CSS classes, and are unaffected by the `kiss-` prefix described here. If you ever styled a popup or a message dialog by targeting a class such as `.popup-content`, `.popup-header`, `.popup-body`, or `.msg-modal` directly (instead of the tags), note that these classes now carry a `kiss-` prefix (`.kiss-popup-content`, `.kiss-popup-header`, `.kiss-popup-body`, `.kiss-msg-modal`) — update any such selectors in application CSS.

15.20.1 Design Tokens

`kiss/Utils.css` declares a block of CSS custom properties on `:root` that every control in this chapter reads its colors and surfaces from, instead of hardcoding them:

```
:root {
   --kiss-ui-bg: #ffffff;
```

```
    --kiss-ui-soft-bg: #f6f8fb;
    --kiss-ui-ink: #232832;
    --kiss-ui-muted: #657184;
    --kiss-ui-ring: rgba(24, 32, 48, 0.10);
    --kiss-ui-ring-strong: rgba(24, 32, 48, 0.18);
    --kiss-ui-accent: #2563eb;
    --kiss-ui-accent-soft: #e9f0ff;
    --kiss-ui-success: #177245;
    --kiss-ui-success-soft: #e5f8ed;
    --kiss-ui-warning: #8a5a12;
    --kiss-ui-warning-soft: #fff3d8;
    --kiss-ui-error: #a12b2b;
    --kiss-ui-error-soft: #fdeaea;
    --kiss-ui-radius: 12px;
    --kiss-ui-shadow: ...;
    --kiss-ui-shadow-hover: ...;
}
```

Because `kiss/Utils.css` always loads before any application stylesheet, these tokens are always defined with opaque, legible, light-theme defaults — a brand-new application that supplies zero application-specific CSS still renders every component in this chapter correctly out of the box; no application theme file is required. To re-theme the component library, redefine the tokens you want to change in a stylesheet loaded *after* `kiss/Utils.css`:

```
/* application.css, loaded after kiss/Utils.css */
:root {
    --kiss-ui-accent: #7c3aed;
    --kiss-ui-accent-soft: #f1e9fe;
    --kiss-ui-radius: 6px;
}
```

Every control that reads `var(--kiss-ui-accent)` (and so on for the other tokens) picks up the override immediately — there is no per-component theming configuration.

15.21 Styling Guidelines

Tip

Avoid `width: 100%` on input controls. This can cause layout issues with the component's internal structure. Use specific pixel widths instead.

```
<!-- Correct -->
<text-input id="name" style="width: 250px;"></text-input>

<!-- Avoid -->
<text-input id="name" style="width: 100%;"></text-input>
```

CSS Grid is the recommended layout approach for form fields:

```
<div style="display: grid;
            grid-template-columns: max-content max-content;
            grid-gap: 10px;">
    <span>First Name:</span>
    <text-input id="first-name" style="width: 250px;"
                maxlength="50" required></text-input>

    <span>Last Name:</span>
    <text-input id="last-name" style="width: 250px;"
                maxlength="50" required></text-input>

    <span>Department:</span>
    <drop-down id="department"
               style="width: 260px;"></drop-down>
</div>
```

15.22 Validation Pattern

The standard validation pattern uses `isError()` on each required field before proceeding:

```
$$('save-btn').onclick(async () => {
    if ($$('first-name').isError('First Name'))
        return;
    if ($$('last-name').isError('Last Name'))
        return;
    if ($$('department').isError('Department'))
```

```
        return;

    // All fields valid --- proceed with save
    const data = {
        firstName: $$('first-name').getValue(),
        lastName: $$('last-name').getValue(),
        department: $$('department').getValue()
    };
    const res = await Server.call(WS, 'save', data);
    if (res._Success) {
        Utils.popup_close();
        refreshGrid();
    }
});
```

`isError()` checks the component's validation rules (e.g., `required`, `min`, `max`). If validation fails, it displays an error message using the label you provide and sets focus to the invalid field. It returns `true` if there is an error, `false` if the field is valid.

Chapter 16

Data Grids

Kiss integrates AG-Grid Community Edition for displaying tabular data. The `AGGrid` wrapper class provides a high-level API for grid creation, data management, row selection, and event handling.

16.1 Creating a Grid

A grid requires three things: an HTML container element, column definitions, and an optional key column for row identification:

```
<div id="emp-grid" style="width: 100%; height: 300px;"></div>
```

```
const columnDefs = [
    {headerName: 'Last Name', field: 'lastName', width: 200},
    {headerName: 'First Name', field: 'firstName', width: 200},
    {headerName: 'Active', field: 'active', width: 100}
];

const grid = new AGGrid('emp-grid', columnDefs, 'employeeId');
grid.show();
```

The three constructor parameters are:

1. **Element ID** — the `id` of the container `<div>`.
2. **Column definitions** — an array of column configuration objects.
3. **Key column** (optional) — the field name that uniquely identifies each row. Used by `selectId()`, `deleteRow()`, and selec-

tion tracking.

Warning

Always call `grid.show()` after creating the grid. The grid is not visible until this method is called. When done with the grid (screen unload), the framework handles cleanup automatically.

16.2 Column Configuration

Each column definition is a JavaScript object with these common properties:

Property	Description
`headerName`	Display text in the column header.
`field`	Property name in the data objects (maps to JSON keys).
`width`	Column width in pixels.
`cellRenderer`	Function that returns custom HTML for each cell.
`valueFormatter`	Function that formats the value for display.
`sortable`	Enable column sorting (default: true).
`filter`	Enable column filtering.
`resizable`	Allow user to resize the column.

16.2.1 Custom Cell Renderers

```
{
    headerName: 'Active',
    field: 'active',
    width: 80,
    cellRenderer: function(params) {
        return params.value === 'Y' ? 'Yes' : 'No';
    }
}
```

16.2.2 Built-in Formatters

`AGGrid` provides static formatters for common data types:

```
{headerName: 'Hire Date', field: 'hireDate',
 valueFormatter: AGGrid.date},       // mm/dd/yyyy
{headerName: 'Created', field: 'created',
 // mm/dd/yyyy hh:mm AM/PM
 valueFormatter: AGGrid.dateTime},
{headerName: 'Start', field: 'startTime',
 valueFormatter: AGGrid.time},       // hh:mm AM/PM
{headerName: 'Salary', field: 'salary',
 valueFormatter: AGGrid.numericFormat,
 mask: 'CD', decimalPlaces: 2}       // $12,345.67
```

16.3 Loading Data

```
async function loadGrid() {
    grid.clear();
    const res = await Server.call(WS, 'list');
    if (res._Success)
        grid.addRecords(res.employees);
}
```

Data management methods:

Method	Description
`addRecords(data)`	Add an array of records to the grid.
`addRecord(data)`	Add a single record.
`setRowData(data)`	Replace all rows with new data.
`clear()`	Remove all rows.
`deleteRow(id)`	Delete the row where the key column equals `id`.
`deleteSelectedRows()`	Delete all currently selected rows.
`updateSelectedRecord(row)`	Replace the selected row's data.

16.4 Row Selection

16.4.1 Selection Events

```
grid.setOnSelectionChanged((rows) => {
    // rows is an array of selected row data objects
```

```
    $$('edit').enable(rows);  // enable if rows selected
    $$('delete').enable(rows);
});

grid.setOnRowDoubleClicked((row) => {
    editRecord(row);
});
```

16.4.2 Reading Selection

```
// single selected row or null
let row = grid.getSelectedRow();
// array of all selected rows
let rows = grid.getSelectedRows();
let count = grid.numberOfSelectedRows();
let idx = grid.getSelectedRowIndex(); // index of selected row
```

16.4.3 Programmatic Selection

```
grid.selectId(employeeId); // select by key column value
grid.deselectAll();        // clear selection
// clear selection from selected rows
grid.clearSelection();
```

16.4.4 Multi-Select

```
grid.multiSelect(); // call before grid.show()
grid.show();
```

16.4.5 Disabling Selection

```
grid.noRowSelection(); // call before grid.show()
```

16.5 Querying Grid Data

```
let allRows = grid.getAllRows();
let first = grid.getFirstRow();
let last = grid.getLastRow();
let row = grid.getRowAtIndex(5);
let count = grid.getNumberOfRows();
let empty = grid.isEmpty();
```

16.6 Row Styling

Apply custom styles to rows based on data:

```
grid.rowStyleFunction((params) => {
    if (params.data.active === 'N')
        return { color: 'gray', fontStyle: 'italic' };
    return null; // default styling
});
```

16.6.1 Row Highlighting

```
grid.highlightRows(3);       // highlight row at index 3
grid.highlightRows([1, 3, 5]); // highlight multiple rows
grid.highlightRows(null);    // remove all highlights
```

16.7 Column Visibility

```
// Hide a column (returns previous visibility state)
let wasVisible = grid.columnShow('salary', false);

// Show a column
grid.columnShow('salary', true);

// Resize columns to fit container
grid.sizeColumnsToFit();
grid.resizeColumns();
```

16.8 Drag and Drop

```
grid.setDragFunction((draggedRow, targetRow) => {
    // reorder records based on drag operation
    reorderRecords(draggedRow.id, targetRow.id);
});
```

16.9 Server-Side Sorting

When a grid shows only part of a result — the first 200 rows of a capped search, or one page of many — sorting in the browser is misleading. It reorders only the rows already on screen, not the rows the user actually wants first. The real sort must happen in SQL, before the limit is applied. Two `AGGrid` methods support this, and the back-end counterpart is `SortBuilder` (Section 10.17).

The division of labor is:

1. The user clicks a column header. The grid does *not* sort; it calls your function with the column's `colId`.
2. Your screen decides the sort policy (which column is primary, which direction) and sends the sort keys to the server.
3. The server maps the keys to fixed SQL with `SortBuilder` and returns the rows already in order.
4. Your screen loads the rows and calls `setSortIndicators()` so the header arrows show what the data is sorted by.

16.9.1 Example: Primary and Secondary Sort

The policy in this example: clicking a column makes it the primary sort, ascending; clicking the primary column again flips its direction; the previous primary becomes the secondary sort. Only two levels are kept.

```
const WS = 'services/OrderService';
const grid = new AGGrid('order-grid', [
    {headerName: 'Customer', field: 'customer',
     width: 220},
    {headerName: 'Placed', field: 'placed',
     valueFormatter: AGGrid.date, width: 110},
    {headerName: 'Status', field: 'status',
     width: 110},
```

```
    {headerName: 'Notes', field: 'notes',
     sortable: false, width: 240}
], 'orderId');
grid.show();

// field is the column's colId; asc is true/false
let sorts = [{field: 'customer', asc: true}];

grid.setServerSort(async (colId) => {
    if (sorts[0].field === colId)
        sorts[0] = {field: colId, asc: !sorts[0].asc};
    else
        sorts = [{field: colId, asc: true},
                 sorts[0]];
    await loadGrid();
});

async function loadGrid() {
    const res = await Server.call(WS, 'search', {
        sort1: sorts[0].field, asc1: sorts[0].asc,
        sort2: sorts[1] ? sorts[1].field : '',
        asc2: sorts[1] ? sorts[1].asc : true
    });
    if (!res._Success)
        return;
    grid.clear();
    grid.addRecords(res.orders);
    grid.setSortIndicators(sorts);
}
loadGrid();
```

The server side of the same example is shown in Section 10.17.5.

16.9.2 setServerSort(fn)

`grid.setServerSort(fn)` turns server-side sorting on. Afterwards a header click (or pressing Enter on a focused header) calls `fn(colId)` and does nothing else. The `colId` is the column's `field`, unless the column definition specifies an explicit `colId`.

- **Rows never move.** The grid never reorders rows itself. They always display in the order they were added (`addRecords()`, `setRowData()`, and so on). The application owns the policy.
- **Which columns are clickable.** Every column is clickable except those whose column definition says `sortable: false`

(and column groups). Clicking an excluded column does nothing. The caller's column definitions are never modified.

- **Timing.** It may be called before or after `show()`.
- **Turning it off.** A falsy argument (`grid.setServerSort(null)`) turns server sorting off and clears any indicators.
- **No effect on other grids.** A grid that never calls the method behaves exactly as before.

The method returns the grid, so calls can be chained.

16.9.3 setSortIndicators(list)

`grid.setSortIndicators(list)` shows the standard AG-Grid sort arrows on the column headers. It never sorts or reorders rows and never calls the function given to `setServerSort()`.

- `list` is an array of `{field, asc}` objects in priority order; the first is the primary sort. `field` is a column's `colId` and `asc` is true for ascending, false for descending.
- **Numbered indicators.** When more than one column is listed, AG-Grid also shows each column's priority number beside its arrow.
- A `field` that has no column, or whose column is hidden, is silently ignored.
- `null` or an empty array clears all indicators.
- It may be called before or after `show()`, and it returns the grid.

Note

`setSortIndicators()` only displays state; it does not remember it. Call it after each reload with the list your screen holds, as the example does.

16.10 Enable/Disable

```
grid.disable();   // disable clicks on grid
grid.enable();    // re-enable clicks
grid.disable(false); // same as enable()
```

16.11 Cleanup

Grids are automatically cleaned up when screens are unloaded via the context management system. For manual cleanup:

```
grid.destroy(); // free all internal data structures
```

16.12 Complete Grid Example

```
'use strict';

(async function () {
    const WS = 'services/EmployeeService';

    const grid = new AGGrid('emp-grid', [
        {headerName: 'Last Name', field: 'lastName',
            width: 200},
        {headerName: 'First Name', field: 'firstName',
            width: 200},
        {headerName: 'Department', field: 'department',
            width: 200},
        {headerName: 'Hire Date', field: 'hireDate',
         valueFormatter: AGGrid.date, width: 120},
        {headerName: 'Active', field: 'active', width: 80,
         cellRenderer: (p) => p.value === 'Y' ? 'Yes' : 'No'}
    ], 'employeeId');
    grid.show();

    async function loadGrid() {
        grid.clear();
        $$('edit-btn').disable();
        $$('delete-btn').disable();
        const res = await Server.call(WS, 'list');
        if (res._Success)
            grid.addRecords(res.employees);
    }
```

```
    loadGrid();

    grid.setOnSelectionChanged((rows) => {
        $$('edit-btn').enable(rows);
        $$('delete-btn').enable(rows);
    });

    grid.setOnRowDoubleClicked(() => editEmployee());

    function editEmployee() {
        const row = grid.getSelectedRow();
        // open edit popup with row data...
    }

    $$('edit-btn').onclick(() => editEmployee());
})();
```

Chapter 17

Popups and Dialogs

Kiss provides a popup system for modal dialogs and several built-in dialog types for messages and confirmations. Popups are the primary mechanism for editing records selected from a grid.

17.1 The Popup Component

A popup is defined in the screen's HTML using the `<popup>` tag. It requires exactly two direct children: `<popup-title>` and `<popup-body>`.

```
<popup id="edit-popup" height="200px" width="450px">
    <popup-title>Edit Employee</popup-title>
    <popup-body>
        <div style="display: grid;
                    grid-template-columns: auto 1fr;
                    gap: 8px; padding: 10px;">
            <span>First Name:</span>
            <text-input id="ep-first" required
                        style="width: 250px;"></text-input>
            <span>Last Name:</span>
            <text-input id="ep-last" required
                        style="width: 250px;"></text-input>
        </div>
        <div style="text-align: right; margin-top: 10px;">
            <push-button id="ep-ok">OK</push-button>
            <push-button id="ep-cancel">Cancel</push-button>
        </div>
    </popup-body>
</popup>
```

Warning

The popup *must* have exactly two direct children: `<popup-title>` and `<popup-body>`. Using a single wrapper `<div>` instead causes a "Cannot set properties of undefined" error at runtime.

```
<!-- WRONG: causes error -->
<popup id="bad-popup">
    <div class="content">...</div>
</popup>

<!-- CORRECT -->
<popup id="good-popup">
    <popup-title>Title</popup-title>
    <popup-body>Content</popup-body>
</popup>
```

17.2 Opening and Closing Popups

```
// Open popup, set focus to a control
Utils.popup_open('edit-popup', 'ep-first');

// Close popup
Utils.popup_close();
```

`Utils.popup_open()` accepts two parameters:

1. **Popup ID** — the `id` of the `<popup>` element.
2. **Focus control** (optional) — the `id` of a component that should receive focus when the popup opens.

`Utils.popup_close()` closes the most recently opened popup. You can optionally pass the popup ID for verification.

17.2.1 Popup Stacking

Multiple popups can be open simultaneously. The framework manages a stack of open popups and increments the z-index for each new popup. Closing a popup pops the stack and restores the previous popup's state.

17.2.2 DOM Preservation After Close

When a popup is closed, its DOM elements are *hidden*, not destroyed. This means:

- All controls within the popup remain accessible.
- Values can be read from popup controls after closing.
- Control references (`$$('popup-control-id')`) continue to work.
- The popup can be reopened without re-initializing its controls.

```
Utils.popup_open('edit-popup');
// ... user interacts with popup ...
Utils.popup_close();

// Controls are still accessible after close
let name = $$('ep-first').getValue(); // works!
```

17.3 Popup Sizing

Specify both `height` and `width` attributes on the `<popup>` tag:

```
<popup id="my-popup" height="300px" width="500px">
```

Size the popup appropriately for its content. Avoid excess whitespace, and test on different screen sizes.

Dynamic sizing is also available:

```
Utils.popup_set_width('my-popup', '600px');
Utils.popup_set_height('my-popup', '400px');
```

17.4 The Standard Edit Pattern

The most common popup use case is editing a record selected from a grid. The pattern uses one popup for both adding and editing:

```
// New record
$$('new-btn').onclick(() => {
```

```
    $$('ep-title').setValue('Add Employee');
    $$('ep-first').clear();
    $$('ep-last').clear();
    Utils.popup_open('edit-popup', 'ep-first');

    $$('ep-ok').onclick(async () => {
        if ($$('ep-first').isError('First Name')) return;
        if ($$('ep-last').isError('Last Name')) return;
        const res = await Server.call(WS, 'create', {
            firstName: $$('ep-first').getValue(),
            lastName: $$('ep-last').getValue()
        });
        if (res._Success) {
            Utils.popup_close();
            loadGrid();
        }
    });

    $$('ep-cancel').onclick(() => Utils.popup_close());
});

// Edit selected record
$$('edit-btn').onclick(() => {
    const row = grid.getSelectedRow();
    $$('ep-title').setValue('Edit Employee');
    $$('ep-first').setValue(row.firstName);
    $$('ep-last').setValue(row.lastName);
    Utils.popup_open('edit-popup', 'ep-first');

    $$('ep-ok').onclick(async () => {
        if ($$('ep-first').isError('First Name')) return;
        if ($$('ep-last').isError('Last Name')) return;
        const res = await Server.call(WS, 'save', {
            id: row.id,
            firstName: $$('ep-first').getValue(),
            lastName: $$('ep-last').getValue()
        });
        if (res._Success) {
            Utils.popup_close();
            loadGrid();
        }
    });

    $$('ep-cancel').onclick(() => Utils.popup_close());
});
```

Tip

Use a `<text-label>` for the popup title so it can be changed programmatically between "Add" and "Edit" modes.

17.5 Message Dialogs

`Utils.showMessage()` displays an informational dialog with an OK button:

```
Utils.showMessage('Info', 'Record saved successfully.');
Utils.showMessage('Error', 'Failed to connect to server.');
```

The dialog is modal and draggable. The title bar is colored based on the title text: red for "Error", blue otherwise. The method returns a Promise that resolves when the user clicks OK.

```
await Utils.showMessage('Info', 'Please review the changes.');
// execution continues after user clicks OK
```

17.6 Confirmation Dialogs

`Utils.yesNo()` displays a Yes/No confirmation dialog:

```
// Callback style
Utils.yesNo('Confirm', 'Delete this record?',
    function () {
        // user clicked Yes
        deleteRecord();
    },
    function () {
        // user clicked No (optional)
    }
);

// Promise style
const answer = await Utils.yesNo('Confirm', 'Delete?');
if (answer) {
    deleteRecord();
}
```

The dialog is modal, draggable, and mobile-responsive (adjusts width for small screens).

17.7 Wait Messages

For long-running operations, display a non-modal wait message:

```
Utils.waitMessage('Processing...');
// ... long operation ...
Utils.waitMessageEnd();
```

Wait messages stack: calling `waitMessage()` while one is already shown replaces the message text; calling `waitMessageEnd()` restores the previous message.

Note

`Server.call()` automatically shows a wait cursor during requests, so you only need `waitMessage()` for custom long operations or when you want a specific message.

Chapter 18

Frontend Utility Classes

Kiss provides a suite of JavaScript utility classes for date/time handling, number formatting, DOM manipulation, and general-purpose operations. This chapter covers each class with examples.

18.1 DateUtils

`DateUtils` handles dates as YYYYMMDD integers — the same format used by the Kiss backend. This avoids timezone ambiguity and makes date arithmetic simple.

```
let today = DateUtils.today();          // e.g., 20260404
let formatted = DateUtils.intToStr4(today); // "04/04/2026"
let short = DateUtils.intToStr2(today); // "04/04/26"
let sql = DateUtils.intToSQL(today);    // "2026-04-04"
// "Sat April 4, 2026"
let long = DateUtils.longFormat(today);
```

18.1.1 Conversion

```
// String to integer
let dt = DateUtils.strToInt("04/04/2026"); // 20260404
let dt2 = DateUtils.SQLtoInt("2026-04-04"); // 20260404

// Universal converter (accepts Date, string, or number)
// today as YYYYMMDD
let dt3 = DateUtils.toInt(new Date());

// Integer to JavaScript Date
```

```
let jsDate = DateUtils.intToDate(20260404);
```

18.1.2 Date Arithmetic

```
let next = DateUtils.intAddDays(20260404, 7); // 20260411
let prev = DateUtils.intAddDays(20260404, -30); // 20260305
let diff = DateUtils.daysDifference(20260404, 20260101); // 93
// 12
let months = DateUtils.monthsDifference(20260404, 20250404);
```

18.1.3 Date Components

```
let y = DateUtils.year(20260404);   // 2026
let m = DateUtils.month(20260404);  // 4
let d = DateUtils.day(20260404);    // 4
let dow = DateUtils.dayOfWeek(20260404); // 6 (Saturday)
let name = DateUtils.dayOfWeekName(20260404); // "Saturday"
let mon = DateUtils.monthName(20260404); // "April"
```

18.1.4 Validation

```
DateUtils.isValid(20260404);    // true
DateUtils.isValid(20261332);    // false (invalid month/day)
DateUtils.isSQLDate("2026-04-04"); // true
```

Note

`DateUtils` is locale-aware. `strToInt()` and `intToStr4()` automatically detect whether the locale uses MM/DD/YYYY or DD/MM/YYYY format.

18.2 TimeUtils

`TimeUtils` handles times as HHMM integers (e.g., 1430 for 2:30 PM).

```
let now = TimeUtils.current();          // e.g., 1430
let ampm = TimeUtils.format(1430);      // "2:30 PM"
```

```
// "02:30 PM" (zero-filled hour)
let zf  = TimeUtils.format(1430, true);
```

18.2.1 Conversion and Parsing

```
// Parse various formats
TimeUtils.strToInt("2:30 PM"); // 1430
TimeUtils.strToInt("14:30");  // 1430
TimeUtils.strToInt("1430");   // 1430

// Universal converter
TimeUtils.toInt(new Date());  // current time as HHMM
```

18.2.2 Time Arithmetic

```
TimeUtils.addHours(1430, 2);   // 1630
TimeUtils.addMinutes(1430, 45); // 1515
TimeUtils.diff(1630, 1430);    // 120 (minutes)
TimeUtils.totalMinutes(1430);  // 870
TimeUtils.minutesToTime(870);  // 1430
```

18.2.3 Time Components

```
TimeUtils.hours(1430); // 14
TimeUtils.minutes(1430); // 30
```

18.3 DateTimeUtils

`DateTimeUtils` handles combined date/time values as YYYYMMDDHHMM integers and provides extensive timezone support.

```
// Current date/time
let now = DateTimeUtils.now(); // e.g., 202604041430

// Create from date + time
let dt = DateTimeUtils.create(20260404, 1430); // 202604041430

// Format for display (locale-aware)
DateTimeUtils.formatDate(202604041430);
```

```
    // "04/04/2026 2:30 PM"
```

18.3.1 Epoch Millisecond Handling

Backend datetime values often arrive as epoch milliseconds. `DateTimeUtils` handles this automatically:

```
// Backend returns epoch millis (e.g., 1737292576000)
let formatted = DateTimeUtils.formatDate(res.createdAt);
    // "01/19/2025 11:36 AM"
```

Warning

Always use `DateTimeUtils.formatDate()` to format backend datetime values. Never manually parse epoch milliseconds with substring operations — that produces garbage like "49/12/1768".

18.3.2 Component Extraction

```
let dt = 202604041430;
DateTimeUtils.getDate(dt); // 20260404 (YYYYMMDD)
DateTimeUtils.getTime(dt); // 1430 (HHMM)
```

18.3.3 DateTime Arithmetic (DST-Aware)

```
let dt = 202604041430;
DateTimeUtils.addDays(dt, 7);   // 202604111430
DateTimeUtils.addHours(dt, 3);  // 202604041730
DateTimeUtils.addMinutes(dt, 90); // 202604041600
DateTimeUtils.addMonths(dt, 1); // 202605041430
DateTimeUtils.addYears(dt, 1);  // 202704041430
```

All arithmetic methods accept an optional timezone parameter and handle DST transitions correctly.

18.3.4 Timezone Support

```
// Get local timezone
DateTimeUtils.localZone();        // "America/New_York"
DateTimeUtils.getLocalTimezoneShortText(); // "EDT"

// Convert between timezones
DateTimeUtils.epochToDisplayEpoch(
    epochMs, 'America/New_York', 'America/Chicago');
```

18.4 NumberUtils

`NumberUtils` provides number comparison and rounding:

```
// Approximate equality (useful for floating-point)
NumberUtils.floatEqual(0.1 + 0.2, 0.3, 0.0001); // true

// Rounding
NumberUtils.round(3.14159, 2);  // 3.14
NumberUtils.roundUp(3.141, 2);  // 3.15
NumberUtils.roundDown(3.149, 2); // 3.14
```

18.5 DOMUtils

`DOMUtils` is the DOM abstraction layer. It provides methods for element access, manipulation, visibility, attributes, events, and dimensions.

Warning

Only code under the `kiss/` directory should use DOMUtils directly. Application code should use the Kiss component APIs (`$$()`, `getValue()`, etc.) instead. This layered architecture ensures consistent DOM manipulation and easier maintenance.

Key DOMUtils categories:

Element access
: `getElement(id)`, `query(selector)`, `queryAll(selector)`

Creation
: `create(tag, attrs)`, `append(parent, html)`

Visibility
`show(el)`, `hide(el)`, `isVisible(el)`

Classes
`addClass(el, cls)`, `removeClass(el, cls)`, `toggleClass(el, cls)`

Attributes
`attr(el, name, val)`, `prop(el, name, val)`, `css(el, prop, val)`

Content
`text(el, text)`, `html(el, html)`, `val(el, value)`

Events
`on(el, event, handler)`, `off(el, event, handler)`

Navigation
`children(el)`, `parent(el)`, `closest(el, selector)`

Dimensions
`width(el)`, `height(el)`, `offset(el)`

Navigation protection
`preventNavigation(isActive, onBack)`

18.5.1 Browser Back Button Prevention

`DOMUtils.preventNavigation()` prevents accidental back-button and page-reload navigation:

```
DOMUtils.preventNavigation(
    function () { return true; }, // active when returns true
    function () {              // called on back button
        Utils.yesNo('Confirm',
            'Are you sure you want to logout?',
            function () { Server.logout(); });
    }
);
```

Note

Chrome blocks navigation prevention until the page has received a user gesture (click, keypress, or touch). Before any interaction, the back button cannot be prevented. After the first click or keypress, full protection activates.

18.6 Utils

`Utils` is the general-purpose utility class. Its most important functions are covered in other chapters:

- `Utils.loadPage()` — Chapter 12
- `Utils.popup_open()` / `popup_close()` — Chapter 17
- `Utils.showMessage()` / `Utils.yesNo()` — Chapter 17
- `$$()` — Chapter 12

Additional utility methods:

Method	Description
`Utils.cleanup()`	Remove event handlers and state from the current screen before loading a new one.
`Utils.showReport(url)`	Open a report URL in a new browser tab.
`Utils.waitMessage(msg)`	Show a non-modal progress message.
`Utils.waitMessageEnd()`	Dismiss the progress message.
`Utils.rescan()`	Activate custom components in newly injected HTML.
`Utils.afterComponentsLoaded(fn)`	Execute a function after all components are loaded.
`Utils.useComponent(name)`	Register a UI component for use.
`Utils.getAppUrl()`	Get the application's base URL.

Method	Description
`Utils.saveData(key, val)`	Save a cross-screen data item by key. Backed by `AppState`, so per the configured store (Chapter 13) it survives page navigation and reload; the value must be JSON-serializable.
`Utils.getData(key)`	Retrieve a cross-screen data item by key.
`Utils.getAndEraseData(key)`	Retrieve a cross-screen data item by key and remove it, returning its prior value.

18.7 MutableString

`MutableString` provides efficient character-level string modifications:

```
let ms = new MutableString("Hello World");
ms.setCharAt(5, '-');       // "Hello-World"
ms.insert(5, " Beautiful"); // "Hello Beautiful-World"
ms.delete(5);               // "Hell Beautiful-World"
ms.append("!");             // "Hell Beautiful-World!"
ms.replace("World", "Day"); // "Hell Beautiful-Day!"
let ch = ms.charAt(0);      // "H"
let len = ms.length();      // 20
let str = ms.toString();    // "Hell Beautiful-Day!"
ms.clear();                 // ""
```

18.8 Editor

`Editor` wraps CKEditor 5 for rich text editing:

```
// Create editor in a container div
const editor = await Editor.create('editor-container');

// Get/set content
let html = editor.getHtml();
let text = editor.getText();
editor.setHtml('<p>Hello <b>World</b></p>');
editor.clear();
```

```
// Read-only mode
editor.readOnly();
editor.readWrite();

// Print editor contents
editor.printEditorContent();

// Cleanup (usually handled by framework)
editor.dispose();
```

Editors are automatically cleaned up when screens unload via the editor context management system, similar to grids.

Chapter 19

Server Communication

All communication between the Kiss frontend and backend uses JSON-RPC via the `Server` class. This chapter covers making calls, handling responses, managing sessions, uploading files, and downloading reports.

19.1 Server.call()

`Server.call()` is the primary communication method. It sends a JSON-RPC request to the backend and returns a Promise with the response:

```
const res = await Server.call("services/Crud", "getRecords", {
    departmentId: 42,
    activeOnly: true
});
if (res._Success) {
    grid.addRecords(res.rows);
}
```

The three required parameters are:

1. **Service path** — the path to the backend service class (e.g., `"services/Crud"`).

2. **Method name** — the method to invoke on the service class.

3. **Input data** — a JavaScript object sent as the `injson` parameter to the backend method. This parameter is optional; omit it if the method requires no input.

`Server.call()` automatically:

- Includes the session UUID in every request.
- Shows a wait cursor while the request is in progress.
- Handles authentication failures (error code 2) by redirecting to login.
- Retries once on transient network errors.
- Displays error messages for server-side exceptions.

19.2 Request/Response Format

Behind the scenes, `Server.call()` sends an HTTP POST to the `/rest` endpoint with a JSON body:

```
// Outgoing request
{
    "_class": "services/Crud",
    "_method": "getRecords",
    "_uuid": "a1b2c3d4-e5f6-...",
    "departmentId": 42,
    "activeOnly": true
}

// Successful response
{
    "_Success": true,
    "_ErrorCode": 0,
    "rows": [ ... ]
}

// Error response
{
    "_Success": false,
    "_ErrorCode": -1,
    "_ErrorMessage": "Department not found"
}
```

The `_Success` field is always present. On error, `_ErrorMessage` contains the error text and `_ErrorCode` indicates the type:

Code	Meaning
0	Success
2	Authentication failure (session expired or invalid)
−1	Server error (exception in service method)

19.3 Error Handling

When the backend throws an exception, `Server.call()` displays the error message to the user automatically. The callback still executes, but `res._Success` is `false`:

```
const res = await Server.call(WS, 'save', data);
if (res._Success) {
    // handle success
    Utils.showMessage('Info', 'Saved successfully');
} else {
    // The error message has already been shown to the user.
    // This branch is optional --- use it only if you need
    // custom error recovery.
}
```

Different backend exception types produce different behaviors:

`UserException`
: The message is displayed to the user. No stack trace is logged on the server.

`ServerException`
: A generic error message is shown. The full stack trace is logged on the server.

`LogException`
: The message is displayed to the user *and* the full stack trace is logged on the server.

19.4 Quiet Calls

`Server.callQuiet()` is a variant of `Server.call()` for lightweight background requests — type-ahead searches, autocomplete lookups, polling, and similar work where the screen should not appear blocked and where a transport failure should be handled silently rather than

interrupting the user. It takes the same arguments as `Server.call()` and returns the same Promise-wrapped response:

```
const res = await Server.callQuiet("services/Customers",
                                   "search", {q: text});
if (res._Success)
    showResults(res.customers);
```

It differs from `Server.call()` in two ways:

- It does *not* change the global wait cursor or busy count, so a rapid sequence of keystroke-driven calls does not make the whole application flicker as busy. The screen owns whatever loading indication it wants to show.
- A transport or JSON-parse failure does *not* raise the framework error popup. Instead the caller receives a `{_Success: false}` result to handle quietly.

In every other respect it behaves like `Server.call()`: the session UUID is attached automatically, and an authentication failure (error code 2) still shows the standard message and routes to the login screen.

Tip

Use `callQuiet()` for background work whose result the user did not explicitly request, and `call()` for actions the user is waiting on (saving a form, running a report). The `<search-input>` component (Chapter 15) is designed to pair with `callQuiet()`.

19.5 Parallel Calls

When a screen needs data from multiple independent services, use `Server.callAll()` to execute them in parallel:

```
const error = await Server.callAll(
    [
        Server.call(WS, 'getDepartments'),
        Server.call(WS, 'getStatuses')
    ],
    function (deptRes) {
```

```
        // process department data
        for (let dept of deptRes.departments)
            $$('department').add(dept.id, dept.name);
    },
    function (statusRes) {
        // process status data
        for (let s of statusRes.statuses)
            $$('status').add(s.id, s.name);
    }
);
if (!error) {
    // all calls succeeded
}
```

`callAll()` waits for all promises to complete before calling any handlers. It returns `true` if any call failed, `false` if all succeeded.

19.6 Session Management

After login, the backend assigns a session token (a UUID). `Server.js` stores this token and includes it automatically in every subsequent request as the `_uuid` field.

The session expires after a configurable period of inactivity (set by `UserInactiveSeconds` in `application.ini`). When a request arrives with an expired UUID, the backend returns error code 2 and `Server.call()` redirects to the login screen.

19.6.1 Inactivity Timeout (Frontend)

The frontend can also enforce an inactivity timeout independently:

```
Server.setMaxInactivitySeconds(900); // 15 minutes
Server.setMaxInactivityMinutes(15); // equivalent
Server.setMaxInactivityHours(1);   // 1 hour
```

The frontend timeout runs a `checkTime()` method that compares the elapsed time since the last user interaction. If the timeout is exceeded, the user is logged out.

19.7 Logging Out

```
Server.logout();
```

`Server.logout()` performs a complete cleanup:

1. Calls the backend `Logout` service to invalidate the session.
2. Clears the stored UUID (and all other persisted client state).
3. Disables back-button protection.
4. Calls `Utils.cleanup()` to release frontend resources.
5. Performs a full page reload of a fresh `index.html`, landing on the login route (`Router.loginHash()`).

The reload in the final step re-boots the tab on the latest cache-busted code, so a tab that stayed open across a redeploy picks up the new release the next time its session ends (Chapter 14).

19.8 File Uploads

File uploads use `Server.fileUploadSend()`, which sends files as multipart/form-data:

```
$$('upload-btn').onclick(async () => {
    const count = $$('file-ctl').numberOfUploadFiles();
    if (count === 0) {
        Utils.showMessage('Error', 'Please select a file');
        return;
    }
    const fd = $$('file-ctl').getFormData();
    const res = await Server.fileUploadSend(
        WS, 'upload', fd, {},
        'Uploading...', 'Upload complete');
});
```

The parameters are:

1. **Service path** and **method name** — same as `Server.call()`.
2. **FormData** — from the `<file-upload>` component's `getFormData()`.
3. **Additional data** — a JavaScript object with extra fields to send alongside the files.
4. **Wait message** — shown during upload.

5. **Success message** — shown after successful upload.

On the backend, the service method accesses uploaded files through the `servlet` parameter:

```
int count = servlet.getUploadFileCount()
String name = servlet.getUploadFileName(0)
BufferedInputStream is =
    servlet.getUploadBufferedInputStream(0)
```

19.9 File Downloads

For reports and exports, the backend generates a temporary file and returns its URL. The frontend opens the URL to trigger the browser's download or display behavior:

```
// Backend generates a PDF report and returns the URL
const res = await Server.call(WS, 'runReport');
if (res._Success)
    Utils.showReport(res.reportUrl);
```

```
// Backend service method
Groff rpt = new Groff("Report-", "My Report", false)
// ... build report ...
outjson.put("reportUrl", rpt.process())
```

Temporary report files are automatically cleaned up by the framework after approximately one day.

19.10 Binary Data

For retrieving binary data (e.g., images), use `Server.binaryCall()`:

```
const res = await Server.binaryCall(WS, 'getImage',
    { id: imageId });
if (res._Success) {
    // res._data contains the binary content as an ArrayBuffer
    const blob = new Blob([res._data]);
    const url = URL.createObjectURL(blob);
    // use the URL in an img tag or picture component
}
```

`binaryCall()` returns the response as an ArrayBuffer. JSON metadata and binary content are separated by a delimiter character, and the parsing is handled automatically.

19.11 A Complete Screen Example

The following example ties together everything in this chapter: loading data, populating controls, saving with validation, and refreshing:

```
'use strict';

(async function () {
    const WS = 'services/Crud';

    // --- Grid setup ---
    const columnDefs = [
        {headerName: 'Last Name', field: 'lastName',
            width: 33},
        {headerName: 'First Name', field: 'firstName',
            width: 33},
        {headerName: 'Phone', field: 'phoneNumber', width: 33}
    ];
    const grid = new AGGrid('grid', columnDefs, 'id');
    grid.show();

    // --- Load data ---
    async function updateGrid() {
        grid.clear();
        $$('edit').disable();
        $$('delete').disable();
        const res = await Server.call(WS, 'getRecords');
        if (res._Success)
            grid.addRecords(res.rows);
    }
    updateGrid();

    // --- Grid selection ---
    grid.setOnSelectionChanged((rows) => {
        $$('edit').enable(rows);
        $$('delete').enable(rows);
    });

    // --- Add new record ---
    $$('new').onclick(() => {
        $$('erp-first-name').clear();
        $$('erp-last-name').clear();
        $$('erp-phone-number').clear();
```

```
        Utils.popup_open('edit-record-popup',
            'erp-first-name');

        $$('erp-ok').onclick(async () => {
            if ($$('erp-first-name').isError('First Name'))
                return;
            if ($$('erp-last-name').isError('Last Name'))
                return;
            const data = {
                firstName: $$('erp-first-name').getValue(),
                lastName: $$('erp-last-name').getValue(),
                phoneNumber: $$('erp-phone-number').getValue()
            };
            const res = await Server.call(WS, 'addRecord',
                data);
            if (res._Success) {
                Utils.popup_close();
                updateGrid();
            }
        });

        $$('erp-cancel').onclick(() => Utils.popup_close());
    });

    // --- Delete with confirmation ---
    $$('delete').onclick(() => {
        Utils.yesNo('Confirm',
            'Delete the selected record?', async () => {
            const row = grid.getSelectedRow();
            const res = await Server.call(WS, 'deleteRecord',
                { id: row.id });
            if (res._Success) updateGrid();
        });
    });

    // --- PDF report ---
    $$('report').onclick(async () => {
        const res = await Server.call(WS, 'runReport');
        if (res._Success)
            Utils.showReport(res.reportUrl);
    });

    // --- CSV export ---
    $$('export').onclick(async () => {
        const res = await Server.call(WS, 'runExport');
        if (res._Success)
            Utils.showReport(res.exportUrl);
    });
})();
```

This is adapted from the actual CRUD example that ships with Kiss. It demonstrates the standard patterns: load data into a grid, enable/disable buttons based on selection, open popups for editing, validate before saving, and confirm before deleting.

Part IV

Application Features

Chapter 20

Authentication and Sessions

Kiss includes a built-in authentication system with session management, configurable inactivity timeout, and browser navigation protection. This chapter covers the login flow, session lifecycle, and customization points.

20.1 The Login Flow

Authentication follows a fixed sequence:

1. The browser submits a username and password to the `Login` service.

2. `Login.groovy` validates the credentials against the `users` table.

3. On success, `UserCache.newUser()` creates a `UserData` object containing a randomly generated UUID session token.

4. The UUID is returned to the browser. `Server.js` stores it and includes it in every subsequent request as the `_uuid` field.

5. On every request, `ProcessServlet` looks up the UUID in the `UserCache`. If the UUID is missing or the session has expired, the request is rejected with error code 2 and the browser redirects to the login screen.

20.2 The Login Service

The `Login.groovy` file in `src/main/backend/` implements two required methods:

```
class Login {
    static UserData login(Connection db, String user,
                          String password, JSONObject outjson,
                          ProcessServlet servlet) {
        Record rec = db.fetchOne(
            "select user_id, user_password from users " +
            "where user_name = ? and user_active = 'Y'", user)
        if (rec == null)
            return null // invalid user

        String pw = rec.getString("user_password")
        if (pw == null || !PasswordHash.verify(password, pw))
            return null

        return UserCache.newUser(user, password,
            (Integer) rec.getInt("user_id"))
    }

    static Boolean checkLogin(Connection db, UserData ud,
                              ProcessServlet servlet) {
        Record rec = db.fetchOne(
            "select user_password from users " +
            "where user_name = ? and user_active = 'Y'",
            ud.getUsername())
        if (rec == null)
            return false

        String pw = rec.getString("user_password")
        return pw != null && PasswordHash.verify(
            ud.getPassword(), pw)
    }
}
```

`login()`
: Called when the user submits credentials. Returns a `UserData` object on success, `null` on failure. You can add data to `outjson` to pass additional information (e.g., user roles) to the frontend.

`checkLogin()`
: Called periodically (approximately every 2 minutes) to revalidate the user's session. This catches cases where a user is

deactivated or their password changes while they are logged in.

Note

Passwords are stored as one-way salted PBKDF2 hashes using `PasswordHash` (§29.8); `login()` verifies a submitted password against the stored hash with `PasswordHash.verify()`. Because the encoded hash is approximately 80 characters long, the `user_password` column must be at least `varchar(255)`.

Because the stored hash embeds its own iteration count (§29.8), `login()` is also the natural place to opportunistically upgrade older hashes to the framework's current work factor. `PasswordHash.needsRehash(stored)` returns `true` when a stored hash was produced with a lower iteration count than the current default; the standard pattern is to check it immediately after a successful `verify()` and re-hash the just-verified password when it does:

```
String pw = rec.getString("user_password")
if (pw == null || !PasswordHash.verify(password, pw))
    return null

if (PasswordHash.needsRehash(pw)) {
    rec.set("user_password", PasswordHash.hash(password))
    rec.update()  // upgrade the work factor in place
}

return UserCache.newUser(user, password,
    (Integer) rec.getInt("user_id"))
```

This lets every stored password gradually migrate to the current work factor as users log in successfully, with no bulk migration required.

20.3 KissInit.groovy

The `KissInit.groovy` file in `src/main/backend/` runs at system startup. It has two initialization phases:

```
class KissInit {
    static void init() {
        // Phase 1: runs before database is open
        MainServlet.readIniFile "application.ini", "main"
```

```
        // Whitelist methods that don't require authentication
        // MainServlet.allowWithoutAuthentication(
        //     "services.PublicService", "getData")

        // Global logout handler
        // UserCache.setLogoutHandler({ UserData ud ->
        //     // Custom cleanup: close resources,
        //     // update timestamps
        // } as Consumer<UserData>)
    }

    static void init2(Connection db) {
        // Phase 2: runs after database is open
        // Initialize application-specific state
        // If you use db, make sure you commit
    }
}
```

`init()`
: Runs before the database connection pool is created. Use it to load configuration and set up authentication rules.

`init2(Connection db)`
: Runs after the database is available. Use it for one-time data initialization. If you modify the database, call `db.commit()`.

20.4 Allowing Unauthenticated Access

By default, every service method requires authentication. To exempt specific methods (e.g., a public API endpoint), call `allowWithoutAuthentication()` in `KissInit.init()`:

```
MainServlet.allowWithoutAuthentication(
    "services.PublicService", "getStatus")
MainServlet.allowWithoutAuthentication(
    "services.PublicService", "getVersion")
```

Whitelisted methods still receive a `Connection` and can access `UserData` if the caller happens to be logged in, but they do not require it.

20.5 Session Management

20.5.1 UserData

`UserData` holds per-user session information:

```
// In a service method
UserData ud = servlet.getUserData()
String username = ud.getUsername()
Object userId = ud.getUserId()
String uuid = ud.getUuid()

// Store custom data in the session
ud.putUserData("role", "admin")
String role = (String) ud.getUserData("role")
```

`UserData` provides:

- Immutable credentials: `getUsername()`, `getPassword()`, `getUserId()`.
- A unique session identifier: `getUuid()`.
- A thread-safe data map for application-specific session data: `putUserData()`, `getUserData()`.
- Last-access tracking for inactivity timeout: `getLastAccessDate()`.

20.5.2 UserCache

`UserCache` manages all active sessions:

- `UserCache.newUser(user, password, userId)` — creates a session and returns `UserData`.
- `UserCache.findUser(uuid)` — looks up a session by UUID.
- `UserCache.removeUser(uuid)` — invalidates a session (triggers the logout handler).

20.5.3 Session Expiration

Sessions expire after a configurable period of inactivity. The timeout is set in `application.ini`:

```
UserInactiveSeconds = 900 # 15 minutes
```

The `UserCache` periodically purges expired sessions (approximately once per minute). When a session is purged, the global logout handler is called if one was registered in `KissInit`.

20.5.4 Logout Handler

Register a global logout handler in `KissInit.init()` to perform cleanup when any user logs out or their session expires:

```
UserCache.setLogoutHandler({ UserData ud ->
    println "User ${ud.getUsername()} logged out"
    // Close user-specific resources
    // Update database logout timestamp
    // Clean up temporary files
} as Consumer<UserData>)
```

20.6 Automatic Authentication in Services

The framework checks authentication before every service method call. Service methods never need to check authentication themselves — by the time your code runs, the session has already been validated.

If a request arrives with an expired or invalid UUID:

1. The framework returns error code 2.
2. `Server.js` on the frontend detects error code 2.
3. The browser is redirected to the login screen.

Immediately after authentication — and before the service method runs — an application-registered `RequestConnectionPreparer` (if any) prepares the request's database connection from the authenticated user, e.g. selecting a per-tenant schema; see Section 8.12.

20.7 Service-Decides Authentication

The default model described above is *authenticate before dispatch*: the framework validates the session *before* a service method is invoked,

so an invalid or expired session returns `_ErrorCode = 2` and the service never runs. This is the right behavior for the vast majority of methods, but some endpoints need to run *before* the user is known — a public landing page, a status probe, a service that returns one thing to anonymous callers and another to authenticated ones, or a method that does some unauthenticated work and only then decides it needs a logged-in user.

For these cases Kiss offers the inverse: *call the service first, then let it decide.* A method registered with `allowWithoutAuthentication()` (§20.4) runs *whether or not* the caller is logged in:

```
// In KissInit.init()
MainServlet.allowWithoutAuthentication(
    "services.PublicService", "landingData")
```

The class name uses the package path with either dots or slashes (it is normalized internally), and the method name matches the service method exactly.

20.7.1 Deciding Inside the Service

A service that runs without enforced authentication receives the usual signature `(JSONObject in, JSONObject out, Connection db, ProcessServlet servlet)`. The `servlet` parameter exposes four helpers for inspecting and enforcing the session at runtime:

`servlet.isLoggedIn()`
: Returns `true` when a valid session is attached to this request. Branch on it to return one result to authenticated callers and another to anonymous ones.

`servlet.getUserData()`
: Returns the `UserData` for the session, or `null` when the caller is not logged in.

`servlet.getUserData(key)`
: Returns a per-session value, or `null` when the key is absent *or* the caller is not logged in. This overload is null-safe — it never throws when there is no session.

`servlet.requireLogin()`
: Aborts the call with the standard `_ErrorCode = 2` response

when no user is logged in; a no-op when the caller is authenticated. Use it once the service has determined it cannot proceed anonymously.

```
class PublicService {

    // Registered with allowWithoutAuthentication(), so it runs
    // whether or not the caller is logged in.
    void landingData(JSONObject injson, JSONObject outjson,
                     Connection db, ProcessServlet servlet) {

        // Public content everyone may see
        outjson.put("announcement", getPublicAnnouncement(db))

        if (servlet.isLoggedIn()) {
            // Personalize for the authenticated user
            UserData ud = servlet.getUserData()
            outjson.put("greeting",
                "Welcome back, " + ud.getUsername())
        } else {
            outjson.put("greeting", "Welcome, guest")
        }
    }

    // A method that does some unauthenticated work, then
    // insists on a logged-in user before doing the
    // sensitive part.
    void submitOrder(JSONObject injson, JSONObject outjson,
                     Connection db, ProcessServlet servlet) {
        validateCart(injson)      // fine to do anonymously

        // from here on a session is required; if absent, the
        // call aborts with _ErrorCode = 2 and nothing
        // below runs
        servlet.requireLogin()

        Object userId = servlet.getUserData().getUserId()
        placeOrder(db, userId, injson)
    }
}
```

20.7.2 How `requireLogin()` Works Across Runners

`requireLogin()` throws `org.kissweb.restServer.LoginRequiredException`, a `RuntimeException`. The framework detects it in `errorReturn` by *walking the cause chain* (not just

checking the top-level exception type) and converts it to exactly the same `_ErrorCode = 2` response that a normally-protected method produces.

Note

Walking the cause chain is what makes this uniform. Each service runner — Groovy, plain Java, compiled Java, and Lisp — wraps a thrown exception differently (for example, reflection wraps it in an `InvocationTargetException`). Because the framework searches the whole chain for a `LoginRequiredException`, `requireLogin()` behaves identically regardless of the language a service is written in — there is no per-runner special-casing.

20.7.3 Closing the Loop on the Front End

The browser side completes the pattern. Any response carrying `_ErrorCode = 2` — whether from a normally-protected method, an expired session, or a `requireLogin()` call — triggers `Server.logout(true)`. That clears `AppState` (the persisted client state, including the session token) and routes to `/login` *carrying the current location as the* `return` *parameter*. After re-authenticating, the user lands back on the page they were on rather than at a generic home screen.

Tip

Pass `true` to `Server.logout(true)` (as the framework does on `_ErrorCode = 2`) to capture the return location for a post-login resume. An intentional, user-initiated logout calls `Server.logout()` with no argument, so the user is simply returned to login with no resume target.

20.8 Server-Restart Detection (Boot ID)

Sessions live in the in-memory `UserCache`, so a back-end restart invalidates every session at once. The *client's* token, however, persists in `AppState` across reloads and (depending on the configured state store) across tabs and restarts. Without help, a browser holding a

token from before the restart would resume straight onto a session that no longer exists on the server.

Kiss closes this gap with a *boot id*: a UUID generated exactly once per server start.

```
// In MainServlet
private static final String bootId =
        // unique per server start
        java.util.UUID.randomUUID().toString();

public static String getBootId() {
    return bootId;
}
```

The boot id is attached to *every* response as the `_BootId` field — both successful responses and the `_ErrorCode = 2` login-failure response include it. Because the value is fixed for the lifetime of a server process and changes only when the process restarts, the front end can compare it to detect a restart.

20.8.1 The Front-End Check

The browser records the boot id at login (`Server.setBootId`, taken from the `Login` response) and stores it in `AppState` alongside the session token. At startup, *before routing begins*, `Server.verifyServerInstance()` makes a single unauthenticated `LoginRequired` call and compares the boot id it receives against the stored one:

```
static async verifyServerInstance() {
    if (!Server.uuid)
        return;  // no persisted session to protect
    const stored = Server.getBootId();
    let res;
    try {
        // returns _BootId
        res = await Server.call('', 'LoginRequired', {});
    } catch (err) {
        // transient error -- let the normal flow handle it
        return;
    }
    const current = res ? res._BootId : null;
    if (!current)
        return;
    if (stored && stored !== current) {
        // stale session from a previous server instance
```

```
            AppState.clear();
            await Utils.showMessage('Notice',
                'The server was restarted. Please log in again.');
        } else if (!stored) {
            Server.setBootId(current);
        }
    }
```

If the boot id changed, the back end was restarted: the persisted session is cleared and the user is forced into a *clean* re-login — no resume — with a "server was restarted" notice. If there is no persisted session, there is nothing to protect and the check returns immediately.

20.8.2 Restart Versus Session Expiry

A restart that happens while the application is already open is still caught: the next service call returns `_ErrorCode = 2` and the normal logout-and-redirect path takes over. The startup check only adds protection for the case where the browser *begins* a session against an already-restarted server.

It is worth contrasting the two recovery paths, because they intentionally differ:

Session expiry
: The server is the same instance; only this user's session timed out. The front end resumes with a `return` URL (§20.7), so after re-authenticating the user lands back where they were.

Server restart
: The server is a *new* instance (a changed boot id); all sessions are gone and any in-flight client state may be inconsistent. The front end forces a clean login with no resume target and tells the user the server was restarted.

Note

The boot id is not a security token — it is a coarse "is this the same server process?" marker. It is fine that it is returned on unauthenticated calls; its only job is to let the client notice that the in-memory `UserCache` it was relying on has been wiped by a restart.

20.9 Login Form Best Practices

Wrap login fields in a `<form>` tag with `autocomplete` attributes to enable browser password managers:

```
<form onsubmit="return false;" autocomplete="on">
    <text-input id="username" required
                autocomplete="username"></text-input>
    <text-input id="password" required password
                autocomplete="current-password"></text-input>
    <push-button id="login">Login</push-button>
</form>
```

The `onsubmit="return false;"` prevents actual form submission since login is handled by JavaScript. The `autocomplete` attributes help password managers identify the username and password fields.

20.10 Browser Back Button Prevention

After login, enable back-button protection to prevent accidental navigation away from the application:

```
DOMUtils.preventNavigation(
    function () { return true; },
    function () {
        Utils.yesNo('Confirm',
            'Are you sure you want to logout?',
            function () { Server.logout(); });
    }
);
```

The first parameter is a function that returns `true` when protection should be active. The second parameter is called when the user presses the back button.

Note

Chrome blocks navigation prevention until the page has received a user gesture (click, keypress, or touch). On a freshly loaded page before any interaction, the back button cannot be prevented. After the user's first interaction, full protection activates.

20.11 Frontend Inactivity Timeout

In addition to the backend session timeout, the frontend can enforce its own inactivity limit:

```
Server.setMaxInactivitySeconds(900); // 15 minutes
Server.setMaxInactivityMinutes(15); // equivalent
Server.setMaxInactivityHours(1);   // 1 hour
```

If the frontend timeout is exceeded, the user is automatically logged out without waiting for the next server request to trigger the backend timeout.

Chapter 21

Reports and Exports

Kiss supports PDF report generation via Groff, CSV export, and general file delivery to the browser. This chapter covers the report generation pipeline, number formatting for aligned columns, and the temporary file management system.

21.1 PDF Reports with Groff

The `Groff` class provides an interface to the groff typesetting system for generating PDF reports with headers, footers, page numbers, and formatted tables.

Warning

The `Groff` class is a thin wrapper around the underlying `groff` typesetting system — the framework does not bundle it. `groff` (and its `tbl` preprocessor) must be installed on whatever machine runs the Kiss backend, in development and in production alike. It is available by default on Linux and macOS. Windows requires a separate installation; see the *Kiss Getting Started Guide* for the procedure. Without `groff` on the host system, every call to `Groff.process()` will fail at runtime.

```
Groff rpt = new Groff("PhoneList-", "Phone List", false)

// Define table columns (tbl format specification)
rpt.startTable("Lw(1i) Lw(1i) Lw(1i)")
```

```
rpt.column("Last Name")
rpt.column("First Name")
rpt.column("Phone Number")
rpt.endTitle()

// Write data rows
List<Record> recs = db.fetchAll(
    "select * from phone order by last_name")
for (Record rec : recs) {
    rpt.column(rec.getString("last_name"))
    rpt.column(rec.getString("first_name"))
    rpt.column(rec.getString("phone_number"))
}

// Generate PDF and return URL for browser
outjson.put("reportUrl", rpt.process())
```

The constructor parameters are:

1. **Filename prefix** — used to generate a unique temporary file.
2. **Report title** — appears at the top of each page.
3. **Landscape** — `true` for landscape, `false` for portrait.

21.1.1 Table Column Formatting

The `startTable()` format string uses `tbl` syntax. Common specifications:

Format	Meaning
`Lw(1.5i)`	Left-aligned, 1.5 inches wide
`Rw(1i)`	Right-aligned, 1 inch wide
`Cw(2i)`	Centered, 2 inches wide
`Nw(1i)`	Numeric (decimal-aligned), 1 inch

21.1.2 Numeric Columns

For formatted numeric output in tables:

```
// Dollar amount with commas, 2 decimals
rpt.column(rec.getDouble("salary"), "CD", 2)

// Integer with commas
rpt.column(rec.getInt("quantity"), "C")
```

The format mask codes match `NumberFormat` (Section 21.3).

21.1.3 Date Columns

```
rpt.dateColumn(rec.getDateAsInt("hire_date")) // MM/DD/YYYY
rpt.dateColumn2(rec.getDateAsInt("hire_date")) // MM/DD/YY
```

21.1.4 Report Configuration

```
rpt.setFooter("Confidential", "Page \\n[%]")
rpt.addPageTitleLine("Department: Engineering")
rpt.noAutoPageHeader() // disable automatic title
rpt.needLines(5)      // page break if < 5 lines remain
```

21.1.5 Generating the PDF

```
String url = rpt.process()       // standard margins
String url = rpt.process(0.5f)   // custom side margin
String path = rpt.getRealFileName() // absolute file path
```

`process()` invokes the groff pipeline, generates a PDF, and returns the HTTP-accessible URL. The file is automatically cleaned up after approximately one day.

21.2 CSV Export

The `DelimitedFileWriter` class writes CSV files with proper quoting and escaping:

```
File f = FileUtils.createReportFile("Export-", ".csv")
DelimitedFileWriter dfw =
    new DelimitedFileWriter(f.getAbsolutePath())

// Header row
dfw.writeField("Last Name")
dfw.writeField("First Name")
dfw.writeField("Salary")
dfw.endRecord()

// Data rows
for (Record rec : records) {
```

```
    dfw.writeField(rec.getString("last_name"))
    dfw.writeField(rec.getString("first_name"))
    dfw.writeField(rec.getDouble("salary"))
    dfw.endRecord()
}
dfw.close()

outjson.put("exportUrl", FileUtils.getHTTPPath(f))
```

`DelimitedFileWriter` automatically handles quoting when field values contain commas, quotes, or newlines. It supports custom delimiters and quote characters via the extended constructor.

21.2.1 Field Writing Methods

Method	Description
`writeField(String)`	Write string field with auto-quoting.
`writeField(int)`	Write integer.
`writeField(double)`	Write double (2 decimal places).
`writeField(char)`	Write single character.
`writeDate(int)`	Write YYYYMMDD as formatted date.
`writeDate(Date)`	Write Date object as formatted date.
`writeNoZero(int)`	Write integer; empty string if zero.
`writeFieldUpperCase(String)`	Write uppercase string.
`endRecord()`	Finalize row with CRLF.

21.3 NumberFormat for Reports

The `NumberFormat` class formats numbers with commas, dollar signs, parentheses for negatives, and fixed-width fields. It is essential for aligned columnar reports.

```
import org.kissweb.NumberFormat

// Currency: ($12,345.35)
NumberFormat.Format(-12345.348, "CDP", 12, 2)
```

```
// Dollar with commas: $12,345.15
NumberFormat.Format(12345.146, "CD", 10, 2)

// Left-justified: $125.15
NumberFormat.Format(125.146, "CDL", 10, 2)

// Zero-filled with parentheses: (00005.15)
NumberFormat.Format(-5.146, "ZCP", 10, 2)
```

Format mask codes:

Code	Meaning
`B`	Blank if zero
`C`	Add commas
`D`	Floating dollar sign
`L`	Left justify
`P`	Parentheses for negatives
`Z`	Zero fill
`U`	Uppercase (for hex)
`R`	Append percent sign

Parameters: `Format(number, mask, width, decimalPlaces)`. Width of 0 means auto-size. Decimal places of -1 means auto-detect. If the number cannot fit in the specified width, asterisks (`***`) are returned.

21.4 Temporary File Management

`FileUtils` manages temporary files for browser delivery:

```
// Create a temporary report file
File f = FileUtils.createReportFile("MyExport-", ".xlsx")
// ... write to f ...

// Get the URL for browser download
String url = FileUtils.getHTTPPath(f)
outjson.put("downloadUrl", url)
```

Files created with `createReportFile()` are placed in a managed temporary directory and automatically deleted after approximately one day.

On the frontend, open the URL to trigger the download:

```
const res = await Server.call(WS, 'runExport');
if (res._Success)
    Utils.showReport(res.downloadUrl);
```

21.5 The PDF Class

For pixel-precise PDF generation beyond what Groff provides, the `PDF` class wraps Apache PDFBox:

```
PDF pdf = new PDF("output.pdf")
pdf.newPage()
pdf.setFont(PDType1Font.HELVETICA_BOLD, 16)
pdf.startText()
pdf.textOut(1, 1, "Employee Report")
pdf.setFont(PDType1Font.HELVETICA, 10)
pdf.textOut(3, 1, "Name: Alice Smith")
pdf.textOut(4, 1, "Department: Engineering")
pdf.endText()

// Draw a horizontal line
pdf.startGraphics()
pdf.drawHLine(5.0f * 72, 72, 7.5f * 72)
pdf.endGraphics()

// Add an image
pdf.imageOut(300, 72, 0.5f, "logo.png")

pdf.endDocument()
```

Key features:

- Text positioning by line/column (1-indexed) or pixel coordinates.
- Graphics primitives: lines, rectangles.
- Image placement with scaling by ratio, width, or height.
- Portrait and landscape orientation.
- Load and modify existing PDFs as templates.

Chapter 22

Scheduled Tasks

Kiss includes a built-in task scheduler that runs Groovy scripts on a cron-like schedule. This chapter covers the cron system, crontab configuration, and writing task scripts.

22.1 The Cron System

The `Cron` class reads a schedule file and executes Groovy scripts at specified intervals. It:

- Checks for scheduled tasks every 60 seconds.
- Runs each task in a separate thread.
- Dynamically reloads the crontab file when it is modified — no restart required.
- Provides a database connection to each task.

The cron system starts automatically when the Kiss server starts, reading from `src/main/backend/CronTasks/crontab`.

22.2 The Crontab File

The crontab file at `src/main/backend/CronTasks/crontab` defines the schedule. The format follows standard Unix cron conventions:

```
# minute hour  day  month  dayOfWeek scriptName
# Run every minute
```

```
* * * * * EveryMinute

# Run at 2 AM every day
0 2 * * * NightlyCleanup

# Run at 9 AM on weekdays (Mon-Fri)
0 9 * * 1-5 WeekdayReport

# Run at noon on the 1st and 15th
0 12 1,15 * * BiMonthlyExport
```

The five time fields are:

Field	Range	Description
Minute	0–59	Minute of the hour
Hour	0–23	Hour of the day (24-hour)
Day	1–31	Day of the month
Month	1–12	Month of the year
Day of week	0–7	0 or 7 = Sunday, 6 = Saturday

Each field supports:

- `*` — matches all values.
- A single number — matches that value exactly.
- A range — e.g., `8-17` for 8 AM through 5 PM.
- A list — e.g., `1,15` for the 1st and 15th.

Lines starting with `#` are comments. The script name is the Groovy file name without the `.groovy` extension.

22.3 Writing a Cron Task

A cron task is a Groovy file in `src/main/backend/CronTasks/`. Every task must have a static `start(Object obj)` method:

```
import org.kissweb.database.Connection
import org.kissweb.DateTime

static void start(Object obj) {
    Connection db = (Connection) obj
    println("Task running at " +
```

```
        DateTime.currentDateTimeFormatted())

    // Perform scheduled work
    db.execute("DELETE FROM temp_data WHERE created < ?",
        DateUtils.addDays(DateUtils.today(), -30))
    db.commit()
}
```

The `obj` parameter is a database `Connection` provided by the framework. Key points:

- The method receives its own database connection, separate from any service request.
- You must call `db.commit()` to persist changes.
- Each invocation runs in its own thread, so long-running tasks do not block other tasks or service requests.
- If the task throws an exception, it is caught and logged — other tasks continue running.

Warning

Always call `db.commit()` in cron tasks. Unlike service methods, the framework does *not* auto-commit for cron tasks.

22.4 Example: Nightly Data Cleanup

```
// CronTasks/NightlyCleanup.groovy
import org.kissweb.database.Connection
import org.kissweb.DateUtils

static void start(Object obj) {
    Connection db = (Connection) obj

    // Delete old audit records
    int cutoff = DateUtils.addDays(DateUtils.today(), -90)
    db.execute("DELETE FROM audit_log WHERE created_date < ?",
        cutoff)

    // Deactivate expired accounts
    db.execute("""
        UPDATE users SET user_active = 'N'
```

```
        WHERE expiration_date < ? AND user_active = 'Y'
    """, DateUtils.today())

    db.commit()
    println("Nightly cleanup completed")
}
```

Crontab entry:

```
0 2 * * * NightlyCleanup
```

22.5 Example: Periodic Report Generation

```
// CronTasks/WeeklyReport.groovy
import org.kissweb.database.*
import org.kissweb.*

static void start(Object obj) {
    Connection db = (Connection) obj

    Groff rpt = new Groff("WeeklyStatus-",
        "Weekly Status Report", false)
    rpt.startTable("Lw(2i) Nw(1i) Lw(1i)")
    rpt.column("Project")
    rpt.column("Hours")
    rpt.column("Status")
    rpt.endTitle()

    List<Record> projects = db.fetchAll(
        "select name, total_hours, status from project " +
        "where active = 'Y' order by name")
    for (Record r : projects) {
        rpt.column(r.getString("name"))
        rpt.column(r.getDouble("total_hours"), "C", 1)
        rpt.column(r.getString("status"))
    }

    rpt.process()
    // Report saved to temp directory
    println("Weekly report generated: " +
        rpt.getRealFileName())
}
```

22.6 Dynamic Crontab Reloading

The cron system monitors the crontab file for changes. If you edit the file while the server is running, the new schedule takes effect automatically on the next 60-second check cycle. No restart is needed.

Tip

During development, use a short interval like `* * * * *` (every minute) to test your cron task, then change to the real schedule when it works correctly.

Chapter 23

File Handling

Kiss provides comprehensive file handling: browser uploads, CSV parsing and generation, FTP transfers, pixel-precise PDF creation, and temporary file management for browser delivery.

23.1 File Uploads

23.1.1 Frontend: The file-upload Component

The `<file-upload>` component renders a file-selection button:

```
<file-upload id="myFile">Upload File</file-upload>
```

Access it through the standard component API:

```
const ctl = $$('myFile');
ctl.clear();            // reset selection
ctl.disable();          // disable the button
ctl.enable();           // enable the button
```

To send the selected file to the backend, use `Server.fileUploadSend()` instead of the regular `Server.call()`:

```
const res = await Server.fileUploadSend(
    'services.FileUpload', 'upload',
    { description: 'Q1 Data' } // additional JSON fields
);
if (res._Success)
    Utils.showMessage('Info', 'File uploaded successfully.');
```

The third parameter is a JSON object of extra fields sent alongside the file. Any `<file-upload>` components on the current screen are automatically included in the multipart request.

23.1.2 Backend: Receiving Uploads

On the backend, the `servlet` parameter provides access to uploaded files:

```
import org.kissweb.restServer.ProcessServlet

void upload(JSONObject injson, JSONObject outjson,
            Connection db, ProcessServlet servlet) {
    String description = injson.getString("description")

    // Uploaded files are addressed by index, starting at 0
    if (servlet.getUploadFileCount() == 0) {
        outjson.put("error", "No file received")
        return
    }

    // Original filename from the browser
    String originalName = servlet.getUploadFileName(0)

    // Save the upload to a temp file, then process it
    String localPath = servlet.saveUploadFile(0)
    File f = new File(localPath)
    // ... read and process f ...

    db.commit()
}
```

Key servlet methods for file uploads (all file arguments are zero-based indexes):

Method	Description
`getUploadFileCount()`	Number of uploaded files.
`getUploadFileName(i)`	Original filename of upload `i` (from the browser).
`getUploadFileType(i)`	File extension of upload `i`.
`saveUploadFile(i)`	Save upload `i` to a temp file; returns its path.
`getUploadBytes(i)`	Upload `i` as a `byte[]`.

Method	Description
`getUploadBufferedInputStream(i)`	Input stream for upload `i`.

Tip

The uploaded file is stored in a temporary location. Copy or process it immediately — it may be cleaned up after the request completes.

23.1.3 Complete Upload Example

This example imports a CSV file of employee records:

```
import org.kissweb.database.Connection
import org.kissweb.DelimitedFileReader

void importEmployees(JSONObject injson, JSONObject outjson,
                     Connection db, ProcessServlet servlet) {
    if (servlet.getUploadFileCount() == 0)
        throw new Exception("No file uploaded")
    String path = servlet.saveUploadFile(0)

    DelimitedFileReader reader =
        new DelimitedFileReader(path)
    int count = 0

    while (reader.nextLine()) {
        Record rec = db.newRecord("employees")
        rec.set("first_name", reader.getString(0))
        rec.set("last_name", reader.getString(1))
        rec.set("email", reader.getString(2))
        rec.addRecord()
        count++
    }
    reader.close()

    db.commit()
    outjson.put("imported", count)
}
```

23.2 Parsing CSV Files

The `DelimitedFileReader` class parses CSV and other delimited files with proper handling of quoted fields, embedded delimiters, and newlines within fields.

23.2.1 Basic Usage

```
import org.kissweb.DelimitedFileReader

DelimitedFileReader reader =
    new DelimitedFileReader("/path/to/data.csv")

while (reader.nextLine()) {
    String name = reader.getString(0) // by column index
    String email = reader.getString(1)
    double salary = reader.getDouble(2)
    int age = reader.getInt(3)
    // process row
}
reader.close()
```

23.2.2 Custom Delimiters

By default, `DelimitedFileReader` uses comma as the delimiter and double-quote as the quote character. You can specify alternatives:

```
// Tab-delimited file (delimiter and quote char
// specified together)
DelimitedFileReader reader =
    new DelimitedFileReader(path, '\t' as char, '"' as char)

// Pipe-delimited with single-quote quoting
DelimitedFileReader reader =
    new DelimitedFileReader(path, '|' as char,
                            '\'' as char)
```

23.2.3 Header-Based Access

If the first row contains column headers, call `readHeader()` to enable access by column name:

```
DelimitedFileReader reader =
```

```
    new DelimitedFileReader(path)
reader.readHeader()

while (reader.nextLine()) {
    String name = reader.getString("Name")
    String email = reader.getString("Email")
    double salary = reader.getDouble("Salary")
}
reader.close()
```

23.2.4 Field Access Methods

Method	Description
`nextLine()`	Advance to the next record. Returns `false` at end of file.
`readHeader()`	Read the first line as column headers for name-based access.
`getString(int)`	Get field by column index (0-based).
`getString(String)`	Get field by column name (requires `readHeader()`).
`getInt(int)`	Get field as integer by index.
`getInt(String)`	Get field as integer by name.
`getDouble(int)`	Get field as double by index.
`getDouble(String)`	Get field as double by name.
`size()`	Number of fields in the current record.
`close()`	Close the file.

23.2.5 Tolerant Header Matching

Files from different sources rarely agree on column titles. Several opt-in methods make name-based access tolerant. Nothing changes unless you call them; by default titles are matched case-insensitively only.

```
DelimitedFileReader reader =
    new DelimitedFileReader(path)
reader.setNormalizeHeaders(true)
reader.addAlias("Company Name", "Company", "Account")
reader.readHeader()
```

```
// Throws IllegalArgumentException naming EVERY
// missing column, not just the first
reader.requireHeaders("Company Name", "Phone")

while (reader.nextLine()) {
    String company = reader.getString("Company Name")
    int    count   = reader.getInt("Count")
    int    start   = reader.getDate("Start Date") // YYYYMMDD
}
reader.close()
```

Method	Description
`setNormalizeHeaders(boolean)`	When on, titles match ignoring case, white space, hyphens, underscores, and a leading byte-order mark, so `"Company Name"`, `"company_name"` and `"COMPANY-NAME"` are the same column. May be called before or after `readHeader()`.
`addAlias(logical, alternates...)`	Register other titles a file may use for a logical column. Lookups by the logical name find the column under any of them.
`getColumnIndex(name)`	Zero-origin index of a column (aliases and normalization apply), or `-1` if absent. Requires `readHeader()`.
`missingHeaders(names...)`	The names, in the order given, that have no matching column. Empty if none.
`requireHeaders(names...)`	Throws `IllegalArgumentException` listing all missing columns.
`getDuplicateHeaders()`	One entry per collision of file columns that resolve to the same logical column. Empty if none.

Method	Description
`getString/getInt/ getDouble/getDate(name)`	Typed getters by name; all honor aliases and normalization. `getDate` returns an `int` in YYYYMMDD form.

23.2.6 Structured Imports with DelimitedFileImport

`org.kissweb.DelimitedFileImport` is an abstract base class for the common "validate the whole file, then load it" import. It has **no database or ORM dependency**. It reads the file and drives the process; your subclass does all persistence (JDBC, Kiss `Connection`, or anything else) in the abstract methods and hooks.

A run proceeds in order:

1. **Header validation.** Missing required columns, duplicate columns, and a file with no recognized column are reported as errors. If the header is acceptable, the optional `checkHeader` hook is called once.

2. **Check pass.** Every row is passed to `checkRow`, which reports problems with `addError`. Nothing should be written.

3. **Parse pass.** Only if there are zero errors, the file is read again and every row is passed to `parseRow`, which does the work. `commitBatch` is called every batch (default 20 rows) and at the end.

4. In `FULL_MODE`, if still error-free, `deactivateMissing` is called.

5. `onFinish(Result)` is called (or `onFailure` after an unexpected exception) and the `Result` is returned.

Because no row is written unless every row passes the check, a bad file never leaves a partial load.

```
import org.kissweb.DelimitedFileImport;
import java.io.FileNotFoundException;

class ContactImport extends DelimitedFileImport {

    ContactImport(String file, int mode)
            throws FileNotFoundException {
```

```
            super("contact", file, mode);
        }

        @Override
        protected void initColumns() {
            // name, required?, aliases...
            addColumn("Last Name", true, "LNAME", "Surname");
            addColumn("Phone", false);
        }

        @Override
        protected void checkRow(int rowNum) {
            if (getString("Last Name").isEmpty())
                addError("Last Name is missing"); // "Row n: ..."
            if (columnHasData("Phone")
                    && getString("Phone").length() < 7)
                addWarning("Phone looks short");
        }

        @Override
        protected void parseRow(int rowNum) {
            // Write getString("Last Name") using YOUR
            // persistence layer; then count it.
            recordsAdded++;
        }

        @Override
        protected void commitBatch() {
            // e.g. connection.commit();
        }
    }

    ContactImport imp = new ContactImport(path,
        DelimitedFileImport.UPDATE_MODE);
    imp.setBatchSize(100);
    imp.setLogPath("/tmp/contact-import.log");
    DelimitedFileImport.Result r = imp.run(); // never throws
    if (!r.isSuccess())
        System.out.println(r.getErrors());
```

Members

Member	Description
`addColumn(name, required, aliases...)`, `addColumn(name)`	Declare a column by logical name (call from `initColumns`). Matching is normalized (see above). Returns the column's `Column` declaration, to which rules are chained (see "Declarative Column Rules" below); ignoring the return value is fine.
`initColumns()`	*Abstract.* Declare the columns.
`checkRow(rowNum)`	*Abstract.* Validate the current row; report with `addError`/`addWarning`. `rowNum` is 1-based and excludes the header.
`parseRow(rowNum)`	*Abstract.* Process the current row; increment the counters.
`checkHeader()`	Optional hook, called once after the title line is accepted and before the first row is checked (by `run`, `runChecked` and `check`). The reader is still on the title line, so `df.getString(i)` is the title of column `i`. Report problems with `addError` (no row number).
`getFileColumnTitles()`	The title of every column of the file, in file order, as the file has it (see below).
`getFileColumnTitle(name)`	The file's own title for a declared column; `""` if the file lacks it.
`getUnknownColumns()`, `getUnknownColumnIndexes()`	The columns that match no declared column or alias: their titles, and their (parallel) column indexes (see below).
`deactivateMissing()`	Hook, `FULL_MODE` only: deactivate records not in the file.
`commitBatch()`	Hook: flush or commit your persistence layer.

Member	Description
`onFinish(Result)`	Hook: called at the end of every run.
`onFailure(Throwable)`	Hook: called when an unexpected exception aborts the run.
`log(level, msg)`	Hook: level is `LOG_INFO`, `LOG_WARNING` or `LOG_ERROR`.
`setBatchSize(n)`	Rows between `commitBatch` calls (default 20).
`setLogPath(path)`	Write the summary to this file using plain `java.io`.
`run()`	Run the import; never throws. Returns a `Result`.
`check()`	Check the whole file without importing anything; returns every problem as a `List<String>` (empty if the file is acceptable). See below.
`runChecked()`	All-or-nothing import that applies the same check phase as `check()`; never throws. Returns a `Result`.
`getString(name)`, `columnHasData(name)`, `colExists(name)`, `fixNum(s)`	Helpers: trimmed value (`""` if absent; already shortened if the column is declared with `truncate()`), non-blank test, column-present test, and removal of `$` and commas from a number.
`getInt`, `getLong`, `getBigDecimal`, `getDouble`, `getLocalDate`, `getLocalDateTime`, `getIntDate`, `getIntTime` `(name)`	Typed getters for columns declared with a type (see below).

Member	Description
`getWarnings()`	The warnings recorded so far (a value shortened by `truncate()`, or added with `addWarning`); the way to read them after `check()`, which returns only problems.

Modes. `APPEND_MODE` adds new records and ignores existing ones; `UPDATE_MODE` adds and updates; `FULL_MODE` adds, updates, and deactivates records absent from the file. The class only passes the mode through (`mode` field); your `parseRow` and `deactivateMissing` implement the behavior.

Counters. Subclasses increment the protected fields `recordsAdded`, `recordsUpdated` and `recordsDeactivated`; `rowsRead` is maintained by the class.

Result. `getErrors()`, `getWarnings()`, `isSuccess()`, `getRowsRead()`, `getRecordsAdded()`, `getRecordsUpdated()`, `getRecordsDeactivated()`, and `getSummary()` (formatted text, also what `setLogPath` writes). Errors and warnings are prefixed with their row number.

The File's Own Titles and Extra Columns

A subclass declares the columns it understands, but a file often carries more. Four protected accessors show what the file actually has; they are filled as soon as the title line has been read, so they can be used in `checkHeader`, `checkRow` and `parseRow`, under `run`, `runChecked` and `check` alike.

- `List<String> getFileColumnTitles()` — *every* column of the title line, by file column index, so element `i` is the title of the column whose value in a data row is `df.getString(i)`. The titles are the file's own text: the reader trims surrounding white space and nothing else is changed (letter case, inner spaces, hyphens and a byte-order mark are kept). An empty title is `""`, which keeps the indexes aligned. (`getFileColumns()`, by contrast, lists only the *declared* columns that are present.)

- `String getFileColumnTitle(String name)` — the title the file uses for a declared column. The column is found exactly

as `getString(name)` finds it, by its logical name or any alias, so the result may be the alias (`"Surname"` for `"Last Name"`); it is `""` if the file does not have the column.

- `List<String> getUnknownColumns()` and `List<Integer> getUnknownColumnIndexes()` — the columns that no declared column reads: the titles (original text, file order) and, in a parallel list, their zero-based column indexes.
- `String getUnknownString(int index)` and `String getUnknownString(String title)` — the value of an unknown column in the current row: trimmed, `""` for a short row, and with the rules of Section 23.2.6 applied (a truncated value is already shortened). By index it takes an index from `getUnknownColumnIndexes()` and throws `IllegalArgumentException` for any other column, since that is a mistake in the program; by title it matches the file's own text exactly and answers `""` if no unknown column has that title.

```
@Override
protected void checkHeader() {
    // keep every extra column as a free-form attribute
    for (int k = 0; k < getUnknownColumns().size(); k++) {
        String title = getUnknownColumns().get(k);
        int col = getUnknownColumnIndexes().get(k);
        if (title.length() > 50)
            addError("The column title '" + title
                + "' is too long to keep");
        extras.put(col, title);
    }
}

@Override
protected void parseRow(int rowNum) {
    for (Map.Entry<Integer, String> e : extras.entrySet())
        save(e.getValue(), df.getString(e.getKey()));
}
```

A column is *unknown* when its title matches neither a declared column's logical name nor any of its aliases. There is no separate matching code for this: the same reader lookup that `getString(name)`, `colExists(name)` and the header validation use decides, so case, white space, hyphens, underscores and non-printable characters are

ignored and aliases count (a title of `"SUR-name"` is the alias `Surname`, not an extra column). A column with an empty title is not listed, since it has no name to report. If the file has two columns for one declared column, which the header validation reports as a duplicate title, only the column a lookup by name reads is recognized and the other is listed as unknown as well. To compare a title with a declared name, use `colExists(name)` (or `df.getColumnIndex(name)`) rather than comparing the text of `getFileColumnTitles()`. The value of a column in the current row is `df.getString(index)`: trimmed, and `""` for a short row. All three lists are unmodifiable.

Checking a File Without Importing It

Sometimes a file must be accepted or refused *before* any import starts — for example, an upload handler that reports a bad file straight back to the user instead of failing later in a background job. `check()` performs exactly the check phase of a run and nothing more. It reads the title line, applies the header validation that `run()` itself uses (there is only one implementation of those rules), verifies that every line can be read, and calls your `checkRow` for every data row. It writes nothing and calls none of `parseRow`, `commitBatch`, `deactivateMissing` or `onFinish`. It returns *every* problem (there is no limit), so the user can fix the whole file at once; the list is empty if, and only if, the file may be imported.

```
ContactImport chk = new ContactImport(path,
    DelimitedFileImport.UPDATE_MODE);
List<String> problems = chk.check();
if (!problems.isEmpty())
    reject(problems);   // nothing was imported
else   // a check consumes its instance: use a new one
    new ContactImport(path,
        DelimitedFileImport.UPDATE_MODE).run();
```

A problem in a data row is reported as "Row *n*: *problem*", where *n* counts the data rows only (blank lines are skipped and the title line is not counted, so the first data row is Row 1). The `checkRow` you already wrote for `run()` supplies those problems, using the same `getString` helpers; there is no second set of row rules. The file-level problems are:

Situation	Message
No title line	The file is missing its first line of column titles.
A required column is absent	The file is missing the required column “Name”. (One message per missing column.)
No declared column is present	The file has no recognizable columns.
The same column twice, even under different accepted names	Duplicate column titles (A, B); each column may appear only once, even under a different accepted name.
A line cannot be parsed	Row *n* could not be read: *reason*. Checking continues.
An I/O failure	The file could not be read after row *n*: *reason*. Checking stops.

A header problem that makes the rows meaningless (no title line, a missing required column, no recognizable column) is reported *alone*: `checkHeader` and `checkRow` are not called. A repeated title is reported *and* the rows are still checked, so one pass shows the user everything; as in the import, the last of the repeated columns is the one a lookup by name reads.

`run()` keeps its original, more technical wording for the same header problems and stops at the first header problem; `check()` and `runChecked()` use the user-facing wording above. An exception thrown by `initColumns`, `checkHeader` or `checkRow` is not turned into a problem; it propagates out of `check()` (after the file is closed), because it is a failure of the program, not a defect of the file. Like `run()`, a check consumes the instance — the file is closed when it returns — and a second use of the same instance throws `IllegalStateException`.

`runChecked()` is the all-or-nothing import built on the same check phase: it checks everything (header and rows together, the identical problems `check()` would report) and imports only if there is nothing to report, then proceeds exactly as `run()` does. Use it when one call should both check and import; use `check()` followed by a new instance’s `run()` when the refusal must happen earlier than the import (the background import then re-checks the file, which is correct if the data the rules read may have changed in between).

Declarative Column Rules

Most row checks ask the same few questions: is a value there, does it fit the database column, is it a number in range, is it a date in range. `addColumn` therefore returns the column's `Column` declaration, and rules are chained onto it in `initColumns`. The rules need no database; a limit that is known only at run time, such as the width of a database column, is passed as a callback.

```
@Override
protected void initColumns() {
    addColumn("Last Name", true, "LNAME", "Surname")
        .notBlank().maxLength(30);
    addColumn("Job Title")
        .maxLength(() -> widthOf("person", "job_title"))
        .truncate();
    addColumn("# Employees").integer().range(0, 1_000_000);
    addColumn("Revenue").decimal().min(0);
    addColumn("Hire Date").date("MM/dd/yyyy")
        .range(LocalDate.of(1990, 1, 1), LocalDate.now());
    addColumn("Activity Time").dateTime("MM-dd-yyyy hh:mm a");
}

@Override
protected void parseRow(int rowNum) {
    int size = getInt("# Employees");        // parsed
    String title = getString("Job Title");   // shortened
    LocalDate hired = getLocalDate("Hire Date"); // or null
}
```

Rule	Meaning
`required()`, `aliases(...)`	The fluent form of the `required` and `aliases` arguments of the long `addColumn`. `required` is a rule about the title line only.
`notBlank()`	When the file has the column, every data row must have a value in it. Independent of `required()`; a column that is both is checked for both.

Rule	Meaning
`maxLength(n)`, `maxLength(IntSupplier)`	A text value may have at most `n` characters (Unicode code points, as a database column counts them). A longer value is a row error, unless `truncate()` is also given. The supplier is called once per run, after the title line is accepted, and only if the file has the column. A result of `Integer.MAX_VALUE` means no limit.
`truncate()`	Shorten a value that is too long instead of rejecting it, and record a warning. Text columns only.
`integer()`, `longInteger()`, `decimal()`	The value must be a whole number that fits an `int`, a whole number that fits a `long`, or a number.
`min(x)`, `max(x)`, `range(lo, hi)`	Inclusive bounds, given after the type: `long`, `double` or `BigDecimal` for a number; `LocalDate` for a `date`; `LocalDateTime` for a `dateTime`. A `null` bound is open.
`date()`, `date(fmt, more...)`, `dateTime(fmt, more...)`	The value must be a date (or date and time) in one of the formats; the first that reads the whole cell is used. `date()` with no format accepts what `DateUtils.parse` accepts.

When the rules run. The rules of every declared column run automatically, once per data row, in `check()`, in the check phase of `runChecked()` and in the check pass of `run()`, *before* `checkRow` is called for the row. A rule that is not met is a row error, every one is recorded (nothing is capped), and `checkRow` is still called, so its own problems follow in the same list. Within a row the messages are in file order. The rules apply only to a column the file has. A blank cell is valid unless the column is `notBlank()`, and a blank cell skips the length, number and date checks, so an empty optional number is not an error and a missing required one is reported once.

Messages. They are plain English and name the column by the title the file uses (an alias if that is what the file has), so the user sees their own header:

Rule broken	Message
`notBlank()`	Row 2: “Last Name” must have a value.
`maxLength(30)`	Row 3: “Job Title” is 45 characters long; at most 30 are allowed.
`integer().range(0, 1_000_000)`	Row 5: “# Employees” must be a whole number from 0 to 1,000,000 (found “50-100”).
`integer().min(0)`	Row 5: “# Employees” must be a whole number of at least 0 (found “-1”).
`decimal().max(100)`	Row 6: “Rate” must be a number of at most 100 (found “100.5”).
`date("MM/dd/yyyy")`	Row 7: “Hire Date” must be a date such as 03/25/2024 (found “13/45/2024”).
`date(...).range(lo, hi)`	Row 8: “Hire Date” must be a date from 01/01/1990 to 12/31/2030 (found “01/01/1980”).

Numbers are read as a person writes them in a spreadsheet: dollar signs, commas and white space are ignored (“$1,500,000”, “1 500”), a leading sign is allowed, and a decimal point may be used. An `integer()` or `longInteger()` rounds a value with a decimal point to the nearest whole number, halves away from zero (“1500.5” is 1501), and then checks the range; `decimal()` keeps the exact value (“$1,234.50” is 1234.50). Text, exponents, percent signs and a second decimal point are errors.

Dates. A format is a `java.time.format.DateTimeFormatter` pattern with these differences, all meant to catch bad data: `yyyy` accepts exactly a four-digit year, so “03-15-24” is refused; `yy` reads a two-digit year with the window `DateUtils` uses; `MM` and `dd` need two digits where `M` and `d` accept one or two; letter case is ignored; the text is read strictly, so February 30 and trailing text are errors; and month names and AM/PM are English. The example in the message is the first format applied to a sample date. A format that is not a valid

pattern, or cannot read the date it writes (a date-time format without a time, a 12-hour `h` without `a`), throws `IllegalArgumentException` from `initColumns`, so the mistake appears the first time the import runs.

Typed getters. `getInt` (for `integer()`), `getLong` (`integer()` or `longInteger()`), `getBigDecimal` and `getDouble` (any number), `getLocalDate` and `getIntDate` (a date or date-time; `getIntDate` is Kiss's YYYYMMDD `int`), `getLocalDateTime` and `getIntTime` (a date-time; `getIntTime` is HHMM) return the value the rules accepted. They are valid in `checkRow` and `parseRow`. A blank cell, a cell that broke its rule (already reported), and a column the file lacks give `0` for the primitive getters and `null` for the object getters, so a `checkRow` may call them without guarding; use `columnHasData` to tell a blank from a zero. Asking for a type the column was not declared with, or for a column that was never declared, throws at once.

Shortening and warnings. A column declared with `maxLength` and `truncate()` is cut to the limit in one place, before the row is checked or parsed, so `getString` returns the same shortened text in `checkRow` and `parseRow` (the reader's `df.getString` remains the file's own text). The cut never splits a character that takes two UTF-16 units. A shortened value is not an error, but it is not silent: each one is recorded, once, in the check pass, as a warning such as `Row 4: "Job Title" was 45 characters long and was shortened to 30 characters.` Warnings never make a file unacceptable. They are in `Result.getWarnings()` (and in the summary) after `run()` or `runChecked()`, and in `getWarnings()` on the import object after `check()`, whose return value is only problems.

A limit from the database. Pass the width as a callback so that Kiss itself needs no database; the callback runs once per run, on the importing thread, after the title line is accepted:

```
addColumn("City").maxLength(() -> {
    try {
        return db.getColumnWidth("address", "city");
    } catch (SQLException e) {
        throw new IllegalStateException(e);
    }
});
```

Mistakes in the declaration (a range before its type, a range on a text column, two types, `truncate()` on a number, a back-

wards range, a bad date format) throw `IllegalStateException` or `IllegalArgumentException` from `initColumns`; like any exception from `initColumns`, `check()` lets it propagate and `run()` reports it in the result.

Rules for the Extra Columns and for Derived Values

Two situations fall outside the rules of a declared column: the values of the columns the subclass did not declare (extra columns it keeps anyway, say as free-form attributes with a database width of their own), and a *part* of a declared value (a street cut out of a full-address cell, which must fit the street column). Both want the same behavior as a declared column, with the same messages and warnings.

Rules for every unknown column. `unknownColumnRules()` returns the `Column` declaration for *all* the unknown columns at once, so the rules above are chained onto it in `initColumns` in the same way:

```
@Override
protected void initColumns() {
    addColumn("Company", true).notBlank();
    unknownColumnRules()
        .maxLength(() -> answerWidth()).truncate();
}

@Override
protected void parseRow(int rowNum) {
    List<String> titles = getFileColumnTitles();
    for (int i : getUnknownColumnIndexes())
        save(titles.get(i), getUnknownString(i)); // shortened
}
// Row 4: "Fax No" was 2300 characters long and was
// shortened to 2000 characters.        (a warning)
```

The rules run exactly as those of a declared column do: once per data row in `check()`, in the check phase of `runChecked()` and in the check pass of `run()`, before `checkRow`; every unmet rule is a row error, in file order together with the declared columns' errors; a truncated value is a warning, recorded once, in the check pass only; and the message names the column by the title the file has (a title longer than 60 characters is shown shortened, with "..."). The run-time limit is read once per run, not once per column, and only if the file has an unknown column. `getUnknownString` returns the value with the rules applied, in `checkRow` and `parseRow` alike, so

the check and the import see the same text; `df.getString(index)` is still the file's own.

Declare the rules in `initColumns`: the set of unknown columns is fixed when the title line has been read. An import that keeps only some of the unknown columns, or none, simply declares no rules (or applies its own selection when it reads them); with nothing declared the unknown columns behave as before. Every rule of a declared column is available, a number or date type included (enforced in every row, the value read as text), except `required()` and `aliases(...)`, which are about matching a title to a declared name and throw `IllegalStateException` here.

Shortening a derived value. `truncate(label, value, max)` cuts any value to `max` characters and records the same warning a truncated column does:

```
String street = truncate(
    "Street (from Company Address)",
    parts.street, widthOf("address", "street"));
// Row 7: "Street (from Company Address)" was 45
// characters long and was shortened to 30 characters.
```

It counts Unicode code points and never splits a character made of two UTF-16 units; a value that fits is returned unchanged, `null` is returned as `null`, and `Integer.MAX_VALUE` means no limit. The label is put between double quotes and should name the value as the user knows it.

It reports once. The value that is checked must be the value that is saved, so call `truncate` wherever the value is computed, in `checkRow` and `parseRow` alike (for instance from one method that both call); both passes get the same text. The warning is recorded only while the rows are not being imported: in the check pass of `run()`, in `check()` and in the check phase of `runChecked()`. The parse pass shortens the same value silently, which is how a cut is listed once, as it is for a declared column. Within one row the same warning (same label, value and limit) is also recorded only once, so several helpers of `checkRow` may reach the same value. A value that is only shortened in `parseRow` is never reported, so shorten it in the check pass too. Called outside the rows, for instance from `checkHeader`, it records a warning without a row number.

23.3 Writing CSV Files

The `DelimitedFileWriter` class writes CSV files with automatic quoting and escaping. Its full API is covered in Chapter 21, Section 21.2. Here is a quick reference:

```
import org.kissweb.DelimitedFileWriter
import org.kissweb.FileUtils

File f = FileUtils.createReportFile("Export-", ".csv")
DelimitedFileWriter dfw =
    new DelimitedFileWriter(f.getAbsolutePath())

// Header
dfw.writeField("Name")
dfw.writeField("Amount")
dfw.endRecord()

// Data
dfw.writeField("Alice Smith")
dfw.writeField(12345.67)
dfw.endRecord()

dfw.close()
String url = FileUtils.getHTTPPath(f)
```

Like `DelimitedFileReader`, the writer supports custom delimiters and quote characters through its extended constructor.

23.4 FTP Transfers

The `FTP` class provides a client for transferring files to and from remote FTP servers.

23.4.1 Connecting and Transferring

```
import org.kissweb.FTP

FTP ftp = new FTP("ftp.example.com", "username", "password")

// Upload a file
ftp.put("local-report.csv", "remote-report.csv")

// Download a file
ftp.get("remote-data.csv", "local-data.csv")
```

```
// Change remote directory
ftp.cd("/incoming/reports")

// Upload to current remote directory
ftp.put("monthly-report.pdf", "monthly-report.pdf")

ftp.close()
```

23.4.2 FTP Methods

Method	Description
`FTP(host, user, pw)`	Connect and authenticate.
`put(local, remote)`	Upload a local file to the remote server.
`get(remote, local)`	Download a remote file to a local path.
`cd(path)`	Change the remote working directory.
`close()`	Disconnect from the server.

Warning

Always call `ftp.close()` when done, preferably in a `try/finally` block, to avoid leaving connections open on the remote server.

23.4.3 Example: Automated Export via FTP

This cron task generates a CSV export and uploads it to a partner's FTP server:

```
// CronTasks/NightlyExport.groovy
import org.kissweb.database.*
import org.kissweb.*

static void start(Object obj) {
    Connection db = (Connection) obj

    // Generate CSV
    File f = File.createTempFile("export-", ".csv")
    DelimitedFileWriter dfw =
```

```
        new DelimitedFileWriter(f.getAbsolutePath())
    dfw.writeField("ID")
    dfw.writeField("Name")
    dfw.writeField("Amount")
    dfw.endRecord()

    List<Record> recs = db.fetchAll(
        "select * from orders where order_date = ?",
        DateUtils.addDays(DateUtils.today(), -1))
    for (Record r : recs) {
        dfw.writeField(r.getString("order_id"))
        dfw.writeField(r.getString("customer_name"))
        dfw.writeField(r.getDouble("amount"))
        dfw.endRecord()
    }
    dfw.close()

    // Upload via FTP
    FTP ftp = null
    try {
        ftp = new FTP("ftp.partner.com", "user", "pass")
        ftp.cd("/incoming")
        ftp.put(f.getAbsolutePath(), "daily-orders.csv")
    } finally {
        if (ftp != null)
            ftp.close()
        f.delete()
    }

    db.commit()
    println("Nightly export uploaded")
}
```

23.5 PDF Generation

Kiss provides two PDF generation approaches:

1. **Groff** — Table-oriented reports with automatic headers, footers, and page numbers (Chapter 21). Requires the `groff` typesetting system to be installed on the host running the backend; it is not bundled with Kiss.

2. **PDF class** — Pixel-precise document creation via Apache PDFBox for forms, certificates, and custom layouts.

The `PDF` class is covered in Chapter 21, Section 21.5. Here is a summary of its key capabilities:

```
import org.kissweb.PDF

PDF pdf = new PDF("output.pdf")
pdf.newPage()

// Text output by line/column (1-indexed)
pdf.setFont(PDType1Font.HELVETICA_BOLD, 16)
pdf.startText()
pdf.textOut(1, 1, "Invoice #12345")
pdf.setFont(PDType1Font.HELVETICA, 10)
pdf.textOut(3, 1, "Customer: Acme Corp")
pdf.textOut(4, 1, "Date: 04/04/2026")
pdf.endText()

// Graphics
pdf.startGraphics()
pdf.drawHLine(5.0f * 72, 72, 7.5f * 72)
pdf.endGraphics()

// Images with scaling
pdf.imageOut(300, 72, 0.5f, "company-logo.png")

pdf.endDocument()
```

Key capabilities:

- Text positioning by line/column or absolute pixel coordinates.
- Graphics primitives: horizontal lines, rectangles.
- Image placement with scaling by ratio, width, or height.
- Portrait and landscape page orientation.
- Load existing PDFs as templates and add content to them.
- Multiple pages with independent content.

23.6 Temporary File Management

`FileUtils` manages temporary files for browser delivery. This is the standard pattern for delivering generated files (reports, exports, documents) to the browser.

23.6.1 Creating and Delivering Files

```
import org.kissweb.FileUtils

// Create a temporary file with a prefix and extension
File f = FileUtils.createReportFile("Invoice-", ".pdf")

// ... write content to f ...

// Get the HTTP-accessible URL
String url = FileUtils.getHTTPPath(f)
outjson.put("downloadUrl", url)
```

On the frontend, open the URL to trigger the download:

```
const res = await Server.call(WS, 'generateInvoice');
if (res._Success)
    Utils.showReport(res.downloadUrl);
```

23.6.2 How It Works

1. `createReportFile()` creates a uniquely named file in a managed temporary directory that is accessible via HTTP.
2. `getHTTPPath()` returns the URL path relative to the application root.
3. `Utils.showReport()` opens the URL in a new browser tab, triggering the download.
4. Files are automatically cleaned up after approximately one day.

23.6.3 Additional FileUtils Methods

Method	Description
`createReportFile(prefix, ext)`	Create a temp file for browser delivery.
`getHTTPPath(File)`	Get the HTTP URL for a temp file.
`readFile(path)`	Read an entire file into a string.
`write(path, content)`	Write a string to a file.

Method	Description
`copy(src, dest)`	Copy a file.
`getMimeType(filename)`	Determine MIME type from extension.

Tip

Use `FileUtils.createReportFile()` for all generated files that need browser delivery. It handles unique naming, proper directory placement, and automatic cleanup.

23.7 External File Storage

The `ExternalFile` class (`org.kissweb.database.ExternalFile`) associates one or more files on disk with a row of any SQL table — useful when content (photos, scanned documents, PDFs, large text blobs) is more naturally kept as a file than as database bytes, while still needing to travel with a specific row's lifecycle (in particular, its deletion). A file's location is always *computed* from the table name, the row's primary key, and (when a row has more than one associated file) a field name — never stored in the database — so relocating the file store to another path or machine requires no data migration.

`ExternalFile` supports two distinct patterns, covered in turn below:

- **Virtual columns** — a file associated with a row through no SQL column at all; the association exists purely in the file's computed path.
- **Hybrid columns** — a real, existing `varchar` column that stores its value inline when it fits and transparently overflows to an external file when it does not.

23.7.1 Configuration

Before any other method on `ExternalFile` is used, the application registers two lambdas once, typically from `KissInit.groovy`:

```
import org.kissweb.database.ExternalFile
```

```
// Where files are stored, evaluated fresh on every call (safe
// to vary, e.g. per tenant)
ExternalFile.setRootSupplier({ -> Config.externalFileRoot() })

// How a (tableName, primaryKey) pair maps to a directory
// under the root
ExternalFile.setDirectoryMapper({ String table, String pk ->
    "${table}/${pk}"
})
```

Calling any other `ExternalFile` method before both are configured throws `IllegalStateException`. Nothing else about primary-key format, sharding, or table layout is assumed — the directory mapper decides all of that, so an application with composite or non-numeric keys, or that wants deeper sharding as row counts grow (e.g. splitting on a key prefix), supplies its own mapping.

23.7.2 Virtual Columns

A virtual column has no backing SQL column; the file's association with a row is entirely computed. Register a logical name once for each file type a table has, giving it the table name and a field name (fictitious — it need not, and normally does not, exist as a real column):

```
ExternalFile.addField("employeePhoto", "employee", "photo")
ExternalFile.addField("employeeResume", "employee", "resume")
```

Save and retrieve by that logical name and the row's primary key:

```
// Save (String, byte[], and Byte[] overloads are all provided)
ExternalFile.saveData("employeePhoto", employeeId, ".jpg",
    imageBytes)

// Read back -- "" / empty array if nothing was ever saved
String resumeText = ExternalFile.getString("employeeResume",
    employeeId, ".txt")
byte[] photoBytes = ExternalFile.getBinary("employeePhoto",
    employeeId, ".jpg")

// Remove
ExternalFile.deleteExternalFile("employeePhoto", employeeId,
    ".jpg")
```

For uploaded files specifically, prefer `saveInputStream()` over `saveData()`: it additionally normalizes EXIF orientation on supported image types (jpg, png, bmp, gif, tiff) so a photo taken on a rotated phone camera does not display sideways.

```
void upload(JSONObject injson, JSONObject outjson,
           Connection db, ProcessServlet servlet) {
    String ext = servlet.getUploadFileType(0)
    ExternalFile.saveInputStream("employeePhoto",
        injson.getString("employeeId"),
        ext, servlet.getUploadBufferedInputStream(0))
}
```

A single row may have several associated files distinguished by field name; every method above accepts an `ExternalField` constructed directly (`new ExternalField(tableName, fieldName)`) as an alternative to a registered logical name, for callers that prefer not to populate the registry.
To deliver a stored file to the browser, generate a temporary HTTP-servable copy:

```
String url = ExternalFile.getURL("employeePhoto", employeeId,
    "Photo-", ".jpg")
outjson.put("photoUrl", url)
```

Key methods:

Method	Description
`addField(name, table, field)`	Register a logical file type.
`saveData(name, pk, ext, data)`	Save (String/byte[]/Byte[]), overwriting any existing file.
`saveInputStream(name, pk, ext, is)`	Save an upload, normalizing EXIF orientation for images.
`getString(name, pk, ext)` / `getBinary(...)`	Read back; `""` / empty array if none saved.
`deleteExternalFile(name, pk, ext)`	Delete; a no-op if the file does not exist.
`externalFileExists(name, pk)`	Does any file exist for this row (any field/extension)?
`deleteAllExternalFiles(table, pk)`	Delete every file for this row, all fields and extensions.

Method	Description
`getURL(name, pk, prefix, ext)`	Temporary HTTP-servable copy of the stored file.

23.7.3 Cascade Delete

`ExternalFile` can participate in the delete callback described in Section 9.6, so a row's files are removed automatically when the row is deleted:

```
// Once, e.g. from a Connection-preparation hook
db.setDeleteCallback(ExternalFile::deleteCallback)

// Once per table, after confirming this is the wanted behavior
ExternalFile.cascadeDeleteFor("employee")
```

Cascade delete is opt-in per table and off by default. Wiring the callback into a `Connection` has no effect on a table until that table is separately opted in with `cascadeDeleteFor()` — an application that has always deleted rows without removing their files does not have that behavior silently change the moment the callback exists elsewhere in the codebase. A table not opted in is simply a no-op on delete. This applies equally to hybrid columns (Section 23.7.4): once a table is opted in, both its virtual-column files and its hybrid-column overflow files are removed on row delete, since both use the same table+primary-key path scheme.

23.7.4 Hybrid Columns: A Real Column That Overflows to a File

A hybrid column is a real, existing `varchar` SQL column whose value is normally small but occasionally too large for the column. `saveHybridColumn()` and `getHybridColumn()` store it inline in the column when it fits, and in an external file (using the same root/directory-mapper configuration as virtual columns) when it does not — transparently, in a single call each way:

```
import org.kissweb.database.ExternalFile

// Save -- decides inline vs. external internally
ExternalFile.saveHybridColumn(db, "note", "body", noteId,
    bodyText)
```

```
// Read -- transparently returns the real value either way
String bodyText = ExternalFile.getHybridColumn(db, "note",
    "body", noteId)
```

This is the right tool when a column's content is normally short but occasionally long (a notes field, a JSON blob, a description that is usually a sentence and occasionally a whole document) and a real column for it already exists, or is one the schema is willing to add. Use the virtual-column methods in Section 23.7.2 instead when there is no real column at all — a photo or a resume, for example, has no reason to have a database column.

How the size decision is made. `saveHybridColumn()` determines the column's declared size (in characters — the same unit a database like PostgreSQL uses for `varchar(n)`) by reading it from the database's own column metadata (the cached `Connection.getColumnWidth()` lookup of Section 8.10); the caller never supplies or hard-codes a threshold, and the metadata lookup is cached per schema, table and column for the life of the application. Data that fits is written directly into the column; data that does not is written to an external file, and a short sentinel value is written into the column in its place so a later `getHybridColumn()` call knows to read the file instead.

Data that happens to look like the sentinel. Because real data could coincidentally begin with the exact bytes used as the sentinel, `saveHybridColumn()` checks for that collision on every save: a value that would be misread is stored with a short escape prefix instead, and `getHybridColumn()` strips it back off. Ordinary data — the overwhelming majority — collides with neither sequence and is stored with no overhead at all. The round trip is exact for every possible input, including a value that merely looks like the sentinel.

Transitions. A later save can move a row's data in either direction. Going from a large value to a small one deletes the now-stale external file; going from one large value to another simply overwrites the same file. A save of `null` or an empty string clears the column and removes any external file for that row. Because the file's location is computed the same way as for virtual columns (table name, primary key, and the column name standing in for the field name), a table

already opted into cascade delete (Section 23.7.3) needs no separate wiring for hybrid columns.

Resolving a value you already have. When the raw column value arrives as part of a larger query (a list or a join), calling `getHybridColumn()` per row would add a SELECT each time. `resolveHybridColumn()` takes the raw value as fetched and returns the real data without querying: `null` stays `null`, ordinary data is returned as-is, an escaped value has its prefix stripped, and the external marker causes the file to be read (an empty string if it is missing). The primary key is needed only to compute the file's path. `getHybridColumn()` itself delegates to it, so the two cannot drift. `isExternal()` tests whether a raw value is the external marker, for example to skip a large field in a list view.

```
// from a larger query
String raw = rec.getString("body")
if (!ExternalFile.isExternal(raw))
    show(raw)
String bodyText = ExternalFile.resolveHybridColumn(db, "note",
    "body", noteId, raw)
```

Clearing a column. `saveHybridColumn()` with `null` stores SQL `NULL` and with an empty string stores an empty string; on a `NOT NULL` column the former fails. `releaseHybridColumn()` removes the external file (if any) and clears the column to `NULL` or to an empty string according to its `nullable` argument, so the caller states the column's nullability rather than choosing between the two save calls.

Method	Description
`saveHybridColumn(db, table, column, pk, data)`	Save, choosing inline vs. external automatically; `null`/empty clears the column and removes any external file.
`getHybridColumn(db, table, column, pk)`	Read, transparently returning the real value regardless of where it is stored.
`resolveHybridColumn(db, table, column, pk, raw)`	Resolve an already-fetched raw column value to the real data with no SELECT.

Method	Description
`isExternal(raw)`	True exactly when a raw column value is the external-storage marker.
`releaseHybridColumn(db, table, column, pk, nullable)`	Remove any external file and clear the column: to `NULL` if `nullable`, else to an empty string.

Warning

As with the rest of `ExternalFile`, a file write or delete is not transactional with the surrounding database transaction — it happens immediately, independent of commit/rollback. `saveHybridColumn()` performs the file operation *before* updating the column, so a failure on the database side leaves at worst an orphaned external file, never a column pointing at a file that was never written. The table must also have a single-column primary key.

Chapter 24

LLM Integration

Kiss includes built-in support for large language models (LLMs) through five classes in the `org.kissweb.llm` package. This chapter covers configuration and usage of each provider, streaming responses, embeddings, and vector search for Retrieval Augmented Generation (RAG).

24.1 Overview

The LLM integration provides:

- **OpenAI** — Chat completions, streaming, and embeddings via the OpenAI API, over either of that provider's two endpoints (Section 24.2.4).
- **Anthropic** — Chat completions and streaming via the Claude API.
- **OpenRouter** — Chat completions and streaming for models from many providers (OpenAI, Anthropic, Google, Meta, Mistral, and others) through one OpenAI-compatible gateway API and one API key.
- **Ollama** — Local model inference for privacy-sensitive or offline deployments.
- **QdrantClient** — Vector database client for RAG workflows.

All the provider classes share a similar pattern: construct with credentials, send a prompt, receive a response. Streaming variants deliver tokens incrementally via callbacks.

24.1.1 Error Handling

The three cloud classes (`OpenAI`, `Anthropic`, and `OpenRouter`) share one error contract. A failed call — a non-2xx HTTP response, or an error event the provider reports mid-stream — throws an exception whose message carries the HTTP status and the provider's raw error body, for example:

```
OpenRouter request failed with HTTP 401:
{"error":{"message":"User not found.","code":401}}
```

The same information remains available afterward from the instance: `getResponseCode()` returns the HTTP status of the last call, and `getResponseString()` returns the raw error body of the last failed call. Because `send()` is built on `stream()`, the contract is identical for both. Wrap calls in `try/catch` and log the exception message — it contains everything the provider said about the failure.

24.1.2 Web Search

`OpenAI`, `Anthropic`, and `OpenRouter` can let the model consult the public web while it generates — for example, to read a prospect's company website. The feature is opt-in: call `setWebSearch(true)` on the instance. By default it is off, and requests are exactly what they would be without the method.

```
Anthropic ai = new Anthropic(key, "claude-sonnet-4-20250514")
ai.setWebSearch(true)
String answer = ai.send("What does acme.example sell?")
```

The provider performs the searches itself; the caller still receives one stream of text, so `send()` and `stream()` are used as usual. What changes on the wire differs by provider:

Class	Request when enabled
`Anthropic`	The Messages API request carries `"tools": [{"type": "web_search_20250305", "name": "web_search"}]`. All text the model produces, including the text on either side of its searches, is returned concatenated.
`OpenAI`	Web search is a feature of the responses endpoint, so the request is always sent there — whatever the model, and even if `setApi(Api.CHAT_COMPLETIONS)` was called — with `"tools": [{"type": "web_search"}]`. The search progress events in the stream are ignored and only the answer text is delivered.
`OpenRouter`	The chat request carries `"plugins": [{"id": "web"}]`; the model id is left unchanged.

Web search is billed by the provider in addition to tokens. If the account or model does not support it (for example, an administrator has disabled it, or an OpenAI reasoning model is run at minimal effort), the provider's HTTP error is thrown as described in Section 24.1.1; the request is not silently retried without search. `Ollama` runs models locally and has no web search option.

24.2 OpenAI

The `OpenAI` class connects to the OpenAI API for chat completions and embeddings.

24.2.1 Construction

```
import org.kissweb.llm.OpenAI

// API key, model, and whether the model is a
// reasoning model (o1, o3, etc.)
OpenAI ai = new OpenAI(apiKey, "gpt-4o", false)

// A reasoning model, with effort control
OpenAI r = new OpenAI(apiKey, "o1", true)
r.setReasoningEffort("high") // "low", "medium", "high"
```

The third constructor parameter declares whether the model is a reasoning model. For a reasoning model the request carries the `reasoning_effort` parameter (set with `setReasoningEffort()`); for a standard model it carries `temperature` and `top_p` instead.

24.2.2 Chat Completions

```
// Simple request
String response = ai.send("Explain SQL joins.")

// With an image (JPEG, PNG, GIF, or WebP)
String answer = ai.send("What's in this image?",
    "/path/to/image.jpg")
```

24.2.3 Streaming Responses

For real-time token delivery (e.g., a chat interface), use the streaming variant. `stream()` takes the prompt, a `Consumer<String>` called for each token as it arrives, and a `Runnable` called once the response is complete:

```
ai.stream("Write a haiku about databases.",
    // onToken: Consumer<String>
    { token -> print(token) },
    { println("\n---\nDone.") })      // onDone: Runnable
```

A four-argument overload adds an image path after the prompt. Errors are thrown as exceptions rather than delivered through a callback — see Section 24.1.1 for the shared error contract.

24.2.4 The Two Endpoints: Chat Completions and Responses

OpenAI serves its models through two different HTTP endpoints. The long-standing one is *chat completions* (`/v1/chat/completions`), the endpoint every OpenAI-compatible gateway implements and the one this class has always used. The newer one is *responses* (`/v1/responses`), which OpenAI now positions as its primary API. The distinction would be an implementation detail except for one fact: some models are served *only* by the responses endpoint. Send such a model to chat completions and the request fails with an HTTP

404 whose body says, in effect, "this model is not supported in `v1/chat/completions`; use `v1/responses` instead." Which models fall on which side changes as OpenAI releases new ones, so Kiss does not rely on a list of names; it combines one naming rule with detection of the failure itself.

Kiss handles this for you. The `Api` enum selects the behavior, and the default — `Api.AUTO` — needs no configuration at all:

- `Api.AUTO` (default) — route automatically: a model whose id ends in `-pro` (for example `o1-pro`, `gpt-5-pro`) is responses-only and goes straight to the responses endpoint; any other model is sent to chat completions, and if the provider reports that it requires the responses endpoint, the identical request is transparently re-sent there.
- `Api.CHAT_COMPLETIONS` — always use chat completions; never fall back.
- `Api.RESPONSES` — always use responses.

The provider has worded that 404 differently over time (naming `/responses`, or saying the model "is not a chat model" with the error parameter set to `model`), so Kiss accepts any of these forms as the signal. A 404 that merely reports an unknown model (error code `model_not_found`) is not such a signal and is surfaced as an ordinary error. The retry happens at most once per call: if the responses endpoint also fails, that failure is thrown and nothing is remembered. For `-pro` models the responses request omits `temperature` and `top_p`, which those models reject.

The fallback is safe precisely because of *when* the failure occurs. The 404 is the HTTP status of the request itself, so it is known before a single token has been streamed — no partial output has reached the caller's `onToken` callback, and there is nothing to undo. The class therefore simply issues the second request and the caller sees one clean stream, never learning that two HTTP calls took place. Had the error arrived mid-stream, after tokens were already delivered, no such retry would be possible; that case still surfaces as an exception (Section 24.1.1).

The result is also cached. When the responses endpoint has successfully served a model after such a fallback, its name is recorded in a static, thread-safe set shared by every `OpenAI` instance in the

JVM, so only the *first* request for that model pays the cost of the failed attempt. Every later request — from any instance — goes straight to the responses endpoint. The set is per-process and is not persisted; a server restart re-learns it on the first call, which costs one extra round trip.

Because `send()` is implemented on top of `stream()` (it streams into a `StringBuilder`), the blocking and streaming calls both get this behavior; there is no separate code path to configure.

Pin the endpoint explicitly only when you have a reason to:

```
// Default: no configuration needed
OpenAI ai = new OpenAI(apiKey, "gpt-5", false)
String answer = ai.send("Explain SQL joins.")

// Pointing at a gateway that implements only
// chat completions -- suppress the fallback
ai.setApi(OpenAI.Api.CHAT_COMPLETIONS)

// A model you know is responses-only -- skip the
// probe request entirely
ai.setApi(OpenAI.Api.RESPONSES)
```

Passing `null` to `setApi()` restores `Api.AUTO`.

What Differs Between the Two

The two endpoints carry the same information in different shapes, and the class translates between them so that neither your code nor your callbacks change:

Concept	Chat completions	Responses
System prompt	a `system` message	top-level `instructions`
Prompt	`messages` array	`input` array
Text content	`type: "text"`	`type: "input_text"`
Image content	`type: "image_url"`	`type: "input_image"`
Reasoning	`reasoning_effort`	`reasoning: {effort}`
Stream shape	message deltas	typed events

The streaming difference is the one worth understanding. Chat

completions emits a series of chunks, each carrying a `delta` of message content, terminated by a `[DONE]` sentinel. The responses endpoint instead emits a sequence of *typed events* describing the whole lifecycle of the response — creation, output-item boundaries, content-part boundaries, text deltas, and completion. Kiss listens for exactly the events that matter and discards the rest:

- `response.output_text.delta` — the generated text; its `delta` field is passed to `onToken`.
- `response.completed` — the response finished normally; `onDone` fires.
- `response.incomplete` — the response was cut short (a token limit, for example). Whatever was produced has already been delivered, so `onDone` fires here too rather than leaving the caller waiting.
- `response.failed` and `error` — a failure reported mid-stream; the raw event is retained in `getResponseString()` and an exception is thrown.
- Everything else — progress events carrying nothing the caller needs; ignored.

`onDone` is guarded so it fires exactly once, whichever event ends the stream, and is called unconditionally at the end if no completion event ever arrived. From the caller's side the two endpoints are indistinguishable: the same `Consumer<String>` receives the same token stream, and the same `Runnable` runs once at the end.

24.2.5 Configuration

```
ai.setTemperature(0.7f)     // creativity (0.0 - 2.0)
ai.setSampling(0.7f)        // top-p nucleus sampling
ai.setReasoningEffort("high") // reasoning models only
ai.setImageDetail("auto")   // "low", "high", "auto"
ai.setApi(OpenAI.Api.AUTO) // endpoint selection (default)

// Override the default OpenAI endpoint URLs
// (useful for proxies, gateways, or compatible endpoints)
OpenAI.setUrl(
    "https://my-proxy.example.com/v1/chat/completions")
OpenAI.setResponsesUrl(
```

```
    "https://my-proxy.example.com/v1/responses")
OpenAI.setModelsUrl(
    "https://my-proxy.example.com/v1/models")
```

The static `setUrl()` and `setResponsesUrl()` methods override the chat completions and responses endpoint URLs respectively. Both are global, affecting all current and future `OpenAI` instances, and are primarily intended for proxies, gateways, or OpenAI-compatible endpoints. If your gateway implements only one of the two, pair the override with `setApi()` so no request is ever sent to the other (Section 24.2.4). `setModelsUrl()` likewise overrides the endpoint `getAvailableModels()` calls (Section 24.2.7), independently of which chat endpoint is selected.

24.2.6 Embeddings

OpenAI can generate vector embeddings for text, which are used for semantic search and RAG. Construct the instance with an embeddings model:

```
OpenAI embedder = new OpenAI(apiKey,
    "text-embedding-3-small", false)
double[] embedding = embedder.getEmbeddings(
    "The quarterly report shows revenue growth.")
// embedding is a double array (e.g., 1536 dimensions)
```

24.2.7 Model Listing

`getAvailableModels()` calls the Models API and returns every model id OpenAI reports, unfiltered and in the order given:

```
List<String> models = ai.getAvailableModels()
```

This does not depend on the model configured in the constructor, so it also works on an instance built with a placeholder model name. Override the endpoint with the static, global `OpenAI.setModelsUrl(url)` for testing, proxies, or gateways.

24.3 Anthropic (Claude)

The `Anthropic` class connects to the Claude API.

24.3.1 Construction and Usage

```
import org.kissweb.llm.Anthropic

Anthropic ai = new Anthropic(apiKey, "claude-sonnet-4-6")
ai.setMaxTokens(2000)

String response = ai.send("Summarize this quarter's results.")

// With an image (JPEG, PNG, GIF, or WebP)
String answer = ai.send("What's in this image?",
    "/path/to/image.jpg")
```

24.3.2 Streaming

```
ai.stream("List the top 5 programming languages.",
    // onToken: Consumer<String>
    { token -> print(token) },
    { println("\n--- done ---") })    // onDone: Runnable
```

As with `OpenAI`, errors surface as thrown exceptions rather than through a callback (Section 24.1.1).

24.3.3 Endpoint and Version Configuration

The `Anthropic` class provides static methods for overriding global defaults:

```
// Override the default Anthropic endpoint URL
// (useful for proxies, gateways, or compatible endpoints)
Anthropic.setUrl(
    "https://my-proxy.example.com/v1/messages")

// Override the anthropic-version header value
Anthropic.setVersion("2023-06-01")

// Override the model-listing endpoint used by
// getAvailableModels()
Anthropic.setModelsUrl(
    "https://my-proxy.example.com/v1/models?limit=1000")
```

All three are static methods that affect all current and future `Anthropic` instances globally.

24.3.4 Anthropic Methods

Method	Description
`Anthropic(key, model)`	Constructor with API key and model name.
`send(prompt [, imagePath])`	Send a prompt (optionally with an image), return the complete response.
`stream(prompt [, imagePath], onToken, onDone)`	Stream tokens to a `Consumer<String>`; run a `Runnable` on completion.
`setMaxTokens(n)`	Set maximum response tokens.
`setTemperature(t)`	Set generation randomness (0.0 – 1.0).
`setSampling(p)`	Set top-p nucleus sampling.
`setWebSearch(b)`	Allow the model to search the web (default off; Section 24.1.2).
`getAvailableModels()`	List available model ids, unfiltered and in the order Anthropic reports them.
`setUrl(url)`	Override the API endpoint URL (static, global).
`setVersion(version)`	Override the `anthropic-version` header (static, global).
`setModelsUrl(url)`	Override the model-listing endpoint URL (static, global).
`getResponseCode()`	HTTP status of the last call.
`getResponseString()`	Raw error body of the last failed call.

24.4 OpenRouter (Multi-Provider Gateway)

The `OpenRouter` class connects to OpenRouter (`openrouter.ai`), a cloud gateway that provides access to models from many providers — OpenAI, Anthropic, Google, Meta, Mistral, and others — through a single, OpenAI-compatible API and a single API key. This makes it easy to switch models (or mix providers) without holding an account with each one. Models are identified with a `provider/model`

string, such as `"openai/gpt-4o"`, `"anthropic/claude-sonnet-4"`, or `"meta-llama/llama-3.1-70b-instruct"`.

24.4.1 Construction and Usage

```
import org.kissweb.llm.OpenRouter

OpenRouter ai = new OpenRouter(apiKey, "openai/gpt-4o")

String response = ai.send("Explain SQL joins.")

// With an image (JPEG, PNG, GIF, or WebP)
String answer = ai.send("What's in this image?",
    "/path/to/image.jpg")
```

24.4.2 Streaming

```
ai.stream("Write a haiku about databases.",
    // onToken: Consumer<String>
    { token -> print(token) },
    { println("\n---\nDone.") })      // onDone: Runnable
```

A four-argument overload adds an image path after the prompt. Errors surface as thrown exceptions carrying the HTTP status and OpenRouter's error body (Section 24.1.1).

24.4.3 Model Fallback Routing

OpenRouter can automatically retry a request with alternate models when the primary model is unavailable or its provider fails:

```
OpenRouter ai = new OpenRouter(apiKey,
    "anthropic/claude-sonnet-4")
ai.setFallbackModels("openai/gpt-4o",
    "meta-llama/llama-3.1-70b-instruct")
```

The alternates are tried in order. Note that OpenRouter validates every model ID up front — an invalid ID anywhere in the request is rejected immediately with HTTP 400; fallback engages only between *valid* models when a provider fails at request time.

24.4.4 Reasoning Effort

OpenRouter normalizes reasoning control across providers. When an effort level is set, it is translated to each model's native reasoning mechanism; models without reasoning support ignore it gracefully:

```
ai.setReasoningEffort("high") // "low", "medium", "high"
```

24.4.5 Attribution Headers

OpenRouter can attribute requests to your application in its usage statistics and rankings. Both settings are optional and have no effect on request processing:

```
ai.setSiteUrl("https://example.com") // HTTP-Referer header
ai.setSiteName("My Application")   // X-Title header
```

24.4.6 OpenRouter Methods

Method	Description
`OpenRouter(key, model)`	Constructor with API key and `provider/model` identifier.
`send(prompt [, imagePath])`	Send a prompt (optionally with an image), return the complete response.
`stream(prompt [, imagePath], onToken, onDone)`	Stream tokens to a `Consumer<String>`; run a `Runnable` on completion.
`setTemperature(t)`	Set generation randomness (0.0 – 2.0).
`setSampling(p)`	Set top-p nucleus sampling.
`setReasoningEffort(e)`	Request reasoning at `"low"`, `"medium"`, or `"high"` effort.
`setImageDetail(d)`	Image analysis detail: `"low"`, `"high"`, `"auto"`.
`setFallbackModels(m...)`	Alternate models tried in order on provider failure.
`setSiteUrl(url)`	Optional `HTTP-Referer` attribution header.

Method	Description
`setSiteName(name)`	Optional `X-Title` attribution header.
`setWebSearch(b)`	Enable the web search plugin (default off; Section 24.1.2).
`getAvailableModels()`	List available model ids, unfiltered and in the order OpenRouter reports them; works without an API key.
`setUrl(url)`	Override the API endpoint URL (static, global).
`setModelsUrl(url)`	Override the model-listing endpoint URL (static, global).
`getResponseCode()`	HTTP status of the last call.
`getResponseString()`	Raw error body of the last failed call.

There is no embeddings method — OpenRouter does not offer an embeddings endpoint; use `OpenAI` or `Ollama` for embeddings.

24.5 Ollama (Local Models)

The `Ollama` class connects to a local Ollama instance for on-premise inference without sending data to external APIs.

24.5.1 Construction

```
import org.kissweb.llm.Ollama

// Connect to local Ollama with a specific model
Ollama ai = new Ollama("http://localhost:11434", "llama3")

// Using default localhost URL
Ollama ai = new Ollama("llama3")
```

24.5.2 Sending a Prompt

The `send()` method posts a single prompt to Ollama's `/api/generate` endpoint and returns the complete response as a

string:

```
String response = ai.send("What is the capital of France?")
```

For multi-turn, role-structured conversations — and for function calling — use `chat()` instead, described next.

24.5.3 Multi-Turn Conversations

`send()` posts a single prompt to Ollama's `/api/generate` endpoint with a hand-built prompt string. The `chat()` method instead posts a structured `messages` array to `/api/chat`, and is the better choice for any conversational or instruction-following use:

- **Correct chat templating.** Each instruct model ships its own chat template with special tokens — Llama 3's `<|start_header_id|>`, Mistral's `[INST]...[/INST]`, and so on. `/api/chat` applies the model's registered template automatically. Inlining "User: ... Assistant: ..." into `/api/generate` emits a format the model was not trained on, degrading role separation and instruction-following — subtly on large models, badly on small ones. This is the single biggest quality lever available here.
- **Clean role separation.** System, user, and assistant turns become explicit fields rather than text the model must parse out of one blob. This improves grounding and reduces prompt-injection ambiguity; query-dependent context (recalled facts, retrieved RAG chunks) goes into the system message and can be refreshed each turn.
- **Tool-calling foundation.** `/api/chat` is also where Ollama exposes the `tools` parameter for function calling — the basis for an agentic loop in which the model itself decides when to invoke a tool. Adopting `/api/chat` now is both a quality win and the foundation that form of integration is built on.

Each message is a `Map<String,String>` carrying at least a `role` (`"system"`, `"user"`, or `"assistant"`) and `content`; any other keys are passed through to Ollama unchanged:

```
import org.kissweb.llm.Ollama
```

```
Ollama ai = new Ollama("http://localhost:11434", "llama3")

List<Map<String,String>> messages = [
    [role: "system",
     content: "You are a concise database expert."],
    [role: "user",
     content: "What is a covering index?"]
]
String answer = ai.chat(messages)
```

The request is sent with `stream: false`, so `chat()` returns the assistant message's `content` as a single string.

A second overload accepts a `JSONArray` of tool descriptors, included as the `tools` parameter of the request when non-empty:

```
import org.kissweb.json.JSONArray

JSONArray tools = new JSONArray()
// ... populate with function/tool descriptors ...
String answer = ai.chat(messages, tools)
```

When the model elects to call a tool, the assistant message's `content` is typically empty and the tool calls appear under the response's `tool_calls` field; call `getResponseString()` to inspect the full raw response in that case.

24.5.4 Streaming Responses

The two methods seen so far — `send(prompt)` and `chat(messages)` — are *blocking*. Each posts its request with `stream: false`, so Ollama answers with a single, complete JSON object; the method blocks until the entire reply has been generated, parses that one object, and returns the assistant's text as a `String`. This is the simplest mode and the right choice whenever the caller only needs the finished answer.

Note

The blocking methods deliberately set `stream: false`. Ollama's default *streaming* reply to `/api/generate` is newline-delimited JSON — a sequence of objects (`{...}\n{...}\n...`) rather than one document — which cannot be parsed as a single JSON object. Requesting `stream: false` makes Ollama return exactly one object.

For a live, token-by-token experience — a chat window that fills in as the model "types" — `Ollama` provides a *streaming* overload of each method. Each adds a single `Consumer<String>` parameter, conventionally named `onToken`:

```
// /api/generate, streamed
String send(String prompt,
    Consumer<String> onToken) throws Exception

// /api/chat, streamed
String chat(List<Map<String,String>> messages,
            Consumer<String> onToken) throws Exception
```

These overloads post the request with `stream: true`. Ollama then replies with its newline-delimited JSON stream — one JSON object per line, emitted as each fragment of the answer is produced. Internally both methods share one engine (the private `streamRequest()` helper) that reads the response a line at a time via the framework's `RestClient.streamCall()`, and for every line:

- parses the line as a JSON object; a non-JSON *keep-alive* line is silently skipped;
- if the object carries an `"error"` field, throws an `IOException` reporting it;
- extracts the incremental text chunk and, when it is non-empty, appends it to the running result *and* passes it to `onToken`;
- stops once the object's `"done"` marker is seen.

The two endpoints carry their incremental text in different places, and each overload knows the right one: the `/api/generate` stream (from `send()`) puts each chunk in the top-level `response` field, while the `/api/chat` stream (from `chat()`) puts it in `message.content`.

The caller never has to know the difference — the chunk delivered to `onToken` is the plain text in both cases.

Crucially, each streaming call serves *both* purposes at once: it pushes every chunk to `onToken` as it arrives *and* returns the complete, assembled reply once the stream is done. A caller that wants the live stream and the final string needs only this one call — there is no separate "collect" step.

```
import org.kissweb.llm.Ollama

Ollama ai = new Ollama("http://localhost:11434", "llama3")

// Each token is printed the instant it is produced,
// so the user sees the answer build up in real time.
String full = ai.send("Write a haiku about databases.",
    { chunk -> print(chunk) }) // onToken: Consumer<String>

println()
// `full` also holds the entire haiku.
```

The same shape applies to `chat()`, which is the preferred path for any conversational use (Section 24.5). Here each chunk is appended to a UI element as it lands:

```
List<Map<String,String>> messages = [
    [role: "system",
     content: "You are a concise database expert."],
    [role: "user",
     content: "Explain a covering index in two sentences."]
]

StringBuilder ui = new StringBuilder()
String reply = ai.chat(messages,
    { chunk -> ui.append(chunk) }) // live append

// `reply` is the complete assistant message;
// `ui` has been growing chunk-by-chunk all along.
```

Because the method assembles the same chunks it streams, the returned string is exactly the concatenation of everything handed to `onToken`:

```
StringBuilder collected = new StringBuilder()
String full = ai.send("Name three SQL databases.",
    { chunk -> collected.append(chunk) })
```

```
assert full == collected.toString() // always true
```

Tip

Passing `null` for `onToken` is allowed: the method still consumes the stream and returns the fully assembled text, just without invoking any per-chunk callback. This is handy when streaming is convenient on the wire but the caller only wants the final result.

These streaming overloads are the foundation for live, token-by-token chat interfaces: a backend service can forward each chunk to the browser as it arrives (for example over Server-Sent Events or a websocket) so the user watches the reply appear word by word, exactly as with hosted providers.

24.5.5 Timeouts and Long Generations

A large model, or a long prompt, can take far longer to respond than an ordinary REST call, and two separate timeouts govern how long Kiss will wait. A long generation can trip either one.

The first is on the call from Kiss to the Ollama server. By default `Ollama` allows up to five minutes for a generation to complete — far more than `RestClient`'s ordinary request timeout, precisely because LLM responses are slow. Raise (or lower) it with `setTimeoutSeconds`:

```
Ollama ai = new Ollama("qwen3.5:27b")
ai.setTimeoutSeconds(900) // allow up to 15 minutes
String reply = ai.send("Write a detailed 2000-word report ...")
```

This request-timeout bound applies to the blocking calls (`send` and `chat`); values of zero or less are ignored, so the five-minute default stays in force unless you explicitly raise it. The streaming overloads instead keep reading for as long as the server keeps sending, so they are governed by the liveness of the stream rather than by a single overall deadline.

The second timeout is on the servlet side. A web service that blocks on a slow generation is, by default, aborted by the servlet container after roughly thirty seconds — after which the browser shows "Error communicating with the server." A service that may

run long must lift that limit itself, as described in Section 6.14. The bundled Ollama demo does exactly this, and also lets the timeout be tuned from `application.ini`:

```
void ask(JSONObject injson, JSONObject outjson,
        Connection db, ProcessServlet servlet) {
    // no servlet-side async timeout
    servlet.setTimeout(0)
    Ollama ai = new Ollama()
    String secs = MainServlet.getEnvironment(
        "OllamaTimeoutSeconds")
    if (secs != null && secs.trim().isInteger())
        // optional, per-deployment
        ai.setTimeoutSeconds(secs.trim().toInteger())
    ai.selectModel(injson.getString("model"))
    outjson.put("textResponse", ai.send(injson.getString(
        "prompt")))
}
```

Warning

Both timeouts matter. Lifting only the servlet timeout still leaves the five-minute Ollama bound; lifting only the Ollama timeout still leaves the roughly thirty-second servlet abort. For a service that can run several minutes, raise both.

The `OllamaTimeoutSeconds` key read above lives in `application.ini` and is consulted by the demo service, not by the framework, so a deployment can tune it without recompiling. Streaming, described earlier in this chapter, sidesteps the problem from the other direction: because data flows continuously the connection never sits idle, so a very long reply both stays alive and shows its progress as it arrives.

24.5.6 Embeddings

Ollama can also generate embeddings using compatible models:

```
double[] embedding = ai.getEmbeddings(
    "Customer satisfaction improved this quarter.")
```

24.5.7 Model Management

```
// List available models
List<String> models = ai.getAvailableModels()

// Check if the configured Ollama server is running
boolean running = ai.isOllamaUp()
```

24.5.8 HTML Conversion

Ollama provides a convenience method to convert Markdown responses to HTML:

```
String markdown = ai.send("Explain REST APIs.")
String html = Ollama.toHtml(markdown)
```

24.6 QdrantClient (Vector Database)

The `QdrantClient` class interfaces with a Qdrant vector database for storing and querying embeddings. This is the foundation for Retrieval Augmented Generation (RAG), where relevant documents are retrieved by semantic similarity and injected into LLM prompts.

24.6.1 Collections

A Qdrant collection is a named set of vectors with a fixed dimensionality and a similarity metric (Kiss creates collections using cosine distance). Collections are managed through *static* methods; the server URL defaults to `http://localhost:6333` and can be changed globally with `setUrl()`. A `QdrantClient` *instance*, by contrast, is bound to one collection and is used for the per-point operations that follow.

```
import org.kissweb.llm.QdrantClient

// Optional: point at a non-default server (static, global)
QdrantClient.setUrl("http://localhost:6333")

// Create a collection for OpenAI embeddings (1536 dims)
QdrantClient.createCollection("documents", 1536)

// Bind a client instance to that collection
QdrantClient qc = new QdrantClient("documents")
```

The other collection-level operations are `QdrantClient.listCollections()` and `QdrantClient.deleteCollection(name)`, both static.

24.6.2 Inserting Vectors

`insertOrUpdate()` stores one embedding together with its source text and metadata. Pass `null` as the chunk ID to have a UUID generated; the method returns the ID actually used. The vector's length must match the collection's dimensionality.

```
import org.kissweb.json.JSONObject

// Generate embedding with OpenAI
OpenAI embedder = new OpenAI(apiKey,
    "text-embedding-3-small", false)
double[] embedding = embedder.getEmbeddings(
    "Revenue increased 15% in Q1.")

// Optional extra metadata, merged into the point's payload
JSONObject meta = new JSONObject()
meta.put("source", "quarterly-report.pdf")
meta.put("page", 3)

// chunkId (null => auto UUID), documentId, sequenceNumber,
// vector, text, otherInfo
String chunkId = qc.insertOrUpdate(
    null, "doc-001", 0, embedding,
    "Revenue increased 15% in Q1.", meta)
```

The supplied `text` is stored in the payload under `text`, alongside `document_id` and `sequence_number`; any keys in `otherInfo` are merged into that same payload.

24.6.3 Searching

`search()` queries the client's own collection and returns a `JSONArray` of the nearest points — each a `JSONObject` with `id`, `score`, and `payload`:

```
// Find the 5 most similar points
double[] queryVec = ai.getEmbeddings(
    "How did revenue perform?")
JSONArray results = qc.search(queryVec, 5)

for (int i = 0; i < results.length(); i++) {
```

```
    JSONObject hit = results.getJSONObject(i)
    println("Score: " + hit.getDouble("score"))
    println("Text: " +
        hit.getJSONObject("payload").getString("text"))
}
```

A three-argument overload, `search(vector, limit, filter)`, accepts a Qdrant filter as a `JSONObject` to restrict results by payload fields.

24.6.4 Record Management

```
// Retrieve a specific record by its point ID
JSONObject record = qc.getRecord(chunkId)

// Delete a record
qc.deleteRecord(chunkId)
```

Payload fields can be made filterable by indexing them with `createFieldIndex(fieldName)` (or the two-argument form that takes an explicit field schema).

24.6.5 QdrantClient Methods

Method	Description
`QdrantClient(collection)`	Bind a client to one collection.
`setUrl(url)`	Set the server URL (static, global).
`createCollection(name, dim)`	Create a collection (static, cosine distance).
`listCollections()`	List collection names (static).
`deleteCollection(name)`	Delete a collection (static).
`insertOrUpdate(chunkId, docId, seq, vec, text, otherInfo)`	Upsert a point; returns its ID.

Method	Description
`search(vec, limit)`	Nearest-neighbor search; returns a `JSONArray`.
`search(vec, limit, filter)`	Filtered nearest-neighbor search.
`getRecord(id)`	Retrieve a point as a `JSONObject`.
`deleteRecord(id)`	Delete a point.
`createFieldIndex(field [, schema])`	Index a payload field for filtering.

24.7 RAG Pattern

Retrieval Augmented Generation combines vector search with LLM prompts. The pattern is:

1. **Index** — Split documents into chunks, generate embeddings, store in Qdrant.
2. **Query** — Generate an embedding for the user's question, search Qdrant for relevant chunks.
3. **Augment** — Include the retrieved chunks as context in the LLM prompt.
4. **Generate** — Send the augmented prompt to the LLM for a grounded response.

```
// 1. User asks a question
String question = "What was Q1 revenue?"

// 2. Search for relevant context
OpenAI embedder = new OpenAI(apiKey,
    "text-embedding-3-small", false)
double[] queryVec = embedder.getEmbeddings(question)
QdrantClient qc = new QdrantClient("documents")
JSONArray results = qc.search(queryVec, 3)

// 3. Build augmented prompt
StringBuilder context = new StringBuilder()
```

```
for (int i = 0; i < results.length(); i++) {
    JSONObject hit = results.getJSONObject(i)
    context.append(hit.getJSONObject("payload").getString(
        "text"))
    context.append("\n")
}

// 4. Generate grounded response
OpenAI ai = new OpenAI(apiKey, "gpt-4o", false)
String answer = ai.send(
    "Answer based on this context:\n" + context +
    "\nQuestion: " + question)
```

24.8 Example: A Chatbot Service

This example implements a complete chatbot with a Groovy backend service and a JavaScript frontend.

24.8.1 Backend Service

```
// services/ChatService.groovy
import org.kissweb.llm.OpenAI
import org.kissweb.restServer.MainServlet

class ChatService {

    void sendMessage(JSONObject injson, JSONObject outjson,
                     Connection db, ProcessServlet servlet) {
        String message = injson.getString("message")
        String apiKey = MainServlet.getEnvironment(
            "OpenAIKey")

        OpenAI ai = new OpenAI(apiKey, "gpt-4o", false)
        ai.setTemperature(0.7f)

        String response = ai.send(message)
        outjson.put("response", response)
    }
}
```

24.8.2 Frontend Screen

```
<!-- screens/ChatScreen.html -->
```

```
<div>
    <h2>AI Assistant</h2>
    <div id="chatHistory"
         style="height:400px; overflow-y:auto;
                border:1px solid #ccc; padding:10px;">
    </div>
    <br>
    <text-input id="userMessage"
                style="width:500px;"></text-input>
    <push-button id="sendBtn">Send</push-button>
</div>
```

```
// screens/ChatScreen.js
const WS = 'services.ChatService';

$$('sendBtn').onclick(async () => {
    const message = $$('userMessage').getValue();
    if (!message) return;

    appendMessage('You', message);
    $$('userMessage').clear();

    Utils.waitMessage('Thinking...');
    const res = await Server.call(WS, 'sendMessage',
        { message: message });
    Utils.waitMessageEnd();

    if (res._Success)
        appendMessage('Assistant', res.response);
});

function appendMessage(sender, text) {
    const history = document.getElementById('chatHistory');
    const div = document.createElement('div');
    div.innerHTML = '<b>' + sender + ':</b> ' + text;
    history.appendChild(div);
    history.scrollTop = history.scrollHeight;
}
```

Tip

Store API keys in `application.ini` and access them via `MainServlet.getEnvironment()`. Never hard-code API keys in service files.

Note

The LLM classes are provider-neutral wrappers. Switching among OpenAI, Anthropic, OpenRouter, and Ollama requires changing only the constructor — the `send()` and `stream()` methods have the same signature across all providers. OpenRouter is a convenient way to trial many models from different vendors before committing to one, since only the model string changes.

Chapter 25

Interfacing with External Systems

A typical Kiss application is more than a self-contained box: it calls third-party APIs, accepts inbound webhooks, and increasingly both exposes tools to AI assistants and consumes tools from them. This chapter covers the seven classes the framework provides for that work:

- `RestClient` — make HTTP calls *out* to external services.
- `RestServerBase` — accept HTTP calls *in* from external services that don't speak Kiss's JSON-RPC.
- `RestServerExample` — a working template that extends `RestServerBase`.
- `MCPServerBase` — expose the application as a Model Context Protocol (MCP) server, the standard way for AI assistants to discover and invoke tools.
- `MCPServerExample` — a working template that extends `MCPServerBase`.
- `MCPClientBase` — consume tools from a *remote* MCP server, the mirror image of `MCPServerBase`.
- `MCPClientExample` — a working template that extends `MCPClientBase`.

The three `*Base`/`*Example` pairs are intentionally parallel: `*Base` carries the protocol boilerplate and lives in `src/main/core/`; `*Example` is a copyable template under `src/main/precompiled/` that you adapt for each endpoint or remote server you work with. The `RestClient` side is symmetric — it is the same set of operations any HTTP client offers, just bound to Kiss's JSON utilities. The server side has more nuance, because there is already a built-in JSON-RPC entry point (`/rest`) used by the Kiss frontend; the classes here are for the cases where that is not the protocol the caller is using.

Note

Use a normal Kiss JSON-RPC service (Chapter 6) for everything called from the Kiss frontend. Reach for the classes in this chapter only when an *external*, non-Kiss client is the caller — typically because that client demands a specific URL, HTTP method, header set, authentication scheme, or response format that does not fit Kiss's JSON-RPC convention.

25.1 RestClient — Calling External APIs

`RestClient` is an HTTP client for making REST calls to external APIs. It supports JSON, string, XML, and streaming response types.

```
import org.kissweb.RestClient

// JSON GET request
JSONObject result = RestClient.jsonCall(
    "GET", "https://api.example.com/users/123")

// JSON POST with body
JSONObject body = new JSONObject()
body.put("name", "Alice")
body.put("email", "alice@example.com")
JSONObject result = RestClient.jsonCall(
    "POST", "https://api.example.com/users", body)

// String response
String html = RestClient.strCall(
    "GET", "https://example.com/page")

// XML response
Document xml = RestClient.xmlCall(
```

```
    "GET", "https://api.example.com/data.xml")

// Streaming response
InputStream stream = RestClient.streamCall(
    "GET", "https://example.com/large-file.zip")
```

Inside a Kiss service the same pattern applies. The HTTP call is just a piece of the service's normal logic:

```
import org.kissweb.RestClient

void fetchWeather(JSONObject injson, JSONObject outjson,
                  Connection db, ProcessServlet servlet) {
    String city = injson.getString("city")
    RestClient rc = new RestClient()
    JSONObject result = rc.jsonGet(
        "https://api.example.com/weather?city=" + city)
    outjson.put("temperature", result.getDouble("temp"))
}
```

25.1.1 Advanced Configuration

```
RestClient client = new RestClient()

// Proxy support
client.setProxy("proxy.corp.com", 8080)

// TLS client certificate
client.setTLSKey("keystore.p12", "password")

// Retry policy for transient failures
client.setRetryPolicy(3, 1000) // 3 retries, 1s delay
```

Method	Description
`jsonCall(method, url)`	HTTP call, parse JSON response.
`jsonCall(method, url, body)`	HTTP call with JSON body.
`strCall(method, url)`	HTTP call, return string.
`xmlCall(method, url)`	HTTP call, parse XML response.
`streamCall(method, url)`	HTTP call, return InputStream.
`setProxy(host, port)`	Configure HTTP proxy.
`setTLSKey(path, pw)`	Set client certificate.

Method	Description
`setRetryPolicy(n, delay)`	Set retry count and delay.

25.2 RestServerBase — Receiving External REST Calls

When an *external* system needs to call into your application (a webhook from a payment processor, an inbound notification from a third-party API, a status callback from a remote job, etc.), that system will not speak Kiss's JSON-RPC protocol. For these cases the framework provides a thin base class, `org.kissweb.RestServerBase`, that lets you expose a conventional servlet-style REST endpoint.

25.2.1 What `RestServerBase` Provides

`RestServerBase` extends `HttpServlet` and adds a small set of static helpers so the endpoint code stays focused on application logic:

- `basicAuthenticate(request, response, realm, user, password)` — validates an HTTP Basic `Authorization` header against an expected username and password, and sets a 401 response on failure.
- `getBodyString(request)` — read the full request body as a `String`.
- `getBodyJson(request)` — read the body as a `JSONObject` (returns `null` if the content type is not `application/json`).
- `getBodyBinary(request)` — read the body as a `byte[]`.
- `setResponse(response, jsonObject)` — write a JSON response with the correct content type and UTF-8 encoding.
- `setResponse(response, contentType, string)` and `setResponse(response, contentType, byte[])` — write text or binary responses.
- `isTextContentType(contentType)` / `isBinaryContentType(contentType)` — best-guess classification of an HTTP content type.

Override any of `doGet`, `doPost`, `doPut`, or `doDelete` to handle the corresponding HTTP method; everything else follows standard Jakarta Servlet conventions.

25.2.2 Configuration — No `web.xml` Required

The `@WebServlet` annotation is the *only* configuration required to register the endpoint. The URL the annotation declares (e.g. `/myservice`) is appended to the application's context path; no `<servlet>` or `<servlet-mapping>` entry in `web.xml` is needed. Servlet 6.x annotation scanning discovers the class at startup automatically.

Note

The framework's `src/main/core/WEB-INF/web.xml` only declares the framework's startup listener and a global CORS filter mapped to `/*`. Your endpoint inherits CORS handling for free; you do not add anything to `web.xml` when adding new external endpoints.

25.2.3 Why It Must Live Under `precompiled/`

Servlet annotation scanning happens once, when the servlet container starts. Classes that the container needs to discover at startup must therefore be compiled into the WAR ahead of time — they cannot be supplied by Kiss's runtime compiler, which only loads files under `src/main/backend/` on demand. For this reason, every class extending `RestServerBase` must be placed under `src/main/precompiled/`, where `./bld -v build` compiles it into `work/exploded/WEB-INF/classes/` before Tomcat starts.

The practical consequence: after adding or modifying a `RestServerBase` subclass, run `./bld -v build` and restart the server. The hot-reload behavior that applies to backend services does *not* apply here.

Tip

`@MultipartConfig` is only needed when the endpoint accepts `multipart/form-data` uploads. Omit it for plain JSON or text endpoints.

25.3 RestServerExample — The Template

A working template lives at `src/main/precompiled/com/mycompany/RestServerExample.java`. Copy it, rename it, and edit it for each external endpoint you need to expose.

```
package com.mycompany;

import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.annotation.MultipartConfig;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.kissweb.RestServerBase;
import org.kissweb.json.JSONObject;
import java.io.IOException;

@WebServlet(urlPatterns="/myservice")
@MultipartConfig
public class RestServerExample extends RestServerBase {

    @Override
    protected void doPost(HttpServletRequest request,
            HttpServletResponse response) throws IOException {
        if (!basicAuthenticate(request, response, null,
                               "username", "password"))
            return;

        JSONObject in = getBodyJson(request);
        // ... process the request ...
        JSONObject out = new JSONObject();
        out.put("ok", true);
        setResponse(response, out);
    }
}
```

The shipped example also stubs out `doGet`, `doPut`, and `doDelete` as override points; delete the methods you don't need.

25.4 MCPServerBase — Exposing Tools to AI Assistants

The Model Context Protocol (MCP) is the open protocol AI assistants such as Claude Desktop and Claude Code use to talk to external tools and data. An MCP server advertises a list of tools (each with a JSON-Schema for its arguments); the assistant chooses when to

call them and feeds the results back into the model's reasoning.

`MCPServerBase` is the framework's MCP server base class. It extends `RestServerBase`, so it inherits the same body-reading and response-writing helpers, the same `@WebServlet`-based registration model, and the same precompiled-directory requirement. On top of that it implements all of the wire-protocol boilerplate — the JSON-RPC 2.0 envelope, the `initialize` handshake, `ping`, notifications, and error replies — so a subclass only has to declare its tools and run them.

25.4.1 The Wire Protocol

Every MCP exchange is a JSON-RPC 2.0 message:

```
// request
{ "jsonrpc": "2.0", "id": 1, "method": "tools/list",
    "params": {} }

// response
{ "jsonrpc": "2.0", "id": 1, "result": { "tools": [ ... ] } }

// notification (no id, no response expected)
{ "jsonrpc": "2.0", "method": "notifications/initialized" }
```

`MCPServerBase.doPost` parses the envelope, dispatches on the `method` field, and either returns a JSON response (for requests) or a 202 Accepted with no body (for notifications). It is declared `final` — subclasses extend behavior through hooks rather than by overriding the dispatcher.

25.4.2 The Three Methods That Matter

A minimum-viable MCP server implements three methods, all of which `MCPServerBase` dispatches automatically:

- `initialize` — handshake. Client sends its protocol version and info; server responds with its protocol version, server info, and *capabilities* (which feature areas it supports).
- `tools/list` — catalog. Server returns an array of tools, each with `name`, `description`, and an `inputSchema` (a JSON Schema describing the expected arguments). The model uses these to decide when and how to call each tool.

- `tools/call` — invocation. Server runs the named tool with the supplied arguments and returns a list of *content blocks* (most commonly a single `type:"text"` block).

`MCPServerBase` also handles `ping` (the MCP keep-alive) and the standard `notifications/*` client-to-server notifications without any subclass involvement.

25.4.3 Subclass Hooks

A subclass implements four required hooks and may override three optional ones:

Method	Purpose
Required	
`getServerName()`	Server name reported during `initialize`.
`getServerVersion()`	Server version reported during `initialize`.
`listTools()`	Return the tool catalog as a `JSONArray`.
`callTool(name, args)`	Execute a tool, return its result object.
Optional	
`authenticate(req, resp)`	Gate every request. Default: allow all.
`getCapabilities()`	Capabilities advertised at `initialize`. Default: tools only.
`dispatchExtension(method, params)`	Handle methods beyond the built-in set (`resources/list`, `prompts/get`, etc.). Default: return null, which produces a method-not-found error.

25.4.4 Helpers

`MCPServerBase` exposes static helpers for the common shapes of MCP data:

- `textBlock(text)` — builds a `{"type":"text","text":...}` content block.

- `toolResult(text)` — builds a complete `tools/call` result containing one text block.
- `toolError(message)` — builds a `tools/call` result with `isError: true`. Use this for tool-specific failures the model should see and react to (see below).
- `buildSchema(title, required, names, types, descriptions)` — builds a JSON-Schema object for a tool's `inputSchema` from parallel arrays.

The class also exposes JSON-RPC error code constants (`ERR_PARSE`, `ERR_INVALID_REQUEST`, `ERR_METHOD_NOT_FOUND`, `ERR_INVALID_PARAMS`, `ERR_INTERNAL`) and a protected `sendError` method for extension handlers that need to fail fast.

25.4.5 Tool Errors vs. Protocol Errors

There are two distinct error channels and they mean different things:

- *Protocol errors* — malformed JSON, unknown method, missing parameters — are returned as JSON-RPC errors (`-32700` parse, `-32600` invalid request, `-32601` method not found, `-32603` internal). The client treats these as transport failures.
- *Tool errors* — "the database lookup failed," "the file you asked about doesn't exist" — are returned as a normal `tools/call` *result* with `isError: true`. The model *sees* the failure and can react to it; it does not see protocol errors.

Use `toolError` (the latter channel) for anything you want the AI to be able to recover from, retry, or report back to the user. Anything thrown out of `callTool` becomes a JSON-RPC `ERR_INTERNAL` response, which the model does not see.

25.4.6 What MCPServerBase Does Not Implement

To stay focused, the base class omits several MCP features. All of them layer on top of the same JSON-RPC envelope and can be added by overriding `dispatchExtension` and `getCapabilities`:

- *Resources* — read-only data the model can request (files, database rows, etc.). Implement `resources/list` and `resources/read`; advertise the `resources` capability.

- *Prompts* — reusable prompt templates the host UI can offer to the user. Implement `prompts/list` and `prompts/get`; advertise the `prompts` capability.
- *Sampling* — server-initiated requests asking the client's model to generate something on the server's behalf.
- *SSE streaming* — responses streamed back as Server-Sent Events instead of a single JSON body. Useful for long-running tools that produce incremental output.
- *Sessions* — the `Mcp-Session-Id` header lets a server keep per-client state across requests.

25.5 MCPServerExample — The Template

A working template lives at `src/main/precompiled/com/mycompany/MCPServerExample.java`. It extends `MCPServerBase`, exposes an endpoint at `/mcp`, and ships with two example tools (`echo` and `add`). Because all of the protocol work is in the base class, the entire example is just the four hooks:

```
@WebServlet(urlPatterns="/mcp")
public class MCPServerExample extends MCPServerBase {

    @Override protected String getServerName() {
        return "kiss-mcp-example";
    }
    @Override protected String getServerVersion() {
        return "1.0.0";
    }

    @Override
    protected JSONArray listTools() {
        JSONArray tools = new JSONArray();

        JSONObject add = new JSONObject();
        add.put("name", "add");
        add.put("description",
            "Add two numbers and return the sum.");
        add.put("inputSchema", buildSchema(
                "Add input",
                // required
                new String[]{"a", "b"},
                new String[]{"a", "b"},
                new String[]{"number", "number"},
```

```
                new String[]{"First addend",
                    "Second addend"}));
        tools.put(add);
        // ... echo tool elided ...
        return tools;
    }

    @Override
    protected JSONObject callTool(String name,
        JSONObject args) {
        switch (name) {
            case "add": {
                double a = args.getDouble("a", 0.0);
                double b = args.getDouble("b", 0.0);
                return toolResult(String.valueOf(a + b));
            }
            // ... echo case elided ...
            default:
                return toolError("Unknown tool: " + name);
        }
    }
}
```

The two methods stay in sync by convention: every entry in `listTools` needs a matching `case` in `callTool`, and vice versa.

25.5.1 Build and Authentication

Like `RestServerExample`, `MCPServerExample` relies on annotation scanning, lives under `src/main/precompiled/`, and requires a build + restart to pick up changes:

```
./bld -v build
```

The shipped example does not authenticate. Most MCP clients support adding an `Authorization` header to outgoing requests, so basic auth drops in by overriding `authenticate`:

```
@Override
protected boolean authenticate(HttpServletRequest req,
                               HttpServletResponse resp) {
    return basicAuthenticate(req, resp, null, "user",
        "secret");
}
```

For OAuth 2.1 bearer-token authentication — which the

MCP specification mandates — no override is needed. Set `OAuthAuthorizationServer` in `application.ini` and every `MCPServerBase` subclass automatically validates incoming tokens against the configured authorization server. Chapter 26 covers this in detail.

25.6 MCPClientBase — Consuming a Remote MCP Server

Where `MCPServerBase` *exposes* tools to an AI assistant, `MCPClientBase` sits on the other end of the same wire: it *consumes* the tools a remote MCP server offers. It is the mirror image of the server base class — the two speak the identical JSON-RPC 2.0 protocol (protocol version `2025-06-18`) and interoperate directly, so one Kiss app can call another Kiss app's `MCPServerBase` endpoint with nothing between them but HTTP.

Two differences from the server side are worth noting up front:

- It is *not* a servlet. There is no `@WebServlet` annotation and no inbound URL; you *instantiate or subclass* it from a backend service and call its methods. Because it does not depend on annotation scanning, `MCPClientBase` itself lives in `src/main/core/` as ordinary framework code rather than under `precompiled/`. (The shipped *template* still lives under `precompiled/` so it sits beside the other examples.)
- HTTP is done through `RestClient` (Section 25.1), so timeouts, proxying, and TLS behave as they do for any other outbound call.

25.6.1 The Lazy Handshake

MCP requires a client to perform an `initialize` handshake — and send a `notifications/initialized` follow-up — before any other request. `MCPClientBase` runs this automatically the first time you call any operation, so the common case needs no ceremony:

```
import org.kissweb.MCPClientBase
import org.kissweb.restServer.MainServlet

class MyMcpClient extends MCPClientBase {
    protected String getServerUrl() {
```

```
        return (String) MainServlet.getEnvironment(
            "RemoteMcpUrl")
    }
}

MyMcpClient client = new MyMcpClient()
// runs initialize on first use
JSONArray tools = client.listTools()
String reply = client.callToolText("echo",
        new JSONObject().put("message", "hi"))
```

Call `initialize()` explicitly only when you want the server's advertised capabilities — `getServerInfo()`, `getServerCapabilities()`, `getNegotiatedProtocolVersion()` — before making any tool calls.

Tip

`MainServlet.getEnvironment()` returns `Object`, not `String`. Groovy assigns it to a `String` variable without complaint, but a `Java` subclass of `MCPClientBase` must cast it: `return (String) MainServlet.getEnvironment("RemoteMcpUrl");`

25.6.2 Subclass Hooks

A subclass implements one required hook and may override four optional ones:

Method	Purpose
Required	
`getServerUrl()`	Full URL of the remote MCP endpoint. Per Kiss convention, read from `application.ini`.
Optional	
`getClientName()`	Client name reported during `initialize`. Default provided.
`getClientVersion()`	Client version reported during `initialize`. Default provided.
`getOAuthProviderName()`	Name of an `[OAuthClient` *`name`*`]` provider to obtain bearer tokens from. Default: none.

Method	Purpose
`applyAuthentication(headers)`	Attach authentication headers to each request. Default: an OAuth bearer token when a provider name is set, otherwise nothing.
`getClientCapabilities()`	Client capabilities sent at `initialize`. Default: none.

25.6.3 Operations

Method	Description
`initialize()`	Run the handshake (once); return the server's `initialize` result.
`listTools()`	Fetch the tool catalog as a `JSONArray`; `nextCursor` pagination is followed automatically.
`callTool(name, args)`	Call a tool; return the raw result object.
`callToolText(name, args)`	Call a tool and return its text content; throws on a tool error.
`ping()`	MCP keep-alive.
`call(method, params)`	Arbitrary JSON-RPC request, for methods without a wrapper.
`notify(method, params)`	Arbitrary JSON-RPC notification.
`setTimeouts(connect, request)`	Connect and request timeouts (defaults 30 s / 60 s).

25.6.4 Reading Tool Results

A `tools/call` result is a list of content blocks plus an optional `isError` flag. Two static helpers cover the common cases:

- `isError(result)` — `true` if the tool reported `isError: true`.
- `textOf(result)` — the `text` of every `type:"text"` block, joined with newlines (non-text blocks are skipped).

`callToolText` combines them: it returns `textOf(result)` but throws `MCPClientException` when `isError` is set — the convenient "give me the text, fail on error" path. Use `callTool` plus the helpers when you need the raw blocks or want to handle a tool error without an exception.

25.6.5 Tool Errors vs. Protocol Errors

This is the client-side mirror of the two error channels described for the server (Section 25.4). A *protocol* failure — a JSON-RPC error from the server, a non-2xx HTTP status, an unparseable body, or a transport failure — is raised as `MCPClientBase.MCPClientException` (a `RuntimeException`); when the failure carries a JSON-RPC error code it is available via `getCode()` (otherwise `0`). A *tool* error the model is meant to see arrives instead as a normal result with `isError: true`, which `callTool` returns without throwing.

25.6.6 Authentication

Authentication is the symmetric counterpart of the server's. The server *validates* incoming bearer tokens (the OAuth 2.1 resource-server role); the client *obtains* them (the relying-party role, Chapter 28). Point the client at a configured `[OAuthClient name]` provider by overriding `getOAuthProviderName()`, and every request carries a transparently-refreshed `Authorization: Bearer` header — discovery, dynamic registration, and token refresh are all handled by the OAuth client.

When no usable token is held and an interactive login is required, `OAuthAuthorizationRequiredException` propagates out of the call. That is a normal control-flow signal, not a failure: catch it where you make the call and start the browser redirect flow.

```
import org.kissweb.oauth.client.OAuthAuthorizationRequiredException
import org.kissweb.oauth.client.OAuthClient

try {
    reply = client.callToolText("echo", args)
} catch (OAuthAuthorizationRequiredException e) {
    String url = OAuthClient.forProvider("myprovider")
            .beginAuthorization("http://localhost:8001")
    // redirect the browser to 'url'; retry after the
    // callback completes.
}
```

For any other scheme — HTTP basic auth, a static API token, a custom header — override `applyAuthentication(headers)` directly instead:

```
@Override
protected void applyAuthentication(JSONObject headers) {
    headers.put("Authorization", "Bearer "
            + (String) MainServlet.getEnvironment(
                "RemoteMcpToken"));
}
```

25.6.7 What MCPClientBase Does Not Implement

Mirroring the server base class, the client omits the same set of features, each of which can be added incrementally on top of the same JSON-RPC envelope: resources, prompts, sampling, server-initiated requests, SSE (`text/event-stream`) streaming responses, and `Mcp-Session-Id` session management. (Because `RestClient` surfaces only the response body, not its headers, session management in particular would require reading response headers.)

25.7 MCPClientExample — The Template

A working template lives at `src/main/precompiled/com/mycompany/MCPClientExample.java`. It extends `MCPClientBase`, reads the remote URL from `application.ini`, and provides a static `demo()` helper that lists the remote server's tools and calls its `echo` tool — the natural counterpart to the `echo` tool in `MCPServerExample`:

```
package com.mycompany;

import org.kissweb.MCPClientBase;
import org.kissweb.json.JSONArray;
import org.kissweb.json.JSONObject;
import org.kissweb.restServer.MainServlet;

public class MCPClientExample extends MCPClientBase {

    @Override
    protected String getServerUrl() {
        return (String) MainServlet.getEnvironment(
            "RemoteMcpUrl");
```

```
    }

    public static String demo() {
        MCPClientExample client = new MCPClientExample();

        // runs initialize
        JSONArray tools = client.listTools();
        JSONObject args = new JSONObject();
        args.put("message", "Hello from the Kiss MCP client");
        return client.callToolText("echo", args);
    }
}
```

The template lives under `precompiled/` so it ships beside `MCPServerExample`; like any class there it is compiled by `./bld -v build` and requires a restart to pick up changes. The file also carries commented-out blocks showing how to wire OAuth authentication (via `getOAuthProviderName()`) or a custom header (via `applyAuthentication()`), as described above. Call `demo()` from any backend service:

```
String reply = com.mycompany.MCPClientExample.demo()
```

Chapter 26

OAuth 2.1 Resource Server

Kiss can act as an *OAuth 2.1 resource server* — validating bearer tokens issued by an external authorization server (Auth0, Okta, Keycloak, Microsoft Entra, Google, or any other RFC 8414 / OpenID Connect compliant provider). This is the half of the OAuth picture that protects an API; the other half (issuing tokens, consent UI, refresh-token handling) is deliberately delegated to a dedicated authorization server.

The primary motivation is the Model Context Protocol. The MCP specification mandates OAuth 2.1 bearer-token authentication, and almost every production MCP server needs to plug into the customer's existing identity provider. Kiss makes this automatic: when the authorization-server URL is set in `application.ini`, every `MCPServerBase` subclass (Chapter 25) starts requiring a valid bearer token on every request, with no code change to the MCP servlet.

The same machinery works for any other servlet that wants OAuth-protected endpoints, MCP or not.

This chapter and the next cover the two server-side roles — validating tokens here, issuing them in Chapter 27. The complementary client role, in which Kiss instead *obtains* tokens from a remote OAuth server and presents them on outbound calls, is covered in Chapter 28.

26.1 What the Resource Server Does

When `OAuthAuthorizationServer` is configured, the framework does five things on every protected request:

1. Parses the `Authorization: Bearer <jwt>` header.

2. Verifies the JWS signature against the authorization server's published JWKS (JSON Web Key Set).

3. Validates the JWT claim set: `iss` matches the configured issuer, `aud` contains the configured resource identifier, `exp` is in the future, `nbf` (if present) is not in the future.

4. Checks that every scope listed in `OAuthRequiredScopes` is present in the token's `scope` or `scp` claim.

5. Returns a 401 (or 403 for an insufficient-scope token) with the proper RFC 6750 / RFC 9728 `WWW-Authenticate` challenge on failure, or makes the validated token available to the rest of the request on success.

It also publishes a discovery document at `/.well-known/oauth-protected-resource` per RFC 9728, so MCP clients can bootstrap their authentication flow.

Note

The resource-server role validates tokens that have already been issued. Token issuance, consent UI, client registration, and refresh-token handling are the authorization server's job. You can plug the resource-server in front of any third-party authorization server (Auth0, Okta, Keycloak, Microsoft Entra, Google, Hydra, Authentik), or you can run Kiss's own embedded authorization server in the same JVM — the common MCP deployment shape, covered in Chapter 27.

26.2 Configuration

All settings live in `src/main/backend/application.ini` and are read at startup via `MainServlet.getEnvironment()`. Only one key contains a URL — the others either default to that URL or to discovery against it — so for the common deployment shape where the resource server and the authorization server are the same Kiss app, the issuer URL appears once in the file and no other URL-bearing key needs to be set:

```
# Required to enable OAuth. Issuer URL of the
# authorization server.
```

```
OAuthAuthorizationServer = https://myapp.example.com

# Optional. Canonical URL of this resource server
# (the "aud" value tokens must carry). Defaults to
# OAuthAuthorizationServer, which is correct when
# this Kiss app is both AS and RS. Set this
# explicitly only when the RS is hosted at a URL
# different from the AS.
# OAuthResourceIdentifier = https://myapp.example.com

# Optional. Comma- or space-separated scopes every
# token must carry. Tokens missing any of these get
# HTTP 403 with error="insufficient_scope".
# OAuthRequiredScopes = mcp:read mcp:write

# Optional. Explicit JWKS URL. If omitted, the
# framework discovers it from
# <OAuthAuthorizationServer>/.well-known/
#     oauth-authorization-server
# (RFC 8414, the OAuth 2.1 standard), falling back
# to <OAuthAuthorizationServer>/.well-known/
#     openid-configuration
# for OIDC-only authorization servers. Set this only
# when neither metadata document is published.
# OAuthJwksUri =
#     https://myapp.example.com/.well-known/jwks.json

# Optional. How long to cache JWKS keys before
# refetching, in seconds. Default 3600.
# OAuthJwksCacheSeconds = 3600

# Optional. Comma-separated list of acceptable JWS
# algorithms. Supported: RS256, RS384, RS512.
# Default: RS256.
# OAuthAllowedAlgorithms = RS256

# Optional. Allowed clock skew when validating
# exp/nbf, in seconds. Default: 60.
# OAuthClockSkewSeconds = 60
```

If `OAuthAuthorizationServer` is absent or blank, the OAuth subsystem is *inert*: the discovery endpoint returns 404, MCP servers accept all requests, and no JWKS fetch is ever attempted. Applications that don't use OAuth pay no runtime cost.

Tip

Most authorization servers publish their issuer URL with a path component (Auth0 uses `https://tenant.auth0.com/`, Keycloak uses `https://kc.example/realms/myrealm`). Set `OAuthAuthorizationServer` to the *exact* value of the `iss` claim in your tokens, modulo a trailing slash (which the validator normalizes).

26.3 Automatic Enforcement for MCP Servers

The default `authenticate()` method in `MCPServerBase` consults `OAuthConfig`. If OAuth is configured, it validates the bearer token; if not, it allows the request (the historical default). So once `OAuthAuthorizationServer` is set, *every* MCP servlet — including `MCPServerExample` and any subclass you write — starts requiring valid bearer tokens without a single line of Java being touched.

The behavior matrix:

No `Authorization` header
: 401 with bare `WWW-Authenticate: Bearer realm="...", resource_metadata="..."`. Per RFC 6750 §3.1, no `error` parameter is included when the client did not attempt authentication.

Non-Bearer scheme (e.g. `Basic ...`)
: Treated identically to no header — the client did not attempt the expected scheme.

Malformed JWT, bad signature, expired, wrong issuer, wrong audience
: 401 with `error="invalid_token"` and a human-readable `error_description`.

Valid JWT missing a required scope
: 403 with `error="insufficient_scope"`.

Valid JWT with all required scopes
: Request proceeds; the validated token is available to tool methods via `BearerTokenValidator.currentToken()`.

The `resource_metadata` parameter in the `WWW-Authenticate` header points the client at the discovery endpoint (see Section 26.5) so it knows where to obtain a token.

26.4 Accessing the Validated Token

Inside any tool method — or anywhere downstream of `authenticate()` — you can read the validated token with a static accessor:

```
import org.kissweb.oauth.BearerTokenValidator;
import org.kissweb.oauth.ValidatedToken;

@Override
protected JSONObject callTool(String name, JSONObject args) {
    ValidatedToken tok = BearerTokenValidator.currentToken();
    String userId = tok.getSubject();      // "sub"
    String email = tok.getClaimString("email");

    if (!tok.hasScope("mcp:write"))
        return toolError(
            "This tool requires the mcp:write scope.");

    // ... your tool logic ...
}
```

The `ValidatedToken` sits in a `ThreadLocal` for the duration of the request and is cleared automatically in `MCPServerBase.doPost`'s `finally` block, so a validated token cannot leak onto a subsequent request that reuses the same servlet-container thread.

26.4.1 ValidatedToken Methods

`getSubject()`
: The `sub` claim — the principal the token represents.

`getIssuer()`
: The `iss` claim.

`getAudience()`
: The `aud` claim values, as a `Set<String>` (one entry if the claim was a string, more if it was an array).

`getScopes()`
: Granted scopes, parsed from either the `scope` claim (space-

separated string) or the `scp` claim (array) — whichever the authorization server uses.

`hasScope(String)`
Convenience scope check.

`getExpiresAtEpochSeconds()`
The `exp` claim as Unix epoch seconds.

`getIssuedAtEpochSeconds()`
The `iat` claim as Unix epoch seconds.

`getClaimString(String)`
Convenience accessor for any other string-valued claim (`email`, `preferred_username`, custom claims, etc.).

`getClaims()`
The full verified claim set as a `JSONObject`. Use for anything not exposed by the named getters.

26.5 Discovery Endpoint

The framework auto-registers a servlet at `/.well-known/oauth-protected-resource` via `@WebServlet` annotation scanning. When OAuth is enabled, this endpoint returns the RFC 9728 metadata document:

```
{
  "resource": "https://myapp.example.com",
  "authorization_servers": ["https://myapp.example.com"],
  "bearer_methods_supported": ["header"],
  "scopes_supported": ["mcp:read", "mcp:write"],
  "resource_signing_alg_values_supported": ["RS256"]
}
```

MCP clients call this endpoint after they see a 401 with `WWW-Authenticate: Bearer ..., resource_metadata="..."`. They use the returned `authorization_servers` value to start an OAuth flow against the right authorization server.

In the common single-Kiss-app deployment, `resource` and `authorization_servers` carry the same value (the issuer URL); in a split deployment where the resource server lives at a different URL from the authorization server, `resource` reflects

`OAuthResourceIdentifier` and `authorization_servers` reflects `OAuthAuthorizationServer`.

When OAuth is *not* configured, the endpoint returns 404 — the same response any non-OAuth resource would give, signaling to clients that this resource is not protected by OAuth.

26.6 Customizing Authentication

The default OAuth-when-configured behavior is right for almost all cases. Override `authenticate()` only when you need different semantics for a particular MCP servlet.

26.6.1 An Explicitly Public MCP Server

Suppose OAuth is configured globally but you have one MCP server that should stay open (a public catalog, a development sandbox, an internal-only server behind a firewall):

```
@Override
protected boolean authenticate(HttpServletRequest req,
                               HttpServletResponse resp) {
    return true;
}
```

26.6.2 A Different Authentication Scheme

To use HTTP basic auth or a custom token scheme instead of OAuth:

```
@Override
protected boolean authenticate(HttpServletRequest req,
        HttpServletResponse resp) throws IOException {
    return basicAuthenticate(req, resp, "MyRealm", "user",
        "secret");
}
```

26.6.3 OAuth Plus Extra Checks

To layer additional logic on top of the OAuth default — for example, multi-tenant routing where the URL path must match a claim in the token:

```
@Override
protected boolean authenticate(HttpServletRequest req,
        HttpServletResponse resp) throws IOException {
    if (!super.authenticate(req, resp))
        return false; // 401/403 already written
    ValidatedToken tok = BearerTokenValidator.currentToken();
    String tenant = tok.getClaimString("tenant");
    if (!tenant.equals(expectedTenant(req))) {
        resp.setStatus(HttpServletResponse.SC_FORBIDDEN);
        return false;
    }
    return true;
}
```

26.7 Using OAuth from Non-MCP Servlets

`BearerTokenValidator` is not MCP-specific. Any servlet — including `RestServerBase` subclasses (Section 25.2) and your own `HttpServlets` — can validate bearer tokens by calling `requireBearerToken`:

```
import org.kissweb.oauth.BearerTokenValidator;
import org.kissweb.oauth.ValidatedToken;

@WebServlet(urlPatterns = "/api/secret")
public class SecretApi extends RestServerBase {
    @Override
    protected void doGet(HttpServletRequest req,
            HttpServletResponse resp) throws IOException {
        try {
            ValidatedToken tok =
                BearerTokenValidator.requireBearerToken(
                    req, resp);
            if (tok == null)
                return;          // 401/403 already written
            // ... protected endpoint logic ...
        } finally {
            BearerTokenValidator.clearCurrentToken();
        }
    }
}
```

Warning

Outside of `MCPServerBase`, the `ThreadLocal` that holds the validated token is *not* cleared automatically. Always wrap the request handler in a `try / finally` and call `BearerTokenValidator.clearCurrentToken()` in the `finally` block to prevent the token from leaking onto a subsequent request that reuses the same servlet thread.

For programmatic validation without HTTP machinery (e.g., from a background job that receives a token via another channel), call `BearerTokenValidator.validate(String)` directly. It throws `TokenValidationException` on a validation failure, `InsufficientScopeException` if required scopes are missing, and `IOException` if the authorization server's JWKS cannot be fetched.

26.8 JWKS Caching and Key Rotation

The framework fetches the authorization server's JWKS lazily on the first request and caches each public key by its `kid` (key id). Two events trigger a refetch:

1. *TTL expiry.* By default keys are cached for one hour (`OAuthJwksCacheSeconds`). Adjust if your authorization server rotates keys more aggressively.

2. *Unknown `kid`.* When a token arrives whose `kid` is not in the cache, the framework treats this as a possible key rotation, refetches the JWKS under a lock, and re-checks. Only one thread fetches at a time, even under concurrent load.

This handles routine key rotation — issuer adds a new key, lets both keys coexist while old tokens drain, then removes the old key — without operator intervention.

26.8.1 Resolving the JWKS URI

The framework needs to know *where* to fetch the JWKS before it can fetch and cache it. Resolution is a three-step waterfall, executed once per process and then cached for the lifetime of the JVM:

1. *Explicit `OAuthJwksUri`.* If the operator has set this key in `application.ini`, that URL is used directly. No discovery requests are made. This is the override path; it bypasses every other step.

2. *RFC 8414 metadata.* The framework GETs `<OAuthAuthorizationServer>/.well-known/oauth-authorization-server` (the OAuth 2.0 Authorization Server Metadata document) and reads the `jwks_uri` field. This is the OAuth 2.1 / MCP standard discovery path, and the one Kiss's own authorization server publishes (Chapter 27). Most third-party providers (Auth0, Okta, Keycloak, Microsoft Entra) publish this document as well.

3. *OpenID Connect Discovery.* If the RFC 8414 step returned 404, or returned a 2xx body that did not contain a `jwks_uri` field, the framework falls back to `<OAuthAuthorizationServer>/.well-known/openid-configuration` (the OIDC discovery document). This step exists for the small number of authorization servers that publish OIDC discovery but not the RFC 8414 metadata document.

If all three steps fail to yield a `jwks_uri`, the framework throws `IOException` with a message that names both URLs tried and points the operator at the `OAuthJwksUri` override.

Why two discovery paths

RFC 8414 is the OAuth 2.1 standard and is what MCP clients themselves use to discover the authorization server's endpoints, so trying it first means the resource server uses the same metadata document the clients do. OIDC discovery, by contrast, predates RFC 8414 and uses a different well-known path; some older or OIDC-focused authorization servers still publish only the OIDC document. Keeping both paths in the waterfall lets Kiss work zero-config against either kind of server.

Soft failures versus hard failures

During discovery the framework draws a deliberate line between two kinds of failure at a metadata endpoint:

Soft failure — fall through to the next step.
HTTP 404 or 410 (the endpoint is not published); a 2xx response that is not valid JSON (the URL serves something else); a 2xx JSON document that does not contain `jwks_uri` (the document is the wrong kind). All three indicate "this discovery endpoint is not the one we want" — so the resolver moves on to the next step.

Hard failure — propagate.
Network error (DNS, refused, timeout); HTTP 401/403 (the endpoint exists but refuses the request); HTTP 5xx (the server is failing). None of these means "the document is not published" — they mean "something specific is wrong, and the operator needs to know." The resolver surfaces these as `IOException` with the specific HTTP status or cause, rather than papering over them by trying the next step (which would typically fail the same way against the same host, producing a misleading "tried everything, nothing worked" diagnostic).

The operational implication: when JWKS discovery fails, the log message is specific. "Could not fetch OAuth metadata from .../.well-known/oauth-authorization-server: connect timed out" tells the operator there's a network problem with the AS, not a configuration problem with the RS. "Could not resolve JWKS URI for issuer ...; neither ... nor ... published a jwks_uri" tells the operator the AS is reachable but doesn't publish either metadata document, and that `OAuthJwksUri` is the way out.

26.9 Security Considerations

Make sure `OAuthResourceIdentifier` reflects this resource server's URL.
The validator rejects any token whose `aud` claim does not contain this value. Without that check, a token issued to a different resource by the same authorization server could be replayed against yours. RFC 8707 (Resource Indicators) defines the authorization-server side of this binding; Kiss enforces the resource side. When the AS and the RS are the same Kiss app the default (which equals `OAuthAuthorizationServer`) is correct; set it explicitly only when the RS is hosted at a URL different from the AS.

`alg=none` is always rejected.
The algorithm dispatch only handles RSA algorithms (RS256/RS384/RS512), so a forged unsigned token cannot slip through even if a misconfigured authorization server somehow issued one.

Run over HTTPS.
Bearer tokens are bearer credentials; whoever intercepts one can use it until it expires. TLS termination is the deployer's responsibility — Kiss does not enforce it.

Brief validation cache.
Validated tokens are cached for 60 seconds, keyed by SHA-256 of the token string, to avoid re-verifying the same JWT on every JSON-RPC call from the same MCP session. The `exp` claim is re-checked on every cache hit. Lower the cache TTL (or bypass via direct `validate()` calls) if you require sub-minute revocation.

JWKS fetch failures surface as 401, not 5xx.
A failed JWKS fetch is logged at `error` level but returned to the client as `invalid_token` to avoid leaking infrastructure details.

No token-type check.
The validator accepts any JWT signed by a key in the JWKS whose `alg` is in the allowed list. Configure your authorization server to issue distinct access tokens and ID tokens; do not configure it to issue access tokens that are reusable as ID tokens or vice versa.

26.10 Components

All classes live in `org.kissweb.oauth`. Application developers do not write or modify any of them — only `application.ini` keys and (optionally) an `authenticate()` override in their own `MCPServerBase` subclass.

`OAuthConfig`
Reads the OAuth settings from `application.ini`. Lazy singleton. Exposes `isEnabled()` for the rest of the framework to check whether OAuth is configured.

`JwksCache`
: Fetches and caches the authorization server's JWKS, keyed by `kid`. Thread-safe. Refetches on unknown `kid` or TTL expiry.

`BearerTokenValidator`
: The main entry point. Parses the `Authorization` header, verifies the JWS signature against the JWKS, validates claims, and writes the appropriate 401 / 403 challenge on failure.

`ValidatedToken`
: Immutable holder for a successfully validated token's claims.

`ProtectedResourceMetadataServlet`
: Serves `/.well-known/oauth-protected-resource` per RFC 9728. Returns 404 when OAuth is not configured.

26.11 A Worked Example

Start from the `MCPServerExample` template (Section 25.3 discusses the related `RestServerExample`). Add this to `application.ini`:

```
OAuthAuthorizationServer = https://your-tenant.auth0.com/
OAuthResourceIdentifier = https://myapp.example.com
OAuthRequiredScopes    = mcp:read mcp:write
```

The split between `OAuthAuthorizationServer` and `OAuthResourceIdentifier` above reflects the usual third-party-provider shape: the AS is hosted by Auth0; the RS is your own application. When the AS is Kiss itself in the same JVM (Chapter 27), only `OAuthAuthorizationServer` is needed — `OAuthResourceIdentifier` defaults to it.
Rebuild and restart:

```
./bld build
```

That's it. Without touching `MCPServerExample.java`, every request to the MCP endpoint now requires a valid bearer token from your Auth0 tenant, carrying both `mcp:read` and `mcp:write` scopes and a matching `aud`.

To make a tool method aware of the calling user — for example, to scope queries to the user's own data — read the token from inside `callTool`:

```
import org.kissweb.oauth.BearerTokenValidator;
import org.kissweb.oauth.ValidatedToken;

@Override
protected JSONObject callTool(String name, JSONObject args) {
    ValidatedToken tok = BearerTokenValidator.currentToken();
    String userId = tok.getSubject();

    switch (name) {
        case "list_my_orders":
            return toolResult(listOrdersForUser(userId));
        case "delete_order":
            if (!tok.hasScope("mcp:write"))
                return toolError(
                    "delete_order requires the mcp:write scope.");

            return toolResult(deleteOrder(userId,
                args.getString("orderId", "")));
        default:
            return toolError("Unknown tool: " + name);
    }
}
```

26.12 Troubleshooting

Discovery endpoint returns 404.
OAuth is not configured. Check that `OAuthAuthorizationServer` is uncommented in `application.ini` and that the server was restarted after the change (configuration is read once at startup).

Every request gets "No JWKS key found for kid=...".
The authorization server is signing tokens with a key that is not in its JWKS, or the discovered JWKS URL is wrong. Check the catalina log for the "Looking for jwks_uri in ..." lines (one per discovery step actually attempted — so up to two, one for RFC 8414 and one for the OIDC fallback) and for the eventual "Fetching JWKS from ..." line. Verify that URL serves the expected JWKS. Override with explicit `OAuthJwksUri` if neither metadata document points where you expect.

Startup or first-request failure with "Could not fetch OAuth metadata from .../.well-known/oauth-authorization-server".
This is a *hard* failure on the RFC 8414 discovery step: a network error, timeout, 401/403, or 5xx response. It is *not* a "the document is not published" condition — the resolver propagated it instead of silently falling through to OIDC discovery (which would typically fail the same way against the same host and produce a misleading diagnostic). Resolve at the authorization server (the metadata endpoint is genuinely unreachable, or the AS is rejecting the request); or set `OAuthJwksUri` explicitly to skip the discovery step entirely.

Startup or first-request failure with "Could not resolve JWKS URI for issuer ...; neither .../.well-known/oauth-authorization-server nor .../.well-known/openid-configuration published a jwks_uri".
Both metadata documents were reachable (so this is not a network problem) but neither contained a `jwks_uri` field — the authorization server is not publishing OAuth metadata in either standard form. Set `OAuthJwksUri` explicitly to the AS's JWKS endpoint.

Every request gets "Token iss '...' does not match expected '...' ".
The `OAuthAuthorizationServer` value must match the `iss` claim in tokens exactly (modulo a trailing slash, which the validator normalizes).

Every request gets "Token aud does not include resource identifier '...' ".
The token's `aud` claim does not include the value of `OAuthResourceIdentifier`. Either reconfigure the authorization server to issue tokens with the right audience or set `OAuthResourceIdentifier` to a value the tokens actually carry.

`BearerTokenValidator.currentToken()` returns null in a tool method.
The OAuth path was not taken for this request — most commonly because OAuth is not configured, or because the MCP

server overrides `authenticate()` with a different scheme.

Tokens validate the first time but fail after the authorization server rotates keys.
The framework refetches the JWKS on encountering an unknown `kid`, so this should self-heal on the next request. If it does not, ensure the authorization server's rotation policy keeps the old key in the JWKS long enough for the cache to refresh.

26.13 What's Not Implemented

To keep the resource-server focused, the following are out of scope — each can be addressed at the validator (where applicable) or by relying on the authorization server for the missing piece. (For Kiss's own authorization server, see Chapter 27.)

- *Token revocation and introspection endpoints.* The RS validates self-contained JWTs using the JWKS; it does not call back to the AS on every request to check for revocation. Use a short access-token TTL if you need tight revocation semantics; refresh-token rotation (Section 27.7) covers the longer-lived secret.

- *ECDSA, EdDSA, HMAC signatures.* Only RSA (RS256, RS384, RS512) is supported. Adding ECDSA support is a localized change to the signature-verification dispatch in `BearerTokenValidator`.

- *Opaque tokens with introspection.* Only self-contained JWTs are validated. Opaque-token validation would require a call to the authorization server's `/introspect` endpoint per request.

- *Demonstrating Proof of Possession (DPoP, RFC 9449).* All validated tokens are bearer tokens.

- *mTLS client certificates as a binding mechanism (RFC 8705).* Bearer tokens are not bound to a transport-layer client certificate.

26.14 Further Reading

A standalone OAuth how-to with copy-paste recipes lives in the repository root as `OAuth.md`. The relevant RFCs:

- *RFC 6749* — The OAuth 2.0 Authorization Framework
- *RFC 6750* — The OAuth 2.0 Authorization Framework: Bearer Token Usage
- *RFC 7515* — JSON Web Signature (JWS)
- *RFC 7519* — JSON Web Token (JWT)
- *RFC 7517* — JSON Web Key (JWK)
- *RFC 8414* — OAuth 2.0 Authorization Server Metadata
- *RFC 8707* — Resource Indicators for OAuth 2.0
- *RFC 9068* — JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens
- *RFC 9728* — OAuth 2.0 Protected Resource Metadata

The Model Context Protocol authorization specification (`https://modelcontextprotocol.io`) describes how MCP clients use the discovery document and the `WWW-Authenticate` challenge to obtain tokens, and is required reading for anyone building MCP servers intended to be used by third-party clients.

Chapter 27

OAuth 2.1 Authorization Server

Chapter 26 covered the half of OAuth that protects an API — validating tokens issued elsewhere. This chapter covers the other half: the authorization server (AS) that *issues* those tokens.

Most Kiss applications that ship an MCP server eventually need both halves. In the typical deployment, a single Kiss instance is the AS, the resource server, and the MCP server: the same JVM authenticates the user, runs the consent UI, issues an access token, and validates that token on the next MCP call. Splitting the AS into a separate service is straightforward (point `OAuthAuthorizationServer` at the external AS, leave `OAuthAsEnabled=false`), but the collocated case is what the framework optimizes for.

The AS is deliberately minimal but spec-compliant. It implements the parts of OAuth 2.1 the MCP specification requires — authorization-code flow with PKCE, refresh-token rotation, dynamic client registration, the metadata and JWKS endpoints — and stops there. Section 27.13 catalogues what is left out and why.

This chapter and the previous one cover Kiss as a server of OAuth. The third role — Kiss as a *client* of a remote OAuth server, obtaining tokens to make its own authorized outbound calls — is the subject of Chapter 28. The two are independent: a single app may issue tokens, validate tokens, and consume a partner's tokens all at once, or use any subset.

27.1 Configuration

All settings live in `src/main/backend/application.ini`. Setting `OAuthAsEnabled=true` activates the AS; everything else has a sensible default. In the common case where the AS and the resource server are the same Kiss app, only one URL setting is needed — and it lives in the resource-server section, not here:

```
# Required.
OAuthAsEnabled = true

# Optional. Canonical URL of this AS. Goes into the iss claim
# of every issued token and forms the base of the metadata
# document. Defaults to OAuthAuthorizationServer (from the
# resource-server config in the preceding section of
# application.ini) when not set --- which is the right value
# when the AS and the RS are the same Kiss app. Set this
# explicitly only when the AS publishes itself under a URL
# different from the RS-side OAuthAuthorizationServer value.
# OAuthAsIssuer = https://myapp.example.com

# Optional defaults shown
# (OAuthAsSqliteFile is relative to applicationPath, or an
# absolute path)
# OAuthAsSqliteFile          = oauth.db
# OAuthAccessTokenTtlSeconds = 3600    # 1 hour
# OAuthRefreshTokenTtlSeconds = 2592000 # 30 days
# OAuthAuthCodeTtlSeconds    = 60      # 1 minute
# OAuthPruneIntervalSeconds  = 900     # 15 minutes
# OAuthAllowDynamicRegistration = true # required for MCP
# OAuthSessionTtlSeconds     = 1800    # AS login cookie
# OAuthKeyId                 = kiss-key-1 # JWT kid
```

When the AS is disabled, every endpoint returns 404 — the same response a server with no AS would give, signaling to clients that this resource is not paired with an AS at this URL.

Tip

The application path (`MainServlet.getApplicationPath()`) is already the backend directory — in dev mode `src/main/backend/`, in a deployed WAR `WEB-INF/backend/`. So the default `OAuthAsSqliteFile=oauth.db` colocates the file with `application.ini`. You generally do not need to change this.
`OAuthAsSqliteFile` may also be set to an *absolute* path (e.g. `/var/lib/myapp/oauth.db`), which is used verbatim. This is useful when you want the runtime state file to live outside the deployed webapp tree so a WAR redeploy cannot touch it — the signing keys, registered clients, and refresh tokens then survive a redeploy unchanged.
The SQLite file is created and its schema initialized automatically on first startup; no setup tool is needed, and the database is completely independent of the application's main database (its own file, its own JDBC connection, no shared schema or connection pool).

27.2 Required App Code: UserAuthenticator

The AS will refuse to issue tokens until the application provides a `UserAuthenticator`. This is the one piece of the AS that cannot be generic: the credential check inevitably knows your user table schema, your LDAP layout, or your IdP's API.

The interface has one method:

```
public interface UserAuthenticator {
    AuthenticatedUser authenticate(String username,
        String password);
}
```

Return an `AuthenticatedUser` on success — carrying the user's subject identifier and an optional bag of extra claims that the framework will copy onto every access token issued for this user. Return `null` on any kind of failure: bad credentials, locked account, user store unreachable. Never throw — the framework cannot distinguish "wrong password" from "LDAP is down" if you throw, and the user-facing render should not leak that distinction anyway. Log

the underlying cause inside your implementation.

For Kiss applications that already have a `Login.groovy` (Chapter 20), a ten-line wrapper is sufficient:

```
// src/main/backend/MyUserAuthenticator.groovy
package com.mycompany

import org.kissweb.oauth.as.AuthenticatedUser
import org.kissweb.oauth.as.UserAuthenticator
import org.kissweb.json.JSONObject

class MyUserAuthenticator implements UserAuthenticator {
    AuthenticatedUser authenticate(String username,
        String password) {
        def row = Login.verifyPassword(username, password)
        if (!row)
            return null
        def extras = new JSONObject()
        extras.put("email", row.email)
        extras.put("name", row.full_name)
        return new AuthenticatedUser(String.valueOf(
            row.user_id), extras)
    }
}
```

Register it in `KissInit.groovy`, after `MainServlet.readIniFile(...)`:

```
AsExtensions.setUserAuthenticator(
    new com.mycompany.MyUserAuthenticator())
```

A few things worth being deliberate about:

Use a stable subject.
The subject (`sub` claim) is the primary identifier every downstream component uses to recognize this user — audit logs, resource-server authorization decisions, custom claims joined to other databases. Prefer the database primary key over a username that can be renamed.

Be deliberate about extra claims.
Anything in `extraClaims` ends up signed into every access token, visible to whatever resource server consumes the token. Email is usually fine; password hashes and internal IDs other systems should not see, are not.

Implementations live in app code.
Either `src/main/backend/` (Groovy or Java, hot-reloadable) or `src/main/precompiled/` (Java with package structure, restart required after change).

27.3 Optional App Code: ConsentProvider and HTML Overrides

27.3.1 ConsentProvider

`ConsentProvider` supplies the human-readable text the consent page shows for each scope a client is requesting. The framework knows scope *names* like `mcp:write` but has no idea what they mean in your application's domain. The default implementation returns the scope name verbatim — functional but unfriendly:

```
public interface ConsentProvider {
    String describeScope(String scope);
    default String getDisplayName() { ... }
}
```

Provide your own to turn cryptic scope names into sentences a user can understand:

```
AsExtensions.setConsentProvider(new ConsentProvider() {
    String describeScope(String s) {
        switch (s) {
            case "mcp:read":
                return "Read your orders, customers, " +
                    "and reports."
            case "mcp:write":
                return "Create, update, and delete records " +
                    "on your behalf."
            default:
                return s
        }
    }
    String getDisplayName() { return "Acme Corp" }
})
```

27.3.2 HTML Overrides

The login and consent pages have minimal built-in HTML that works out of the box. To restyle or rebrand, drop replacement files in `src/main/frontend/oauth/`:

- `src/main/frontend/oauth/login.html`
- `src/main/frontend/oauth/consent.html`

Both use simple `{placeholder}` substitution. The supported placeholders:

`{asName}`
: `ConsentProvider.getDisplayName()`. Available in login and consent.

`{clientName}`
: The DCR `client_name` (or the client id if no name was registered). Available in login and consent.

`{flowId}`
: Opaque flow id assigned to this in-flight authorization request. Must round-trip in the form as a hidden field named `flow`. Available in login and consent.

`{userSubject}`
: Subject of the user who just logged in. Available in consent only.

`{scopeList}`
: Pre-rendered `<li>` items for each requested scope, with the description from `ConsentProvider`. Available in consent only.

`{errorBlock}`
: An error `<div>`, or empty when there is nothing to show. Available in login and consent.

Both forms must POST back to `/oauth/authorize` with a hidden `stage=login` or `stage=consent` and the `flow` id. The defaults in `AuthorizationServlet.java` are the working reference — copy them and rework the styling.

Note

Templates are loaded once at first use and cached for the lifetime of the JVM. Edits to your template files take effect after a server restart. Hot reload does not apply — the servlet container does not re-scan template files between requests.

27.4 The Five Endpoints

All five are auto-registered via `@WebServlet` annotations; no `web.xml` entries required.

`GET /.well-known/oauth-authorization-server`
: RFC 8414 metadata. Returns the issuer URL, the endpoint URLs, the supported grant types, response types, PKCE methods, token endpoint auth methods, and signing algorithm. Cached one hour at the client. Returns 404 when the AS is disabled.

`GET /oauth/jwks`
: A single-key JWKS document carrying the AS's current RSA-2048 public key. `kid` matches the `kid` on every issued JWT, so a resource server can fetch this once and validate signatures locally without further round-trips. Cached five minutes.

`GET /oauth/authorize`
: Authorization endpoint. Validates the request, then renders the login page (no session) or the consent page (session present). On approval, redirects the user-agent back to the client's `redirect_uri` with `?code=...&state=...`; on denial, with `?error=access_denied&...`. Errors that prevent the redirect (unknown client, redirect URI mismatch) render as 400 HTML — the framework refuses to redirect to a URI not registered for the client.

`POST /oauth/token`
: Token endpoint. Handles `authorization_code` and `refresh_token` grants. All responses include `Cache-Control: no-store`.

`POST /oauth/register`
: RFC 7591 dynamic client registration. Anyone can POST;

the response carries a freshly-generated `client_id` and, for confidential clients, a `client_secret`. Disable via `OAuthAllowDynamicRegistration=false` if your deployment does not need open registration.

27.5 The Authorization Code Flow Step by Step

The flow as it plays out across browser, AS, and client:

1. Client begins the dance: builds an authorization URL and either redirects the user-agent there (web client) or opens it in a browser (native / desktop / MCP client):

```
GET /oauth/authorize
    ?response_type=code
    &client_id=<id>
    &redirect_uri=<exact-match-of-registered>
    &scope=mcp:read mcp:write
    &state=<csrf nonce>
    &code_challenge=<S256 hash of verifier>
    &code_challenge_method=S256
    # optional RFC 8707
    &resource=https://myapp.example.com
```

2. AS validates everything. Unknown `client_id` or mismatched `redirect_uri` render as 400 HTML and stop here. Other errors (unknown scope, missing PKCE) redirect back to the client with `?error=...`.

3. If the user has no session, the AS renders the login page. The form's hidden `flow=<id>` keys an in-memory record of the pending request.

4. User submits credentials. The AS calls `UserAuthenticator.authenticate()`. On failure, re-render the login page with an error. On success, set an `oauth_session` cookie (HttpOnly, Secure when over TLS, scoped to `/oauth/`, 30-minute TTL by default) and render the consent page.

5. User clicks Allow. The AS generates an opaque authorization code (60-second TTL, single-use), stores it in memory along

with the user, scopes, audience, redirect URI, PKCE challenge, and any extra claims from the authenticator, then 302-redirects to:

```
HTTP 302
Location: <redirect_uri>?code=<code>&state=<state>
```

6. Client receives the redirect, extracts the code, and POSTs to the token endpoint:

```
POST /oauth/token
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code
&code=<code>
&redirect_uri=<must match step 1 exactly>
&client_id=<id>
&code_verifier=<PKCE verifier>
```

7. AS verifies: client exists, code is valid and not consumed, redirect URI matches, PKCE `SHA256(code_verifier) == code_challenge`. Then it mints an RS256 access token (RFC 9068 `at+jwt` profile) carrying `iss / sub / aud / exp / iat / nbf / jti / scope / client_id` plus any extra claims, generates an opaque refresh token, stores the refresh token, and returns:

```
HTTP 200 OK
Cache-Control: no-store
Content-Type: application/json

{
  "access_token": "eyJ...",
  "token_type":   "Bearer",
  "expires_in":   3600,
  "refresh_token": "...",
  "scope":        "mcp:read mcp:write"
}
```

8. Client uses the access token against the resource server (Chapter 26) until it expires, then refreshes (Section 27.7).

Subsequent `/authorize` requests from the same browser within the session window skip the login step — the `oauth_session` cookie

identifies the user, and only consent is shown. Logout is implicit: the cookie expires (default 30 minutes) or the user clears it.

27.6 Dynamic Client Registration

DCR is enabled by default because the MCP specification requires it. Anyone reachable from the AS can POST a client metadata document to `/oauth/register` and receive back a freshly-generated client id (and, for confidential clients, a one-time-displayed secret). Persisted to the AS's SQLite database (`oauth.db`).

```
curl -X POST http://localhost:8001/oauth/register \
  -H "Content-Type: application/json" \
  -d '{
    "redirect_uris": ["http://localhost:54545/cb"],
    "client_name": "Claude Desktop",
    "scope": "mcp:read mcp:write",
    "token_endpoint_auth_method": "none"
  }'
```

`token_endpoint_auth_method=none` produces a *public client* (no secret). Public clients authenticate at the token endpoint via PKCE alone; this is the right shape for native and desktop MCP clients that cannot keep a secret.

Any other supported value (`client_secret_basic` or `client_secret_post`, the default) produces a *confidential client*. The server generates a 256-bit secret, returns it in the response exactly once, and stores only its SHA-256 hash. There is no recovery if the client loses the secret — it must re-register.

For production where DCR is not appropriate, set `OAuthAllowDynamicRegistration=false` and provision clients programmatically from startup code:

```
import org.kissweb.oauth.as.*

ClientStore.get().register(new RegisteredClient(
    "claude-prod",
    // pre-computed SHA-256 hex of secret
    "sha256:abc123...",
    "Claude Production",
    ["https://prod.example/cb"],
    ["mcp:read", "mcp:write"] as Set,
    ["authorization_code", "refresh_token"] as Set,
    System.currentTimeMillis() / 1000))
```

There is no RFC 7592 client-management endpoint in v1 — to update or delete a registered client, use `ClientStore.remove(clientId)` from app code.

27.7 Refresh-Token Rotation and Reuse Detection

OAuth 2.1 mandates refresh-token rotation for the `refresh_token` grant, and Kiss implements it. Every successful refresh:

1. Issues a brand-new refresh token in the same *family*. The family id is a UUID assigned when the original authorization-code grant happened; every rotation in the same logical session keeps the same family id.
2. Marks the old refresh token as rotated, with its `rotated_to_jti` pointing at the successor.
3. Returns the new pair (access + refresh) to the client.

If a *rotated* refresh token is ever presented again — which an honest client never does because it has already upgraded to the successor — the AS treats this as evidence of token theft and **revokes the entire family**. Every refresh token in the family is deleted from the AS database; the next refresh attempt by anyone in the family (including the legitimate client) returns `invalid_grant`. The legitimate user has to log in again.

Note

Rotation does not prevent the initial theft. It bounds the damage: the attacker cannot keep refreshing indefinitely without the legitimate client noticing the next time *it* tries to refresh, at which point both attacker and legitimate client lose access.

This is the OAuth 2.1 stolen-refresh-token detection mechanism described in RFC 6749bis §6.1. Implementation-wise, rotation issues two writes (mark the old token rotated; insert the successor) inside one SQLite transaction so the on-disk state is never observed mid-rotation; family revocation is a single indexed `DELETE`. Both complete in milliseconds at MCP-scale traffic — see Section 27.8.

27.8 Persistence: The oauth.db Database

All long-lived AS state lives in a private SQLite database — a single file (`oauth.db` by default), independent of the application's main database. The AS opens its own JDBC connection against this file, shares no connection pool with the rest of the application, and requires no operator setup beyond pointing at a file path. The schema is materialized via `CREATE TABLE IF NOT EXISTS` on every open, so a fresh install creates the file automatically.

```
CREATE TABLE oauth_keys (
    kid         TEXT PRIMARY KEY,
    private_key TEXT NOT NULL,         -- base64 PKCS#8
    public_key  TEXT NOT NULL,         -- base64 X.509
    created_at  INTEGER NOT NULL
);

CREATE TABLE oauth_clients (
    client_id          TEXT PRIMARY KEY,
    -- null for public clients
    client_secret_hash TEXT,
    client_name        TEXT,
    redirect_uris      TEXT,           -- space-joined
    allowed_scopes     TEXT,           -- space-joined
    allowed_grant_types TEXT,          -- space-joined
    created_at         INTEGER NOT NULL
);

CREATE TABLE oauth_refresh_tokens (
    jti              TEXT PRIMARY KEY,
    family_id        TEXT NOT NULL,
    client_id        TEXT NOT NULL,
    user_sub         TEXT,
    user_extra_claims TEXT,            -- base64 JSON
    scopes           TEXT,             -- space-joined
    audience         TEXT,
    created_at       INTEGER NOT NULL,
    expires_at       INTEGER NOT NULL,
    -- null for active, set for rotated
    rotated_to_jti   TEXT
);
CREATE INDEX idx_oauth_refresh_family
    ON oauth_refresh_tokens (family_id);
CREATE INDEX idx_oauth_refresh_expires
    ON oauth_refresh_tokens (expires_at);

CREATE TABLE oauth_meta (
    meta_key   TEXT PRIMARY KEY,
```

```
    -- holds schema_version etc.
    meta_value TEXT NOT NULL
);
```

Authorization codes (60-second TTL) live in memory only — losing one on restart costs the user one click of the Allow button, not worth a write.

27.8.1 Transactions

Every mutating operation commits as a normal SQLite transaction. Refresh-token rotation in particular issues both writes (mark the old token; insert the successor) inside one transaction, so the on-disk state is never observed mid-rotation. A JVM crash before commit leaves the previous state intact — no half-written rows, no all-keys-lost catastrophe.

27.8.2 Lazy Pruning

There is no cron job. Every OAuth endpoint call checks an in-memory `lastPrunedAt` timestamp; if `OAuthPruneIntervalSeconds` (default 15 minutes) has elapsed since the last prune, the call issues a single `DELETE ... WHERE expires_at <= ?` for refresh tokens and drops expired in-memory codes. The `expires_at` column is indexed so the sweep is cheap regardless of table size. The throttle timestamp is in memory only — a restart causes the first call to re-prune, which is exactly what we want.

27.8.3 Concurrency Model

Single-process only. SQLite serializes writers internally, and the Kiss `Connection` is shared across threads under a JVM-wide monitor on `OAuthSqliteStore`. Two Kiss JVMs writing to the same `oauth.db` will fail with SQLite's standard `database is locked` error rather than silently corrupting state. For high availability, run a single AS instance behind a load balancer, or swap in alternative implementations of `ClientStore`, `RefreshTokenStore`, and `KeyManager` that talk to a shared server-class database.

27.8.4 Scale

Per-row overhead is well under a kilobyte. Tens of thousands of active refresh tokens fit comfortably in a single-MB file with sub-millisecond point lookups (`jti`, `family_id`, and `expires_at` are all indexed). Even after years of accumulated registered clients, the database remains small enough to ship in backups and trivially small to inspect.

27.8.5 Inspection

Operators with the `sqlite3` CLI can inspect the file in place:

```
sqlite3 /path/to/oauth.db '.tables'
sqlite3 /path/to/oauth.db \
  'SELECT client_id, client_name FROM oauth_clients;'
```

Take a copy first if the AS is running.

27.8.6 Build-Time Safety

`./bld build` copies `src/main/backend/` into the deployed WAR, but the framework's build script excludes `oauth.db` (and SQLite's transactional sidecar files `oauth.db-journal`, `-wal`, `-shm`) from the copy. An accidental `oauth.db` in the source tree will never overwrite the runtime file in the deployed app. Also add the same set of names to `.gitignore` — runtime state should never be checked in.

For deployments where even `WEB-INF/backend/` is rewritten on redeploy (e.g. a full WAR replacement rather than an in-place update), set `OAuthAsSqliteFile` to an absolute path outside the webapp directory — the file is then completely untouched by the deploy and the AS retains its signing keys, registered clients, and refresh tokens across upgrades.

27.9 Security Considerations

HTTPS in production.
: The session cookie carries the user's identity for the duration of the consent flow; access and refresh tokens are bearer credentials. TLS termination is the deployer's responsibility — Kiss does not enforce it. Cookies are automatically marked `Secure` when the request is HTTPS.

PKCE is mandatory.
S256 only; `plain` is rejected even when explicitly requested. Public clients depend entirely on PKCE for authentication.

Exact redirect-URI matching.
Per OAuth 2.1 §4.1.2.1, no wildcards, no scheme/host normalization, no fragments. The `redirect_uri` on `/authorize` must equal one of the client's registered URIs byte-for-byte, and the same value must come back on `/token`. The framework refuses to redirect to a non-registered URI even when the client says to.

Refresh-token rotation is mandatory.
See Section 27.7. Without rotation, a leaked refresh token is good for 30 days with no detection mechanism.

Audience binding.
If the client supplied a `resource` parameter on `/authorize`, the issued tokens carry it as the `aud` claim and the audience is enforced on `/token` refresh. Otherwise, tokens are bound to the AS issuer URL — any resource server that trusts this AS will accept them.

Client secret storage.
Only SHA-256 of the secret is stored. The full secret is returned to the client exactly once during DCR; there is no recovery if it's lost.

`alg=none` cannot be issued.
`TokenIssuer` signs with `SHA256withRSA` directly; there is no algorithm-from-header dispatch on the issuing side.

Authorization codes are single-use.
`AuthorizationCodeStore.consume()` returns the code and removes it from the store in one synchronized operation. A replay attempt finds nothing.

Client secrets sent in basic auth are constant-time compared
against the stored hash to avoid leaking partial matches via timing.

No initial-access-token gate on DCR.
Anyone reachable from the AS can register a client. This is

desirable for MCP (clients you've never met need to onboard) but means the registry can grow unboundedly under attack. Disable DCR with `OAuthAllowDynamicRegistration=false` if open registration is not acceptable for your deployment.

27.10 Components

All classes live in `org.kissweb.oauth.as`. Application developers do not write or modify any of them — only the two interfaces `UserAuthenticator` and `ConsentProvider` (in app code), plus `application.ini` keys.

`AuthorizationServerConfig`
: Reads AS config from `application.ini`. Lazy singleton.

`OAuthSqliteStore`
: Opens and initializes the schema of the AS's private SQLite database, provides the shared JDBC `Connection` to the other stores, and holds the in-memory prune-throttle timestamp.

`KeyManager`
: Generates RSA-2048 keypair on first start, persists as base64 PKCS#8 / X.509 in the `oauth_keys` table, exposes private key for signing and public JWK for publishing.

`ClientStore`
: CRUD for registered clients in the `oauth_clients` table.

`RefreshTokenStore`
: CRUD for refresh tokens in the `oauth_refresh_tokens` table. Implements rotation and family revocation via indexed SQL.

`AuthorizationCodeStore`
: In-memory single-use codes with TTL expiry.

`TokenIssuer`
: Builds and signs RS256 JWTs per RFC 9068; generates random opaque codes and refresh-token values.

`PkceValidator`
: RFC 7636 S256 verifier.

`AsExtensions`
: Static registry where the application installs its `UserAuthenticator` and `ConsentProvider`.

`UserAuthenticator` ***(interface)***
: Application implements: verify credentials, return `AuthenticatedUser` or null.

`ConsentProvider` ***(interface)***
: Application implements: per-scope human description, AS display name.

`AuthenticatedUser`
: Subject identifier plus extra claims to embed in tokens.

`RegisteredClient`, `RefreshToken`, `AuthorizationCode`
: Immutable record types.

`MetadataServlet`, `JwksPublishServlet`
: RFC 8414 and JWKS endpoints.

`AuthorizationServlet`
: Login, consent, code issuance.

`TokenServlet`
: Code exchange and refresh rotation.

`DynamicClientRegistrationServlet`
: RFC 7591.

27.11 A Worked Example

Take a Kiss application that already has a working MCP server (Chapter 25) and an existing user table. Adding self-hosted OAuth takes three changes.

27.11.1 Step 1: Turn on the AS in application.ini

When the AS and the resource server are the same Kiss app at the same URL — the common MCP shape — the issuer URL appears exactly once. `OAuthAsIssuer` defaults to `OAuthAuthorizationServer`; `OAuthResourceIdentifier` also defaults to `OAuthAuthorizationServer`; and `OAuthJwksUri` defaults

to RFC 8414 discovery against the same URL (Section 26.8.1), which Kiss's own AS publishes (next bullet of the next section). So the minimum config is:

```
# Authorization-server config
OAuthAsEnabled = true

# Resource-server config so MCP servlets validate tokens
# issued here
OAuthAuthorizationServer = https://myapp.example.com
OAuthRequiredScopes    = mcp:read mcp:write
```

If the AS and RS are split across different URLs — unusual for MCP but supported — set `OAuthAsIssuer`, `OAuthAuthorizationServer`, and `OAuthResourceIdentifier` to their respective canonical URLs.

27.11.2 Step 2: Provide a UserAuthenticator

```
// src/main/backend/MyUserAuthenticator.groovy
package com.mycompany

import org.kissweb.oauth.as.AuthenticatedUser
import org.kissweb.oauth.as.UserAuthenticator
import org.kissweb.json.JSONObject

class MyUserAuthenticator implements UserAuthenticator {
    AuthenticatedUser authenticate(String username,
        String password) {
        def row = Login.verifyPassword(username, password)
        if (!row) return null
        def extras = new JSONObject()
        extras.put("email", row.email)
        return new AuthenticatedUser(String.valueOf(
            row.user_id), extras)
    }
}
```

27.11.3 Step 3: Register the authenticator and consent provider

```
// KissInit.groovy, in init() after readIniFile(...)
AsExtensions.setUserAuthenticator(
    new com.mycompany.MyUserAuthenticator())
AsExtensions.setConsentProvider(new ConsentProvider() {
```

```
    String describeScope(String s) {
        switch (s) {
            case "mcp:read": return "Read your data."
            case "mcp:write": return "Modify your data."
            default:         return s
        }
    }
    String getDisplayName() { return "Acme MCP" }
})
```

27.11.4 That's It

Rebuild and restart:

```
./bld build
```

The first MCP client to connect will:

1. Discover the resource via `GET` https://myapp.example.com/.well-known/oauth-protected-resource.
2. Discover the AS via `GET` https://myapp.example.com/.well-known/oauth-authorization-server — the same metadata document Kiss's own resource server consults (Section 26.8.1) to find the JWKS, so the AS+RS pair needs no `OAuthJwksUri` setting.
3. Register itself via `POST` https://myapp.example.com/oauth/register.
4. Open a browser to https://myapp.example.com/oauth/authorize?... for the user to log in and approve.
5. Exchange the code at `POST` https://myapp.example.com/oauth/token.
6. Use the access token on MCP calls; refresh as needed.

No further code on your side.

27.12 Troubleshooting

Every `/oauth/...` returns 404.
AS not enabled. Set `OAuthAsEnabled = true` and restart (configuration is read once at startup).

`/oauth/authorize` returns 500 with "No UserAuthenticator has been registered."
Add an `AsExtensions.setUserAuthenticator(...)` call to `KissInit.groovy`.

`/oauth/token` returns `invalid_grant`: "redirect_uri does not match...".
OAuth 2.1 requires the same `redirect_uri` on `/authorize` and on `/token`, byte-for-byte. Make sure the client is sending an identical value at both calls (trailing slashes, port numbers, casing all matter).

`/oauth/token` returns `invalid_grant`: "PKCE code_verifier does not match...".
The client either supplied a different verifier than the challenge was generated from, or built the challenge wrong. Check that the client is computing `base64url(SHA-256(verifier))` (no padding) and submitting the original verifier on `/token`.

`/oauth/token` returns `invalid_grant`: "Refresh token has been rotated; family revoked".
The client (or someone with a copy of an old refresh token) submitted a rotated token. All tokens in the family are now revoked; the user must re-authenticate.

Login page rejects valid credentials.
Check `tomcat/logs/catalina.out` for exceptions thrown by your `UserAuthenticator`. Exceptions render as "Invalid username or password" to the user but are logged in full.

`oauth.db` (or its `-journal`/`-wal`/`-shm` sidecars) appeared in `src/main/backend/` and got committed.
Add the same set of names to `.gitignore` immediately. The build excludes them from the WAR copy, but git does not know about that exclusion.

Issued tokens fail validation at the resource server with "Token iss does not match expected".
`OAuthAsIssuer` (used as `iss` on issued tokens) must equal `OAuthAuthorizationServer` (used by the validator to check `iss`). Set both to the same canonical URL.

27.13 What's Not Implemented

The framework deliberately ships a minimal-but-spec-compliant AS. Out of scope for v1:

- *Algorithms other than RS256.* Only RS256 is issued (and only RS{256,384,512} can be validated). ECDSA (ES256), EdDSA, HMAC are not implemented. ES256 in particular is a localized addition to `TokenIssuer` and `BearerTokenValidator`.
- *Key rotation tooling.* The framework supports a single current key. The JWKS structure can accommodate a published "previous" key during a rotation window, but the operational steps (generate new, keep old until last refresh expires, drop old) are not yet built.
- *RFC 9449 DPoP* (proof-of-possession tokens). All issued tokens are plain bearer tokens.
- *RFC 8705 mTLS* client authentication or token binding.
- *RFC 7592* client-management endpoint (update or delete a registered client via HTTP). Use `ClientStore.remove()` from Java code.
- *RFC 7009 revocation* and *RFC 7662 introspection* endpoints. Revocation can be done by deleting the refresh token (or calling `RefreshTokenStore.revokeFamily()`); introspection is generally unnecessary when access tokens are JWTs the resource server can validate locally.
- *OpenID Connect.* No `openid` scope handling, no ID tokens, no userinfo endpoint. The framework is OAuth 2.1 only; layer OIDC on top, or run a separate OIDC server.
- *Multi-tenancy or realms.* Single AS per Kiss instance.
- *Shared or clustered AS storage.* The AS backs onto a private SQLite file (`oauth.db`). This is intentional — the AS does not couple to the application's main database, and operators do not need to provision a schema. For a multi-instance HA deployment, swap in alternative implementations of `ClientStore`, `RefreshTokenStore`, and `KeyManager` that talk to a shared server-class database; the rest of the AS code does not need to change.

27.14 Further Reading

A standalone OAuth reference (covering both the resource and authorization server sides) lives at the repository root as `OAuth.md`.

Relevant RFCs:

- *RFC 6749* — The OAuth 2.0 Authorization Framework
- *RFC 7636* — Proof Key for Code Exchange by OAuth Public Clients (PKCE)
- *RFC 7517* — JSON Web Key (JWK)
- *RFC 7519* — JSON Web Token (JWT)
- *RFC 7591* — OAuth 2.0 Dynamic Client Registration Protocol
- *RFC 8414* — OAuth 2.0 Authorization Server Metadata
- *RFC 8707* — Resource Indicators for OAuth 2.0
- *RFC 9068* — JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens
- *draft-ietf-oauth-v2-1* (a.k.a. OAuth 2.1) — the consolidation that mandates PKCE and refresh-token rotation

The Model Context Protocol authorization specification (`https://modelcontextprotocol.io`) describes how MCP clients discover the AS, register dynamically, and obtain tokens; required reading for anyone building MCP clients or servers intended to interoperate with the broader MCP ecosystem.

Chapter 28

OAuth 2.1 Client

The previous two chapters covered the two server-side OAuth roles: validating tokens (Chapter 26) and issuing them (Chapter 27). This chapter covers the third role, the mirror image of both: Kiss as an OAuth 2.1 *client* (relying party), obtaining tokens from a *remote* authorization server and presenting them on outbound calls.

Where the resource server *consumes* tokens that arrive and the authorization server *produces* tokens for others, the client *acquires* tokens for itself. A Kiss application uses the client role when it needs to call some other organization's OAuth-protected API — a remote MCP server, a partner's REST endpoint, any service that speaks OAuth 2.1. The framework drives the full authorization-code + PKCE dance, discovers the remote server's endpoints, registers itself when necessary, captures the browser redirect, persists the resulting tokens, and refreshes them automatically — so application code is reduced to "give me a valid token for provider `X`."

The three roles are independent. An app can use any one, any two, or all three. A Kiss-hosted MCP server typically runs the resource and authorization servers together (Chapter 27); the client role is what an app reaches for when it is the one making the protected call rather than receiving it. And like the other two, the client role is *inert until configured*: with no `[OAuthClient *]` section in `application.ini`, the callback servlet returns 404, no database is touched, and no network request is ever made.

28.1 What the Client Does

When at least one provider is configured, the framework does four things on your behalf:

1. *Discovers* the remote authorization server starting from the resource URL: it reads the RFC 9728 protected-resource metadata to learn which authorization server protects the resource, then reads that server's RFC 8414 metadata (falling back to OpenID Connect discovery) to learn the authorization, token, and registration endpoints. The result is cached.
2. *Registers* the client: it uses a pre-configured `client_id` when one is set, or performs RFC 7591 Dynamic Client Registration as a public (PKCE-only) client when none is. The result is cached.
3. *Drives the authorization-code + PKCE flow*: it builds the authorization URL, captures the redirect at a built-in callback servlet (`/oauth/client/callback`), and exchanges the returned code for tokens.
4. *Refreshes* access tokens automatically when they expire — single-flight per provider, honoring refresh-token rotation.

Note

The client role is generic. It has no knowledge of *what* the remote server does — MCP, a plain REST API, anything else. It obtains a bearer token and makes it available; sending that token on an actual request is the application's job (typically an `Authorization: Bearer` header on a `RestClient` call, Section 25.1).

28.2 Configuration

Each remote server is declared in its own section named `[OAuthClient <name>]`, where `<name>` is a label you choose. The label is used both in code (`OAuthClient.forProvider("<name>")`) and as the key under which that provider's discovery document, registration, and tokens are persisted. Multiple providers are supported — one section each.

```
[OAuthClient myprovider]
; remote resource/server URL (required)
Url          = https://remote.example.com/mcp
; space- or comma-separated (optional)
Scopes       = mcp:read mcp:write
; blank => Dynamic Client Registration (RFC 7591)
ClientId     =
; only if pre-registered + confidential
ClientSecret =
```

`Url` *(required)* The remote resource or server URL. Discovery starts from this URL's origin; audience binding (RFC 8707) uses the full value. A provider section with no `Url` is logged and ignored.

`Scopes`
(optional) Space- or comma-separated scopes to request. Omit to request none.

`ClientId`
(optional) A pre-registered client id. Leave blank to trigger Dynamic Client Registration against the remote server.

`ClientSecret`
(optional) A pre-registered client secret, only for a confidential client. Leave blank for the usual public, PKCE-only client.

Warning

Unlike the resource-server and authorization-server keys, which live in the `[main]` section, the `[OAuthClient *]` sections are their own named sections. Kiss's flat environment map (`MainServlet.getEnvironment()`) only carries `[main]`, so these sections are read directly from the ini file by `OAuthClientConfig` via `IniFile.load(...)`, `getSectionNames()`, and `get(section, key)` (Chapter 5).

A few optional keys, which *do* live in `[main]`, tune the client globally:

```
# Base URL used to build the redirect_uri
# (<base>/oauth/client/callback). If unset, it is
# taken from the base URL passed to beginAuthorization(),
```

```
# or derived from the request that starts the flow. The
# remote server must accept this redirect_uri (registered
# automatically when using Dynamic Client Registration).
# OAuthClientRedirectBaseUrl = http://localhost:8001

# Treat an access token as expired this many seconds
# before its real expiry, so a refresh happens slightly
# early. Default 60.
# OAuthClientRefreshSkewSeconds = 60

# The ini file the [OAuthClient *] sections are read
# from. Default application.ini. Exists mainly so tests
# can point at a fixture.
# OAuthClientConfigFile = application.ini
```

There is no client state-file key: cached registrations and tokens go into the same `oauth.db` the authorization server uses (Section 28.7).

28.3 Using the Client from App Code

The entire API surface an application touches is the `OAuthClient` facade. Obtain one for a configured provider with `forProvider`, then either start an interactive login or ask for a token:

```
import org.kissweb.oauth.client.OAuthClient;
import org.kissweb.oauth.client.OAuthAuthorizationRequiredException;

OAuthClient client = OAuthClient.forProvider("myprovider");

// Before making outbound calls, ensure we are authorized:
if (!client.isAuthorized()) {
    // Send the browser to this URL to log in / consent. The
    // base URL becomes <base>/oauth/client/callback; pass null
    // to use OAuthClientRedirectBaseUrl from application.ini.
    String authorizeUrl = client.beginAuthorization(
        "http://localhost:8001");
    // ... redirect the browser to authorizeUrl ...
}

// On every outbound call --- refreshes the access token as
// needed (discovery and registration happened in
// beginAuthorization):
try {
    String token = client.getAccessToken();
    JSONObject headers = new JSONObject();
    headers.put("Authorization", "Bearer " + token);
    // ... call the remote API with RestClient ...
} catch (OAuthAuthorizationRequiredException e) {
    // No usable token and none obtainable without a fresh
    // login --- kick off beginAuthorization() again.
}
```

The methods:

`static OAuthClient forProvider(String name)`
: Returns the client bound to the named provider. Throws `OAuthClientException` if no `[OAuthClient <name>]` section is configured.

`boolean isAuthorized()`
: True when a usable token is held — either an access token that is still valid, or a refresh token that can mint a new one. A cheap, non-throwing pre-check before starting an interactive flow.

`String beginAuthorization(String redirectBaseUrl)`
: Runs discovery and (if needed) registration, generates a PKCE verifier/challenge and a random `state`, records them in the in-memory pending registry, and returns the authorization URL the browser must visit. Pass the externally-reachable base URL, or `null` to use `OAuthClientRedirectBaseUrl`. Throws `OAuthClientException` if no base URL is available or discovery/registration fails.

`String getAccessToken()`
: Returns a currently-valid access token, refreshing transparently when the cached one is expired (within the skew margin). Throws `OAuthAuthorizationRequiredException` when no token is held and none can be obtained without an interactive login; throws `OAuthClientException` on an unexpected failure.

`void clearTokens()`
: Discards stored tokens for the provider (logout), while keeping the cached discovery and registration so a later login skips re-discovery.

`String getProviderName()`
: The provider label this client is bound to.

Tip

`OAuthAuthorizationRequiredException` is a normal control-flow signal, not an error condition. It means "there is no token and I cannot get one silently — send the user through `beginAuthorization()`." Catch it where you make outbound calls and turn it into a redirect to the login URL. Its `getProvider()` method returns the provider name that needs authorization.

28.4 The Authorization-Code Flow Step by Step

The flow as it plays out across the application, the browser, and the remote authorization server:

1. The app calls `client.beginAuthorization(base)`. The framework resolves discovery (from cache, or fetched and cached on first use), ensures a client registration (the configured `ClientId`, or DCR — again cached), generates a fresh PKCE verifier/challenge and a random `state`, stores a `PendingAuthorization` keyed by that state, and returns the authorization URL:

```
GET <authorization_endpoint>
    ?response_type=code
    &client_id=<id>
    &redirect_uri=<base>/oauth/client/callback
    &scope=mcp:read mcp:write
    &state=<random CSRF nonce>
    &code_challenge=<S256 hash of verifier>
    &code_challenge_method=S256
    # RFC 8707 audience binding
    &resource=<provider Url>
```

2. The app redirects the browser to that URL. The user logs in and consents *at the remote authorization server* — Kiss is not involved in that step.

3. The remote server 302-redirects the browser back to `<base>/oauth/client/callback?code=...&state=...`.

4. `OAuthCallbackServlet` handles the redirect. It validates the returned `state` against the pending registry — the entry is single-use and short-lived, which is the CSRF defense — then calls `OAuthClient.completeAuthorization`, which POSTs to the remote token endpoint:

```
POST <token_endpoint>
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code
&code=<code>
&redirect_uri=<must match step 1 exactly>
&client_id=<id>
&code_verifier=<PKCE verifier>
&resource=<provider Url>
```

   The returned tokens are persisted, and the servlet renders a small "Authorization complete — you may close this window" page.

5. Subsequent `client.getAccessToken()` calls return the stored access token, refreshing it when needed (Section 28.5).

Note

The callback servlet is a framework OAuth endpoint, in the same family as the authorization-server servlets under `org.kissweb.oauth.as`. It legitimately renders a small HTML completion page: the "backend never generates HTML" rule (Chapter 3) governs application business endpoints, not OAuth protocol endpoints. When no provider is configured, the servlet returns 404 — the same inertness convention the resource-server discovery endpoint follows.

28.5 Token Refresh and Rotation

When `getAccessToken()` finds the access token expired — treating it as expired `OAuthClientRefreshSkewSeconds` (default 60) before its real expiry so the refresh happens slightly early — it performs a `refresh_token` grant against the remote token endpoint.

The refresh is **single-flight per provider**. Concurrent callers for the same provider serialize on a per-provider lock; a thread that

wakes after another has already refreshed simply re-reads and uses the fresh token rather than refreshing again. This prevents a burst of concurrent outbound calls from firing a storm of redundant (and potentially rotation-tripping) refresh requests.

The client **honors refresh-token rotation**. If the token response carries a new `refresh_token`, it replaces the stored one; if it does not, the existing refresh token is kept. This works correctly whether the remote server rotates refresh tokens (as Kiss's own authorization server does, Section 27.7) or not.

If the remote server rejects the refresh with `invalid_grant` — for example, the refresh token was revoked, or its family was invalidated by the server's reuse detection — the client deletes the stored tokens and throws `OAuthAuthorizationRequiredException`, so the next attempt cleanly starts a fresh login rather than retrying a dead token.

28.6 Discovery and Registration in Detail

28.6.1 Discovery

Discovery resolves three endpoints — `authorization_endpoint`, `token_endpoint`, and (optionally) `registration_endpoint` — from nothing but the provider's `Url`. It follows the same well-known ladder the resource-server side uses, but in the client direction:

1. *Protected-resource metadata.* GET `<origin>/.well-known/oauth-protected-resource` (RFC 9728) and read the first entry of `authorization_servers`. That value is the issuer.

2. *Authorization-server metadata.* GET `<issuer>/.well-known/oauth-authorization-server` (RFC 8414), falling back to `<issuer>/.well-known/openid-configuration` (OpenID Connect discovery) if the first is absent. Read the endpoint URLs from the returned document.

If step 1 yields nothing — the resource publishes no protected-resource metadata — the provider URL's own origin is treated as the issuer and step 2 is attempted directly. So a server that is simultaneously the resource server and its own authorization server

still resolves with no special configuration. If the discovered metadata lacks an `authorization_endpoint` or `token_endpoint`, discovery fails with `OAuthClientException`.

The resolved metadata is cached in `oauth.db` keyed by provider name; subsequent flows reuse it without re-fetching.

28.6.2 Registration

If the provider section sets a `ClientId`, that id (with the optional `ClientSecret`) is used directly — no registration request is made.

Otherwise the framework performs RFC 7591 Dynamic Client Registration against the discovered `registration_endpoint`, registering a *public* client:

```
POST <registration_endpoint>
{
  "client_name": "Kiss OAuth Client (myprovider)",
  "redirect_uris": ["<base>/oauth/client/callback"],
  "grant_types": ["authorization_code", "refresh_token"],
  "response_types": ["code"],
  "token_endpoint_auth_method": "none",
  "scope": "mcp:read mcp:write"
}
```

`token_endpoint_auth_method=none` produces a public client whose security rests entirely on PKCE — the right shape for an application that cannot keep a client secret confidential, and what MCP servers expect. The returned `client_id` (and any `client_secret`, though public registrations rarely return one) is cached. If the provider has no configured `ClientId` *and* the remote server advertises no registration endpoint, registration fails with a configuration error pointing you at the `ClientId` key.

Note

Registration is keyed on the redirect URI as well as the provider. If you start a flow with a different redirect base URL than the cached registration used, the framework re-registers for the new redirect URI rather than reusing a registration the remote server would reject.

28.7 Persistence: The Shared oauth.db

The client does *not* open its own database. It persists into the same SQLite file the authorization server uses (`oauth.db` by default; configured by `OAuthAsSqliteFile`, Section 27.8), reusing `OAuthSqliteStore`'s single JDBC connection and its JVM-wide monitor. All access goes through the Kiss `org.kissweb.database` API (Chapters 8–9) — never raw JDBC.

Three tables, keyed by provider name, are created lazily (`CREATE TABLE IF NOT EXISTS`) on the first client read or write and sit alongside the authorization-server tables in that one file:

```
CREATE TABLE oauth_client_discovery (
    provider              TEXT PRIMARY KEY,
    issuer                TEXT,
    authorization_endpoint TEXT NOT NULL,
    token_endpoint        TEXT NOT NULL,
    registration_endpoint TEXT,
    fetched_at            INTEGER NOT NULL
);

CREATE TABLE oauth_client_registration (
    provider      TEXT PRIMARY KEY,
    client_id     TEXT NOT NULL,
    -- null for a public client
    client_secret TEXT,
    redirect_uri  TEXT NOT NULL,
    created_at    INTEGER NOT NULL
);

CREATE TABLE oauth_client_tokens (
    provider      TEXT PRIMARY KEY,
    access_token  TEXT NOT NULL,
    refresh_token TEXT,
    token_type    TEXT,
    scope         TEXT,
    expires_at    INTEGER NOT NULL,
    created_at    INTEGER NOT NULL
);
```

The in-flight `PendingAuthorization` (the `state` plus PKCE verifier for a single login in progress) is held in memory only — single-use and short-lived (10-minute TTL), so losing it on a restart costs the user at most a re-click of the login button.

Because the client shares the authorization server's file and lock, the concurrency, build-safety, and inspection notes from the AS

persistence section (Section 27.8) apply equally here. In particular:

- *Single-process.* The shared connection serializes AS and client access under one monitor; two JVMs against the same file collide with SQLite's `database is locked` rather than corrupting state.
- *Build safety.* `./bld build` excludes `oauth.db` (and its `-journal`/`-wal`/`-shm` sidecars) from the WAR copy; add the same names to `.gitignore`.
- *Client-only deployments still use `oauth.db`.* An application that uses the client role but not the authorization server still gets an `oauth.db` file; the AS tables are created but remain empty, and the client tables hold the live state.

28.8 Security Considerations

PKCE is mandatory.
S256 only. Dynamic Client Registration registers a public client (`token_endpoint_auth_method=none`) whose security rests entirely on PKCE.

`state` is a CSRF token.
The callback proceeds only when the returned `state` matches a pending, unexpired, single-use entry. A forged or replayed callback finds no matching entry and is rejected.

Redirect capture is a fixed framework servlet
at `/oauth/client/callback`. The remote server must accept that `redirect_uri`; DCR registers it automatically, otherwise whitelist it when you pre-register the client. Exact-match redirect rules at the remote server mean the base URL you use must match what was registered.

Audience binding.
Each authorization and token request carries `resource=<provider Url>` (RFC 8707) so the issued token's audience is bound to the intended remote server, not replayable elsewhere.

Token storage.
Access and refresh tokens are stored at rest in `oauth.db`, unencrypted, exactly like the authorization server's refresh tokens. Protect the file with filesystem permissions, and in production set `OAuthAsSqliteFile` to an absolute path outside the deployed webapp tree.

Run over HTTPS in production.
The authorization code and the tokens are bearer credentials in transit; TLS termination is the deployer's responsibility.

28.9 Components

All classes live in `org.kissweb.oauth.client`. Application developers touch only `OAuthClient` (plus the two exceptions); everything else is internal machinery, listed here for orientation and debugging.

`OAuthClient`
The facade: `forProvider`, `beginAuthorization`, `isAuthorized`, `getAccessToken`, `clearTokens`. The only class application code normally references.

`OAuthClientConfig`
Reads the `[OAuthClient *]` sections and the client `[main]` keys. Lazy singleton.

`OAuthClientProvider`
Immutable parsed provider: name, URL, scopes, optional client id/secret.

`OAuthMetadataDiscovery`
RFC 9728 → RFC 8414 / OIDC endpoint discovery via `RestClient`.

`DynamicClientRegistration`
RFC 7591 registration of a public client.

`Pkce`
Client-side S256 verifier/challenge generator.

`OAuthClientStore`
Persists discovery, registration, and tokens into the shared `oauth.db`.

`PendingAuthorization`
: In-memory, single-use `state` + PKCE verifier for an in-flight login (10-minute TTL).

`OAuthCallbackServlet`
: `/oauth/client/callback` — validates `state`, exchanges the code for tokens, renders the completion page.

`DiscoveryMetadata`, `ClientRegistration`, `TokenSet`
: Immutable value records for the three persisted kinds of state.

`OAuthClientException`
: Unrecoverable client error (discovery, registration, or token failure).

`OAuthAuthorizationRequiredException`
: Control-flow signal that an interactive login is required.

28.10 A Worked Example

Suppose a Kiss application needs to call a remote MCP server at `https://partner.example.com/mcp` that is OAuth-protected and supports dynamic registration. Two changes get it connected.

28.10.1 Step 1: Declare the provider in application.ini

```
[OAuthClient partner]
Url    = https://partner.example.com/mcp
Scopes = mcp:read mcp:write
# ClientId left blank => the framework will self-register
# via Dynamic Client Registration.
```

If the app runs behind a fixed public URL, also set the redirect base in `[main]` so the redirect URI is stable across requests:

```
OAuthClientRedirectBaseUrl = https://myapp.example.com
```

28.10.2 Step 2: Drive the flow from a service

A pair of JSON-RPC service methods (Chapter 6) — one to start the login, one to make the authorized call:

```
import org.kissweb.oauth.client.OAuthClient
import org.kissweb.oauth.client.OAuthAuthorizationRequiredException
import org.kissweb.RestClient
import org.kissweb.json.JSONObject

class PartnerService {

    // Returns a URL the front end should open in a browser tab
    // so the user can log in at the partner. Empty when
    // already authorized.
    void getLoginUrl(JSONObject injson, JSONObject outjson,
                     Connection db, ProcessServlet servlet) {
        OAuthClient client = OAuthClient.forProvider("partner")
        if (client.isAuthorized()) {
            outjson.put("loginUrl", "")
            return
        }
        // null => OAuthClientRedirectBaseUrl
        outjson.put("loginUrl", client.beginAuthorization(
            null))
    }

    // Makes an authorized outbound call to the partner
    // MCP server.
    void listRemoteTools(JSONObject injson, JSONObject outjson,
                         Connection db, ProcessServlet servlet) {
        OAuthClient client = OAuthClient.forProvider("partner")
        try {
            // refreshes as needed
            String token = client.getAccessToken()
            JSONObject headers = new JSONObject()
            headers.put("Authorization", "Bearer " + token)
            headers.put("Content-Type", "application/json")
            RestClient rc = new RestClient()
            JSONObject body = new JSONObject()
            body.put("method", "tools/list")
            JSONObject resp = rc.jsonCall("POST",
                "https://partner.example.com/mcp",
                body.toString(), headers)
            outjson.put("tools", resp.getJSONArray("tools"))
        } catch (OAuthAuthorizationRequiredException ignore) {
            // front end then calls getLoginUrl
            outjson.put("needsLogin", true)
        }
    }
}
```

28.10.3 What Happens at Runtime

The first time `getLoginUrl` runs, `isAuthorized()` is false, so `beginAuthorization()` discovers the partner's endpoints, dynamically registers this app as a public client, and returns an authorization URL. The user opens it, logs in and consents at the partner, and the

browser lands on `/oauth/client/callback`, which exchanges the code and stores the tokens. From then on, `listRemoteTools` calls `getAccessToken()` and gets a valid token straight from `oauth.db`, with the framework silently refreshing it whenever it ages out. No further interactive step until the refresh token itself is revoked or expires.

28.11 Troubleshooting

`/oauth/client/callback` returns 404.
No client provider is configured. Add an `[OAuthClient <name>]` section with a `Url` and restart (configuration is read once at startup).

`forProvider("x")` throws "No OAuth client provider named 'x' ".
The section header must be exactly `[OAuthClient x]` (note the single space and exact label), and it must contain a non-empty `Url`; a section with no `Url` is logged and skipped.

`beginAuthorization` throws "No redirect base URL available".
Pass a base URL to `beginAuthorization()` or set `OAuthClientRedirectBaseUrl` in `[main]`.

Discovery fails with "Could not fetch authorization-server metadata".
The provider's origin publishes neither `/.well-known/oauth-authorization-server` nor `/.well-known/openid-configuration`, or they are unreachable. Confirm the `Url` is correct and the remote server is up; check `tomcat/logs/catalina.out` for the per-URL discovery debug lines.

Registration fails with "no pre-registered ClientId and ... does not support Dynamic Client Registration".
The remote server advertises no `registration_endpoint`. Pre-register out of band and set `ClientId` (and `ClientSecret` if confidential) in the provider section.

Callback shows "Invalid or expired authorization state".
The `state` did not match a pending entry — the login took

longer than the 10-minute TTL, the server was restarted mid-flow (pending entries are in-memory), or the callback was reached without a matching `beginAuthorization()`. Start the login again.

`getAccessToken()` keeps throwing `OAuthAuthorizationRequiredException`.
Either no tokens are stored yet (run the interactive flow), or the stored refresh token was rejected with `invalid_grant` and discarded. The remote server may have revoked the refresh-token family (Section 27.7); the user must log in again.

Tokens vanish after redeploy.
`oauth.db` was inside the webapp tree and got rewritten by the WAR deploy. Set `OAuthAsSqliteFile` to an absolute path outside the deployed directory.

28.12 What's Not Implemented

Consistent with the rest of the Kiss OAuth support, the client role is deliberately scoped. Out of scope for v1:

- *Loopback redirect capture (RFC 8252).* Redirect capture is via the `/oauth/client/callback` servlet only. The loopback-listener pattern that CLI and desktop tools with no web server use is not implemented; it can be added later behind the same `OAuthClient` API.
- *Path-based protected-resource discovery.* Protected-resource metadata is fetched from the origin-root well-known path (`<origin>/.well-known/oauth-protected-resource`); the RFC 9728 path-insertion variant is not implemented. When no protected-resource document is published, the provider URL's origin is treated as the issuer and RFC 8414 / OIDC discovery is attempted directly.
- *Discovery TTL.* Discovery is cached indefinitely and re-fetched only when absent — there is no time-based refresh of endpoint metadata. Clear the `oauth_client_discovery` row to force re-discovery.
- *Shared or clustered storage.* Single-process, sharing the authorization server's `oauth.db` concurrency model (Section 27.8).

- *RSA-only / bearer tokens.* No DPoP (RFC 9449) or mTLS (RFC 8705) binding, consistent with the resource-server and authorization-server roles.

28.13 Further Reading

The standalone OAuth reference at the repository root, `OAuth.md`, covers all three roles — resource server, authorization server, and client — in one place.

Relevant RFCs:

- *RFC 6749* — The OAuth 2.0 Authorization Framework
- *RFC 7591* — OAuth 2.0 Dynamic Client Registration Protocol
- *RFC 7636* — Proof Key for Code Exchange by OAuth Public Clients (PKCE)
- *RFC 8414* — OAuth 2.0 Authorization Server Metadata
- *RFC 8707* — Resource Indicators for OAuth 2.0
- *RFC 9728* — OAuth 2.0 Protected Resource Metadata
- *draft-ietf-oauth-v2-1* (OAuth 2.1) — the consolidation that mandates PKCE and refresh-token rotation

The Model Context Protocol authorization specification (`https://modelcontextprotocol.io`) describes the client side of obtaining and presenting tokens, and is the relevant reference when the remote server is itself an MCP server.

Part V

Utilities and Advanced Topics

Chapter 29

Utility Classes Reference

Kiss provides a comprehensive set of utility classes in the `org.kissweb` package. This chapter surveys each class with its purpose and key methods.

29.1 StringUtils

`StringUtils` provides over 50 methods for string manipulation. All methods are static and null-safe — passing `null` returns a sensible default rather than throwing an exception.

29.1.1 Substring Operations

```
import org.kissweb.StringUtils

// Take first N characters (safe if string is shorter)
StringUtils.take("Hello World", 5)   // "Hello"

// Drop first N characters
StringUtils.drop("Hello World", 6)   // "World"

// Substring by start index and length (count, not end index)
StringUtils.substring("Hello World", 0, 5) // "Hello"
```

`take()` is the bounds-safe truncating helper; `substring(str, from, n)` returns `n` characters starting at `from` and is not bounds-checked.

29.1.2 Trimming

```
// Strip whitespace from the right, left, or both ends
StringUtils.rightStrip("Hello ")     // "Hello"
StringUtils.leftStrip(" Hello")      // "Hello"
StringUtils.centerStrip(" Hello ")   // "Hello"
```

29.1.3 Joining and Splitting

```
// Join a list with a delimiter
StringUtils.join(["a", "b", "c"], ", ") // "a, b, c"

// sprintf-style formatting
StringUtils.sprintf("%s has %d items",
    "Cart", 5)                          // "Cart has 5 items"
```

29.1.4 Testing and Conversion

```
// Null or empty check
StringUtils.isEmpty(null)   // true
StringUtils.isEmpty("")     // true
StringUtils.isEmpty(" ")    // false

// HTML to plain text
StringUtils.htmlToText(
    "<p>Hello <b>World</b></p>") // "Hello World"

// Fuzzy matching (candidates Vector first, then the target)
Vector<String> candidates =
    new Vector<>(["color", "column", "collar"])
StringUtils.findClosestMatch(candidates, "colour")
    // "color"
```

29.1.5 Key StringUtils Methods

Method	Description
`take(str, n)`	First `n` characters (safe).
`drop(str, n)`	String after dropping first `n` characters.
`substring(str, from, n)`	`n` characters starting at `from` (length, not end index).

Method	Description
`rightStrip(str)`	Trim trailing whitespace.
`leftStrip(str)`	Trim leading whitespace.
`centerStrip(str)`	Trim whitespace from both ends.
`join(strings, delim)`	Join a `String[]` with a delimiter.
`sprintf(fmt, args...)`	Format a string (like C sprintf).
`isEmpty(str)`	True if null or empty string.
`htmlToText(html)`	Strip HTML tags and decode entities.
`findClosestMatch(candidates, target)`	Closest Levenshtein match; candidates is a `Vector<String>`.

29.2 DateUtils

`DateUtils` handles dates as YYYYMMDD integers. This format avoids timezone ambiguity and simplifies date arithmetic and database storage.

```
import org.kissweb.DateUtils

int today = DateUtils.today()           // e.g., 20260404
Date d = DateUtils.toDate(20260404)     // Java Date
int dt = DateUtils.toInt(new Date())    // YYYYMMDD int

String s = DateUtils.format("MM/dd/yyyy", 20260404)
    // "04/04/2026"

int next = DateUtils.addDays(20260404, 7) // 20260411

int y = DateUtils.year(20260404)        // 2026
int m = DateUtils.month(20260404)       // 4
int d = DateUtils.day(20260404)         // 4
```

Method	Description
`today()`	Current date as YYYYMMDD.
`toInt(Date)`	Convert Date to YYYYMMDD.
`toDate(int)`	Convert YYYYMMDD to Date.
`format(fmt, int)`	Format date with pattern.
`addDays(int, n)`	Add/subtract days.

Method	Description
`year(int)`, `month(int)`, `day(int)`	Extract date components.

29.3 TimeUtils

`TimeUtils` handles times as HHMM integers (e.g., 1430 for 2:30 PM).

```
import org.kissweb.TimeUtils

int now = TimeUtils.now()               // e.g., 1430
String mil = TimeUtils.formatMilitary(1430) // "14:30"
String ampm = TimeUtils.formatAMPM(1430) // "2:30 PM"
int t = TimeUtils.parse("2:30 PM")      // 1430

int h = TimeUtils.hour(1430)           // 14
int m = TimeUtils.minutes(1430)        // 30
```

29.4 DateTime

`DateTime` wraps Java's `ZonedDateTime` for combined date/time operations with timezone awareness.

```
import org.kissweb.DateTime

DateTime dt = new DateTime()        // now
DateTime dt2 = new DateTime(someDate) // from java.util.Date

int date = dt.getIntDate()   // YYYYMMDD
int time = dt.getIntTime()   // HHMM
Date javaDate = dt.getDate() // java.util.Date

// "MM/dd/yyyy h:mm a"
String s = dt.format()
// custom pattern
String s2 = dt.format("MM/dd/yyyy h:mm:ss a")

dt = dt.addDays(7)    // each returns a new DateTime
dt = dt.addHours(3)
dt = dt.addMinutes(45)
```

Note

Use `DateTime` when you need combined date/time handling. Use `DateUtils` and `TimeUtils` when you only need the date or time component.

29.5 NumberUtils and NumberFormat

29.5.1 NumberUtils

`NumberUtils` provides numeric comparison and rounding:

```
import org.kissweb.NumberUtils

// Approximate equality for floating-point
NumberUtils.doubleEqual(0.1 + 0.2, 0.3, 0.0001) // true

// Rounding
NumberUtils.round(3.14159, 2)     // 3.14
NumberUtils.roundUp(3.141, 2)     // 3.15
NumberUtils.roundDown(3.149, 2)   // 3.14

// Parsing
double d = NumberUtils.parseDouble("1,234.56")
```

29.5.2 NumberFormat

`NumberFormat` provides full-featured number formatting for reports and displays. It is covered in detail in Chapter 21, Section 21.3. Quick reference:

```
import org.kissweb.NumberFormat

// Currency: ($12,345.35)
NumberFormat.Format(-12345.348, "CDP", 12, 2)

// With commas: 1,234,567
NumberFormat.Format(1234567, "C", 10, 0)

// Zero-filled: 00042
NumberFormat.Format(42, "Z", 5, 0)

// Percent: 85.5%
NumberFormat.Format(85.5, "R", 6, 1)
```

Format mask codes: `B` (blank if zero), `C` (commas), `D` (dollar sign), `L` (left justify), `P` (parentheses for negatives), `Z` (zero fill), `U` (uppercase), `R` (percent sign).

29.6 Choosing a Cryptography or Hashing Utility

Kiss provides four distinct classes for cryptographic operations, covered in the sections that follow. They are kept deliberately separate rather than merged behind one facade — a facade would put a reversible-encryption call and a one-way-hash call one method name apart, which is exactly the ambiguity that invites misuse (e.g. "just encrypting a password reversibly" instead of hashing it). Use the table below to pick the right one for a given need:

Need	Use
Recover the original value later (reversible encryption)	`Crypto` (§29.7)
Store or verify a user's password (never recoverable)	`PasswordHash` (§29.8)
Plain digest, checksum, or content fingerprint — no secret key involved	`Hash` (§29.9)
Keyed digest/MAC, or a deterministic "blind index" for equality search over an encrypted column	`Hmac` (§29.10)

Two smaller shared utilities support all four: `ConstantTime` (timing-safe byte-array comparison) and `RandomUtil` (a shared `SecureRandom`-backed helper), both covered in §29.11.

29.7 Crypto

`Crypto` provides reversible AES-256-GCM authenticated encryption for strings and byte arrays — data a legitimate caller must later recover in plaintext. It is the wrong tool for passwords; see

`PasswordHash` (§29.8) for those. Every encrypted value is a self-describing, versioned container carrying its own random nonce and its own 128-bit authentication tag, so tampering or corruption is detected on decrypt rather than silently producing garbage plaintext. A `Crypto` instance may safely be shared across threads.

29.7.1 Two Ways to Supply the Key

`Crypto` offers two independent construction paths, for two different situations:

- **Password-based** (`new Crypto(password)`, `Crypto.deriveOnce(password, salt)`) — the AES key is derived from a human-choosable password (and optional salt) with PBKDF2 (65,536 iterations, 256-bit derived key). PBKDF2 exists to slow down brute-forcing a low-entropy secret, so this path makes sense when the input really is a password.
- **Raw key** (`Crypto.fromKey(byte[])`, `Crypto.fromKeyBase64(String)`) — the caller supplies an already high-entropy 256-bit key directly, with no PBKDF2, no salt, and no per-call KDF cost. This is the recommended path for bulk/high-volume field encryption, e.g. a master key loaded once from `application.ini` and reused for many rows. Use `Crypto.generateKey()` / `Crypto.generateKeyBase64()` to provision a fresh key.

```
import org.kissweb.Crypto

// Password-based (low-volume, or an actual human passphrase)
Crypto crypto = new Crypto(myPassword)
String encrypted = crypto.encryptWithRandomSalt("Hello World")
String decrypted = crypto.decryptWithRandomSalt(encrypted)

// Raw 256-bit key, no PBKDF2 -- recommended for
// bulk/high-volume field encryption (e.g. a master key from
// application.ini)
// provision once
String key = Crypto.generateKeyBase64()
Crypto fast = Crypto.fromKeyBase64(key)
String encrypted2 = fast.encryptWithRandomSalt("Hello World")
String decrypted2 = fast.decryptWithRandomSalt(encrypted2)
```

Tip

Applying PBKDF2 to a key that is already high-entropy adds latency without adding security — there is nothing for PBKDF2 to usefully slow down. Prefer `fromKey`/`fromKeyBase64` whenever the "password" is really a machine-generated key rather than a human passphrase.

Regardless of which path is used, per-value non-determinism (the same plaintext encrypting to different ciphertext each time) comes solely from the random 12-byte GCM nonce generated on every encrypt call — never from the key derivation — so no security property differs between the two paths; only the KDF cost does.

29.7.2 Explicit and Random Salts

```
// With an explicit salt
String enc = crypto.encrypt("salt123", "sensitive data")
String dec = crypto.decrypt("salt123", enc)

// With a random salt embedded in the output
String enc2 = crypto.encryptWithRandomSalt("sensitive data")
String dec2 = crypto.decryptWithRandomSalt(enc2)
```

`deriveOnce(password, salt)` builds a password-based instance exactly like `new Crypto(password)`, except the PBKDF2 derivation for that salt runs immediately at construction rather than lazily on the first encrypt/decrypt call. The output is byte-for-byte compatible with (and decryptable by) a plain `new Crypto(password)` instance — it changes only *when* the PBKDF2 cost is paid, not the format or the key. This is useful when a caller wants that one-time cost to land at a predictable point (e.g. once at the start of a batch) rather than on whichever value happens to be encrypted first.

Note

Key derivation is cached, not repeated per call. For a password-based instance, the PBKDF2-derived key for a given (password, salt) pair is computed at most once per instance and reused for every later call with that same salt — no construction path re-derives the key on every encrypt/decrypt call.

29.7.3 Versioned Container and Backward Compatibility

Every value produced by the current format is a self-describing container: a magic prefix, a version byte, the salt (if any), the random nonce, and the GCM ciphertext with its authentication tag. String outputs are additionally prefixed `$KC1$` and Base64 encoded. Two versions share this envelope shape:

- `VERSION_1` — produced by the password-based path; the salt length is embedded and may be non-zero.
- `VERSION_2` — produced by `fromKey`/`fromKeyBase64`; no salt applies, so the salt length is always zero. A `VERSION_2` value can only be decrypted by an instance constructed via `fromKey`/`fromKeyBase64` — a password-based instance throws `IllegalStateException`, since there is no password or derivation to fall back on.

Data produced by older (pre-GCM) versions of this class — unauthenticated `AES/ECB` — remains transparently decryptable: decryption auto-detects the legacy format and falls back to the original algorithm. All new encryptions always use one of the current GCM formats. A peer running code that predates `VERSION_2` can still decrypt everything it always could, but cannot decrypt values produced via `fromKey`/`fromKeyBase64` on a newer version.
Key methods (each has a `String` and a `byte[]` form unless noted):

Method	Description
`new Crypto(password)`	Create a password-based instance (PBKDF2, `VERSION_1`).
`Crypto.deriveOnce(password, salt)`	Same as above, but derives the key immediately at construction.
`Crypto.fromKey(key256)`	Create a raw-key instance from 32 bytes; no PBKDF2 (`VERSION_2`).
`Crypto.fromKeyBase64(base64Key)`	Same, from a Base64-encoded 256-bit key.
`Crypto.generateKey()`	Generate a fresh random 256-bit key.
`Crypto.generateKeyBase64()`	Same, Base64 encoded.
`encrypt(value)`	Encrypt using the instance password/key.
`encrypt(salt, value)`	Encrypt with an explicit salt (ignored in raw-key mode).
`decrypt(value)`	Decrypt a value.
`decrypt(salt, value)`	Decrypt a value encrypted with a salt.
`encryptWithRandomSalt(value)`	Encrypt with a random salt embedded in the output.
`decryptWithRandomSalt(value)`	Decrypt a random-salt value.

Warning

Store passwords and raw keys securely (e.g., in `application.ini`), not in source code.

Warning

`Crypto` is for data you must be able to recover. Never use it to store user passwords — those must be hashed one-way with `PasswordHash` (§29.8).

29.8 PasswordHash

`PasswordHash` stores user passwords using one-way salted PBKDF2-HMAC-SHA256 hashing (600,000 iterations, the OWASP 2023 floor, with a random 16-byte salt per password). Passwords are hashed, never encrypted: there is intentionally no way to recover the original password, only to verify a candidate against a stored hash, in constant time (§29.11).

```
import org.kissweb.PasswordHash

// When setting or changing a password:
String stored = PasswordHash.hash(plainTextPassword)

// When authenticating:
boolean ok = PasswordHash.verify(enteredPassword, stored)
```

Key methods (all static):

Method	Description
`hash(password)`	Hash a plain-text password for storage.
`verify(password, stored)`	Constant-time verify a candidate against a stored hash.
`isHashed(stored)`	Whether a stored value is in `PasswordHash` format.
`needsRehash(stored)`	Whether a stored hash should be replaced with a fresh one on next successful login.

The stored value is a self-describing string — the literal `pbkdf2`, the iteration count, the Base64 salt, and the Base64 hash, each separated by a `$` — and is approximately 80 characters long, so the database column that holds it must be at least `varchar(255)`.

29.8.1 Upgrading the Work Factor: `needsRehash`

Because the iteration count is embedded in the stored value, `PasswordHash` can raise its default work factor over time without invalidating hashes that already exist. `needsRehash(stored)` returns `true` when a stored hash's embedded iteration count is lower than the class's current default — or when `stored` is not even in

`PasswordHash` format at all. It is purely advisory: it never invalidates `stored` and has no bearing on `verify()`, which keeps verifying any value it always could.
The standard pattern is to check `needsRehash` immediately after a successful `verify()` and, if it returns `true`, re-hash the just-verified plain-text password and save it — gradually migrating every stored password up to the current work factor as users log in, with no bulk migration required:

```
boolean ok = PasswordHash.verify(enteredPassword, stored)
if (ok && PasswordHash.needsRehash(stored))
    // upgrade in place, save 'stored'
    stored = PasswordHash.hash(enteredPassword)
```

29.9 Hash

`Hash` provides general-purpose, unkeyed cryptographic digests — SHA-256, SHA-384, SHA-512, and their SHA-3 counterparts — for content fingerprints, dedup/change detection, checksums, and cache keys, where the input is not secret and speed is a feature rather than a liability. It is pure JDK (`MessageDigest` has supported SHA-3 natively since Java 9), so no external dependency is required for any algorithm here.

Warning

`Hash` is not the right tool for passwords (use `PasswordHash`, §29.8) or for anything needing a secret key or proof of origin (use `Hmac`, §29.10). A plain digest has neither property.

```
import org.kissweb.Hash

byte[] digest = Hash.sha256("some content")
String hex    = Hash.sha256Hex("some content")
String b64    = Hash.sha256Base64("some content")

// Streamed, so the whole file is never loaded into memory
String fileHash = Hash.hashFileHex(new File("upload.bin"),
    "SHA-256")
```

Each of `sha256`, `sha384`, `sha512`, `sha3_256`, and `sha3_512` is avail-

able in three forms: a raw `byte[]` digest (from a `byte[]` or a UTF-8 `String`), a lowercase-hex `...Hex` form, and a standard-Base64 `...Base64` form.

Method	Description
`sha256(data)` / `sha256Hex(data)` / `sha256Base64(data)`	SHA-256 digest, raw/hex/Base64.
`sha384(data)` / `sha384Hex(data)` / `sha384Base64(data)`	SHA-384 digest, raw/hex/Base64.
`sha512(data)` / `sha512Hex(data)` / `sha512Base64(data)`	SHA-512 digest, raw/hex/Base64.
`sha3_256(data)` / `sha3_256Hex(data)` / `sha3_256Base64(data)`	SHA3-256 digest, raw/hex/Base64.
`sha3_512(data)` / `sha3_512Hex(data)` / `sha3_512Base64(data)`	SHA3-512 digest, raw/hex/Base64.
`hash(inputStream, algorithm)`	Digest a stream in fixed-size chunks; caller closes the stream.
`hashFile(file, algorithm)`	Digest a file's content, streamed.
`hashFileHex(file, algorithm)`	Same, lowercase-hex encoded.

29.10 Hmac

`Hmac` provides keyed message authentication codes — HMAC-SHA256, HMAC-SHA384, and HMAC-SHA512 — pure JDK (`javax.crypto.Mac`, no external dependency). Unlike `Hash`, every digest here is parameterized by a secret key: only someone holding the key can produce or verify a matching tag for a given message.

```
import org.kissweb.Hmac

byte[] mac = Hmac.hmacSha256(secretKey,
    "message to authenticate")
String hex = Hmac.hmacSha256Hex(secretKey,
    "message to authenticate")
```

As with `Hash`, each algorithm is available as a raw `byte[]` MAC plus lowercase-hex and standard-Base64 convenience forms:

Method	Description
`hmacSha256(key, message)` / `hmacSha256Hex(...)` / `hmacSha256Base64(...)`	HMAC-SHA256, raw/hex/Base64.
`hmacSha384(key, message)` / `hmacSha384Hex(...)` / `hmacSha384Base64(...)`	HMAC-SHA384, raw/hex/Base64.
`hmacSha512(key, message)` / `hmacSha512Hex(...)` / `hmacSha512Base64(...)`	HMAC-SHA512, raw/hex/Base64.

To compare an HMAC output against an expected value, use `ConstantTime.equals` (§29.11) rather than `Arrays.equals` or `String.equals`, which return as soon as they find the first differing byte and so can leak timing information about secret-derived values.

29.10.1 The Blind-Index Pattern

`Crypto`'s AES-GCM encryption (§29.7) is deliberately non-deterministic — the same plaintext encrypts to different ciphertext every time, via a fresh random nonce — which is exactly what makes it safe. It also means `WHERE encrypted_column = ?` cannot work directly against it. A *blind index* restores equality lookup: store a deterministic, keyed HMAC of the (normalized) plaintext in a separate column alongside the encrypted one, and query against that instead:

```
// write path
row.encryptedValue = Crypto.fromKey(fieldEncryptionKey)
                        // reversible, for display/export
                        .encryptWithRandomSalt(value);
row.blindIndex = Hmac.hmacSha256Hex(fieldIndexKey,
```

```
                            // deterministic, for lookup only
                            normalize(value));

// read path -- an indexed equality lookup, no decryption
// of any row
String candidateIndex = Hmac.hmacSha256Hex(fieldIndexKey,
    normalize(candidateValue));
// ... SELECT ... WHERE blind_index = candidateIndex ...
```

Two responsibilities stay with the caller, not with `Hmac`, because they are application- and data-shape-specific decisions:

- **Key separation** — `fieldIndexKey` must be a *different* key from `fieldEncryptionKey`. Never reuse one secret for two primitives; keep each as its own entry in `application.ini`.
- **Normalization** — normalize the plaintext before hashing (trim whitespace, strip punctuation, fix case, etc., as appropriate) so formatting variance — not a real difference in the underlying value — does not defeat equality lookup. What "normalized" means is inherently specific to the value being indexed.

29.11 ConstantTime and RandomUtil

29.11.1 ConstantTime

`ConstantTime` provides a single method, `equals(byte[] a, byte[] b)`, that compares two byte arrays without an early return on the first differing byte, so its running time depends only on the arrays' length, not their content. A naive `Arrays.equals`-style comparison returns as soon as it finds a mismatch, which can — in principle — leak how many leading bytes matched to an attacker measuring response time. Use it whenever comparing a MAC, password hash, or blind-index value against an expected value.

```
import org.kissweb.ConstantTime;

boolean same = ConstantTime.equals(expectedMac, suppliedMac);
```

`PasswordHash` (§29.8) uses `ConstantTime` internally to verify a candidate password's derived hash against the stored one.

29.11.2 RandomUtil

`RandomUtil` provides shared, cryptographically strong random values backed by one lazily-seeded `SecureRandom` instance, so new code has one obvious place to get random key material or tokens from rather than each caller instantiating its own `SecureRandom`.

```
import org.kissweb.RandomUtil;

// e.g. a raw Crypto key
byte[] key  = RandomUtil.randomBytes(32);
// a human-typeable random code
String token = RandomUtil.randomAlphaNumeric(20);
```

Method	Description
`randomBytes(n)`	`n` cryptographically strong random bytes.
`randomAlphaNumeric(len)`	Random string of `len` upper/lowercase letters and digits.

`Crypto.generateKey()` (§29.7) is built on `RandomUtil.randomBytes`.

29.12 Image

`Image` is a static utility class for server-side image manipulation. It does not need to be instantiated. Operations include resizing, in-place rotation, EXIF orientation correction, recompression of both raster images and PDFs, and conversion of HEIC files to PNG. Supported raster formats for resize, rotate, and recompression are JPEG, BMP, PNG, and GIF. PDF compression works by recompressing every embedded image in the document at the specified quality level.

Note

`Image` depends on two external libraries: EXIF reading uses `metadata-extractor` and PDF compression uses `Apache PDFBox`. Both must be present on the classpath for the respective operations. HEIC decoding additionally requires one external command-line tool on the host's `PATH` (see §29.12.6).

29.12.1 Compression

```
import org.kissweb.Image;

// Compress a JPEG to 70% quality
byte[] smaller = Image.compressImage("jpg", originalBytes,
    0.7f);

// Compress a PDF (recompresses all embedded images)
byte[] smallerPdf = Image.compressImage("pdf", pdfBytes, 0.6f);
```

The `quality` parameter is a value between 0.0 and 1.0; lower values produce smaller files with more visible loss. If `type` is `null` or the format has no writer, the original byte array is returned unchanged.

29.12.2 File I/O

```
// Write raw bytes to a file (creates or overwrites)
Image.writefile("/var/uploads/photo.jpg", imageBytes);

// Read an entire file into a byte array
byte[] data = Image.readfile("/var/uploads/photo.jpg");
```

29.12.3 Resizing

```
// Resize so the longer axis is at most 1024 pixels;
// aspect ratio is preserved.
byte[] thumb = Image.resizeImage(originalBytes, "jpg", 1024);
```

If both axes are already within 10% of the target size, the original bytes are returned as-is. Unsupported types are also returned unchanged.

29.12.4 Rotation

`rotateLeft` and `rotateRight` operate directly on a file, writing the result back in place. Exceptions are silently swallowed and the file is left unchanged on failure.

```
// Rotate the file 90 degrees counter-clockwise
Image.rotateLeft("/var/uploads/photo.jpg");
```

```
// Rotate the file 90 degrees clockwise
Image.rotateRight("/var/uploads/photo.jpg");
```

To rotate an in-memory `BufferedImage` by an arbitrary angle, use `rotateImage`. For 90° and 270° rotations the output dimensions are swapped (width becomes height and vice versa); all other angles keep the original dimensions.

```
BufferedImage rotated = Image.rotateImage(src, 90); // CW
```

29.12.5 EXIF Orientation

Cameras embed an orientation tag in JPEG EXIF metadata to record how the device was held when the photo was taken. Without correcting for this tag, an image that looks correct in a camera roll may appear rotated or flipped when displayed in a browser or stored without the EXIF metadata.

`getExifOrientation` reads the tag from raw image bytes and returns a value from 1 to 8 (EXIF standard), where 1 means no correction is needed. It returns 1 on any failure, including missing EXIF data or an unsupported format.

`applyExifOrientation` applies the appropriate rotation and/or flip for a given orientation value. It has two overloads: one that accepts and returns a `BufferedImage`, and one that accepts and returns a byte array in a named format.

```
// --- Auto-orient an upload before storing it ---

// 1. Read orientation BEFORE any destructive operation
//    (compression and resize strip EXIF data)
int orientation = Image.getExifOrientation(uploadBytes);

// 2. Resize for storage
byte[] resized = Image.resizeImage(uploadBytes, "jpg", 2048);

// 3. Apply orientation correction to the resized bytes
byte[] corrected =
    Image.applyExifOrientation(resized, orientation, "jpg");

// 4. Store the corrected image
Image.writefile("/var/uploads/photo.jpg", corrected);
```

Tip

Always call `getExifOrientation` on the original upload bytes *before* resizing or compressing. Both operations discard EXIF metadata, so the orientation tag will no longer be readable afterwards.

29.12.6 HEIC to PNG Conversion

`convertHeicToPng` decodes a HEIC image and re-encodes it as PNG. HEIC uses the HEVC codec, for which no production-quality pure-Java decoder exists, so the method delegates to an external command-line converter chosen from the host's `PATH`. The first tool found from the following list is used, and the choice is cached for the life of the JVM:

1. `sips` — built into macOS; no install required.
2. `heif-convert` — Linux; part of the `libheif-tools` or `libheif-examples` package.
3. `magick` — ImageMagick 7 with the HEIC delegate; typical Windows install path.
4. `ffmpeg` — universal fallback; official prebuilt binaries are available for Linux, macOS, and Windows.

Two overloads are provided, one for byte arrays and one for file paths:

```
import org.kissweb.Image;

// Byte-array in / byte-array out (uses temp files internally)
byte[] pngBytes = Image.convertHeicToPng(heicBytes);

// File-to-file
Image.convertHeicToPng("/var/uploads/photo.heic",
                       "/var/uploads/photo.png");
```

Note

If none of the supported tools is on `PATH`, both overloads throw `IOException` with a message that lists the acceptable installs. To enable HEIC support on each platform:

- **macOS:** nothing to install — `sips` ships with the OS.
- **Linux:** `apt install libheif-examples` (Debian/Ubuntu) or `dnf install libheif-tools` (Fedora/RHEL).
- **Windows:** install ImageMagick 7 (with the HEIC delegate selected) or ffmpeg; both provide standard installers.

Warning

Each conversion launches an external process and is bounded by a five-minute timeout. Do not call `convertHeicToPng` inside a tight loop on large batches without considering throughput; the per-call process startup cost dominates small-image workloads.

Key methods:

Method	Description
`compressImage(type, bytes, q)`	Compress image or PDF bytes at quality `q` (0.0–1.0).
`writefile(fname, bytes)`	Write bytes to a file.
`readfile(fname)`	Read a file into a byte array.
`resizeImage(bytes, type, size)`	Resize to max dimension `size` px, preserving aspect ratio.
`rotateLeft(fileName)`	Rotate file 90° counter-clockwise in place.
`rotateRight(fileName)`	Rotate file 90° clockwise in place.
`rotateImage(src, angle)`	Rotate a `BufferedImage` by `angle` degrees clockwise.
`getExifOrientation(bytes)`	Read EXIF orientation tag (1–8); returns 1 on failure.

Method	Description
`applyExifOrientation(img, o)`	Apply EXIF transform to a `BufferedImage`.
`applyExifOrientation(bytes, o, fmt)`	Apply EXIF transform to a byte array; return corrected bytes.
`convertHeicToPng(bytes)`	Decode HEIC bytes, return PNG bytes (external tool).
`convertHeicToPng(heicPath, pngPath)`	Convert a HEIC file to a PNG file (external tool).

29.13 IniFile

`IniFile` provides a thread-safe parser for INI configuration files with section support, comments, and typed value access.

```
import org.kissweb.IniFile

IniFile ini = IniFile.load("config.ini")

// Read values from a section
String host = ini.get("database", "host")
int port = ini.getInt("database", "port")
boolean debug = ini.getBoolean("app", "debug")
double rate = ini.getDouble("billing", "rate")

// Write values
ini.put("app", "version", "2.1")
ini.save()
```

INI file format:

```
[database]
host = localhost
port = 5432

[app]
debug = true
version = 2.0
```

Method	Description
`IniFile.load(path)`	Load and parse an INI file (static).
`get(section, key)`	Get a string value.
`getInt(section, key)`	Get an integer value.
`getDouble(section, key)`	Get a double value.
`getBoolean(section, key)`	Get a boolean value.
`getSection(section)`	Get all keys in a section as a map.
`getSectionNames()`	Get the names of every section in the file.
`put(section, key, val)`	Set a value.
`removeValue(section, key)`	Set a key's value to null (key remains).
`removeKey(section, key)`	Remove a key/value pair from a section.
`removeSection(section)`	Remove an entire section.
`save()`	Write changes to disk.

`getSectionNames()` and `removeSection()` let you walk and manipulate an INI file whose section names aren't known at compile time — for example, persisting one logical record per section. This is what the OAuth authorization server (Chapter 27) does to track registered clients and refresh tokens, with one section per entry keyed by id.

Note

The framework's own `application.ini` uses a simplified format without section headers. The `IniFile` class supports both formats — sectionless keys are accessed with `null` as the section parameter.

29.14 JsonPath

`JsonPath` provides XPath-like access to `JSONObject` structures using dot-separated paths. It returns `null` for missing paths instead of throwing exceptions, making it safe for navigating deeply nested JSON.

```
import org.kissweb.JsonPath
```

```
JSONObject data = ... // complex nested JSON

// Safe navigation with dot paths
String name = JsonPath.getString(data, "user.name")
int age = JsonPath.getInteger(data, "user.profile.age")
double salary = JsonPath.getDouble(
    data, "user.employment.salary")
boolean active = JsonPath.getBoolean(
    data, "user.isActive")

// Get a nested object
Object nested = JsonPath.getObject(
    data, "user.address")
```

Method	Description
`getObject(json, path)`	Get any value at path.
`getString(json, path)`	Get string at path.
`getInteger(json, path)`	Get integer at path.
`getDouble(json, path)`	Get double at path.
`getBoolean(json, path)`	Get boolean at path.

29.15 Base64

`Base64` provides encoding and decoding for binary data:

```
import org.kissweb.Base64

// Encode
String encoded = Base64.encode("Hello World".getBytes())

// Decode
byte[] decoded = Base64.decode(encoded)
String original = new String(decoded)
```

29.16 URLBuilder

`URLBuilder` constructs URLs with proper parameter encoding:

```
import org.kissweb.URLBuilder

String url = new URLBuilder("https://api.example.com/search")
```

```
    .addParameter("q", "hello world")
    .addParameter("page", "1")
    .addParameter("filter", "name=John&age>30")
    .build()
// "https://api.example.com/search
//     ?q=hello+world&page=1&filter=name%3DJohn%26age%3E30"
```

29.17 XML

`XML` provides parsing and transformation utilities for XML documents:

```
import org.kissweb.XML

// Parse XML string
Document doc = XML.parse(xmlString)

// Parse XML file
Document doc = XML.parseFile("data.xml")

// Transform with XSLT
String result = XML.transform(doc, xsltDoc)
```

Note

XML parsing rejects DTDs and external entities, so it is not susceptible to XML external entity (XXE) attacks. Documents that declare a `<!DOCTYPE>` are refused.

29.18 LRUCache and LRUSet

`LRUCache` and `LRUSet` are thread-safe collections with automatic eviction based on size and age.

29.18.1 LRUCache

A key/value cache that evicts the least recently used entries:

```
import org.kissweb.LRUCache

// Max 1000 entries, expire after 3600 seconds
LRUCache<String, UserProfile> cache =
```

```
    new LRUCache<>(1000, 3600)

// Store and retrieve
cache.put("user-123", profile)
UserProfile p = cache.get("user-123")

// Check existence
if (cache.containsKey("user-123")) { ... }
```

29.18.2 LRUSet

A set with the same eviction policy but without values:

```
import org.kissweb.LRUSet

// Track recently seen IPs (max 10000, expire 600s)
LRUSet<String> recentIPs = new LRUSet<>(10000, 600)

recentIPs.add("192.168.1.100")
boolean seen = recentIPs.contains("192.168.1.100")
```

29.19 ThreadLevelCache

`ThreadLevelCache` provides a per-thread key/value cache. Each thread gets its own isolated storage, making it useful for request-scoped caching without synchronization.

```
import org.kissweb.ThreadLevelCache

// Store value for current thread
ThreadLevelCache.put("currentUser", userData)

// Retrieve (returns null if not set)
Object val = ThreadLevelCache.get("currentUser")
```

29.20 ArrayUtils

`ArrayUtils` converts between primitive and boxed byte arrays:

```
import org.kissweb.ArrayUtils

// Boxed Byte[] -> primitive byte[]
```

```
byte[] prim = ArrayUtils.toPrimitiveByteArray(boxed)

// Primitive byte[] -> boxed Byte[]
Byte[] boxed2 = ArrayUtils.toWrapperByteArray(prim)
```

29.21 BuildUtils

`BuildUtils` executes system commands:

```
import org.kissweb.BuildUtils

// Run a shell command (synchronous)
BuildUtils.runShell("pg_dump -h localhost -U user " +
    "-d mydb -f backup.sql")
```

The method waits for the command to complete. Use it for system-level operations like database backups, file conversions, or external tool invocations.

29.22 JsonTemplate

`JsonTemplate` substitutes `${NAME}` placeholders in a JSON template with values from a map, then parses the result. The template must be valid JSON (or a path to a JSON file), and `fill()` returns a `JSONObject`:

```
import org.kissweb.JsonTemplate
import org.kissweb.json.JSONObject

Map<String, String> values = [
    "name": "Alice Smith",
    "date": "04/04/2026"
]

// Single quotes keep ${...} literal (Groovy does not
// interpolate single-quoted strings).
String template =
    '{ "greeting": "Dear ${name}",' +
    '  "when": "${date}" }'

JSONObject result = JsonTemplate.fill(template, values)
String greeting = result.getString("greeting")
```

29.23 GroovyUtils

`GroovyUtils` provides helpers for Groovy integration, primarily used internally by the framework's service discovery and compilation system.

29.24 Summary

The following table provides a quick-reference guide to all utility classes:

Class	Purpose
`StringUtils`	String manipulation, trimming, joining, HTML conversion.
`DateUtils`	Dates as YYYYMMDD integers.
`TimeUtils`	Times as HHMM integers.
`DateTime`	Combined date/time with timezone support.
`NumberUtils`	Numeric comparison, rounding, parsing.
`NumberFormat`	Advanced number formatting for reports.
`Crypto`	Reversible AES-GCM encryption and decryption.
`PasswordHash`	One-way salted PBKDF2 password hashing and verification.
`Hash`	Unkeyed SHA-256/384/512 and SHA-3 digests.
`Hmac`	Keyed HMAC-SHA256/384/512 MACs and the blind-index pattern.
`ConstantTime`	Timing-safe byte-array comparison.
`RandomUtil`	Shared `SecureRandom`-backed random bytes/strings.
`IniFile`	INI configuration file parser.
`JsonPath`	Safe dot-path navigation of JSON.
`Base64`	Binary encoding and decoding.
`URLBuilder`	URL construction with parameter encoding.
`XML`	XML parsing and XSLT transformation.
`LRUCache`	Thread-safe key/value cache with eviction.
`LRUSet`	Thread-safe set with eviction.

Class	Purpose
`ThreadLevelCache`	Per-thread key/value storage.
`ArrayUtils`	Byte array primitive/boxed conversions.
`BuildUtils`	System command execution.
`JsonTemplate`	`${NAME}` placeholder substitution.
`GroovyUtils`	Groovy integration helpers.
`Image`	Resize, rotate, EXIF correction, image/PDF compression, and HEIC-to-PNG conversion.

Chapter 30

Desktop Applications

Kiss applications can be packaged as desktop applications using Electron, providing a native window experience while running the same web-based code. This chapter covers the Electron integration, configuration, and packaging.

30.1 Overview

Electron wraps a Chromium browser and a Node.js runtime into a standalone executable. A Kiss application runs inside this wrapper: Electron launches the embedded Tomcat backend, waits for it to become ready, and then opens the frontend in a native window. The user sees a desktop application rather than a browser tab.

The key advantage is code reuse: the same backend services, frontend screens, and database layer work identically in both web and desktop modes. Only the launch mechanism differs.

30.2 Prerequisites

Desktop mode requires:

- **Node.js** — version 16 or later.
- **npm** — included with Node.js.
- **Electron** — installed as a dev dependency via `npm install`.

The Kiss project includes a `package.json` that declares Electron as a dependency. Run `npm install` in the project root to install it:

```
cd /path/to/Kiss
npm install
```

30.3 How It Works

The Electron integration is driven by `start-electron.js` in the project root. When launched, it:

1. Spawns the Java backend via `./bld start-backend`.
2. Polls the backend (`http://localhost:8001` by default — the `backendUrl` constant at the top of `start-electron.js`, which must match `portBase` + 1 from `Tasks.java`) with exponential backoff (500 ms intervals, 60-second timeout) until it is ready.
3. Opens a native window pointing to the frontend.
4. On shutdown, gracefully stops the backend via `./bld stop-backend`.

30.4 Launching the Desktop Application

```
# Start in desktop mode
npm start
```

This runs the `start` script defined in `package.json`, which launches Electron with `start-electron.js` as the main process.

30.5 Backend URL Detection

The stock launcher loads the window directly from the Tomcat origin, so the pages it displays carry the build-stamped `SystemInfo.sameOriginBackend` flag and call the backend at their own origin (Section 12.9) — no URL configuration in the frontend at all.

If an application instead packages the frontend files into Electron and loads them via `file://`, `index.js` detects that protocol and defaults the backend to `http://localhost:8001` (the backend of the default port block):

```
if (window.location.protocol === "file:") {
    // Desktop mode: backend is on localhost
    Server.setURL("http://localhost:8001");
}
```

An application using a different `portBase` sets `SystemInfo.backendUrl` explicitly in that case. The same frontend code works in both web and desktop modes.

30.6 Development Mode

The `start-electron.js` file includes a `devMode` flag. When enabled:

- Chrome DevTools are available (right-click, Inspect).
- Hot reload works with Ctrl+R.
- `nodeIntegration` is enabled for debugging.

For production builds, set `devMode` to `false` to disable these features for security.

30.7 Packaging and Distribution

To distribute the application as a standalone installer, use Electron's build tools:

```
# Install electron-builder
npm install electron-builder --save-dev

# Build for the current platform
npx electron-builder
```

This produces platform-specific installers:

- **Windows** — `.exe` installer or `.msi` package.
- **macOS** — `.dmg` disk image.
- **Linux** — `.deb`, `.rpm`, or `.AppImage`.

The installer includes the Electron runtime, the Kiss frontend, and the Java backend with its embedded Tomcat.

30.8 Differences from Web Deployment

Aspect	Web Mode	Desktop Mode
Backend location	Remote server	Local machine
Launch method	Browser URL	Electron window
Protocol	`http://` or `https://`	`file://`
CORS	Must be configured	Relaxed
File system	No access	Available via Node.js
Updates	Automatic (server)	Requires new installer
Multi-user	Yes	Typically single-user

Tip

Desktop mode is ideal for internal tools, data entry applications, or scenarios where a server-based deployment is impractical. The same codebase serves both web and desktop users.

Note

The Java runtime (JRE/JDK 17+) must be installed on the user's machine for desktop mode. The Electron wrapper handles the browser and Node.js runtime, but the Kiss backend still requires Java.

Chapter 31

Testing

Kiss integrates JUnit 5 for unit testing. This chapter covers running the test suite, writing new tests, and testing strategies for Kiss applications.

31.1 Running Tests

Run the full test suite with:

```
./bld -v test
```

This command:

1. Compiles all source code (core, precompiled, and test classes).
2. Packages the test classes with JUnit dependencies into `KissUnitTest.jar`.
3. Executes the tests using JUnit Platform Console Launcher.
4. Reports results to the terminal.

The `-v` flag enables verbose output, showing each test class and method as it runs.

31.2 The Existing Test Suite

Kiss ships with 19 test classes. The core utility and database tests below live in `src/test/core/org/kissweb/` (with

`QueryBuilderTest` under `database/`); an additional six OAuth test classes live under `oauth/` (`BearerTokenValidatorTest`, `JwksDiscoveryTest`, `oauth/as/AuthorizationServerTest`, and `oauth/client/{OAuthClientConfigTest, OAuthClientStoreTest, PkceTest}`):

Test Class	What It Tests
`ArrayUtilsTest`	Byte array primitive/boxed conversions.
`Base64Test`	Base64 encoding and decoding.
`DateTimeTest`	`DateTime` wrapper operations.
`DateUtilsTest`	YYYYMMDD date arithmetic and formatting.
`IniFileTest`	INI file parsing with sections and types.
`JsonPathTest`	Dot-path JSON navigation.
`NumberFormatTest`	Number formatting with masks.
`NumberUtilsTest`	Rounding and floating-point comparison.
`QueryBuilderTest`	SQL query building (the largest suite).
`StringUtilsTest`	String manipulation methods.
`TimeUtilsTest`	HHMM time operations.
`URLBuilderTest`	URL construction and parameter encoding.
`XMLTest`	XML parsing and transformation.

The `QueryBuilderTest` is the most comprehensive, containing over 100 tests that verify SELECT, WHERE, JOIN, subquery, CTE, and UNION generation against expected SQL output.

31.3 Test Dependencies

Kiss uses JUnit 5.11 (Jupiter). The required JARs are included in the `libs/` directory:

- `junit-jupiter-5.11.0.jar` — aggregator module.
- `junit-jupiter-api-5.11.0.jar` — annotations and assertions.

- `junit-jupiter-engine-5.11.0.jar` — test engine.
- `junit-jupiter-params-5.11.0.jar` — parameterized tests.
- `junit-platform-console-standalone-1.11.0.jar` — command-line launcher.

No external test framework installation is needed — everything is bundled.

31.4 Writing Tests

Place test classes in `src/test/core/org/kissweb/` with the package declaration `org.kissweb`:

```
package org.kissweb;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class MyUtilTest {

    @Test
    public void testBasicOperation() {
        String result = MyUtil.process("input");
        assertEquals("expected", result);
    }

    @Test
    public void testNullHandling() {
        assertNull(MyUtil.process(null));
    }

    @Test
    public void testException() {
        assertThrows(IllegalArgumentException.class,
            () -> MyUtil.process(""));
    }
}
```

31.4.1 Common Assertions

Assertion	Purpose
`assertEquals(expected, actual)`	Verify equality.
`assertNotEquals(a, b)`	Verify inequality.
`assertTrue(condition)`	Verify condition is true.
`assertFalse(condition)`	Verify condition is false.
`assertNull(value)`	Verify value is null.
`assertNotNull(value)`	Verify value is not null.
`assertThrows(type, executable)`	Verify exception is thrown.
`assertDoesNotThrow(executable)`	Verify no exception.

31.4.2 Setup and Teardown

Use `@BeforeEach` and `@AfterEach` for per-test initialization:

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.AfterEach;

public class ServiceTest {
    private Connection db;

    @BeforeEach
    void setUp() {
        // Initialize test state
    }

    @AfterEach
    void tearDown() {
        // Clean up resources
    }

    @Test
    void testService() {
        // test using initialized state
    }
}
```

31.4.3 Parameterized Tests

JUnit 5 supports parameterized tests for testing multiple inputs:

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

public class DateUtilsTest {

    @ParameterizedTest
    @CsvSource({
        "20260101, 2026, 1, 1",
        "20261231, 2026, 12, 31",
        "20260404, 2026, 4, 4"
    })
    void testDateComponents(int date,
            int year, int month, int day) {
        assertEquals(year, DateUtils.year(date));
        assertEquals(month, DateUtils.month(date));
        assertEquals(day, DateUtils.day(date));
    }
}
```

31.5 Testing Database Operations

Database tests can use an in-memory SQLite database to avoid external dependencies. Create a `Connection` with SQLite and a temporary file:

```
import org.kissweb.database.Connection;
import org.junit.jupiter.api.io.TempDir;
import java.nio.file.Path;

public class DatabaseTest {
    @TempDir
    Path tempDir;

    @Test
    void testInsertAndFetch() throws Exception {
        String dbPath = tempDir.resolve("test.db")
            .toString();
        Connection db = new Connection(
            Connection.ConnectionType.SQLite,
            dbPath);

        // Create table
        db.execute(
            "CREATE TABLE test_data (" +
            "id INTEGER PRIMARY KEY, " +
            "name TEXT, value REAL)");
```

```
        // Insert
        Record rec = db.newRecord("test_data");
        rec.set("name", "Alice");
        rec.set("value", 42.5);
        rec.addRecord();
        db.commit();

        // Fetch and verify
        Record fetched = db.fetchOne(
            "SELECT * FROM test_data " +
            "WHERE name = ?", "Alice");
        assertNotNull(fetched);
        assertEquals("Alice",
            fetched.getString("name"));
        assertEquals(42.5,
            fetched.getDouble("value"), 0.001);

        db.close();
    }
}
```

Tip

Use JUnit 5's `@TempDir` annotation to create temporary directories for SQLite databases. The directory and its contents are automatically cleaned up after the test.

31.6 Testing Services

Service methods can be tested by constructing `JSONObject` inputs and calling the method directly. Since service methods take standard parameters (`JSONObject`, `Connection`, `ProcessServlet`), you can create test instances:

```
import org.kissweb.json.JSONObject;

@Test
void testServiceMethod() throws Exception {
    // Set up database (SQLite for isolation)
    Connection db = createTestDatabase();
    setupTestData(db);

    // Build input
    JSONObject injson = new JSONObject();
```

```
        injson.put("employeeId", "EMP001");

        // Build output container
        JSONObject outjson = new JSONObject();

        // Call the service method directly
        EmployeeService svc = new EmployeeService();
        svc.getEmployee(injson, outjson, db, null);

        // Verify output
        assertEquals("Alice",
            outjson.getString("firstName"));
        assertEquals("Engineering",
            outjson.getString("department"));

        db.close();
    }
```

Note

When testing services directly, pass `null` for the `ProcessServlet` parameter if the service does not use it. If the service accesses session data via `servlet.getUserData()`, you will need to mock or stub the servlet.

31.7 Testing Strategies

Unit tests
: Test individual utility methods and pure logic. These are fast and have no external dependencies. Most of Kiss's existing tests are in this category.

Integration tests
: Test service methods with a real (SQLite) database. These verify that SQL queries, Record operations, and business logic work together correctly.

Frontend testing
: Kiss's frontend is plain JavaScript without a build step. Use browser developer tools for interactive debugging, or integrate a tool like Playwright or Cypress for automated UI testing.

Warning

Always use parameterized SQL queries in tests, just as in production code. Never concatenate user input into SQL strings, even in test code, to maintain consistent security practices.

Chapter 32

Deployment

This chapter covers building, packaging, and deploying a Kiss application to production environments.

32.1 Building a WAR File

The standard deployment artifact is a WAR (Web Application Archive) file:

```
./bld war
```

This command:

1. Compiles all Java source code (core, precompiled, and application classes).
2. Copies frontend assets, backend scripts, and configuration files into an exploded directory structure.
3. Copies `web-secure.xml` into `web.xml` to enable production CORS restrictions.
4. Packages everything into `work/Kiss.war`.
5. Copies `web-unsafe.xml` back into `web.xml` so local development continues with the permissive CORS configuration.

The resulting WAR file is a standard Jakarta EE 11 (Servlet 6.1) archive containing:

```
Kiss.war
  index.html            # Frontend entry point
  index.js              # Application bootstrap
  kiss/                 # Framework frontend files
  screens/              # Application screens
  WEB-INF/
    web.xml             # Servlet configuration (secure)
    classes/            # Compiled Java classes
    lib/                # JAR dependencies
    backend/            # Groovy/Lisp service files
    application.ini     # Application configuration
    log4j2.xml          # Logging configuration
```

32.2 Web Configuration Variants

Kiss maintains two source variants of the servlet configuration in `src/main/core/WEB-INF/`:

`web-unsafe.xml`
: Development configuration. Allows CORS from any origin (`*`), enabling rapid frontend development from any port or device. This is the default during development.

`web-secure.xml`
: Production configuration. Restricts CORS to specific origins (`localhost:8000` and IntelliJ's debug port 63342), preventing cross-origin attacks from arbitrary domains. This is automatically used when building the WAR file.

The deployed `web.xml` is not maintained by hand — it is a copy that `bld` writes from one of the two variants above based on whether you are building for development or for production.

Warning

Never deploy with the permissive CORS configuration (`web-unsafe.xml`) in production. The wildcard CORS policy allows any website to make requests to your backend. The WAR build automatically substitutes `web-secure.xml`; update its allowed origins to match your production domain.

32.3 Deploying to Tomcat

32.3.1 Embedded Tomcat

Kiss ships with an embedded Tomcat 11 in the `tomcat/` directory. To deploy:

1. Build the WAR: `./bld war`
2. Copy to Tomcat: `cp work/Kiss.war tomcat/webapps/ROOT.war`
3. Start Tomcat: `tomcat/bin/startup.sh`

The embedded Tomcat listens on the application's back-end port — 8001 with the default `portBase` (Section 4.5). The `bld` system stamps the port into `tomcat/conf/server.xml` on every `setup-tomcat` run.

Note

A production WAR deployed to the embedded Tomcat (`webapps/ROOT.war` — `./bld build` places one there automatically) does not survive the next development run: `develop` and `start-backend` remove any stale `ROOT.war` before starting Tomcat. If it were left in place, Tomcat would re-explode the WAR over the `ROOT` directory at startup, replacing the development tree those tasks stage there — most visibly `web.xml`, whose production CORS allow-list rejects the cross-origin development front end. Treat a WAR deployed to the embedded Tomcat as a production-like test, not a persistent installation.

32.3.2 External Tomcat

To deploy to a standalone Tomcat 11 installation:

1. Build the WAR: `./bld war`
2. Copy `work/Kiss.war` to Tomcat's `webapps/` directory, typically renaming it to your application's name (for example `YourApplication.war`).

3. Restart Tomcat (or wait for auto-deploy if enabled).
4. Access the application at `http://hostname:port/YourApplication/`.

Note

The WAR filename becomes the application's URL context path. Tomcat takes the deployed file name (minus the `.war` extension) and uses it as the first path segment when serving the application: `Kiss.war` is reached at `/Kiss/`, `Payroll.war` at `/Payroll/`, and so on. Kiss builds the WAR as `Kiss.war` by default, so when copying to a production server you will usually rename it to whatever name you want to appear in the URL — for example `cp work/Kiss.war /var/lib/tomcat/webapps/YourApplication.war`.
To deploy as the root application (no path segment in the URL), rename the WAR to `ROOT.war` or configure the Tomcat context path explicitly.

32.4 Other Servlet Containers

The WAR file is compatible with any Jakarta EE 11 (Servlet 6.1) container:

- **Tomcat 11** — the reference container for Kiss.
- **Jetty 12** — lightweight alternative.
- **WildFly** — full Jakarta EE application server.
- **GlassFish** — reference Jakarta EE implementation.

Deployment procedures vary by container but follow the standard WAR deployment process.

32.5 Split-System Deployment

For high-traffic applications or CDN-based frontends, the frontend and backend can be deployed on separate servers.

32.5.1 Frontend Server

Serve the frontend as static files using any web server:

- Nginx, Apache, or a CDN for HTML, CSS, and JavaScript files.
- No server-side processing needed — the frontend is pure static content.

32.5.2 Backend Server

Deploy the WAR file (or just the backend portion) to a servlet container.

32.5.3 Connecting Them

Configure the backend URL in the frontend's `index.js` by setting `SystemInfo.backendUrl` before the framework initializes:

```
// In index.js, before Server initialization
SystemInfo.backendUrl = "https://api.myapp.com";
```

The frontend's `Server.js` checks for this value and uses it as the base URL for all backend calls. When not set, it assumes the backend is co-located (same origin).

Note

In a split deployment, update `web-secure.xml` to include the frontend's origin in the CORS allowed-origins list.

32.6 Production Configuration Checklist

Before deploying to production, verify the following settings in `application.ini`:

Setting	Production Value
`DatabaseType`	Your database type (PostgreSQL, MySQL, MSSqlServer, Oracle).
`DatabaseHost`	Production database hostname.
`DatabaseName`	Production database name.

Setting	Production Value
`DatabaseUser`	Database username with appropriate permissions.
`DatabasePassword`	Database password.
`MaxWorkerThreads`	Match to expected concurrent request load (default: 30).
`UserInactiveSeconds`	Session timeout in seconds (default: 900 = 15 min).

Additional steps:

- Use `web-secure.xml` and update allowed CORS origins.
- Configure `log4j2.xml` for production log levels and file rotation.
- Ensure the database schema is up to date.
- Set JVM memory parameters (`-Xmx`, `-Xms`) in the Tomcat startup script.
- Store all passwords hashed (SHA-256) in the `users` table.
- Test the application with the production database before going live.

32.7 Logging in Production

Server logs are written to `tomcat/logs/catalina.out`. The default `log4j2.xml` configuration sets the root logger to `ERROR` level with console output:

```
# log4j2.xml pattern
%d{yyyy-MM-dd HH:mm:ss} [%-5p] [%-25c{2}] - %m%n
```

For production, consider:

- Setting the root level to `WARN` or `ERROR` to reduce log volume.
- Adding a `RollingFileAppender` for log rotation.
- Configuring specific loggers for application packages at `DEBUG` level during troubleshooting.

- Using `Configurator.setLevel()` to change log levels at runtime without a restart (see Chapter 29).

32.8 Performance Tuning

Worker threads

The `MaxWorkerThreads` setting in `application.ini` controls the size of the service request thread pool. Set this to match the expected number of concurrent requests. A typical starting point is 2–4 times the number of CPU cores.

Connection pool

C3P0 connection pool settings are configured in `application.ini`. Key parameters include:

- `MaxPoolSize` — maximum database connections (default: varies by config). A good starting point is 4 times the number of CPU cores.
- `MinPoolSize` — minimum idle connections to maintain.
- `MaxIdleTime` — seconds before idle connections are closed.
- `DatabaseTestOnCheckin` — validate connections when returned to the pool.

JVM memory

Set `-Xmx` (maximum heap) and `-Xms` (initial heap) in the Tomcat startup script or `CATALINA_OPTS` environment variable:

```
export CATALINA_OPTS="-Xms512m -Xmx2g"
```

Static file caching

In a split deployment, configure the frontend web server (Nginx, Apache) with appropriate cache headers for static assets. Kiss frontend files rarely change after deployment.

Tip

Monitor `tomcat/logs/catalina.out` during the first days of production deployment. Watch for connection pool exhaustion warnings, slow query logs, and memory pressure indicators.

Chapter 33

Building a Complete Application

This chapter walks through building a real application from scratch, applying everything covered in the book. The application is a project tracking system that manages employees, projects, and assignments with full CRUD operations, a data grid, popups, and reports.

33.1 Requirements

The system tracks:

- **Departments** — organizational units.
- **Employees** — name, department, active status.
- **Projects** — name, start date, status (Active, On Hold, Completed).
- **Assignments** — which employees are assigned to which projects, with role and hours allocated.

Users can:

- List, create, edit, and delete records for each entity.
- Search and filter employee and project lists.
- View a project's roster of assigned employees.

- Generate a PDF project status report.

- Export assignment data to CSV.

33.2 Database Schema

Place this SQL in a setup script or run it against your database:

```
CREATE TABLE department (
    department_id CHAR(36) PRIMARY KEY,
    name          VARCHAR(100) NOT NULL
);

CREATE TABLE employee (
    employee_id   CHAR(36) PRIMARY KEY,
    first_name    VARCHAR(50) NOT NULL,
    last_name     VARCHAR(50) NOT NULL,
    department_id CHAR(36)
        REFERENCES department(department_id),
    hire_date     INTEGER,
    active        CHAR(1) DEFAULT 'Y'
);

CREATE TABLE project (
    project_id    CHAR(36) PRIMARY KEY,
    name          VARCHAR(100) NOT NULL,
    start_date    INTEGER,
    status        VARCHAR(20) DEFAULT 'Active'
);

CREATE TABLE assignment (
    assignment_id CHAR(36) PRIMARY KEY,
    project_id    CHAR(36)
        REFERENCES project(project_id),
    employee_id   CHAR(36)
        REFERENCES employee(employee_id),
    role          VARCHAR(50),
    hours_allocated NUMERIC(6,1)
);
```

Primary keys use UUID strings (36 characters) generated by the backend. Dates are stored as YYYYMMDD integers, consistent with Kiss conventions.

33.3 Backend Services

Each entity gets its own Groovy service in `src/main/backend/services/`.

33.3.1 DepartmentService

```
// services/DepartmentService.groovy
import org.kissweb.database.*
import org.kissweb.restServer.ProcessServlet
import org.kissweb.json.JSONObject

class DepartmentService {

    void list(JSONObject injson, JSONObject outjson,
              Connection db, ProcessServlet servlet) {
        outjson.put("departments",
            db.fetchAllJSON(
                "select * from department order by name"))
    }

    void newRecord(JSONObject injson, JSONObject outjson,
                   Connection db, ProcessServlet servlet) {
        Record rec = db.newRecord("department")
        rec.set("department_id",
            UUID.randomUUID().toString())
        rec.set("name", injson.getString("name"))
        rec.addRecord()
        db.commit()
    }

    void save(JSONObject injson, JSONObject outjson,
              Connection db, ProcessServlet servlet) {
        Record rec = db.fetchOne(
            "select * from department " +
            "where department_id = ?",
            injson.getString("departmentId"))
        rec.set("name", injson.getString("name"))
        rec.update()
        db.commit()
    }

    void delete(JSONObject injson, JSONObject outjson,
                Connection db, ProcessServlet servlet) {
        db.execute(
            "delete from department " +
            "where department_id = ?",
            injson.getString("departmentId"))
```

```
            db.commit()
        }
    }
```

33.3.2 EmployeeService

```
    // services/EmployeeService.groovy
    import org.kissweb.database.*
    import org.kissweb.restServer.ProcessServlet
    import org.kissweb.json.JSONObject

    class EmployeeService {

        void search(JSONObject injson, JSONObject outjson,
                    Connection db, ProcessServlet servlet) {
            QueryBuilder qb = db.newQueryBuilder()
            qb.select("employee.employee_id",
                "employee.first_name",
                "employee.last_name", "employee.hire_date",
                "employee.active",
                "department.name as department_name")
            qb.leftJoin("employee", "department_id",
                "department", "department_id")

            // Dynamic search filters
            String lastName = injson.optString("lastName", "")
            if (!lastName.isEmpty())
                qb.where("employee.last_name like ?",
                    lastName + "%")

            String active = injson.optString("active", "")
            if (!active.isEmpty())
                qb.where("employee.active = ?", active)

            qb.orderBy("employee.last_name")
            qb.orderBy("employee.first_name")

            outjson.put("employees", qb.fetchAllJSON())
        }

        void get(JSONObject injson, JSONObject outjson,
                 Connection db, ProcessServlet servlet) {
            Record rec = db.fetchOne(
                "select * from employee " +
                "where employee_id = ?",
                injson.getString("employeeId"))
            outjson.put("employee", rec.toJSON())
```

```
    }

    void newRecord(JSONObject injson, JSONObject outjson,
                   Connection db, ProcessServlet servlet) {
        Record rec = db.newRecord("employee")
        rec.set("employee_id",
            UUID.randomUUID().toString())
        rec.set("first_name",
            injson.getString("firstName"))
        rec.set("last_name",
            injson.getString("lastName"))
        rec.set("department_id",
            injson.getString("departmentId"))
        rec.set("hire_date",
            injson.getInt("hireDate"))
        rec.set("active", "Y")
        rec.addRecord()
        db.commit()
    }

    void save(JSONObject injson, JSONObject outjson,
              Connection db, ProcessServlet servlet) {
        Record rec = db.fetchOne(
            "select * from employee " +
            "where employee_id = ?",
            injson.getString("employeeId"))
        rec.set("first_name",
            injson.getString("firstName"))
        rec.set("last_name",
            injson.getString("lastName"))
        rec.set("department_id",
            injson.getString("departmentId"))
        rec.set("hire_date",
            injson.getInt("hireDate"))
        rec.set("active",
            injson.getString("active"))
        rec.update()
        db.commit()
    }

    void delete(JSONObject injson, JSONObject outjson,
                Connection db, ProcessServlet servlet) {
        db.execute(
            "delete from employee where employee_id = ?",
            injson.getString("employeeId"))
        db.commit()
    }
}
```

33.3.3 ProjectService

```
// services/ProjectService.groovy
import org.kissweb.database.*
import org.kissweb.restServer.ProcessServlet
import org.kissweb.json.JSONObject

class ProjectService {

    void list(JSONObject injson, JSONObject outjson,
              Connection db, ProcessServlet servlet) {
        String status = injson.optString("status", "")
        String sql = "select * from project"
        if (!status.isEmpty())
            sql += " where status = ?"
        sql += " order by name"

        if (!status.isEmpty())
            outjson.put("projects",
                db.fetchAllJSON(sql, status))
        else
            outjson.put("projects",
                db.fetchAllJSON(sql))
    }

    void get(JSONObject injson, JSONObject outjson,
             Connection db, ProcessServlet servlet) {
        Record rec = db.fetchOne(
            "select * from project " +
            "where project_id = ?",
            injson.getString("projectId"))
        outjson.put("project", rec.toJSON())
    }

    void newRecord(JSONObject injson, JSONObject outjson,
                   Connection db, ProcessServlet servlet) {
        Record rec = db.newRecord("project")
        rec.set("project_id",
            UUID.randomUUID().toString())
        rec.set("name", injson.getString("name"))
        rec.set("start_date",
            injson.getInt("startDate"))
        rec.set("status",
            injson.optString("status", "Active"))
        rec.addRecord()
        db.commit()
    }

    void save(JSONObject injson, JSONObject outjson,
```

```
                Connection db, ProcessServlet servlet) {
        Record rec = db.fetchOne(
            "select * from project " +
            "where project_id = ?",
            injson.getString("projectId"))
        rec.set("name", injson.getString("name"))
        rec.set("start_date",
            injson.getInt("startDate"))
        rec.set("status", injson.getString("status"))
        rec.update()
        db.commit()
    }

    void delete(JSONObject injson, JSONObject outjson,
                Connection db, ProcessServlet servlet) {
        // Delete assignments first (referential integrity)
        db.execute(
            "delete from assignment " +
            "where project_id = ?",
            injson.getString("projectId"))
        db.execute(
            "delete from project " +
            "where project_id = ?",
            injson.getString("projectId"))
        db.commit()
    }
}
```

33.3.4 AssignmentService

```
// services/AssignmentService.groovy
import org.kissweb.database.*
import org.kissweb.restServer.ProcessServlet
import org.kissweb.json.JSONObject

class AssignmentService {

    void listForProject(JSONObject injson,
                        JSONObject outjson,
                        Connection db,
                        ProcessServlet servlet) {
        QueryBuilder qb = db.newQueryBuilder()
        qb.select("assignment.assignment_id",
            "assignment.role",
            "assignment.hours_allocated",
            "employee.first_name", "employee.last_name")
        qb.leftJoin("assignment", "employee_id",
```

```
            "employee", "employee_id")
        qb.where("assignment.project_id = ?",
            injson.getString("projectId"))
        qb.orderBy("employee.last_name")

        outjson.put("assignments", qb.fetchAllJSON())
    }

    void newRecord(JSONObject injson, JSONObject outjson,
                   Connection db, ProcessServlet servlet) {
        Record rec = db.newRecord("assignment")
        rec.set("assignment_id",
            UUID.randomUUID().toString())
        rec.set("project_id",
            injson.getString("projectId"))
        rec.set("employee_id",
            injson.getString("employeeId"))
        rec.set("role", injson.getString("role"))
        rec.set("hours_allocated",
            injson.getDouble("hoursAllocated"))
        rec.addRecord()
        db.commit()
    }

    void save(JSONObject injson, JSONObject outjson,
              Connection db, ProcessServlet servlet) {
        Record rec = db.fetchOne(
            "select * from assignment " +
            "where assignment_id = ?",
            injson.getString("assignmentId"))
        rec.set("role", injson.getString("role"))
        rec.set("hours_allocated",
            injson.getDouble("hoursAllocated"))
        rec.update()
        db.commit()
    }

    void delete(JSONObject injson, JSONObject outjson,
                Connection db, ProcessServlet servlet) {
        db.execute(
            "delete from assignment " +
            "where assignment_id = ?",
            injson.getString("assignmentId"))
        db.commit()
    }
}
```

33.4 Frontend Screens

The frontend consists of HTML screens and their companion JavaScript files in `src/main/frontend/screens/`.

33.4.1 Employee List Screen

The employee list demonstrates the standard Kiss pattern: a search form, a data grid, and an edit popup.

```
<!-- screens/EmployeeList.html -->
<div>
    <h2>Employees</h2>

    <div style="margin-bottom: 10px;">
        <text-input id="searchLastName"
            style="width:200px;"></text-input>
        <drop-down id="searchActive"
            style="width:120px;"></drop-down>
        <push-button id="searchBtn">Search</push-button>
        <push-button id="newBtn">New Employee</push-button>
    </div>

    <div id="employeeGrid"
         style="width:100%; height:400px;"></div>
</div>

<popup id="employeePopup" width="500px" height="380px">
    <popup-title>Employee Detail</popup-title>
    <popup-body>
        <div style="display:grid;
                    grid-template-columns: 120px 1fr;
                    gap: 8px;">
            <label>First Name:</label>
            <text-input id="epFirstName"
                style="width:200px;"></text-input>

            <label>Last Name:</label>
            <text-input id="epLastName"
                style="width:200px;"></text-input>

            <label>Department:</label>
            <drop-down id="epDepartment"
                style="width:200px;"></drop-down>

            <label>Hire Date:</label>
            <date-input id="epHireDate"
                style="width:150px;"></date-input>
```

```
            <label>Active:</label>
            <drop-down id="epActive"
                style="width:100px;"></drop-down>
        </div>
        <div style="margin-top:15px; text-align:right;">
            <push-button id="epSave">Save</push-button>
            <push-button id="epCancel">Cancel</push-button>
        </div>
    </popup-body>
</popup>
```

```
// screens/EmployeeList.js
const WS = 'services.EmployeeService';
let grid;
let editMode; // 'new' or 'edit'
let editId;   // employee_id being edited

// Initialize screen
$$('searchActive').add('', '(all)');
$$('searchActive').add('Y', 'Active');
$$('searchActive').add('N', 'Inactive');
$$('searchActive').setValue('Y');

// Set up the grid
grid = new AGGrid('employeeGrid', [
    { headerName: 'Last Name',
      field: 'last_name', width: 150 },
    { headerName: 'First Name',
      field: 'first_name', width: 150 },
    { headerName: 'Department',
      field: 'department_name', width: 150 },
    { headerName: 'Hire Date',
      field: 'hire_date', width: 120,
      valueFormatter: AGGrid.date },
    { headerName: 'Active',
      field: 'active', width: 80 }
], 'employeeGrid');
grid.show();

// Search
$$('searchBtn').onclick(search);
search(); // initial load

async function search() {
    const res = await Server.call(WS, 'search', {
        lastName:
            $$('searchLastName').getValue(),
```

```
        active: $$('searchActive').getValue()
    });
    if (res._Success)
        grid.addRecords(res.employees);
}

// Grid double-click to edit
grid.setOnRowDoubleClicked(openEdit);

async function openEdit() {
    const row = grid.getSelectedRow();
    if (!row) return;

    editMode = 'edit';
    editId = row.employee_id;

    const res = await Server.call(WS, 'get',
        { employeeId: editId });
    if (!res._Success) return;

    const emp = res.employee;
    await loadDepartments();
    $$('epFirstName').setValue(emp.first_name);
    $$('epLastName').setValue(emp.last_name);
    $$('epDepartment').setValue(emp.department_id);
    $$('epHireDate').setValue(emp.hire_date);

    $$('epActive').clear();
    $$('epActive').add('Y', 'Yes');
    $$('epActive').add('N', 'No');
    $$('epActive').setValue(emp.active);

    Utils.popup_open('employeePopup');
}

// New employee
$$('newBtn').onclick(async () => {
    editMode = 'new';
    editId = null;

    await loadDepartments();
    $$('epFirstName').clear();
    $$('epLastName').clear();
    $$('epHireDate').clear();

    $$('epActive').clear();
    $$('epActive').add('Y', 'Yes');
    $$('epActive').add('N', 'No');
    $$('epActive').setValue('Y');
```

```
    Utils.popup_open('employeePopup');
});

// Save
$$('epSave').onclick(async () => {
    if ($$('epFirstName').isError('First name'))
        return;
    if ($$('epLastName').isError('Last name'))
        return;

    const data = {
        firstName: $$('epFirstName').getValue(),
        lastName: $$('epLastName').getValue(),
        departmentId: $$('epDepartment').getValue(),
        hireDate: $$('epHireDate').getValue(),
        active: $$('epActive').getValue()
    };

    let res;
    if (editMode === 'new') {
        res = await Server.call(
            WS, 'newRecord', data);
    } else {
        data.employeeId = editId;
        res = await Server.call(WS, 'save', data);
    }

    if (res._Success) {
        Utils.popup_close('employeePopup');
        search();
    }
});

// Cancel
$$('epCancel').onclick(() => {
    Utils.popup_close('employeePopup');
});

// Helper: load department dropdown
async function loadDepartments() {
    const res = await Server.call(
        'services.DepartmentService', 'list');
    if (!res._Success) return;

    $$('epDepartment').clear();
    $$('epDepartment').add('', '(select)');
    for (const dept of res.departments) {
        $$('epDepartment').add(
```

```
            dept.department_id, dept.name);
    }
}
```

This screen demonstrates several key Kiss patterns:

- Dynamic search with `QueryBuilder` on the backend.
- AGGrid for data display with built-in date formatting.
- Popup reuse for both new and edit operations.
- Drop-down population from a related service.
- Validation with `isError()` before saving.

33.4.2 Project Dashboard

The project dashboard shows the project list and, when a project is selected, its assigned employees in a detail grid:

```
<!-- screens/ProjectDashboard.html -->
<div>
    <h2>Projects</h2>
    <div style="margin-bottom: 10px;">
        <drop-down id="statusFilter"
            style="width:150px;"></drop-down>
        <push-button id="filterBtn">Filter</push-button>
        <push-button id="newProjectBtn">
            New Project</push-button>
        <push-button id="reportBtn">
            Status Report</push-button>
        <push-button id="exportBtn">
            Export CSV</push-button>
    </div>
    <div id="projectGrid"
         style="width:100%; height:250px;"></div>

    <h3 id="rosterTitle"
        style="margin-top:15px;">Assignments</h3>
    <div id="assignmentGrid"
         style="width:100%; height:200px;"></div>
</div>
```

```
// screens/ProjectDashboard.js (abbreviated)
const ProjWS = 'services.ProjectService';
const AsgnWS = 'services.AssignmentService';
```

```
let projectGrid, assignmentGrid;
let selectedProjectId = null;

// Status filter
$$('statusFilter').add('', '(all)');
$$('statusFilter').add('Active', 'Active');
$$('statusFilter').add('On Hold', 'On Hold');
$$('statusFilter').add('Completed', 'Completed');
$$('statusFilter').setValue('Active');

// Project grid
projectGrid = new AGGrid('projectGrid', [
    { headerName: 'Project', field: 'name',
      width: 200 },
    { headerName: 'Start Date', field: 'start_date',
      width: 120, valueFormatter: AGGrid.date },
    { headerName: 'Status', field: 'status',
      width: 120 }
], 'projectGrid');
projectGrid.show();

// Assignment grid
assignmentGrid = new AGGrid('assignmentGrid', [
    { headerName: 'Employee',
      valueGetter: (p) =>
          p.data.last_name + ', ' + p.data.first_name,
      width: 200 },
    { headerName: 'Role', field: 'role', width: 150 },
    { headerName: 'Hours', field: 'hours_allocated',
      width: 100, type: 'numericColumn' }
], 'assignmentGrid');
assignmentGrid.show();

// Load projects
$$('filterBtn').onclick(loadProjects);
loadProjects();

async function loadProjects() {
    const res = await Server.call(ProjWS, 'list',
        { status: $$('statusFilter').getValue() });
    if (res._Success)
        projectGrid.addRecords(res.projects);
}

// When a project is selected, load its assignments
projectGrid.setOnSelectionChanged(() => {
    const row = projectGrid.getSelectedRow();
    if (row) {
        selectedProjectId = row.project_id;
```

```
        loadAssignments(selectedProjectId);
    }
});

async function loadAssignments(projectId) {
    const res = await Server.call(
        AsgnWS, 'listForProject',
        { projectId: projectId });
    if (res._Success)
        assignmentGrid.addRecords(res.assignments);
}
```

33.5 Reports

33.5.1 PDF Status Report

Add a report method to `ProjectService`:

```
void statusReport(JSONObject injson, JSONObject outjson,
                  Connection db,
                  ProcessServlet servlet) {
    Groff rpt = new Groff("ProjectStatus-",
        "Project Status Report", false)

    rpt.startTable("Lw(2.5i) Lw(1i) Lw(1i) Nw(1i)")
    rpt.column("Project")
    rpt.column("Start Date")
    rpt.column("Status")
    rpt.column("Hours")
    rpt.endTitle()

    List<Record> projects = db.fetchAll("""
        select p.name, p.start_date, p.status,
            coalesce(sum(a.hours_allocated), 0)
                as total_hours
        from project p
        left join assignment a
            on p.project_id = a.project_id
        group by p.name, p.start_date, p.status
        order by p.name
    """)

    for (Record r : projects) {
        rpt.column(r.getString("name"))
        rpt.dateColumn(r.getInt("start_date"))
        rpt.column(r.getString("status"))
        rpt.column(r.getDouble("total_hours"), "C", 1)
```

```
    }

    outjson.put("reportUrl", rpt.process())
}
```

On the frontend:

```
$$('reportBtn').onclick(async () => {
    const res = await Server.call(
        ProjWS, 'statusReport');
    if (res._Success)
        Utils.showReport(res.reportUrl);
});
```

33.5.2 CSV Export

```
void exportAssignments(JSONObject injson,
                       JSONObject outjson,
                       Connection db,
                       ProcessServlet servlet) {
    File f = FileUtils.createReportFile(
        "Assignments-", ".csv")
    DelimitedFileWriter dfw =
        new DelimitedFileWriter(f.getAbsolutePath())

    // Header
    dfw.writeField("Project")
    dfw.writeField("Employee")
    dfw.writeField("Role")
    dfw.writeField("Hours")
    dfw.endRecord()

    List<Record> recs = db.fetchAll("""
        select p.name as project_name,
            e.last_name || ', ' || e.first_name
                as employee_name,
            a.role, a.hours_allocated
        from assignment a
        join project p
            on a.project_id = p.project_id
        join employee e
            on a.employee_id = e.employee_id
        order by p.name, e.last_name
    """)

    for (Record r : recs) {
        dfw.writeField(r.getString("project_name"))
```

```
            dfw.writeField(r.getString("employee_name"))
            dfw.writeField(r.getString("role"))
            dfw.writeField(r.getDouble("hours_allocated"))
            dfw.endRecord()
        }

        dfw.close()
        outjson.put("exportUrl", FileUtils.getHTTPPath(f))
    }
```

On the frontend:

```
$$('exportBtn').onclick(async () => {
    const res = await Server.call(
        ProjWS, 'exportAssignments');
    if (res._Success)
        Utils.showReport(res.exportUrl);
});
```

33.6 Navigation

After login, set up a simple navigation framework in `index.js`:

```
// After successful login
function showMainScreen() {
    Utils.loadPage('screens/MainLayout', 'appBody');
}
```

```
<!-- screens/MainLayout.html -->
<div style="display:flex; gap:10px;
            padding:10px; background:#f0f0f0;">
    <push-button id="navEmployees">
        Employees</push-button>
    <push-button id="navProjects">
        Projects</push-button>
</div>
<div id="contentArea"></div>
```

```
// screens/MainLayout.js
$$('navEmployees').onclick(() => {
    Utils.loadPage('screens/EmployeeList',
        'contentArea');
});
```

```
$$('navProjects').onclick(() => {
    Utils.loadPage('screens/ProjectDashboard',
        'contentArea');
});

// Load default screen
Utils.loadPage('screens/ProjectDashboard',
    'contentArea');
```

33.7 Lessons Learned

Building this application illustrates several key patterns:

Service simplicity
: Each service method is typically 10–30 lines. The framework handles authentication, connection management, and JSON serialization.

Popup reuse
: A single popup serves both "new" and "edit" modes, controlled by a flag variable. This halves the amount of UI code.

QueryBuilder for search
: Dynamic search screens benefit from `QueryBuilder`'s conditional `where()` clauses. Each filter adds a condition only when the user provides a value.

Grid as the center
: Most list screens follow the same pattern: search form at the top, grid in the middle, edit popup on double-click, action buttons that call backend methods.

Reports are services
: PDF reports and CSV exports are just service methods that return a URL. The frontend opens the URL in a new tab.

UUID primary keys
: Generated on the backend with `UUID.randomUUID().toString()`, avoiding database-specific auto-increment mechanisms.

Commit explicitly
: Every service method that modifies data ends with `db.commit()`. The framework does not auto-commit.

Tip

Start with the database schema, then build the backend services, then the frontend screens. This bottom-up approach ensures each layer has a solid foundation before the next layer is built.

Part VI

Appendices

Appendix A

API Quick Reference

This appendix provides lookup tables for the most commonly used Kiss APIs. For full documentation, see the JavaDoc (backend) and JSDoc (frontend).

A.1 Connection

Method	Description
`execute(sql, args...)`	Execute INSERT, UPDATE, DELETE.
`fetchOne(sql, args...)`	Fetch single Record (or null).
`fetchAll(sql, args...)`	Fetch all Records as List.
`fetchAllJSON(sql, args...)`	Fetch as JSONArray.
`query(sql, args...)`	Return Cursor for iteration.
`exists(sql, args...)`	True if any records match.
`newRecord(table)`	Create Record for insertion.
`newCommand()`	Create Command for fine control.
`newQueryBuilder()`	Create QueryBuilder with auto-joins.
`commit()`	Commit current transaction.
`rollback()`	Rollback current transaction.
`getSchemaGraph()`	Get or build the SchemaGraph.
`getDBType()`	Get the database type.

A.2 Record

Method	Description
`set(column, value)`	Set a column value.
`addRecord()`	INSERT the record.
`update()`	UPDATE changed columns only.
`delete()`	DELETE the record.
`getString(column)`	Get String value.
`getChar(column)`	Get char value.
`getInt(column)`	Get int value.
`getShort(column)`	Get short value.
`getLong(column)`	Get long value.
`getFloat(column)`	Get float value.
`getDouble(column)`	Get double value.
`getBigDecimal(column)`	Get BigDecimal value (for NUMERIC / DECIMAL).
`getDateAsInt(column)`	Get date as YYYYMMDD int.
`getDateTime(column)`	Get as java.util.Date.
`toJSON()`	Convert to JSONObject.
`copy(src)`	Copy column values from another Record into this one.
`getTableName()`	Get the source table name.

A.3 Command

Method	Description
`execute(sql, args...)`	Execute a statement.
`fetchOne(sql, args...)`	Fetch one Record.
`fetchAll(sql, args...)`	Fetch all Records.
`query(sql, args...)`	Return Cursor.
`fetchAllJSON(sql, args...)`	Fetch all Records as JSONArray.
`newQueryBuilder()`	Create a QueryBuilder bound to this Command.
`close()`	Release JDBC resources.

A.4 Cursor

Method	Description
`isNext()`	Advance; true if row available.
`getRecord()`	Get current Record.
`close()`	Close the cursor.

A.5 QueryBuilder

Method	Description
`select(columns...)`	Add SELECT columns.
`distinct()`	Enable SELECT DISTINCT.
`where(cond, params...)`	Add WHERE condition (AND).
`startOr()` / `endOr()`	Begin/end OR group.
`startAnd()` / `endAnd()`	Begin/end AND group.
`orderBy(col)`	ORDER BY ascending.
`orderByDesc(col)`	ORDER BY descending.
`groupBy(col)`	GROUP BY.
`having(cond, params...)`	HAVING condition.
`limit(max)`	Limit result rows.
`join(fromTbl, fromCol, toTbl, toCol [, alias])`	INNER JOIN.
`leftJoin(fromTbl, fromCol, toTbl, toCol [, alias])`	LEFT JOIN.
`rightJoin(fromTbl, fromCol, toTbl, toCol [, alias])`	RIGHT JOIN.
`whereIn(col, subquery)`	WHERE col IN (subquery).
`whereNotIn(col, subquery)`	WHERE col NOT IN.
`whereExists(subquery)`	WHERE EXISTS.
`whereNotExists(subquery)`	WHERE NOT EXISTS.
`with(name, sql)`	Common Table Expression.
`union(qb)`	UNION.

Method	Description
`unionAll(qb)`	UNION ALL.
`fetchAll()`	Execute, return List<Record>.
`fetchOne()`	Execute, return Record or null.
`fetchAllJSON()`	Execute, return JSONArray.
`query()`	Execute, return Cursor.
`build()`	Generate SQL string.
`getParameters()`	Get parameter values.

A.6 SortBuilder

Method	Description
`key(key, exprs...)`	Register a sort key with one or more expressions; returns this.
`join(joinSql, keys...)`	Join emitted once when any listed key is used; returns this.
`text(col)`	Static. Case-insensitive text; empty/whitespace is blank.
`zeroBlank(col)`	Static. Number or int date; 0 is blank.
`raw(sql)`	Static. Verbatim SQL; must yield NULL when blank.
`textSql(type, col)`	Static. The `text()` SQL for a `ConnectionType`.
`build(type, sorts, default, tiebreak...)`	Return a `Result`. Blanks last; unknown keys rejected.
`Sort.of(key, asc)`	A requested sort.
`Result.orderBy()`	`order by ...` clause, or empty.
`Result.joins()`	Needed joins, each once, trailing space.
`Result.joinList()`	Needed joins as a list.
`Result.rejectedKeys()`	Unknown non-blank keys (client text).

A.7 SchemaGraph

Method	Description
`fromDatabase(conn)`	Build from DB metadata.
`addForeignKey(fromTbl, fromCol, toTbl, toCol)`	Add a relationship (also a composite `String[]` overload).
`findJoinPath(tables, rootTable)`	BFS shortest join path (returns `List<Edge>`).
`getEdges(table)`	Get all edges for a table.
`saveToFile(path)`	Serialize to file.
`loadFromFile(path)`	Deserialize from file.

A.8 Frontend: Server.js

Method	Description
`Server.call(svc, meth, data)`	Call backend service (async).
`Server.callAll( promises...)`	Parallel service calls.
`Server.fileUploadSend( cls, meth, fd, injson)`	Upload file(s); `fd` is a `FormData`.
`Server.logout()`	End session; full page reload onto the login route.
`Server.setURL(url)`	Set backend base URL.
`Server.setMaxInactivity Seconds(n)`	Frontend timeout.
`Server.setMaxInactivity Minutes(n)`	Frontend timeout (minutes).

A.9 Frontend: Component Methods

Common methods available on all components via `$$('id')`:

Method	Description
`getValue()`	Get current value.
`setValue(val)`	Set value.
`clear()`	Clear/reset.
`enable()` / `disable()`	Enable or disable.

Method	Description
`show()` / `hide()`	Show or hide.
`isError(msg)`	Validate; show error if invalid.
`focus()`	Set keyboard focus.
`onChange(fn)`	Set change handler.
`onclick(fn)`	Set click handler (buttons).
`add(value, label)`	Add option (drop-down, list-box).
`readOnly()` / `readWrite()`	Toggle editability.

A.10 Frontend: Utils.js

Method	Description
`Utils.loadPage(path, target)`	Load screen into element.
`Utils.popup_open(id)`	Open a popup.
`Utils.popup_close(id)`	Close a popup.
`Utils.showMessage(title, msg)`	Message dialog.
`Utils.yesNo(title, msg, yesFn, noFn)`	Confirmation dialog.
`Utils.waitMessage(msg)`	Show progress indicator.
`Utils.waitMessageEnd()`	Hide progress indicator.
`Utils.showReport(url)`	Open URL in new tab.
`Utils.cleanup()`	Clean up current screen.
`Utils.rescan()`	Activate injected components.
`Utils.useComponent(name)`	Register a component.
`Utils.getAppUrl()`	Get application base URL.
`Utils.saveData(key, val)`	Save cross-screen data by key (persisted via AppState).
`Utils.getData(key)`	Retrieve cross-screen data by key.
`Utils.getAndEraseData(key)`	Retrieve cross-screen data by key and remove it.

A.11 Frontend: DateUtils.js

Method	Description
`DateUtils.today()`	Today as YYYYMMDD int.
`DateUtils.intToStr4(dt)`	Format as MM/DD/YYYY.
`DateUtils.intToStr2(dt)`	Format as MM/DD/YY.
`DateUtils.strToInt(str)`	Parse date string to int.
`DateUtils.intAddDays(dt, n)`	Add days.
`DateUtils.daysDifference(d1, d2)`	Days between dates.
`DateUtils.year(dt)`	Extract year.
`DateUtils.month(dt)`	Extract month.
`DateUtils.day(dt)`	Extract day.
`DateUtils.isValid(dt)`	Validate date int.

A.12 Frontend: AGGrid

Method	Description
`new AGGrid(id, cols, tag)`	Create grid.
`show()`	Render the grid.
`addRecords(data)`	Load/replace data.
`getSelectedRow()`	Get selected row data.
`getSelectedRows()`	Get all selected (multi).
`getAllRows()`	Get all row data.
`setOnSelectionChanged(fn)`	Selection handler.
`setOnRowDoubleClicked(fn)`	Double-click handler.
`getNumberOfRows()`	Row count.
`clear()`	Remove all rows.
`destroy()`	Clean up grid.
`setServerSort(fn)`	Header clicks call `fn(colId)` instead of sorting; rows never move. Falsy turns it off.
`setSortIndicators(list)`	Show sort arrows from `[{field, asc}]` in priority order (numbered if more than one); `null` clears.

Built-in column formatters:

Formatter	Description
`AGGrid.date`	YYYYMMDD to MM/DD/YYYY.
`AGGrid.time`	HHMM to H:MM AM/PM.
`AGGrid.dateTime`	Date/time epoch to MM/DD/YYYY H:MM AM/PM.
`AGGrid.numericFormat`	Number with commas.

Appendix B

Groovy Primer

This appendix is a crash course in Groovy for Java developers. It covers the language features most relevant to writing Kiss backend services. For a more detailed treatment, see Chapter 7. An additional standalone overview — "Groovy Overview," covering what Groovy is, why it was chosen for Kiss, its strengths and weaknesses, and how it is invoked from the framework — ships with the project as `manual/GroovyOverview.html`.

B.1 Why Groovy?

Groovy is a JVM language that is nearly a superset of Java. Valid Java is (mostly) valid Groovy. The key advantages for Kiss development:

- **No compilation step** — Groovy files are compiled automatically by the framework when they change.
- **Less boilerplate** — optional semicolons, optional types, streamlined syntax.
- **Full Java interop** — call any Java class, use any Java library, no bridges or wrappers.
- **String interpolation** — embed expressions directly in strings.

B.2 Basic Syntax

```
// Semicolons are optional
def name = "Alice"
int count = 42

// Type inference with def
def list = [1, 2, 3]         // ArrayList
def map = [name: "Alice", age: 30] // LinkedHashMap

// String interpolation (GStrings)
println "Hello, ${name}! Count is ${count}."

// Multi-line strings
def sql = """
    SELECT *
    FROM employee
    WHERE active = 'Y'
"""
```

B.3 Differences from Java

Java	Groovy Equivalent
`String[] a = new String[]{"a"};`	`def a = ["a"]`
`map.get("key")`	`map.key` or `map["key"]`
`for (String s : list)`	`for (s in list)` or `list.each { }`
`obj != null ? obj.m() : null`	`obj?.m()` (safe navigation)
`val != null ? val : def`	`val ?: def` (Elvis operator)
`import needed`	Many common imports automatic
`checked exceptions`	All exceptions unchecked
`private fields`	Properties with auto getters/setters

B.4 Variables and Types

Groovy supports both static and dynamic typing:

```
// Dynamic typing (any type)
def x = "hello"
x = 42          // valid: x is now an Integer

// Static typing (type-checked)
String name = "Alice"
int count = 42
double rate = 3.14

// Type casting
String s = (String) someObject
int n = someString as int // Groovy-style cast
```

B.5 Closures

Closures are anonymous functions — Groovy's most powerful feature:

```
// Single parameter (implicit 'it')
[1, 2, 3].each { println it }

// Named parameters
[1, 2, 3].collect { n -> n * 2 } // [2, 4, 6]

// Multi-parameter
files.sort { File a, File b ->
    Long.compare(b.lastModified(), a.lastModified())
}

// As a variable
def greet = { name -> "Hello, ${name}!" }
println greet("Alice") // "Hello, Alice!"
```

Warning

When using closures with Kiss framework methods that expect Java functional interfaces (e.g., `Consumer<UserData>`), you must add an explicit cast: `{ ud -> ... } as Consumer<UserData>`.

B.6 String Features

```
// GStrings (double quotes) support interpolation
```

```
def name = "World"
println "Hello, ${name}"    // Hello, World
println "2 + 2 = ${2 + 2}" // 2 + 2 = 4

// Regular strings (single quotes) --- no interpolation
println 'Hello, ${name}'    // Hello, ${name}

// Triple-quoted (multi-line, preserves whitespace)
def msg = """
    Dear ${name},
    Welcome aboard.
"""

// Useful string methods
"Hello".contains("ell")     // true
"Hello World".split(" ")    // ["Hello", "World"]
"hello".capitalize()        // "Hello"
"  hello ".trim()           // "hello"

// SHA-256 hashing (Kiss extension)
"password".sha256()         // 64-char hex string
```

B.7 Collections

```
// Lists
def nums = [1, 2, 3, 4, 5]
nums << 6                   // append
nums += [7, 8]              // concatenate
nums[0]                     // 1 (first)
nums[-1]                    // 8 (last)
nums[1..3]                  // [2, 3, 4] (range)

// Maps
def config = [host: "localhost", port: 5432]
config.host                 // "localhost"
config["port"]              // 5432
config.each { k, v -> println "${k}=${v}" }

// Filtering and transforming
nums.findAll { it > 3 }     // [4, 5, 6, 7, 8]
nums.collect { it * 2 }     // [2, 4, 6, ...]
nums.sum()                  // 36
nums.max()                  // 8
```

B.8 Control Structures

```
// Standard if/else
if (count > 10) {
    println "Many"
} else {
    println "Few"
}

// Switch (supports ranges, types, closures)
switch (x) {
    case 1..5:   println "small"; break
    case 6..10:  println "medium"; break
    case String: println "text"; break
    default:     println "other"
}

// For loops
for (item in list) { ... }
for (int i = 0; i < 10; i++) { ... } // Java-style

// While
while (condition) { ... }
```

B.9 Null Safety

```
// Safe navigation operator (?.)
String city = employee?.address?.city
// Returns null if employee or address is null
// (no NullPointerException)

// Elvis operator (?:)
String name = inputName ?: "Unknown"
// Uses "Unknown" if inputName is null or empty

// Combining both
String dept = emp?.department?.name ?: "Unassigned"
```

B.10 Common Patterns in Kiss Services

```
// Service method signature
void methodName(JSONObject injson, JSONObject outjson,
```

```
                Connection db, ProcessServlet servlet) {
    // Read input
    String id = injson.getString("id")

    // Database operations
    Record rec = db.fetchOne(
        "select * from table where id = ?", id)

    // Null check with safe navigation
    String name = rec?.getString("name") ?: ""

    // Return data
    outjson.put("name", name)
    outjson.put("records", db.fetchAllJSON(
        "select * from table order by name"))

    // Commit if modified
    db.commit()
}
```

B.11 Working with Java

Groovy code can call any Java class directly:

```
// Java classes work seamlessly
import java.time.LocalDate
import java.util.UUID

def today = LocalDate.now()
def id = UUID.randomUUID().toString()

// Kiss Java classes
import org.kissweb.database.Connection
import org.kissweb.DateUtils

int date = DateUtils.today()
```

Java code can also call Groovy classes, though in Kiss the typical pattern is Groovy services calling Java framework classes, not the reverse.

B.12 Common Pitfalls

Default package

Classes without a `package` declaration cannot be imported

from packaged Groovy files. Always use a package for shared utility classes.

Closure type inference

Groovy's type inference can cause `ClassCastException` in closure-based operations. Use explicit `for` loops when this occurs:

```
// May fail with type issues
def files = dir.listFiles().findAll {
    it.name.endsWith(".sql") }

// Safer alternative
File[] all = dir.listFiles()
List<File> sqlFiles = []
for (File f : all) {
    if (f.name.endsWith(".sql"))
        sqlFiles.add(f)
}
```

Optional syntax can hurt readability

Optional semicolons and optional parentheses on method calls are convenient but can reduce readability, especially in chained or nested expressions. Prefer parentheses on method calls when the intent would otherwise be unclear.

Parentheses ambiguity

While parentheses are optional in some cases, omitting them can cause Groovy to interpret a method call as property access:

```
// Ambiguous: method call or property?
println list.size  // works (Groovy magic)
println list.size() // clearer, always works
```

No `char` type

Groovy has no distinct character literal — `'x'` is a `String`, not a `char`. When a character is needed (for example, calling a Java API that takes a `char`), use an explicit cast:

```
char c1 = (char) 'x'
char c2 = 'x' as char
```

Floating-point literals are `BigDecimal`

A literal like `3.14` is a `BigDecimal`, not a `double`. This has caused many subtle bugs — for example, when passing a literal to an overloaded method, the `BigDecimal` overload is selected instead of the `double` overload, or arithmetic produces unexpected scale or rounding behavior. When a `double` is required, use the `d` suffix:

```
def a = 3.14    // BigDecimal
double b = 3.14d // double
```

`==` vs `.equals()`

In Groovy, `==` calls `.equals()`, not reference equality. Use `.is()` for reference comparison.

Appendix C

IDE Setup

Kiss can be developed in any text editor, but an IDE provides code completion, debugging, and navigation. This appendix covers setup for the most common IDEs.

C.1 General Approach

Kiss does not use Maven or Gradle, so IDE setup is manual but straightforward. The key steps for any IDE are:

1. Open the Kiss root directory as a project.
2. Configure source directories.
3. Add library JARs to the project classpath.
4. Set the Java SDK to version 17 or later.
5. Exclude build output and dependency directories.

The source directories to configure are:

Directory	Type	Contains
`src/main/core`	Source root	Framework Java code
`src/main/precompiled`	Source root	Shared utilities
`src/main/backend`	Source root	Groovy/Java services

Directory	Type	Contains
`src/test/core`	Test root	JUnit test classes

Library JARs are in the `libs/` directory. Add all `*.jar` files to the project classpath.

Warning

Do not modify files under `src/main/core/` or `src/main/frontend/kiss/`. These are framework files. Application code goes in `src/main/backend/` (services) and `src/main/frontend/` (screens).

C.2 IntelliJ IDEA

C.2.1 Community Edition

1. **Open project**: File → Open → select the Kiss root directory.
2. **Set SDK**: File → Project Structure → Project → SDK → Java 17+.
3. **Mark source roots**: In Project Structure → Modules:
   - Mark `src/main/core` as Sources.
   - Mark `src/main/precompiled` as Sources.
   - Mark `src/test/core` as Tests.
4. **Add libraries**: Project Structure → Libraries → Add → Java → select the `libs/` directory.
5. **Exclude directories**: Mark `work/`, `node_modules/`, `target/`, and `tomcat/` as Excluded.

C.2.2 Ultimate Edition

In addition to the Community Edition steps, Ultimate Edition provides:

- **Groovy support** — code completion, refactoring, and error highlighting in `.groovy` service files.

- **JavaScript debugging** — set breakpoints in frontend code and debug with Chrome.
- **Database tool** — connect to your database and run queries from within the IDE.
- **HTTP client** — test backend service endpoints directly.

Tip

IntelliJ IDEA is the recommended IDE for Kiss development due to its excellent Groovy and Java support. The Community Edition is free and sufficient for most development tasks.

Note

IntelliJ IDEA is the only IDE known to support interactive debugging of Kiss's dynamically loaded backend services. Because Kiss compiles Groovy and Java service files at runtime rather than at build time, breakpoints set in service source files are only honored by IntelliJ's debugger. Other IDEs (Eclipse, NetBeans, VS Code) can edit and navigate these files but cannot reliably step through them while the server is running. If interactive debugging of services matters to your workflow, use IntelliJ.

C.3 Eclipse

1. **Import project**: File → Import → General → Existing Projects into Workspace → select the Kiss root directory.
2. **Configure build path**: Right-click project → Properties → Java Build Path:
   - Source tab: Add `src/main/core`, `src/main/precompiled`.
   - Libraries tab: Add External JARs → select all JARs in `libs/`.
3. **Install Groovy plugin** (optional): Help → Eclipse Marketplace → search for "Groovy Development Tools" → Install.

4. **Set JRE**: Properties → Java Build Path → Libraries → set JRE System Library to Java 17+.

C.4 NetBeans

1. **Open project**: File → Open Project → select the Kiss root directory (or create a new Java project with existing sources).
2. **Add source folders**: Project Properties → Sources → Add Folder for `src/main/core` and `src/main/precompiled`.
3. **Add libraries**: Project Properties → Libraries → Add JAR/Folder → select all JARs in `libs/`.
4. **Set platform**: Project Properties → Libraries → Java Platform → Java 17+.

C.5 VS Code

1. **Open workspace**: File → Open Folder → select the Kiss root directory.
2. **Install extensions**:
   - **Extension Pack for Java** (Microsoft) — Java language support, debugging, project management.
   - **Groovy** (optional) — syntax highlighting for `.groovy` files.
3. **Configure classpath**: Create a `.vscode/settings.json` with:

```
{
    "java.project.sourcePaths": [
        "src/main/core",
        "src/main/precompiled"
    ],
    "java.project.referencedLibraries": [
        "libs/**/*.jar"
    ]
}
```

4. Frontend HTML/CSS/JavaScript support is built in — no additional extensions needed.

C.6 General Tips

- **Use the terminal for building and running.** Run `./bld develop` in a terminal alongside the IDE. The IDE is for editing and navigation; the `bld` script handles compilation, server startup, and hot reload.
- **Do not create IDE-specific build configurations** for Kiss. The `bld` script is the authoritative build system. IDE build features may conflict with the framework's compilation order.
- **Exclude large directories** (`work/`, `node_modules/`, `target/`, `tomcat/`) from indexing to keep the IDE responsive.
- **Frontend editing** does not require special IDE support. Kiss frontend code is plain HTML, CSS, and JavaScript with no build step. Any editor with basic web support works well.
- **Groovy editing** benefits from IDE support (code completion, import management, error checking), but is not strictly required. Groovy syntax is close enough to Java that a Java-configured editor provides a good experience.

Note

Remember that Kiss compiles and loads service files automatically when they change. You do not need to trigger a build from the IDE — just save the file and the framework handles the rest.

Index

$$(), 119
_uuid, 128

accordion, 181
AGGrid, 191
Anthropic, 294
app-mode, 146
app-version, 146
application.ini, 33, 332
AppState, 117, 127, 128, 214
ArrayUtils, 415
AsExtensions, 365
AuthenticatedUser, 351, 365
authentication
    service-decides, 234
AuthorizationCodeStore, 364
AuthorizationServerConfig, 364
AuthorizationServlet, 365
avatar-badge, 184

badge-chip, 183
Base64, 413
BearerTokenValidator, 343
blind index, 404
boot id, 237
bootstrap kernel, 147
bootstrap.js, 116, 149
BuildUtils, 22, 27, 416

C3P0, 19, 74
cacheBust, 149
cacheBustHtmlResources, 150
check-box, 173
clickjacking, 159
ClientStore, 364
combo-box, 167
Command, 65
components.js, 161
Connection, 65
    getColumnWidth, 73
ConnectionType, 66
ConsentProvider, 353, 365
ConstantTime, 405
Content-Security-Policy
    connect-src, 156
    script-src, 155
CORS, 432
Cron, 249
Crypto, 396
Cursor, 65

date-input, 164
DateTime (backend), 394
DateTimeUtils.js, 209
DateUtils (backend), 393
DateUtils.js, 207
decimal-hours, 165
DelimitedFileImport, 261
    check, 267
    Column, 269
    getFileColumnTitles, 265
    getUnknownColumns, 266
    getUnknownString, 266
    maxLength, 270

notBlank, 269
truncate, 270
truncate(label, value, max), 274
unknownColumnRules, 273
DelimitedFileReader, 258
DelimitedFileWriter, 245, 275
design tokens, 187
DOMUtils, 19, 115
DOMUtils.js, 117, 211
DOMUtils.preventNavigation, 212
downloadAll, 30
drop-down, 166
duration-input, 165
DynamicClientRegistration, 382
DynamicClientRegistrationServlet, 365

Editor.js, 214
Electron, 419
ExternalFile, 280

fetchHTML, 150
file-upload, 175
file-upload component, 255
FileUtils, 247, 278
FTP, 275

Groff, 243
GroovyUtils, 417

Hash, 402
Hmac, 403
hot reload, 15
HSTS, 160

Image, 406
index.html, 116, 146
IniFile, 411

JSON-RPC, 16
JsonPath, 412
JsonTemplate, 416
JUnit, 423
JwksCache, 343
JwksPublishServlet, 365

KeyManager, 364
Kiss Framework, 3
KissBuildUtils, 22
KissInit.groovy, 231

list-box, 172
LogException, 50
loginHash, 141
LRUCache, 414
LRUSet, 414

MainServlet, 14
MCPClientBase, 313
MCPClientExample, 313
MCPServerBase, 313, 334
MCPServerExample, 313, 343
menu-button, 179
MetadataServlet, 365
MutableString.js, 214

native-date-input, 164
NumberFormat, 246
NumberFormat (backend), 395
NumberUtils (backend), 395
NumberUtils.js, 211
numeric-input, 163

OAuthAuthorizationRequiredException, 383
OAuthCallbackServlet, 383
OAuthClient, 374, 382
OAuthClientConfig, 382
OAuthClientException, 383
OAuthClientProvider, 382
OAuthClientStore, 382
OAuthConfig, 342

OAuthMetadataDiscovery, 382
OAuthSqliteStore, 364
Ollama, 299
  streaming, 301
OpenAI, 289
  Api enum, 291
OpenRouter, 296
org.kissweb.oauth, 342
org.kissweb.oauth.as, 364
org.kissweb.oauth.client, 382

panel-card, 182
PasswordHash, 401
PDF, 248, 278
PendingAuthorization, 383
picture, 176
Pkce, 382
PkceValidator, 364
popup, 201
ProcessServlet, 14
ProtectedResourceMetadata-
    Servlet, 343
push-button, 174

QdrantClient, 306
QueryBuilder, 87
QueueManager, 14

radio-button, 174
RandomUtil, 406
Record, 77
RefreshTokenStore, 364
RegisteredClient, 365
RequestConnectionPreparer, 74
RestClient, 313
RestServerBase, 313
RestServerExample, 313
Router, 127, 133
routes.js, 117, 135

sameOriginBackend, 124
SchemaGraph, 107
SchemaGraph.Edge, 111
search-input, 176
section-title, 183
security audit, 3
SecurityHeadersFilter, 154
segmented-control, 178
server restart detection, 237
Server.js, 132, 217
ServerException, 50
service-decides authentication,
    234
session management, 233
`setServerSort`, 197
`setSortIndicators`, 198
smart-chooser, 170
SortBuilder, 101
stampVersion, 152
starter-theme.css, 186
streaming
  Ollama, 301
StringUtils, 391
SystemInfo.js, 116
SystemInfo.stateStore, 128

Tasks.java, 22
text-input, 163
text-label, 175
textbox-input, 175
ThreadLevelCache, 415
time-input, 164
TimeUtils (backend), 394
TimeUtils.js, 208
toast, 186
TokenIssuer, 364
TokenServlet, 365
Tomcat, 433

URLBuilder, 413
UserAuthenticator, 351, 365
UserCache, 18, 233

UserData, 18, 233
UserException, 49
Utils.css, 187
Utils.js, 213
Utils.loadPage, 115
Utils.popup_close, 202
Utils.popup_open, 202
Utils.showMessage, 205
Utils.toast, 185
Utils.yesNo, 205

ValidatedToken, 343

WAR file, 431
web search
    LLM, 288

XML, 414