Kiss

Getting Started Guide
30 April 2026

by Blake McBride

This guide was typeset with the TEX typesetting system developed by Donald Knuth utilizing the Texinfo format.

Short Contents

Table of Contents

1 Introduction

The *KISS Framework* is an application development framework for developing web-based business applications, portable desktop applications, and command-line utilities. The main home for *Kiss* is https://github.com/blakemcbride/Kiss

Kiss' focus is on simplicity and development speed. By being simple to develop in, the development and support of the application can occur more rapidly. Simplicity is achieved by abstracting away as much common functionality as possible so that developer lines of code are maximally applicable to the application solution rather than infrastructure and support of the framework. Throughout the framework, business-normal defaults have been employed in order to minimize commonly needed functionality.

Another goal of the *Kiss* framework is to be a complete web-based application development solution. *Kiss* isn't a browser solution alone, nor is it a back-end solution. *Kiss* includes solutions for both ends – although the two sides may largely be used independently.

Kiss attempts to create a consistent interface. This can greatly simplify code even in simple cases. For example, in terms of an input text control, why would you disable/enable it with:

```
$('#id').prop('disable', false);
$('#id').prop('disable', true);
```

and then hide/show it with:

```
$('#id').hide();
$('#id').show();
```

Kiss provides a consistent interface. With *Kiss*, you would do:

```
$$('id').disable();
$$('id').enable();
$$('id').hide();
$$('id').show();
```

Kiss is designed to be simple to get started with, simple to learn, and simple to use. *Kiss* does this while supporting important technologies such as micro-services, front-end components, and SQL.

The term *single page application* has several, subtly different meanings. One meaning is that the entire application code is bundled into a single file or HTTP GET request. In that sense, *Kiss* is not a single page application. This makes no sense for a business application that could have hundreds of screens.

Another meaning of the term *single page application* is that there is only a single `html` tag and all of the remaining pages are modifications of the original `html` tag contents. In this sense, *Kiss* is a *single page application*. *Kiss* applications lazy-load as needed. Browser cache is leveraged to minimize Internet traffic.

Kiss is used in a production environment and built by someone with more than 30 years of experience as a framework designer and a business application software engineer. So *Kiss* is not a proof of concept.

Kiss was built as a solution to the challenges faced by the author when developing web-based business applications. As such, *Kiss* is more a solution for business application development than for the development of public facing company presentation web sites.

Another goal of *Kiss* is to keep the front-end and back-end as independent of each other as possible. To this end, communications between the front-end and back-end occur via REST services and JSON. This accomplishes two things. First, it allows your organization to be best prepared for the ever-evolving software environment. Pieces can be changed and enhanced without causing massive rewrites of the entire system. The second advantage is that by pushing as much processing to the front-end as possible (rendering the display on the front-end), the system can better scale.

1.1 The Kiss Book

For a comprehensive, tutorial-style guide to the *Kiss* framework, see *Kiss: A Complete Guide to the Web Application Framework* by Blake McBride, available on Amazon at `https://a.co/d/035V1VEl`. The book provides in-depth coverage of backend and frontend development, database operations, authentication, reporting, deployment, and a complete worked application from start to finish.

This guide provides the essential reference information needed to set up, configure, and develop with *Kiss*. The book offers deeper explanations, more examples, and guided tutorials.

1.2 Writing Sophisticated Applications

Kiss comes as a very basic running application. Since the front-end and back-end can be changed while the system is running, it is intended that a developer would start with this basic application and build it into the desired application.

The basic application the system comes with does not represent the kind of application that can be built with *Kiss*. It is intentionally kept basic so that it is easy to see how it works. In a real application, all of the possibilities made available with *HTML* and *CSS* are at your disposal without limitation. In other words, the only limitations to the sophistication of the application are your imagination and the amount of work you're willing to put into it.

Kiss comes with a complete front-end framework that includes most of the things needed for a modern business web application. In fact, it is currently being used as the sole framework in some very large production systems. It works well. However, there is nothing in *Kiss* that prevents you from using other front-end frameworks such as *React*, *Angular*, *Vue*, etc. In fact, except for dealing with web services and authentication, you could exclusively use your preferred front-end framework.

The *Kiss* back-end is a complete back-end. It is currently being used by large web applications involving many hundreds of screens and SQL tables and thousands of users. The *Kiss* back-end contains many built-in features to efficiently handle many, many simultane-

ous users. Like the front-end, the *Kiss* back-end is designed to be as simple as possible to build production-quality web applications. For example, a web service can be built with a single source file containing just a handful of lines of code. Everything in *Kiss* is designed to be simple to understand, simple to use, and yet be fully compiled and run at top speed.

1.3 *Kiss* Highlights

Some highlights of the *Kiss* system include:

1.3.1 Back-end Highlights

1. Micro services - add, change, or delete a web service on a running system.
2. Each web method is in a single file and is very simple. No configuration files or setup code are needed.
3. Easy access to common SQL databases with support for nested queries without cursor interference.
4. All REST services are stateless. However, the system fully authenticates each request.
5. Changes to web services occur immediately on a running system, without the need to reboot the application.
6. A growing class library to handle common business application needs.
7. The back-end framework is written in Java, and the system is portable to Linux or Windows servers.
8. Web services may be written in Groovy, Java or Common Lisp. Python, JavaScript, Ruby, and Scala are expected to follow soon.
9. User authentication
10. Asynchronous back-end REST services (via a queue and thread pool) provide support for heavy loads and high throughput.
11. A powerful and convenient class library for dealing with SQL persistence that supports PostgreSQL, Microsoft SQL Server, MySQL, Oracle, and SQLite.
12. Built-in interface to the *Ollama*, *OpenAI*, and *Anthropic* LLM servers used to add AI features to your application.

1.3.2 Front-end Highlights

1. Build your own HTML components, thus encapsulating any amount of code into a simple, custom HTML tag.
2. Browser cache control. Never ask your users to clear their browser cache again.
3. All code is written in JavaScript/HTML/CSS. No need for a complex build and debug process, nor any need to learn yet another language.
4. Growing list of included business oriented components designed to provide simple access to fully functional business components.
5. Straight forward means of designing your own components without a lot of hidden and unpredictable magic.
6. System is small and concise, rather than hundreds of megabytes other systems take up.
7. Consistent and simple API.

1.3.3 Back-end Web Service Example

The following example depicts a complete back-end web service. The path to the file is its URL. The class name is the web method name.

The file is a text file, but compiled code gets executed. Authentication occurs before `main` is called.

Simply drop the file in place and the web service and method become immediately available on a running system. Changes to the service take effect immediately (no need to reboot the server app). There are no configuration files or other code that needs to be changed.

For example, the following file is located in the `services` directory.

```
class MyWebService {
    void myWebMethod(JSONObject injson, JSONObject outjson) {
        int num1 = injson.getInt("num1");
        int num2 = injson.getInt("num2");
        outjson.put("result", num1 + num2);
    }
}
```

1.3.4 Front-end Web Service Usage Example

The following front-end example utilizes the web service defined in the previous sub-section.

```
let data = {
    num1:  22,
    num2:  11
};
let res = await Server.call("services/MyWebService", "myWebMethod", data);
if (res._Success) {
    let result = res.result;
    //...
}
```

1.4 Supported Environments

1.4.1 Development Environment

The following development platforms are supported by the *Kiss* framework:

- Linux
- macOS
- Windows
- WSL under Windows
- BSD
- OpenIndiana[1]

[1] The SQLite interface does not work on OpenIndiana due to reasons unrelated to OpenIndiana. However, *Kiss* works fine with the other supported databases on OpenIndiana. Also, the demo that *Kiss* comes

- Haiku[2]

1.4.2 Production Environment

The following production platforms are supported by the *Kiss* framework:

- Linux
- Windows Server
- BSD
- OpenIndiana

1.4.3 Databases Supported

The following database servers are supported by the *Kiss* framework:

- PostgreSQL
- Microsoft SQL Server
- MySQL
- Oracle
- SQLite

1.4.4 Java

The system is tested with Java versions 17, 21, 25. Any Java version above 17 is expected to work.

1.5 REST vs. JSON-RPC

Throughout this guide, the communications mechanism between the front-end and back-end is often referred to as *REST*. Strictly speaking, *Kiss* uses *JSON-RPC* (Remote Procedure Call) — all calls use *POST*, return data as JSON, and use application-level error codes rather than *HTTP* status codes for operation failures. The term *REST* is used loosely because it is more widely recognized, and the underlying transport is standard HTTP.

For a detailed discussion of the differences and the design rationale, see the *Kiss* book.

1.6 HTML component usage

To use a component, add to HTML:

```
<my-component></my-component>
```

Add to JavaScript:

```
Utils.useComponent('MyComponent');
```

The component can put any HTML in the component location, have any functionality, have its own modal windows, and use other components. The component can have custom

with will not run completely on OpenIndiana because it depends on SQLite. Once another database is configured, all is well.

[2] The SQLite interface does not work on Haiku due to reasons unrelated to Haiku. However, *Kiss* works fine with the other supported databases on Haiku. Also, the demo that *Kiss* comes with will not run completely on Haiku because it depends on SQLite. Once another database is configured, all is well.

and non-custom attributes (like style). Non-custom attributes do what you'd expect them to do.

The system also supports tag-less components. This provides an easy way to package arbitrary blocks of code (that can have screens too).

1.7 System Maturity And Stability

The Kiss system has been used in production environments for several years. Additionally, several commercial applications utilize *Kiss*. In spite of this, however, *Kiss* is constantly being adjusted in response to additional needs, evolving environments, and bug fixes.

We use *Kiss* daily in a Linux and PostgreSQL environment. Therefore, it is best tested there. While we support all of the listed environments, they receive a bit less testing. If you encounter a problem, please reach out to us. It is probably easy for us to fix, and we are happy to do so.

1.8 Getting All Source Code

Source code for all of *Kiss* and its dependencies is freely available. The builder program located at `src/main/precompiled/Tasks.java` contains the paths to all of the external dependencies (those not included in the *Kiss* distribution). The following lists the paths to the internal dependencies (those included with *Kiss*):

`abcl.jar` https://common-lisp.net/project/armedbear

`SimpleWebServer.jar (only used during development)`
https://github.com/blakemcbride/SimpleWebServer

1.9 Support, Contact, And Links

The *Kiss* main website is at https://kissweb.org

Source code is at https://github.com/blakemcbride/Kiss

Public discussion and support are available at
https://github.com/blakemcbride/Kiss/discussions

Issue tracking is at https://github.com/blakemcbride/Kiss/issues

Commercial support is available. Contact us via email at blake@mcbridemail.com

1.10 License

1.11 Acknowledgments

The Kiss design, code, documentation, and web site were written by Blake McBride. The author gratefully acknowledges and appreciates, among others, the following:

Apache Groovy located at https://groovy-lang.org

JSON-Java located at https://github.com/stleary/JSON-java (although I am using a modified version available at https://github.com/blakemcbride/JSON-java)

C3P0 located at https://www.mchange.com/projects/c3p0

Texinfo located at https://www.gnu.org/software/texinfo/

ABCL project located at https://common-lisp.net/project/armedbear

Melaine Sarbey (melswildart@gmail.com) for creating the Kiss logo.

2 System Setup

2.1 Important

The *Kiss* system comes with its own build system, so it doesn't use build systems such as *ant*, *maven*, or *gradle*. It works under Linux, macOS, Windows, etc. This included build system will make things such as downloading remote libraries, installing and configuring a web server, building *Kiss*, and running *Kiss* for development purposes easy, intelligent, and automatic. This new build system can be used in conjunction with your favorite IDE.

The build system (called *bld*) is small and written in Java. Its source code is included with the *Kiss* source code. So, a Java compiler will be needed before anything will work. Java 17, 21, 25, and above are supported.

When building the system for the first time, *BLD* will automatically download and cache required libraries (jar files), install and configure the development server (Tomcat), build the system, and it can even be used to launch the development-mode server. After building the system the first time, application development proceeds without any need for re-compiles or re-builds.

The program that runs *BLD* is "*bld*" under Unix-like systems and "*bld.cmd*" under Windows. In this guide, "*bld*" will be shown as "*./bld*". The "./" is required under Linux and macOS but not under Windows. When running on Windows, use "bld" rather than "./bld" and the same for all the other commands shown.

Note. The *Kiss* system includes a file named `pom.xml`. However, *Kiss* is *not* a Maven project. The `pom.xml` file is only included to provide *GitHub* with a list of dependencies. Unfortunately, this file sometimes confuses IDEs into interpreting the existence of that file as the project being a *Maven* project. Therefore, in some instances, it is a good idea to delete that file prior to IDE configuration.

The `build.xml` file is also not used by the *Kiss* system per se. It is an integration with *ant* for IDE integration.

2.2 Super-Quick-Start

This is the simplest and shortest path to a running system. It assumes:

1. you have a JDK installed
2. you have your JAVA_HOME and JRE_HOME environment variables set correctly
3. you have an Internet connection

After doing a `git clone`, all that is needed is the following:

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

This will build the system, install Tomcat, deploy the app, and run the server. At this point you will be able to go to your browser at the following URL:

```
http://localhost:8000
```

Be sure not to use port 8080. Although port 8080 will appear to work, you will not be able to do front-end development while the system is running. Port 8000 will allow front-end development while the system is running.

At this point, you can do all development without any build procedures. You can add or change anything on the front-end or back-end while the system is running.

All application back-end code is located under the `src/main/backend` directory.

All application front-end code is located under the `src/main/frontend` directory.

If you change anything in either place, the system will notice the change and deliver it with the next request.

2.3 Quick-Start Checklist

This is a more detailed and expanded description of the same super-quick-start.

The following enumerates the steps necessary to get the system up and running:

1. See [Pre-requisites], page 11,
2. See [Download Kiss], page 12,
3. See [Setup and Configuration], page 13,
4. The development server can be run without an IDE by running: `./bld develop`
5. Once the server starts up, you can access it on your browser by going to `http://localhost:8000` You can also debug the back-end by attaching to the process at port 9000.

   Alternatively, your IDE can be configured to run the development process entirely through it.

Once the development server is running under *bld*, you can stop it by hitting any key.

2.4 Runtime Environments

As shipped, there are two different environments that *Kiss* may run in as follows:

1. Development
2. Production

The *Production* scenario is created with a single command (`./bld war`) and produces a single `war` file (located in the `work` directory) that can be deployed to your production environment.

Before doing anything with the *Development* environment, it is important that you have the *JAVA_HOME* and *JRE_HOME* environment variables set correctly to the root of your JDK (Java Developer Kit). Doing this varies according to the OS you are using and various other Java installation possibilities. There are plenty of instructions on the Internet for this.

The *Development* environment consists of two servers. One serves the back-end REST services, and the second serves the front-end HTML, CSS, and JavaScript files. By using this method, both front-end and back-end source files can be changed on a running system and take effect immediately without any builds, compiles, server reboots, re-deploys, or file copies. (This is also true of a production environment – with a single server – when the new files are put in place.)

Back-end REST services are debugged, and edited through the IDE. Saving a source file is all that is needed to have it take effect.

The front-end (HTML, CSS, and JavaScript files) are served by a simple server supplied with the *Kiss* system. This server is only used during the development process. See [Front-end Development], page 27. (Source code to this server is available at `https://github.com/blakemcbride/SimpleWebServer`) Debugging the front-end is done through the browser debugger. There is no setup, and the front-end server runs by executing a single command.

2.5 Pre-requisites

Kiss requires a few common pre-requisites as follows:

1. Java JDK 17, 21, 25, +.
2. SQL Database Server — *Kiss* comes with the *SQLite* database. However, in a production system you're likely to prefer a more substantial database server. *Kiss* supports a variety of SQL database servers including PostgreSQL, Microsoft SQL Server, MySQL, Oracle, and SQLite.
3. IDE — Although *Kiss* does not require an IDE, one is strongly recommended (e.g. IntelliJ (`https://www.jetbrains.com/idea`), VSCode, NetBeans, Eclipse)
4. GIT source code control system
5. Groff — *Kiss* utilizes the open-source *groff* typesetting system in order to produce PDF reports.

On Linux, most of these packages may be obtained utilizing the normal package manager that comes with your distribution.

The following sub-sections detail how to get those pre-requisites for Windows and macOS.

2.5.1 Windows

You may obtain the requirements under Windows as follows.

1. Java JDK 17, 21, 25 or above for many OSs at:
   - `https://www.azul.com/downloads`
   - `https://aws.amazon.com/corretto`
   - `https://adoptopenjdk.net`
   - `https://developers.redhat.com/products/openjdk/download`
2. *Kiss* comes with the SQLite database. However, you may install other databases such as PostgreSQL, Microsoft SQL Server, MySQL, Oracle, or SQLite.

3. IDE (e.g. IntelliJ (https://www.jetbrains.com/idea), VSCode, NetBeans, Eclipse)

Correctly setting the *JAVA_HOME* and *JRE_HOME* environment variables to the root of your JDK is necessary. Setting this varies from OS to OS and also depends on where it gets installed. Instructions for setting this variable are all over the Internet.

The remaining requirements may be obtained through the *Chocolatey* Windows Package Manager located at https://chocolatey.org Their "community" version is free. We used their "Individual" install. The following packages should be installed after installing the choco package manager:

- choco install groff
- choco install strawberryperl
- choco install ghostscript
- choco install git

After installing these programs, you must exit and re-start any terminals you have running.

2.5.2 macOS

macOS requirements may be obtained via the *brew* package manager available at https://brew.sh Once `brew` is installed,you can install the remainder of the requirements with the following commands:

- brew install openjdk@17
- (Add the following line to your `~/.zshrc` file and then re-start your terminal.) export JAVA_HOME="$(brew –prefix)/opt/openjdk@17"
- brew install git
- brew install groff

2.6 Download Kiss

Kiss is located at https://github.com/blakemcbride/Kiss

It can be downloaded via the following command:

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

2.7 Quick Test

Once everything has been set up, you can easily build and run the system with the following commands:

```
git clone https://github.com/blakemcbride/Kiss.git
cd Kiss
./bld -v develop (or "bld -v develop" under Windows)
```

Once the system is up, you can go to the following address with your browser: `http://localhost:8000`

2.8 Documentation

The *Kiss* documentation consists of three parts; this guide, the detailed back-end API documentation contained in the JavaDocs, and the detailed front-end API documentation. The JavaDocs do not come with the system, but you can generate them yourself with what is provided. See [Creating JavaDocs], page 22.

This guide may be created in two forms. The first is in an HTML form. The system comes with this. You can also generate a nicely formatted PDF file with the following commands (if you have all of the formatting tools installed):

```
cd manual
make Kiss.pdf
```

Updates to the HTML file are achieved with the following commands:

```
cd manual
make
```

All of the documentation can be accessed with your browser. For example, if the root of *Kiss* is located at `/my/home/path/kiss` then you will be able to access the three manuals at the following URLs:

```
file:///my/home/path/kiss/manual/man/index.html

file:///my/home/path/kiss/work/javadoc/index.html

file:///my/home/path/kiss/manual/jsdoc/index.html
```

2.9 Setup and Configuration

The system is configured by the contents of a single file:
`src/main/backend/application.ini` A reboot of the web server is required if any of the parameters in this file are changed.

When the system starts up, the file
`src/main/backend/KissInit.groovy` is run. System startup code can be added here. A reboot of the web server is required if this file is changed.

Given that *Kiss* is for business applications, it authenticates its users. In order for this to work, there is usually a database of valid users. This information is persisted in an SQL database. Therefore a database is normally required. However, for testing purposes, if no database is configured, the system will still run and allow any username and password to succeed.

If you are using an alternative data store for user management, you can force authentication without configuring a SQL database by setting `RequireAuthentication = true` in `application.ini`. When this option is enabled, the system will delegate authentication to `Login.groovy` even when no SQL database is configured. See [Authentication], page 28.

As shipped, the system comes configured as follows:

Database type	SQLite

Host	localhost
Database	DB.sqlite
Database user	[empty]
Database user password	[empty]

Valid options for the Database type are as follows:

- PostgreSQL
- MicrosoftServer
- MySQL
- Oracle
- SQLite

Support for other databases is easy to add.

setMaxWorkerThreads defines how many REST services may be processed in parallel. Service requests beyond this are placed in a FIFO queue and processed as worker threads become available. This capability drastically improves the system's ability to handle a large number of simultaneous users.

The remaining parameters should be self-explanatory. Use the format shown in the example.

Although *Kiss* comes with a default demo database, another one should be configured in live or more substantial development environments. An SQL script file named `schema.sql` is included with the system to initialize said database. Application-specific tables may be added to this database.

The default username is *kiss*, and the default password is *password*

2.10 Bypassing Authentication

On rare occasions, it is necessary to bypass authentication. In other words, one should be able to execute a web service without being logged in. One example of this is registering a new user. If the user isn't already a valid user, they can't execute services in order to register.

Kiss has a way of dealing with this scenario. This is done in the same configuration file where all of the configuration options are located specifically in the file `src/main/backend/KissInit.groovy`. In that file, you are able to specify specific web methods that may be called without authentication. That file has an example.

It should be noted that it is recommended that something like a *captcha* be used to at least ensure that you are communicating with a human. Additionally, it should be noted that the danger of providing unauthenticated services is somewhat mitigated by the combination of *HTTPS* and *CORS*.

2.11 Building The System

Although the system may be built with the included build system (called *bld*) *or* your favorite IDE, the *bld* system should be used for the initial step which downloads the external dependencies (jar files).

The build system included with *Kiss* (called *bld*) has been tested on Linux, macOS, Windows, and several other OSs. The system also includes an *Ant* build file (named *build.xml*) that is only used for IDE integration with the included *bld* system.

The build system included with Kiss is written in Java and is located under the *src/main/core/org/kissweb/builder/* directory. This build system also includes two driver batch files / shell scripts used to build and run the build system. All that is needed to use this system is a Java compiler. (As a side note - this build system is generic and can be used to build other types of projects.)

The build process is run from the command-line. No IDE is necessary. There is no specific IDE integration. None is needed because the system is rarely built. After the first build, application development is done without any build process.

The build system, which comes in source form, must be built before it can be used to build *Kiss*. However, the build system gets built automatically by *bld*.

You can see what operations it can perform by typing:

```
./bld list-tasks            [Linux, macOS, BSD, etc.]
    -or-
bld list-tasks              [Windows]
```

Those tasks that require prior tasks will evoke the dependent tasks automatically. The system is smart enough not to repeat tasks that are unneeded.

The main tasks that will be of interest to you are as follows:

`help` This will display a help message.

`list-tasks`
: This will list all of the available tasks in *bld*.

`libs` This task is only required if you intend to use your IDE to build the remainder of the system. It installs the required dependencies.

`develop` This will cause the entire system to build (not repeating unnecessary steps) and start up the back-end (Tomcat) and front-end servers used in development mode. The system will be started in the foreground, meaning *bld* will not exit until you are done with the entire system. The system will be available from your local browser at `http://localhost:8000`. You may debug the application by attaching to the running *Tomcat* server at port 9000.

This only works from the command line and not from within an IDE. If you want to run the system from within the IDE, see the following targets.

`run-backend`
: This will build and run the development back-end server. The back-end will be started and run in the background.

`stop-backend`
: This will stop the background back-end development server.

`run-frontend`
: This will run the development front-end server in the background.

`stop-frontend`
: This will stop the background back-end development server.

`build`
: This performs all of the steps necessary to set up and build the system but doesn't start up the Tomcat server.

`war`
: This will cause the system to generate the single file needed by a production system. It will end up in *work/Kiss.war*

Other tasks which may be useful are as follows:

`clean`
: This task removes all files built but retains files that were downloaded from repositories (although *bld* caches those files anyway).

`realclean`
: This removes all built and downloaded files so the system should be everything you need to build it without any extraneous files.

`javadoc`
: This task creates the Javadoc files that end up in `work/javadoc`

`jar`
: This creates a jar file named `work/Kiss.jar` This is useful when you want to use the *Kiss* system as a library in another system.

`kisscmd`
: This task creates a command-line JAR that can be used in a non-web, command-line application. This is useful when creating applications that perform various utility functions. This JAR cannot be used in any web environment. See `src/main/core/org/kissweb/Main.java`

See [Important], page 9, and See [Quick-Start Checklist], page 10.

2.11.1 Using an IDE

Most IDEs can be used to develop and debug the application. There are two ways to do this as follows:

1. Using *bld* to build and run the development environment.
2. Using your IDE to build and run the development environment.

Using *bld* to build and run the development environment is the easiest to start off with but is somewhat klunky on an ongoing basis. Its main advantage is that it is portable and doesn't require a lot of IDE configuration. The back-end development server is configured and started by *bld* by simply running *./bld develop* After that, the IDE can be used to debug it by *attaching* to port 9000.

Kiss also comes with an *ant* build that simply calls the native *bld* program. This is useful as an bridge between *bld* and the IDE since many IDEs support *ant*.

Using your IDE to run the entire process is a bit tedious to initially setup but makes the entire process simpler from that point forward. Unfortunately, configuring your IDE is

completely different for each IDE. Instructions for setting up some of the IDEs are located under the `manual/IDE-Setup` directory.

2.12 Kiss Framework Updates

The *Kiss* framework is continually enhanced. *Kiss* includes a mechanism to upgrade your application that will work well if you follow certain rules.

Certain portions of the *Kiss* directory hierarchy are intended to be core parts of *Kiss* and untouched by the developer. (Of course, you can change anything you like; it is just that your ability to perform an update to the *Kiss* system will be hampered.) If you have changes needed to the core of the system, my suggestion is to send me the changes so I can incorporate that additional functionality into the system for everyone. Otherwise, it will be increasingly difficult for you to receive additions made to the core of the system.

The following directories (along with their sub-directories) are not intended to be changed:

- src/main/core
- src/main/frontend/kiss
- the scripts located in the root of the *Kiss* system

The process of upgrading your application is as follows:

1. Back up your existing application.
2. In your application, make a note of the last entry in the file named `KissChangeLog.txt`.
3. Create or update an untouched, virgin clone of the *Kiss* system in another directory using `git clone` or `git pull`.
4. While remaining in the virgin *Kiss* clone directory (and not your application directory), run the following command:

```
./upgrade-kiss <path-to-your-application>     (Linux & macOS)

upgrade-kiss <path-to-your-application>       (Windows)
```

5. Look at the file named `KissChangeLog.txt` in your application directory noting the new entries. These new entries will describe what has changed in the system and what, if any, additional steps may be required to upgrade your system.

Caveat

While this upgrade process upgrades the majority of the system, there are a few files that are not auto-updated to be sure not to clobber possible application-specific code that may have been added. Therefore, the following files will need to be manually verified.

- `src/main/precompiled/Tasks.java`

  This file is left untouched by the upgrade process. You may have custom tasks or modified existing tasks. Be sure to check `buildForeignDependencies()`.

- `libs`

  The upgrade process installs new or updated JAR files that are part of the *Kiss* system. However, you may have added application-specific JAR files. After the upgrade, be

sure to manually delete any outdated JAR files that come with *Kiss* that have been upgraded by this process.

2.13 What Do I Do With It Now?

What you have at this point is the beginnings of your new application. *Kiss* is provided as a running and deployable system. It is expected that you will modify what's there to suit your application needs.

Besides this narrative, you will need the JavaDocs located under the `work/javadoc` directory for back-end API-specific documentation and front-end API reference located in the *manual/jsdoc* directory.

For a guided, tutorial-style walkthrough of building applications with *Kiss* — from architecture through a complete working application — see *Kiss: A Complete Guide to the Web Application Framework*, available on Amazon.

3 Orientation

The entire source code comes with the system and is convenient when debugging. *Kiss* has been designed to segregate *Kiss* framework code from your application code for two reasons:

1. It makes it easy to differentiate framework code from your application code minimizing the code you have to look at when developing your application.
2. It makes upgrading the *Kiss* framework simple and accurate.

Naturally, in order for this to work, it is important that you not change framework code and put all of your application code in the correct areas.

3.1 Back-end Application Files

Some back-end code is normal code that gets pre-compiled at build time, as most systems do — as the *Kiss* core framework code is. However, some *Kiss* code (the majority of the application code) is *hot-loaded* at runtime. What this means is that code that is hot-loaded can be added, changed, or deleted on a running system. There is no need to bring the system down or rebuild anything. In a development environment, development and debugging occur while the system is running continuously. There is no need to bring the system down or rebuild anything. A production system can be updated while the system is running without the need to reboot the server or upset normal user use of the system.

All back-end code resides in the following directories:

`src/main/backend`
: The vast majority of your application-specific code goes in this directory tree. All files in this directory tree (except `KissInit.groovy` and `application.ini`) are hot-loaded.

  This directory contains the following:

  `services`
  : This is where all of your web services will be located. You can structure sub-directories to it as needed. Web services can call any core *Kiss* code, *jar* files, or `precompiled` code directly. However, although web services can call other hot-loaded files located in the `backend` tree, a special syntax is required. There is an example of this in `services/MyGroovyService.groovy`.

    Unlike other coding philosophies that prefer thin web services that call core code, *Kiss* strongly prefers thick web services that contain the bulk of your application code. This maximizes the use and value of the hot-load facility.

  `scripts`
  : This directory contains code that is not web services but common application code that is to be hot-loaded. However, there is a big negative associated with this code. While hot-loaded web services can directly call *Kiss* core code, *jar* files, and `precompiled` code, they can't call methods in the scripts directory directly. They must use a convoluted syntax enabling the hot-loading of those methods. An example of this exists in the `services/MyGroovyService.groovy`.

`application.ini`
: This file configures the system parameters. It sets various system parameters such as database configuration, threads, and other system parameters. See [Setup and Configuration], page 13,

`KissInit.groovy`
: This file is read once upon system startup and is not hot-loaded. It loads `application.ini` and can be used to perform other startup tasks. See [Setup and Configuration], page 13,

`Login.groovy`
: This is the code that validates a user's login. It is kept here because it is common for application customization of the login process.

`CronTasks`
: Files in this directory support the ability to auto-start processes at scheduled and recurring dates and times ala Unix *cron*. See the files in that directory for documentation.

`src/main/precompiled`
: Code in this directory tree is compiled at build time and not hot-loaded. This directory tree is used to store common application-specific code that gets compiled into the system. Whenever code in this directory tree is changed, the application must be rebuilt.

`src/main/core`
: *Kiss* core back-end framework code resides in this directory tree. Nothing in this directory tree should be modified by you because it will be overwritten when *Kiss* is upgraded. (See [updates], page 17)

3.2 Front-end Application Files

Files under the `src/main/frontend` directory represent the front-end of the application.

All files under the `src/main/frontend/kiss` directory are part of the *Kiss* system. Nothing in this directory tree should be modified by you because it will be overwritten when *Kiss* is upgraded. (See [updates], page 17)

`index.html` and `index.js` are also part of the *Kiss* system and aren't normally modified. They contain code that ensures that browsers load updated code.

`index.html` contains three important variables that assure that users correctly load front-end code that has been changed rather than using their cached version. These variables are as follows:

`SystemInfo.softwareVersion`
: This can be set to any unique string. When the system is in production use, if this string changes, the browser will load new copies of all front-end files (which it will cache for future use). If any front-end files are changed, this variable should be changed. The users will get the new screens after they log out and back into the system.

`SystemInfo.controlCache`
: Setting this variable to `true` tells the system to observe the value in `SystemInfo.softwareVersion` and re-load the front-end screens if its value changes.

  The value of this variable is often set to `false` during the development process to avoid double-loading of certain files. When debugging the front-end, be sure to disable the browser cache.

`SystemInfo.releaseDate`
: This variable is used to track the release date of a given front-end. It is used for display purposes only.

In addition to differentiating between development and production environments, `index.js` is also used to detect the user's device type (e.g., desktop, tablet, mobile) and, if desired, load different application screens.

`login.html` and `login.js` represent the user login page and should be modified to suit your needs.

Other directories such as `page1` represent other user pages and would be the application-specific screens you create. The included `page1` directory is only an example page.

3.3 Database

Kiss supports Microsoft SQL Server, Oracle, PostgreSQL, MySQL, and SQLite.

As shipped, *Kiss* comes configured with an embedded SQLite server and database. While this is fine for a demo or small application, a real database should be configured for real use. The included database is located in the `backend` directory and is named `DB.sqlite`

Although *Kiss* has no preferred database, PostgreSQL is strongly recommended because it is free, full-featured, fast, rock solid, and portable on all major platforms.

See [Setup and Configuration], page 13,

3.4 Single Page Application

Kiss applications are single-page applications in the sense that there is a single `<body>` tag and all other pages essentially get placed into that tag on a single page. However, *Kiss* is not a single-page application in the sense that the entire application gets loaded with a single `GET` request. This doesn't make sense for a large business application in which many hundreds of pages may exist. *Kiss* lazy-loads pages as they are used, and except for browser cache, eliminates them once another page is loaded.

3.5 Controlling Browser Cache

As discussed previously, the user's browser cache can be controlled from the file `src/main/frontend/index.html`. In that file, you will see two lines that look as follows:

```
SystemInfo.softwareVersion = "1";  // version of the entire system
SystemInfo.controlCache = false;   // normally true but use false during
                                   // debugging
```

If `SystemInfo.controlCache` is set to `true`, each time `SystemInfo.softwareVersion` is incremented, all users starting the application will be forced to load new code from the server and not use their browser's cache. Once they download the new version, normal browser cache activity will occur.

Users using the system when front-end files are changed will not get the updated files until they log out and back into the system. In a development environment with browser cache disabled, the developer will see all screen changes upon reload, regardless of the variable settings. In fact, it is better to set `SystemInfo.controlCache` to `false` in a development environment to avoid a needless duplicate load of `index.html`.

3.6 Creating JavaDocs

JavaDocs for the *Kiss* system will need to be created. They are created from the command line by issuing the following command:

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

The JavaDoc files end up in the `work/javadoc` directory.

3.7 Deploying A Kiss Application

The only file needed to deploy the application is `Kiss.war` It can be built by typing `./bld war` at a command prompt. `Kiss.war` ends up in the `work` directory. If you have your IDE create the `Kiss.war` file, it will likely not work. The *Kiss* system requires a special build process because application files are distributed in source form. Therefore, `bld` should be used to create the production WAR file.

If using *Tomcat*, `Kiss.war` should be placed in the `webapps` directory. When *Tomcat* starts, it will see the file, unpack it, and run it. The application will be available at `[HOST]/Kiss`

Renaming `Kiss.war` to `ABC.war`, for example, will cause the application path to change to `[HOST]/ABC`

3.8 Learning The System

In order to start getting a feel for how `Kiss` applications function, in terms of the back-end, look at files in the `src/main/backend/services` directory. With `Kiss` you can develop applications in several different languages. The `services` example shows the same code in all of the supported languages.

In terms of the front-end, see the example files under `src/main/frontend/page1`

4 Developing

This section details the development process with *Kiss*. The details provided will be for IntelliJ but can be adapted to other IDEs. The use of an IDE is tremendously beneficial because of the graphical debugging and intelligent code completion capabilities.

A typical development environment includes two separate web servers running on two different ports (all running on your single development machine). One serves the back-end REST services, and the other serves the front-end HTML, JavaScript, and CSS files. This arrangement allows both front-end and back-end development without any compiles, server reboots, file copies, or deployments. Back-end and front-end files can be edited and saved. Their changes take effect immediately.

Of course, in a production environment, only a single web server would be utilized, and front-end and back-end changes take effect immediately there too.

4.1 Overview

The following details the normal steps to boot up a development environment. Once this environment is set up, it may remain active for the whole day. There is rarely any reason to re-build or re-boot the development environment.

Note that all of the command-prompt commands are executed from the root of your application.

1. In a command-prompt, type: `./view-log` (not needed on Windows)
   This is where back-end messages appear. This command is not needed under Windows.
2. In a second command-prompt, type: `./bld develop` (`bld develop` on Windows)
   This builds the system and runs the back-end and front-end servers.
3. In the IDE, attach to the Java process at port 9000.
   This will allow you to debug the back-end.
4. From your browser, go to `http://localhost:8000`
   This is where you view and interact with your application.
5. Open the *Developer tools* from within the browser.
   This is where you debug the front-end.
6. Be sure to disable network caching on the browser. (Otherwise, changes you make to the front-end will not immediately appear in the browser.)

At this point, development, testing, and debugging can occur unabated. There should be no need to rebuild or bring anything down.

Front-end changes will appear as soon as you re-load the page on your browser. Back-end changes will take effect immediately.

4.1.1 IDE

While development with *Kiss* does not require an IDE, most developers utilize an IDE (Integrated Development Environment). *Kiss* includes an effort to assist with IDE integration (such as an *ant* interface to our build system).

See the files under the `manual/IDE-Setup` directory for further information.

4.2 Back-end Development

The back-end works differently in development and production environments. Although in both environments back-end and front-end changes take effect immediately, the setup of the production environment copies all files into the production environment, whereas in a development environment the source and production code are split. In order to facilitate rapid and easy development, it is important that source files be used rather than the production copies during the development process.

The system automatically detects the location of the application source in most configurations. However, this may be explicitly set via a system environment variable (`KISS_ROOT`). The value of this environment variable should be the absolute path of the root of the application source code. The directory it indicates should have a sub-directory named `src/main/backend`.

4.2.1 REST Server

There are two different methods of running the back-end REST server as follows.

4.2.1.1 IDE

Running the IDE back-end requires the following:

1. The IDE is completely configured.
2. The system was built with the IDE.

Typically, the IDE manages a Tomcat instance and serves up the core back-end code serving the REST services. That code detects that it is running under an IDE and re-routes all back-end application files back to the source directories.

Although back-end files are edited in source form and run in compiled form, back-end code can be debugged (including breakpoints) as if it were compiled before the system was booted.

Once the back-end server is up, application files changed under the `src/main/backend` directory will take effect immediately.

4.2.1.2 BLD

Utilizing the included *Kiss build system (bld)*, the many steps required to install and configure Tomcat and the IDE are unnecessary. The whole process (for the back-end portion) can be done as follows:

1. From any stage (including having just downloaded the *Kiss* system) type the following:
   `./bld develop`

   Note: Remember that all commands that start with `./` would drop the `./` under Windows.

At this point, the back-end will be running. From within your IDE, you can attach to the process at port 9000 to debug the back-end (including breakpoints, etc.).

Remember, however, that you won't be able to use or debug the application until the front-end server is started too.

4.2.2 Application Code

All communications between the front-end and back-end occur over REST services you define. Each REST service exists in its own file or class. Web methods are methods within those classes.

As architected, directories under `src/main/backend` represent the application's REST services. Each class/file under that directory represents a web service. The name of the class is the name of the web service.

Instance methods within the web service class represent REST methods for that web service. Each web method is passed four arguments as follows:

`JSONObject injson`
: This represents the data that came from the front-end.

`JSONObject outjson`
: This represents the data being returned to the front-end. It is pre-initialized with an empty `JSONObject`.

`Connection db`
: This is a pre-opened connection to the defined database (defined in `backend/application.ini`)

`MainServlet servlet`
: This is a rarely used servlet context argument.

Basically what happens is:

1. The front-end makes a REST service call.
2. The `Kiss` back-end receives the request.
3. The user gets authenticated.
4. A new database connection is formed, and a new database transaction is started.
5. The requested web service is identified (and loaded and compiled if needed).
6. The `outjson` object is filled in by the web service that is to be returned to the front-end.
7. Upon completion of the REST service, *Kiss* commits the transaction, closes the database connection, and returns `outjson` to the front-end.
8. If, however, the REST service throws an exception, *Kiss* rolls back the transaction, closes the database, and sends an error return to the front-end.

Additional class and instance methods, that are not web methods, may be defined and used within web service classes.

Of course during this process, `Kiss` handles many possible error conditions.

4.2.3 Cached User Data

During the login process, user-specific data may be cached. This often occurs in the `login` Groovy method of the `Login` application-specific class.

Every web service has access to this data via the following method:

```
servlet.getUserData().getUserData("dataName")
servlet.getUserData().putUserData("dataName", "dataValue")
```

See [Authentication], page 28,

4.2.4 Programming Languages

At this point, *Kiss* supports the development of back-end REST services in the Java, Groovy, or Common Lisp languages. Groovy was added first because it was easy, worked with the IDE well, and did all that was needed. (See the document *GroovyOverview* included with *KISS*.) Java was added simply due to its natural integration with the rest of the system.

With *Kiss*, different web services can be written in different languages. You are not forced to use one or the other.

Common Lisp (ABCL) was added due to this author's love of that language. Unlike Groovy and Java, Lisp has an impedance mismatch with the core *Kiss* system that is written in Java. For example, in Java, one can have two methods with the same name in the same class that differ only by their argument types. This is not part of the Common Lisp language. Also, the foreign function interface in Lisp requires some Lisp code to make the connection clear. Due to this connection code having to run, the Lisp interface is very slow on the first call. It is, however, reasonably fast on all subsequent calls. The code that interfaces Java to Lisp is under the `src/main/core/org/kissweb/lisp` directory.

Due to the easy and natural connection between Java and many other JVM languages, interfaces to those languages are very easy. It is anticipated that support for many of these other languages (such as Scala, Clojure, JRuby, Jython, and Kotlin) will likely follow, especially if there are requests.

4.2.5 Cron

Kiss has the ability to run any number of commands at specified intervals. For example, you could run a process every hour and another process every Tuesday at 3 PM, etc. This facility echoes the standard Unix or Linux `cron` facility.

The file that determines what gets run and when is `src/main/backend/CronTasks/crontab`. All tasks must be in the `Groovy` language and exist in the `src/main/backend/CronTasks` directory.

All contents of the `src/main/backend/CronTasks` can be changed on a running system. The change will be noticed and take effect. There is no need to restart the system.

Look at the files in the `src/main/backend/CronTasks` directory for samples and further documentation.

4.2.6 CORS

For security reasons, web servers prevent web services from an application that doesn't come from the same origin. This is known as Cross-Origin Resource Sharing or CORS. *Kiss* fully supports this security standard.

Kiss automatically enables full CORS protection in production environments and allows CORS in the development environment.

4.2.7 Unit Tests

Kiss supports `jUnit` unit tests. The tests are located under the `src/test` directory. The directory hierarchy under the `src/test` should mirror that of the `src/main` directory.

Unit tests may be run as follows:

1. Add or change unit tests or code to be tested.
2. Build *KissUnitTest.jar* by executing:

   ```
   ./bld unit-tests
   ```

   This creates the `work/KissUnitTest.jar` file which contains the entire *Kiss* system and unit tests.

All tests can be run by executing the following command:

```
java -jar work/KissUnitTest.jar  execute \
  --class-path work/KissUnitTest.jar  --scan-class-path
```

Alternatively, a single test can be run as follows:

```
java -jar work/KissUnitTest.jar \
  execute \
  --class-path work/KissUnitTest.jar \
  --select-class org.kissweb.IniFileTest
```

where `org.kissweb.IniFileTest` is the full path to the desired test.

4.3 Front-end Development

A separate server is used so that development files will be served rather than the front-end that was present when the back-end server was started. This is the same whether you are using the IDE or *bld* back-end server modes.

Once the back-end and front-end servers are running, the front-end can be debugged through your browser debugger (F12 on Chrome).

In order to make this work, there are two steps that need to be followed.

1. The front-end needs to know where the back-end is located. This is controlled in the file `src/main/frontend/index.js`. That file contains a line that looks as follows:

   ```
   Server.setURL('http://localhost:8080');
   ```

   As shipped, that setting should be good in most cases. Adjust as needed.
2. A server needs to be running to serve the front-end code. The *Kiss* system comes with a simple server that performs this function. It is in a file named `SimpleWebServer.jar` on the root of the *Kiss* system. From the root of the *Kiss* system, run the following command to run the front-end server:

   ```
   ./serve
   ```

   or on Windows:

```
serve.cmd
```

(Source code to this simple server is available at https://github.com/blakemcbride/SimpleWebServer)

Once the front-end and back-end servers are up, you can access the development environment through your browser at `http://localhost:8000`

If you disable the browser cache through the browser developer console settings, changes you make to the front-end will appear by just reloading the page. On Chrome, for example, the browser cache can be turned off by going into the page debugger (F12), then `settings`, and then selecting `Disable cache (while DevTools is open)`

You can now develop the front-end portion of your application by editing files under the `src/main/frontend` directory.

4.3.1 Mobile Interface

It is often necessary to support mobile devices. Although many applications support mobile applications through pages that use responsive design, it is often the case that the pages are so different between various platforms that the use of whole new pages is simpler and more effective – especially for complex screens.

Although *Kiss* has always supported responsive design, *Kiss* also supports the ability to have different pages for different platforms. If you look at `src/main/frontend/index.js` you will see how `Kiss` handles this. Basically, `Kiss` detects the platform and loads different pages based on the platform.

4.4 Reports And Exports

Creating reports and exports requires both front-end and back-end components. The back-end usually creates the file to be sent to the front-end. It then returns a path that the front-end can use to download the file.

Kiss provides the infrastructure needed to support this facility. *Kiss* also manages and cleans up report files that are no longer needed.

In terms of producing reports, *Kiss* leverages the facilities provided by the common groff/tbl/mm utilities publicly available. These facilities automatically handle paging, titles, page numbering, tables, and overall formatting of your reports. See https://www.gnu.org/software/groff

In terms of producing CSV export files, see the back-end `DelimitedFileWriter` class.

When files are produced by the back-end, they are sent to the front-end by just providing the URL to the file. At that point, the front-end `Utils.showReport` takes the URL returned by the back-end and downloads the file.

4.5 Authentication

Kiss has built-in authentication. However, each application has its own method of storing and validating users. Additionally, each application may have its own user-specific data it

may want to retain between web service calls. *Kiss* has a generic and easy way of handling these needs.

Application-specific user login and data are handled by the Groovy file located in the `backend` directory named `Login.groovy`. That file must have two methods: `login` and `checkLogin`. See that file for more details.

By default, authentication requires a SQL database to be configured. If no SQL database is configured, the system bypasses authentication entirely. However, if your application uses a non-SQL data store for user management, you can enforce authentication by setting `RequireAuthentication = true` in `application.ini`. When enabled, the system calls `Login.groovy` for authentication regardless of whether a SQL database is configured. Note that the `db` parameter passed to the `login` and `checkLogin` methods will be `null` when no SQL database is configured, so your `Login.groovy` must handle this appropriately.

See [Cached User Data], page 25,

4.6 The Build System

Kiss does not use *Ant*, *Maven*, or *Gradle*. It includes its own build system called *bld*, which is designed to handle *Kiss*' microservice architecture where the core is compiled ahead of time but application code is compiled at runtime.

bld is made up of two Java files: `src/main/core/org/kissweb/BuildUtils.java` (framework code, do not modify) and `src/main/precompiled/Tasks.java` (application-specific build procedures). It can download and cache dependencies, and only performs work that is needed. An *Ant* bridge (`build.xml`) is included for IDE integration.

When upgrading *Kiss* (See [updates], page 17), `Tasks.java` is never overwritten, so you must manually reconcile any changes.

For a deeper discussion of the build system design and customization, see the *Kiss* book.

4.7 Using Artificial Intelligence

New AI/LLM systems have appeared (like *Claude Code*) that can develop whole applications. Fully embracing this new technology, *Kiss* now includes files to help facilitate this mode of development. In particular, *Kiss* includes two files as follows:

`AI/KnowledgeBase.md`
: This file explains the *Kiss* framework to an AI system.

`AI/ApplicationDetails.md`
: This is a template you can use to describe the application you are building.

In order to use these, follow the following steps:

1. Edit `ApplicationDetails.md` to describe the application you will be building.
2. Start up the AI system and tell it to read those two files.

From that point, you can instruct the AI to build your application.

You may run *Kiss* and test the application while the AI is developing your application. If any errors appear, feed them to the AI and it will fix them.

5 Front-end API

The front-end API is all the facilities that run on the browser. This includes HTML, CSS, JavaScript, image files, etc. The *Kiss* back-end does not produce or modify HTML or JavaScript code. These files are served, unaltered, by the server as they are on the back-end disk. The `Kiss` model is that the browser receives these files from the server, and that they include all the code that the browser needs to perform its function. Besides these static files, all data is communicated between the back-end and front-end via REST services.

Having all of the display logic running on the front-end or user's browser makes a lot of sense for the following reasons:

1. Minimize the dependence the front-end and back-end have on each other. This means that one end can be changed without necessitating the need for the other to change. In other words, they are minimally dependent on each other.
   a. In this rapidly changing environment, minimizing dependencies means minimizing the amount of code that has to be changed as technology changes.
   b. Code is easier to understand and maintain since you don't have four totally different languages in the same file.
2. Push as much processing to the client side so that the back-end can scale easier.

All of this leads to the following:

1. Shorter development time
2. Easier to maintain
3. Be most prepared for future changes
4. Reduces server costs
5. Reduced development time and cost

The front-end API is documented in the file `manual/jsdoc/index.html`.

5.1 Calling REST Services

On the front-end, the class `Server` is what deals with the REST communications between the front-end and back-end. In it, there are a handful of methods that deal with the environment, such as the back-end URL. All communications between the back-end and front-end are done with JSON.

The way it works is the login service requires a username and password, and it returns a login token (UUID). That token is used in all future calls, and it gets automatically invalidated after a certain amount of non-use time. There is no state kept on the back-end. Each REST call must login to the back-end with the provided token in order to be authenticated to communicate.

The method used to communicate is named `Server.call()`. It is passed the path to the REST service, the REST service method name, and a JSON object that is sent to the back-end method. A `Promise` is returned that is used to obtain the result of the call. This can also be used with `async/await`.

There is also a `logout` method that simply erases the login token so that future communications cannot occur.

The `Server` class also includes a method (`fileUploadSend`) that makes it easy to upload files.

5.2 Kiss Components

Although HTML provides what is needed for real applications, it provides those facilities at too low a level to be useful without a lot of custom code. Custom components (tags) allow you to encapsulate that advanced behavior into what is used as and appears as native functionality.

Kiss provides the ability to create your own custom HTML tags or elements as well as use those provided by *Kiss*. There are two principal user methods that make this work. `Utils.useComponent` is used to load either a *Kiss* defined component or one you define yourself (there is no difference). This loads the JavaScript file that defines the new tag. All of the components that come with the *Kiss* system are under the `src/main/frontend/kiss/component` directory. You can see those files for examples of how custom tags are defined.

New application pages are loaded with the `Utils.loadPage` method. In addition to loading the HTML and JavaScript code associated with that page, this method performs the processing necessary to make the components work. It does this intelligently so that, for example, one component can use another component without any special loading order requirement.

Briefly, the code that describes the custom tag must describe what the tag is replaced with. Ultimately, it must boil down to straight HTML, CSS, and JavaScript code.

5.2.1 Tagless Components

Let's say you have a pop-up window that allows a user to search for employees, products, or whatever. The user gets a variety of search capabilities and the selected item is returned. Let's further say that you need this functionality in several places within your application. These are tagless components. They aren't placed with a custom tag. They are a response to an event like a button push. Tagless components allow you to encapsulate a block of functionality (including pop-up windows) into a neat package that can be reused in any number of places.

The method used to load tagless components is `Utils.useTaglessComponent`. Later, when the tagless component is needed, one would execute `Kiss.MyComponent.run(in_data, on_exit)` (where `MyComponent` is the name of your tagless component). `in_data` represents possible data passed to the component on entry. `on_exit` is a function that gets executed when the component exits. Arguments passed to `on_exit` are determined by the component.

5.3 Modal Popup Windows

Kiss supports draggable, modal popup windows. *HTML* is used to describe the layout of the popup, and JavaScript is used to control the appearance of the popup.

The *HTML* portion is represented as a top-level `popup` tag that represents the entire popup. Within the top-level `popup` there must be two child tags named `popup-title` and `popup-body`. The first represents the single line header, and the second represents the body of the popup window.

The top-level `popup` tag's attribute section must contain the following:

`id="my-popup"`
: an ID is needed to reference the popup

`width="600px" height="300px"`
: Set the height and width of the popup

An example is as follows:

```
<popup id="my-popup" width="600px" height="300px">
    <popup-title>The title</popup-title>
    <popup-body>
        The content
        <div style="display: inline-block; position: absolute; bottom: 20px; right: 20
            <push-button id="cancel" style="margin-left: 15px;">Cancel</push-button>
            <push-button id="ok" style="margin-left: 15px;">Ok</push-button>
        </div>
    </popup-body>
</popup>
```

There are only two JavaScript functions used to control the popup.

`Utils.popup_open(id [, focus-id])`
: open the popup indicated by the `id`, and if `focus-id` is present, set initial focus to that control

`Utils.popup_close()`
: close the most recent popup

Your defined responses to the buttons on the popup determine your dealings with the data on the popup and when to close it.

5.4 File Uploads

Kiss includes facilities that make it easy to upload files. The HTML uses a `<file-upload>` control, and the `Server.fileUploadSend()` method handles the transfer to the back-end. The `file-upload` control provides helper methods such as `numberOfUploadFiles()` and `getFormData()`.

On the back-end, the servlet provides methods to access uploaded files: `getUploadFileCount()`, `getUploadFileName()`, `getUploadBufferedInputStream()`, and `saveUploadFile()`.

See the included example application and the *Kiss* book for complete front-end and back-end file upload examples.

5.5 Utilities

Kiss includes an ever-growing set of utilities to help deal with common tasks. These utilities are located under the `src/main/frontend/kiss` directory and have names such as `DateTimeUtils.js`, `DateUtils.js`, `TimeUtils.js`, `Utils.js`, etc. These utilities are documented in the front-end API documentation.

5.6 Controlling Browser Cache

Browsers have a mind of their own in terms of deciding when to use their cache for a file and when to download a new one. This can cause no end of trouble when code gets changed. Some user files end up being old from the browser cache, and others are freshly downloaded. The old and new files don't agree with each other and all sorts of errors occur.

Kiss includes a facility to assure that all files are downloaded afresh whenever the application changes while still taking maximal advantage of the browser cache when the files have not changed. The only cost for this capability is the requirement that the `index.html` file *always* gets loaded afresh. To that end, *Kiss* has code to ignore browser cache and always load `index.html` afresh.

`index.html` contains two variables named `SystemInfo.softwareVersion` and `SystemInfo.controlCache`. Assuming `SystemInfo.controlCache` is `true`, *Kiss* has code that forces the browser to reload all files whenever `SystemInfo.softwareVersion` changes. After the code is re-loaded, the browser cache will work as normal to maximally cache the files until the next `SystemInfo.softwareVersion` change.

5.7 Additional Resources

Although not a part of the *Kiss* system, there are some very valuable technologies and libraries that have been used with *Kiss* in order to create some very powerful solutions.

The first is the Lovefield library, which adds SQL capabilities on the browser side. Data is persisted on the user's browser and remains through browser or machine reboots. The library is located at https://github.com/google/lovefield

A recent technology that has been used to enable browser applications that run when there is no Internet connection is called *Service Workers*. There is a package at https://developers.google.com/web/tools/workbox that makes working with service workers very easy.

For in-depth coverage of the front-end component system, grid integration, popup windows, and utility classes, see the *Kiss* book.

6 Back-end API

In addition to the full API provided by the Java system and any additional JAR files you add, *Kiss* comes with an API that assists with the development of business application with *Kiss*. These API's may be broadly grouped as follows:

1. Database API
2. JSON API
3. Utilities

An overview of these APIs is contained in this chapter. Detailed documentation is contained in the JavaDocs (See [Creating JavaDocs], page 22) and in the *Kiss* book, which provides comprehensive coverage with worked examples.

6.1 Database API

Kiss comes with a powerful library for accessing SQL databases. Code for this is located under `org.kissweb.database` It is currently being used in production environments. This API provides the following benefits:

- Automatic connection and statement pooling
- Vastly simpler API than bare JDBC
- Handling of parameterized arguments
- Auto generation of SQL for single record adds, edits, and deletions
- Auto handling for cases of cursor interference on nested queries
- Supports transactions out-of-the-box

As shipped, this library supports PostgreSQL, Microsoft Server, Oracle, MySQL, and SQLite.

The detailed documentation for the database utilities is in the JavaDocs which you must generate. (See [Creating JavaDocs], page 22.) This section provides an overview.

The *Kiss* database routines revolve around four main classes as follows:

`Connection`
: This represents a connection to an SQL database.

`Command`
: This represents a single action or command against the database.

`Cursor`
: If the action is a `select`, the `Cursor` represents a pointer into the result set.

`Record`
: This class represents a single row within a table or result set.

The `Connection` class contains several convenience methods that are used in simple cases where only a single action is being performed. These methods should not be used when multiple simultaneous actions are taking place at once (by that single thread). This issue is not a problem in multi-user or multi-threaded situations. It is only a problem when a single thread is doing one action while another action is still open.

You will notice that your REST services are passed a *Connection* argument. *Kiss* automatically forms a unique connection for each REST service call and closes it when the call is done. Therefore, you will not normally need to create your own connection. Additionally, *Kiss* automatically starts a new transaction with each REST service and commits it when the service is done. However, if the service throws any exception, the transaction is rolled back instead.

You would not normally write SQL for single record adds, updates, and deletes. Using the *Record* API, `Kiss` automatically generates these statements for you.

In addition to the above, these utilities provide full support for transactions and parameters.

6.2 Microservices

Microservices are classes that may be added, changed, or deleted while the system is running. In spite of this, however, all microservices are fully compiled and run at full speed. This has two major advantages.

First, in a development environment, all development may be done without the need to bring the development server down, rebuild, re-deploy, and reboot the development server. This means development time is significantly reduced. Additionally, IDE debuggers function in an environment such as this, so the debugging process may proceed normally.

Second, in a production environment, the system may be upgraded without disrupting existing users on the system. Of course, users actively using the exact features you have just changed could be affected (if their front-end and back-end do not agree). A simple re-try would put everything back in sync.

Kiss microservices are on a class basis. What that means is that a microservice is always a single and whole class. You cannot have more than one class in a microservice. Microservices can call core components of the system just as regular methods can. However, if one wishes to have one microservice call another microservice, the calling mechanism is a little more clunky. However, using this clunkier mechanism retains all of the dynamic features of the system.

Defining microservices in Kiss involves defining a normal class just as you would write a class in any circumstance. No special configuration, wiring, declarations, or additional steps are required. Microservice additions, changes, and deletions take effect as soon as you save the source file. All microservices are compiled at runtime by the system automatically, so there is no compilation step that you need to perform.

The only caveat to the above is that remote microservices (described below) expect a certain method signature (standard arguments). This is only to assure that the front-end and back-end can communicate as expected. Local microservices do not have this requirement.

6.2.1 Microservice Language

In Kiss, microservices can be written in Java, Groovy, or Common Lisp. However, Groovy microservices have been used exclusively in all current environments that we are aware of. Therefore, Groovy microservices are best tested.

The reasons Groovy was used are as follows:

1. Groovy was the easiest and most natural to implement.
2. Groovy runs as fast as Java and has full and natural access to all Java facilities.
3. The actual loading of Groovy services is the fastest of the three languages.
4. Groovy is largely a super-set of Java so if you know Java, you basically know Groovy. Learning of additional Groovy facilities can occur over time.
5. Groovy offers a small number of conveniences over Java.

Having said all this, however, all three languages are fully supported and there are no known bugs in any of the supported languages.

6.2.2 Types of Microservices

There are two types of microservices as follows:

1. Remote (REST) Services
2. Local Services

Remote services are generally called by a (likely JavaScript) front-end or a remote client typically over *HTTP* or *HTTPS*. Local services are services that are called locally, from within the system as, for example, one microservice calling a method in a different microservice.

6.2.3 Remote Microservices

In *Kiss*, remote microservices appear to the outside world as typical *REST* services. They may be called by a *JavaScript* front-end, web service client, or any other facility that can talk to *REST* services.

Kiss REST services are asynchronous *HTTP* or *HTTPS* services. Internally they are processed utilizing a thread pool to assure optimal CPU utilization. The number of threads in the thread pool is controlled by a configuration parameter given in the `application.ini` file.

Kiss comes with *JavaScript* code so that `Kiss REST` services can be easily accessed from a typical front-end. This code resides in a single file and can and has been used by alternative front-ends such as *Angular* and `React`.

There is no need to handle authentication. *Kiss* handles that automatically. So, when your web method is called, you know who they are and that they have been authenticated.

6.2.4 Defining Remote Services

Remote REST microservices typically reside under the `backend/services` directory. You can organize them any way you like. A microservice is equivalent to a class. The class

name is the microservice name. The methods in that class that have a certain signature (a particular set of arguments) are the web methods.

A REST web microservice has the following signature:

```
void myMethod(JSONObject injson,
              JSONObject outjson,
              Connection db,
              ProcessServlet servlet) {
...
}
```

'`injson`' This is a `JSON` object that contains all of the arguments passed in from the front-end.

'`outjson`' This is a `JSON` object that will contain all of the results sent back to the front-end. Whatever is put in this object gets sent back to the front-end.

'`db`' This is a database connection that can be used to access the SQL database. The connection is unique and independent of all other services.

'`servlet`' This object provides access to various system facilities uniquely related to this call.

See [JSON], page 39, and the `JavaDoc` for additional information.

6.2.5 Local Microservices

Local microservices are simply regular classes. They typically reside in the `backend/services` directory organized any way you like. Although methods within a particular microservice/class can call each other in the normal way, there is an extra step required for one microservice to call a method in a different microservice. One of the reasons for this is so the system can be certain the latest version of the service is loaded and that it is fully compiled before you attempt to use it.

In Groovy, a method in one class/microservice can call a method in a different class/microservice via the following methods.

- GroovyService.run
- GroovyService.getMethod

See the *Javadoc*.

6.2.6 Exceptions

Kiss defines three exception classes named `UserException`, `LogException` and `ServerException`. These exceptions are designed to be thrown from within your web services.

`UserException` is used to abort the web service due to some sort of user error. An error message and possible error code are returned to the front-end. The front-end displays the error message in a popup and tells the web front-end that the web service failed by returning `_Success = false`. Nothing is logged on the back-end. This was not an error in the back-end. It is good for notices to the user that something was wrong with their data.

`LogException` is like `UserException` except that it additionally leaves the message on the back-end log. This is useful for debugging purposes.

`ServerException` is like `UserException` except that it additionally leaves a message and stack trace on the back-end log. This is useful for debugging purposes.

6.3 JSON

The first two arguments to all REST methods are `injson` and `outjson`. `injson` is a `JSONObject` that contains the data passed in *from* the front-end. `outjson` is a pre-initialized, empty `JSONObject` that will be *returned to* the front-end. The REST service should read the data passed in from `injson`, perform any needed processes, and put the result into `outjson` to be returned to the front-end.

The two main data types are `JSONObject` and `JSONArray`. Common methods include `getString()`, `getInt()`, `getBoolean()`, `getDouble()`, `getJSONArray()`, `has()`, and `put()`. See the JavaDocs for the complete API.

6.4 Large Language Models

Kiss provides a high-level interface to the *Ollama* large language model (LLM), the *OpenAI* (chatGPT) server, and the *Anthropic* (Claude) server. This makes it easy for you to add LLM features to your application.

6.4.1 Ollama

Before this can function, however, you must have access to an *Ollama* server, either running on your local machine or a remote machine.

- See https://ollama.com for information about installing the *Ollama* server.
- The *Kiss* class `org.kissweb.llm.Ollama` implements the interface.
- *Kiss* comes with a complete demo of this functionality.

6.4.2 OpenAI (chatGPT)

Unlike the *Ollama* server, which runs locally, the *chatGPT* server runs on the cloud-based *OpenAI* server. In order to use this server, you must have an account with them. Once you get an account, you will need an API Key to interface with their server. Once you have this API Key, you may query *chatGPT*.

The single *KISS* class that interfaces with *OpenAI* is named *OpenAI*.

Typical usage would look as follows:

```
String apiKey = "xxxxxxxxxxxxxx";
OpenAI chatGPT = new OpenAI(apiKey, "gpt-3.5-turbo")
String result = chatGPT.send("Who is president Bush?")
```

The static method `setUrl(String url)` can be used to override the default OpenAI endpoint URL. This is useful for proxies, gateways, or OpenAI-compatible endpoints. Since it is a global setting, changing the URL affects all instances.

See that class for documentation.

6.4.3 Anthropic (Claude)

Like *OpenAI*, the *Anthropic* server runs in the cloud. In order to use it, you must have an account with Anthropic. Once you get an account, you will need an API Key to interface with their server. Once you have this API Key, you may query *Claude*.

The single *KISS* class that interfaces with *Anthropic* is named *Anthropic*.

Typical usage would look as follows:

```
String apiKey = "xxxxxxxxxxxxxx";
Anthropic claude = new Anthropic(apiKey, "claude-sonnet-4-20250514");
String result = claude.send("Who is president Bush?");
```

The class also supports streaming responses and image inputs.

The static method `setUrl(String url)` can be used to override the default Anthropic endpoint URL. This is useful for proxies, gateways, or Anthropic-compatible endpoints. The static method `setVersion(String version)` can be used to override the `anthropic-version` request header value (e.g., `2023-06-01`). Both are global settings that affect all instances.

See that class for documentation.

6.5 Utilities

Kiss includes an ever-growing set of utilities to help deal with common tasks. These utilities are located under the `src/main/core/org/kissweb` directory and have names such as `DateTime.java`, `NumberFormat`, etc. These utilities are documented in the JavaDocs.

6.6 Global Logout Handler

Kiss provides a mechanism to execute custom code whenever a user logs out (manually or due to inactivity timeout). The handler is configured in `KissInit.groovy` using `UserCache.setLogoutHandler()`, which accepts a `Consumer<UserData>`.

A basic example:

```
UserCache.setLogoutHandler({ UserData ud ->
    println "User ${ud.getUsername()} is logging out"
} as Consumer<UserData>)
```

The `UserData` object provides methods such as `getUsername()`, `getUserId()`, `getUuid()`, and `getUserData(String key)`. If you need database access in the handler, open a new connection with `MainServlet.openNewConnection()` and close it in a `finally` block.

To log out from the front-end, call `Server.logout()`.

For detailed examples and usage patterns, see the *Kiss* book and the JavaDocs.

7 SQL Query Generator

In systems with many interrelated SQL tables, writing queries with the correct joins is complex and error-prone. Developers must manually trace foreign key relationships, determine join paths, and construct multi-table queries — work that the schema itself already encodes.

Kiss provides an automatic SQL query generator that solves this problem. The developer specifies what columns to select, what conditions to apply, and how to order the results. The system determines the join path automatically and produces the correct SQL.

All classes are in the `org.kissweb.database` package:

`SchemaGraph`
: Models the database schema as a graph where tables are nodes and foreign key relationships are edges. Finds the shortest join path between tables using BFS (breadth-first search).

`QueryBuilder`
: Fluent API to specify SELECT, WHERE, ORDER BY, GROUP BY, HAVING, and explicit joins. Uses a `SchemaGraph` to automatically determine the required joins and produce the final SQL.

`SchemaGraph.Edge`
: Represents a single or composite foreign key relationship between two tables.

7.1 Quick Start

7.1.1 Building the Schema Graph

The schema graph must be built once at application startup. There are several ways to create it.

7.1.1.1 From Live Database Metadata

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

This reads JDBC `DatabaseMetaData` for all tables and their foreign keys. Composite (multi-column) foreign keys are automatically grouped by constraint name. Works on all five supported databases (PostgreSQL, MySQL, SQLite, SQL Server, Oracle).

7.1.1.2 Programmatic Declaration

```
SchemaGraph graph = new SchemaGraph();
graph.addForeignKey("employee", "department_id",
                    "department", "department_id");
graph.addForeignKey("order_line",
    new String[]{"order_id", "product_id"},
    "order_product",
    new String[]{"order_id", "product_id"});
```

7.1.1.3 Hybrid

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

```
graph.addForeignKey("audit_log", "user_id",
                    "employee", "employee_id");
```

7.1.1.4 Cached per Schema (Thread-Safe)

```
// First call builds from the database and caches
SchemaGraph graph = SchemaGraph.fromDatabase(conn, "mySchema");

// Subsequent calls return the cached instance immediately
SchemaGraph graph = SchemaGraph.fromDatabase(conn, "mySchema");

// Clear cache if schema changes
SchemaGraph.clearSchemaCache("mySchema");  // one entry
SchemaGraph.clearSchemaCache();            // all entries
```

7.1.1.5 Cached (for Fast Startup)

```
// First run: read from database, save cache
SchemaGraph graph = SchemaGraph.fromDatabase(conn);
graph.saveToFile("schema-cache.txt");

// Subsequent runs: load from cache (sub-millisecond)
SchemaGraph graph = SchemaGraph.loadFromFile("schema-cache.txt");
```

7.1.2 Building and Executing a Query

The simplest approach is to create a `QueryBuilder` directly from a `Connection`. The connection's schema graph is used automatically, and the no-argument `fetchAll()`, `fetchOne()`, and `fetchAllJSON()` methods execute against that connection:

```
List<Record> records = conn.newQueryBuilder()
    .select("employee.first_name", "employee.last_name")
    .select("department.name AS dept_name")
    .select("building.address")
    .where("project.project_id = ?", projectId)
    .where("employee.active = 'Y'")
    .orderBy("employee.last_name")
    .orderBy("employee.first_name")
    .fetchAll();

for (Record r : records) {
    String name = r.getString("first_name");
    String dept = r.getString("dept_name");
}
```

Alternatively, a `QueryBuilder` can be created with an explicit `SchemaGraph` and a connection passed to the execute methods:

```
List<Record> records = new QueryBuilder(graph)
    .select("employee.first_name", "employee.last_name")
    .select("department.name AS dept_name")
```

```
        .select("building.address")
        .where("project.project_id = ?", projectId)
        .where("employee.active = 'Y'")
        .orderBy("employee.last_name")
        .orderBy("employee.first_name")
        .fetchAll(conn);

    for (Record r : records) {
        String name = r.getString("first_name");
        String dept = r.getString("dept_name");
    }
```

The system automatically generates SQL such as:

```
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, employee.first_name
```

7.2 Further Information

The `QueryBuilder` API includes many additional features not covered in this overview, including:

- OR / AND condition grouping (`startOr()`, `endOr()`, `startAnd()`, `endAnd()`)
- Subqueries in WHERE (`whereIn()`, `whereNotIn()`, `whereExists()`, `whereNotExists()`)
- Explicit joins for self-joins and controlling join paths (`join()`, `leftJoin()`, `rightJoin()`)
- Aggregate helpers (`selectCount()`, `selectSum()`, `selectAvg()`, `selectMin()`, `selectMax()`)
- GROUP BY / HAVING
- Common Table Expressions (CTEs) via `with()`
- UNION / UNION ALL
- LIMIT
- DISTINCT

For the complete API reference, detailed examples, and edge case documentation, see the *Kiss* book and the JavaDocs. (See [Creating JavaDocs], page 22.)

7.3 Source Files

`src/main/core/org/kissweb/database/SchemaGraph.java`
: Schema graph model: table/edge management, BFS path finding, schema caching.

`src/main/core/org/kissweb/database/QueryBuilder.java`
: Fluent query builder: SELECT, WHERE, ORDER BY, GROUP BY, HAVING, explicit joins, aggregate helpers, build, and execute methods.

`src/test/core/org/kissweb/database/QueryBuilderTest.java`
: 108 unit tests covering all functionality.

8 Command Line Utility

In addition to utilizing *Kiss* as a web application development system, *Kiss* also provides a command line interface. This interface allows you to build quick but powerful utilities to do things like updating a database, parsing a CSV file, interfacing with a third-party REST service, and more. Basically, all of *Kiss* is available except the *Kiss* REST server.

Although this system supports *PostgreSQL* out-of-the-box, it also supports any of the other databases with a slightly more complex command line.

8.1 Building The Utility

The build system (*bld*) has the ability to build a *JAR* file named *KissGP.jar*. The *GP* stands for *Groovy* and *PostgreSQL*. Basically, it is a *JAR* file capable of running *Groovy* scripts in the context of all of the *Kiss* utilities including access to a *PostgreSQL* database.

(It should be noted that although *Groovy* scripts are text / source files, they nevertheless run at full compiled speed because they are compiled at runtime.)

To build the *JAR* file, type the following:

```
./bld KissGP
```

A file named *KissGP.jar* will be created in the *work* directory. That, in addition to the *JDK*, is all that is needed to use the system.

8.2 Using The System

To use the *Kiss* command line interface you first create the *Groovy* program you would like to run. It may use all of the *Groovy* and *Kiss* APIs. For example, let's start with something simple. Create the following file named *test1.groovy*:

```
static void main(String [] args) {
    println "Hello world!"
}
```

You can then run the program as follows:

```
java -jar KissGP.jar test1
```

test1.groovy can be extended arbitrarily to perform any function needed.

8.3 Databases Other Than PostgreSQL

Although *KissGP.jar* comes bundled with support for the *PostgreSQL* database, *KissGP.jar* can support databases other than *PostgreSQL* by adding the driver for the database and using a slightly more complex command line as follows.

To use *KissGP.jar* with Microsoft SQL Server, for example, you'd have to include the database driver for it in addition to the *KissGP.jar* file and the file with your program. The command line would look as follows:

```
java -cp mssql-jdbc-8.2.0.jre8.jar -jar KissGP.jar test1
```

The same is true of the other databases. Please note that the drivers you'll need are already in the *libs* directory.

9 Split System

Kiss comes as a complete system that includes both the front-end and back-end. This works well for most situations. However, sometimes a web application may have a single back-end that serves several different front-ends. Or, perhaps you prefer to keep the front-end and back-end projects separate. *Kiss* has a mechanism to support the ability to split the front-end and back-end.

This chapter will detail the steps needed to accomplish this.

9.1 Back-end-only System

A back-end-only system is a complete system minus the front-end portion of the system. The git repo is also deleted since it is meaningless in this scenario.

9.1.1 Creating

In order to create a back-end-only system you start with a complete *Kiss* system cleanly checked out and delete the front-end portion. That leaves you with a back-end-only system. However, before doing that, you must install the third-party libraries. The following commands will accomplish this whole process.

```
./bld libs                    (Linux or macOS)
./remove-frontend remove
    or
bld libs                      (Windows)
remove-frontend remove
```

Once these commands have been run, what you have left is a back-end-only system. The git repository it came in will have been deleted as well as the front-end portion of the system. The procedure for developing, deploying, and upgrading the system will be slightly modified as described herein.

9.1.2 Developing

The main difference between developing on a back-end-only system vs. a whole system is just that the command used to start the back-end is slightly different. The locations of all the back-end pieces are the same. Once running, the system may be modified while running just the same.

Use the following command to build and run the back-end:

```
./bld develop-backend         (Linux or macOS)
     or
bld develop-backend           (Windows)
```

Viewing the server log is done in the same way as before.

9.1.3 Deploying

Deploying the back-end is done just as before except now, instead of a single file containing the whole system, you have a single file that represents the back-end only. When deploying

the front-end and back-end portions, they will be treated as two separate systems by the server.

9.1.4 Upgrading

Upgrading a back-end-only project is done the same way as upgrading a full system except that after the upgrade, you'll need to run the `remove-frontend` script again. See [System Updates], page 17,

9.2 Front-end-only System

A front-end-only system is a complete system minus the back-end portion of the system. The git repo is also deleted since it is meaningless in this scenario.

9.2.1 Creating

In order to create a front-end-only system you start with a complete *Kiss* system cleanly checked out and delete the back-end portion. That leaves you with a front-end-only system. However, before doing that, you must install the third-party libraries. The following commands will accomplish this whole process.

```
./bld libs                    (Linux or macOS)
./remove-backend remove
    or
bld libs                      (Windows)
remove-backend remove
```

Once these command have been run, what you have left is a front-end-only system. The git repository it came in will have been deleted as well as the back-end portion of the system. The procedure for developing, deploying, and upgrading the system will be slightly modified as described herein.

9.2.2 Developing

Before running the front-end, you should start the back-end as described above.

There are no steps required to edit front-end files. However, you must serve them. This can be accomplished with the following command:

```
./serve          (Linux or macOS)
    or
serve            (Windows)
```

Presuming the back-end is running, you can access the running system at `http://localhost:8000`

As before, you can edit front-end files while the system is running.

9.2.3 Deploying

The front-end must be distributed as a separate *war* file named *frontend.war*. This file can be created with the following command:

```
./make-frontend          (Linux or macOS)
```

```
      or
make-frontend            (Windows)
```

From your server's perspective, the front-end and back-end are two different systems.

9.2.4 Upgrading

Upgrading a front-end-only project is done the same way as upgrading a full system except that after the upgrade, you'll need to run the `remove-backend` script again. See [System Updates], page 17,

10 Browser-Based Distribution

As an alternative to packaging a *Kiss* application as a desktop application with Electron, the application can be distributed as a standalone, browser-based package. This approach uses the system's default web browser as the user interface and requires no additional frameworks beyond Java.

The resulting distribution is a single zip file that works on Linux, macOS, and Windows — a single package for all platforms.

10.1 Prerequisites

The target system must have the following installed:

- Java 17 or later

No other dependencies are required. The distribution includes its own embedded Tomcat server.

10.2 Building the Distribution

10.2.1 Step 1: Build the Application

```
./bld build              [Linux / macOS]
    or
bld build                [Windows]
```

This compiles the application and deploys it to the embedded Tomcat server.

10.2.2 Step 2: Create the Distribution

Run the distribution build script from the project root, passing the desired distribution name as an argument:

```
./build-dist MyApp
```

This creates `work/MyApp.zip` ready for distribution to end users. The name argument determines both the zip filename and the directory name inside the zip.

10.3 What the Distribution Contains

`start.sh` Launcher script for Linux and macOS. Checks for Java, starts the server, waits for it to be ready, and opens the default web browser.

`stop.sh` Shutdown script for Linux and macOS.

`start.bat` Launcher script for Windows.

`stop.bat` Shutdown script for Windows.

`tomcat/` The embedded Tomcat server with the application pre-deployed in `tomcat/webapps/ROOT/`.

10.4 End User Instructions

1. Install Java 17 or later if not already installed.
2. Unzip the distribution zip file.
3. Run the appropriate start script:

```
./start.sh              [Linux / macOS]
    or
start.bat               [Windows]
```

4. The default web browser will open to `http://localhost:8080`.
5. To shut down the application, run:

```
./stop.sh               [Linux / macOS]
    or
stop.bat                [Windows]
```

10.5 Comparison with Desktop Application

Feature	**Browser Distribution**	**Electron Desktop**
Cross-platform from single package	Yes	No (build per OS)
Dedicated application window	No (browser tab)	Yes
Custom application menus	No	Yes
Distribution size	~30 MB	~150 MB+ per OS
Requires Java	Yes	Yes
Requires Node.js / npm	No	Yes

11 Desktop Applications

Portable desktop applications can be built utilizing *Kiss* in conjunction with Electron (https://electronjs.org). These applications would be portable to Linux, macOS, Windows, BSD, and other platforms with a single codebase.

Both the front-end and back-end are built exactly as a browser-based application would be built. The only difference is how the system is run.

11.1 Setup

Before a desktop application can be built, the *electron* system must be installed. This, in turn, is dependent on *nodejs* and *npm*.

11.1.1 Installing Nodejs & Npm

Full instructions to install *nodejs* and *npm* can be found at https://nodejs.org

Linux systems that use the *apt* package manager can install *nodejs* and *npm* with the following command:

```
sudo apt install nodejs npm
```

Linux systems that use the *dnf* package manager can install *nodejs* and *npm* with the following command:

```
sudo dnf install nodejs
```

Other systems can install *nodejs* and *npm* as instructed at https://nodejs.org

11.1.2 Installing Electron

You can install *electron* on any OS from within the *KISS* application with the following command:

```
npm install electron --save-dev
```

11.2 Building & Running

The *KISS* application is built no differently than if it were a browser application. You can use the following command:

You can build the system with the following command:

```
./bld build              [all OS except Windows]
    or
bld build                [Windows]
```

The system can be run as an application by executing the following command:

```
npm start
```

11.3 Debug Mode

If you are developing and need the front-end developer tools, simply edit the file named *start-electron.js* Change the value of the variable named *devMode* from *false* to *true*. Then just start normally.

When you do this, you will get the beloved developer tools window along with the application.