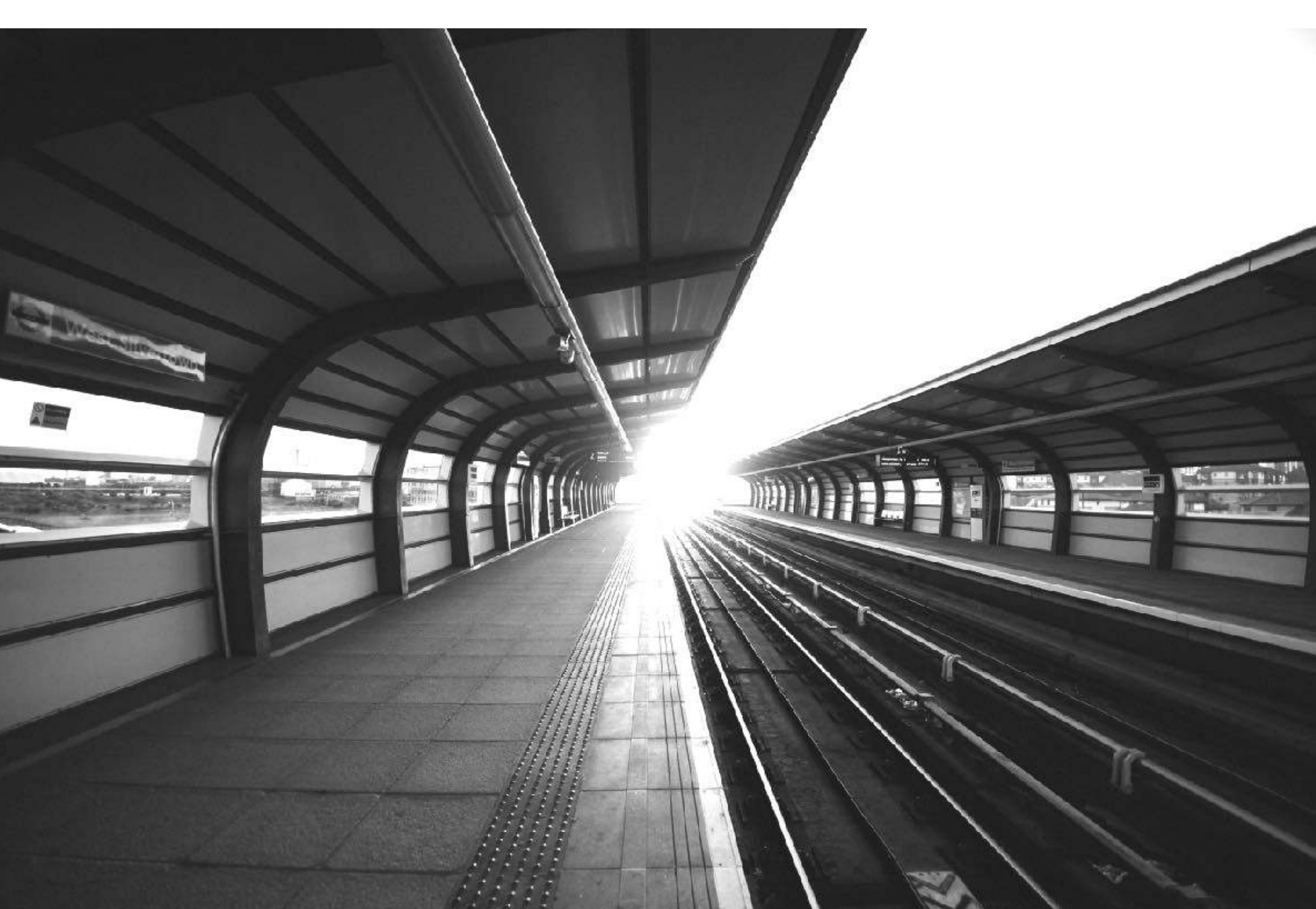
A gentle **guide** to **asynchronous programming** with **Eclipse Vert.x** for **Java developers**

By Julien Ponge, Thomas Segismont & Julien Viet



A gentle guide to asynchronous programming with Eclipse Vert.x for Java developers

Julien Ponge, Thomas Segismont, Julien Viet

Version 1.1.0, 2017-10-19

Table of Contents

1. Introduction 2
  1.1. About this guide 2
  1.2. What is Vert.x? 2
  1.3. Core Vert.x concepts 3
    1.3.1. Threading and programming models 3
    1.3.2. Event bus 5
2. A minimally viable wiki written with Vert.x 7
  2.1. Bootstrapping a Maven project 7
  2.2. Adding the required dependencies 8
  2.3. Anatomy of a verticle 9
  2.4. A word on Vert.x future objects and callbacks 10
  2.5. Wiki verticle initialization phases 11
    2.5.1. Database initialization 12
    2.5.2. Notes about logging 14
    2.5.3. HTTP server initialization 15
  2.6. HTTP router handlers 16
    2.6.1. Index page handler 17
    2.6.2. Wiki page rendering handler 20
    2.6.3. Page creation handler 24
    2.6.4. Page saving handler 25
    2.6.5. Page deletion handler 26
  2.7. Running the application 27
3. Refactoring into independent and reusable verticles 28
  3.1. Architecture and technical choices 28
  3.2. The HTTP server verticle 29
  3.3. The database verticle 32
    3.3.1. Configurable SQL queries 32
    3.3.2. Dispatching requests 35
    3.3.3. Reducing the JDBC client boilerplate 36
  3.4. Deploying the verticles from a main verticle 39
4. Refactoring to Vert.x services 42
  4.1. Maven configuration changes 42
  4.2. Database service interface 44
  4.3. Database service implementation 45
  4.4. Exposing the database service from the database verticle 47
  4.5. Obtaining a database service proxy 49
5. Testing Vert.x code 52
  5.1. Getting started 52

5.2. Testing database operations . . . 53
6. Integrating with a 3rd-party web service . . . 56
6.1. Scenario: backing up to GitHub Gist . . . 56
6.2. Updating the database service . . . 59
6.3. The web client API . . . 60
6.4. Creating anonymous Gists . . . 61
7. Exposing a web API . . . 64
7.1. Web sub-routers . . . 65
7.2. Handlers . . . 66
7.2.1. Root resource . . . 66
7.2.2. Getting a page . . . 67
7.2.3. Creating a page . . . 67
7.2.4. Updating a page . . . 68
7.2.5. Deleting a page . . . 69
7.3. Unit testing the API . . . 69
8. Securing and controlling access . . . 73
8.1. HTTPS support in Vert.x . . . 73
8.2. Access control and authentication . . . 74
8.2.1. Adding Apache Shiro authentication to routes . . . 75
8.2.2. Supporting features based on roles . . . 78
8.3. Authenticating web API requests with JWT . . . 83
8.3.1. Adding JWT support . . . 83
8.3.2. Using JWT tokens . . . 86
8.3.3. Adapting the API test fixture . . . 88
9. Reactive programming with RxJava . . . 90
9.1. Enabling the RxJava APIs . . . 90
9.2. Deploying verticles in order . . . 90
9.3. Partially *"Rxifying"* `HttpServerVerticle` . . . 91
9.3.1. Import RxJava versions of Vert.x classes . . . 91
9.3.2. Use delegate on a *"Rxified"* `vertx` instance . . . 92
9.4. Executing authorization queries concurrently . . . 92
9.5. Working with database connections . . . 93
9.6. Bridging the gap between callbacks and RxJava . . . 93
9.7. Data flows . . . 94
10. Client-side web application using AngularJS . . . 95
10.1. Single page application . . . 95
10.2. Vert.x Backend . . . 96
10.2.1. Simplifying the HTTP verticle code . . . 96
10.2.2. Exposed routes . . . 97
10.3. AngularJS frontend . . . 98
10.3.1. Application view . . . 98

10.3.2. Application controller 101
11. Real-time web features using cross-border messaging over the event bus 105
11.1. Setting up the SockJS event bus bridge 105
11.1.1. On the server 105
11.1.2. On the client 106
11.2. Sending the Markdown content over the event bus for processing 106
11.3. Warning the user when the page is modified 107
12. Conclusion 110
12.1. Summary 110
12.2. Going further 111
12.3. That’s all folks! 111

You can read the latest published version of the guide at http://vertx.io/docs/guide-for-java-devs/

Acknowledgements

This document has received contributions from Arnaud Esteve and Marc Paquette.

Chapter 1. Introduction

This guide is a gentle introduction to asynchronous programming with Vert.x, primarily aimed at developers familiar with mainstream non-asynchronous web development frameworks and libraries (e.g., Java EE, Spring).

1.1. About this guide

We assume that the reader is familiar with the Java programming language and its ecosystem.

We will start from a wiki web application backed by a relational database and server-side rendering of pages; then we will evolve the application through several steps until it becomes a modern single-page application with "real-time". [1: Note that the widespread usage of the term "real-time" in the context of web technologies shall not be confused with *hard* or *soft* real-time in specialized operating systems.] web features. Along the way you will learn to:

1. Design a web application with server-side rendering of pages through templates, and using a relational database for persisting data.
2. Cleanly isolate each technical component as a reusable event processing unit called a *verticle*.
3. Extract Vert.x services for facilitating the design of verticles that communicate with each other seamlessly both within the same JVM process or among distributed nodes in a cluster.
4. Testing code with asynchronous operations.
5. Integrating with third-party services exposing a HTTP/JSON web API.
6. Exposing a HTTP/JSON web API.
7. Securing and controlling access using HTTPS, user authentication for web browser sessions and JWT tokens for third-party client applications.
8. Refactoring some code to use reactive programming with the popular RxJava library and its Vert.x integration.
9. Client-side programming of a single-page application with AngularJS.
10. Real-time web programming using the unified Vert.x event bus integration over SockJS.

The source of both this document and the code examples are available from https://github.com/vert-x3/vertx-guide-for-java-devs. We welcome issue reports, feedback and pull-requests!

1.2. What is Vert.x?

> Eclipse Vert.x is a toolkit for building reactive applications on the JVM.
>
> — Vert.x website

Eclipse Vert.x (which we will just call *Vert.x* in the remainder of this document) is an opensource project at the Eclipse Foundation. Vert.x was initiated in 2012 by Tim Fox.

Vert.x is not a framework but a toolkit: the core library defines the fundamental APIs for writing asynchronous networked applications, and then you can pick the useful modules for your application (e.g., database connection, monitoring, authentication, logging, service discovery, clustering support, etc). Vert.x is based on the Netty project, a high-performance asynchronous networking library for the JVM. Vert.x will let you access the Netty internals *if need be*, but in general you will better benefit from the higher-level APIs that Vert.x provides while not sacrificing performance compared to *raw* Netty.

Vert.x does not impose any packaging or build environment. Since Vert.x core itself is just a regular Jar library it can be embedded inside applications packaged as a set of Jars, a single Jar with all dependencies, or it can even be deployed inside popular component and application containers.

Because Vert.x was designed for asynchronous communications it can deal with more concurrent network connections with less threads than synchronous APIs such as Java servlets or `java.net` socket classes. Vert.x is useful for a large range of applications: high volume message / event processing, micro-services, API gateways, HTTP APIs for mobile applications, etc. Vert.x and its ecosystem provide all sorts of technical tools for building end-to-end reactive applications.

While it may sound like Vert.x is only useful for demanding applications, the present guide also states that Vert.x works very well for more traditional web applications. As we will see, the code will remain relatively easy to comprehend, but if the application needs to face a sudden peak in traffic then the code is already written with the essential ingredient for scaling up: *asynchronous processing of events*.

Finally, it is worth mentioning that Vert.x is *polyglot* as it supports a wide range of popular JVM languages: Java, Groovy, Scala, Kotlin, JavaScript, Ruby and Ceylon. The goal when supporting a language in Vert.x is not just to provide access to the APIs, but also to make sure that the language-specific APIs are idiomatic in each target language (e.g., using Scala futures in place of Vert.x futures). It is well-possible to develop different technical parts of a Vert.x application using different JVM languages.

1.3. Core Vert.x concepts

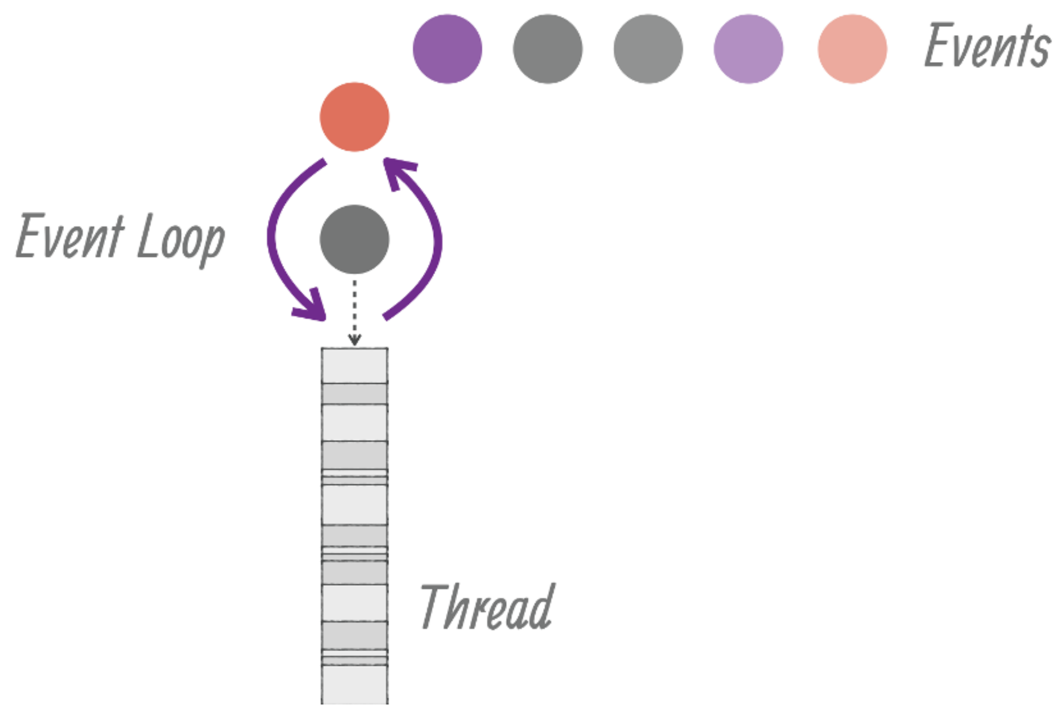
There are 2 key concepts to learn in Vert.x:

1. what a *verticle* is, and
2. how the *event bus* allows verticles to communicate.

1.3.1. Threading and programming models

Many networking libraries and frameworks rely on a simple threading strategy: each network client is being assigned a thread upon connection, and this thread deals with the client until it disconnects. This is the case with Servlet or networking code written with the `java.io` and `java.net` packages. While this *"synchronous I/O"* threading model has the advantage of remaining simple to comprehend, it hurts scalability with too many concurrent connections as system threads are not cheap, and under heavy loads an operating system kernel spends significant time *just* on thread scheduling management. In such cases we need to move to *"asynchronous I/O"* for which Vert.x provides a solid foundation.

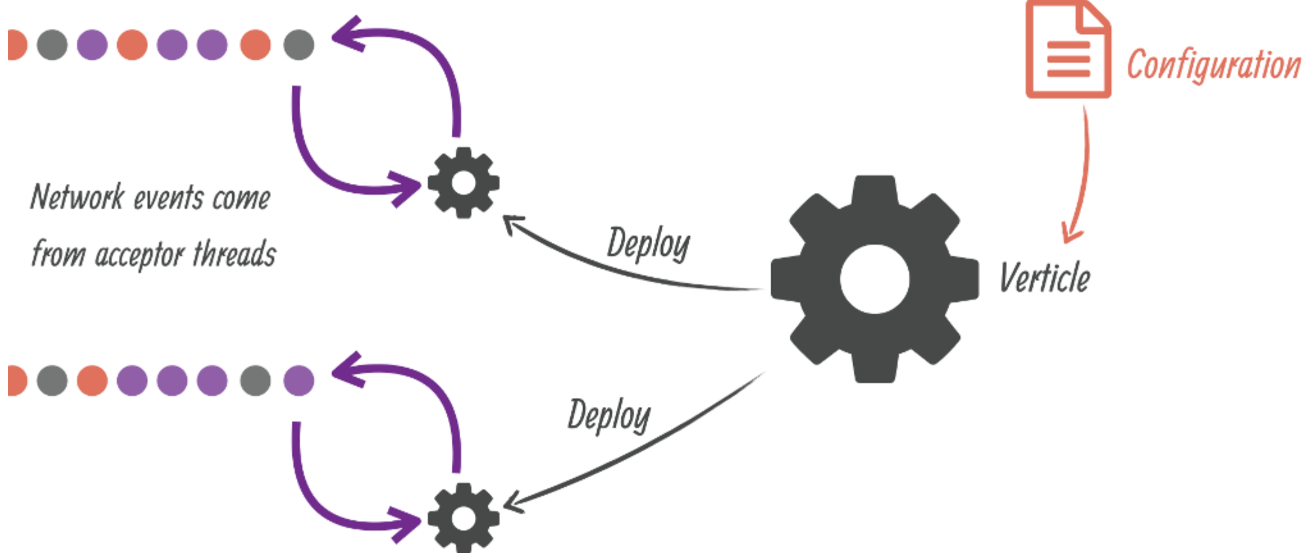
The unit of deployment in Vert.x is called a *Verticle*. A verticle processes incoming events over an *event-loop*, where events can be anything like receiving network buffers, timing events, or messages sent by other verticles. Event-loops are typical in asynchronous programming models:



Each event shall be processed in a *reasonable* amount of time to not block the event loop. This means that *thread blocking* operations shall not be performed while executed on the event loop, exactly like processing events in a graphical user interface (e.g., freezing a Java / Swing interface by doing a slow network request). As we will see later in this guide, Vert.x offers mechanisms to deal with blocking operations outside of the event loop. In any case Vert.x emits warnings in logs when the event loop has been processing an event for *too long*, which is also configurable to match application-specific requirements (e.g., when working on slower IoT ARM boards).

Every event loop is attached to a thread. By default Vert.x attaches 2 event loops per CPU core thread. The direct consequence is that a regular verticle always processes events on the same thread, so there is no need to use thread coordination mechanisms to manipulate a verticle state (e.g, Java class fields).

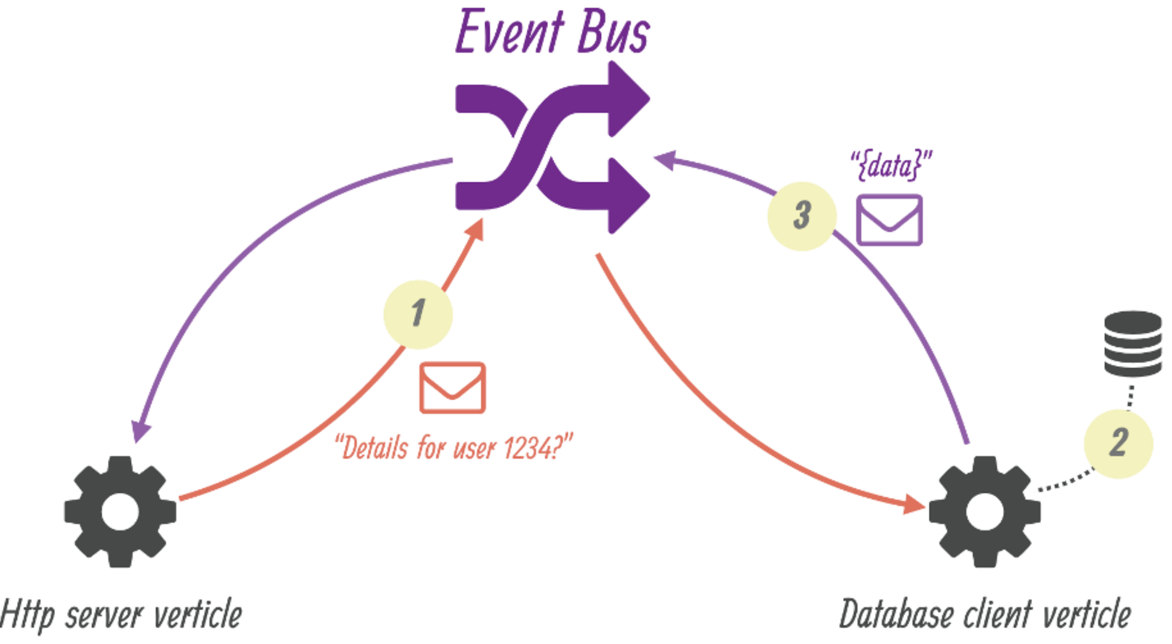
A verticle can be passed some configuration (e.g., credentials, network addresses, etc) and a verticle can be deployed several times:



Incoming network data are being received from accepting threads then passed as events to the corresponding verticles. When a verticle opens a network server and is deployed more than once, then the events are being distributed to the verticle instances in a round-robin fashion which is very useful for maximizing CPU usage with lots of concurrent networked requests. Finally, verticles have a simple start / stop life-cycle, and verticles can deploy other verticles.

1.3.2. Event bus

Verticles form technical units of deployments of code in Vert.x. The Vert.x *event bus* is the main tool for different verticles to communicate through asynchronous message passing. For instance suppose that we have a verticle for dealing with HTTP requests, and a verticle for managing access to the database. The event bus allows the HTTP verticle to send a request to the database verticle that performs a SQL query, and responds back to the HTTP verticle:



The event-bus allows passing any kind of data, although JSON is the preferred exchange format since it allows verticles written in different languages to communicate, and more generally JSON is a popular general-purpose semi-structured data marshaling text format.

Message can be sent to *destinations* which are free-form strings. The event bus supports the following communication patterns:

1. point-to-point messaging, and
2. request-response messaging and
3. publish / subscribe for broadcasting messages.

The event bus allows verticles to transparently communicate not just within the same JVM process:

- when network clustering is activated, the event bus is *distributed* so that messages can be sent to verticles running on other application nodes,
- the event-bus can be accessed through a simple TCP protocol for third-party applications to communicate,
- the event-bus can also be exposed over general-purpose messaging bridges (e.g, AMQP, Stomp),
- a SockJS bridge allows web applications to seamlessly communicate over the event bus from JavaScript running in the browser by receiving and publishing messages just like any verticle would do.

Chapter 2. A minimally viable wiki written with Vert.x

The corresponding source code is in the `step-1` folder of the guide repository.

We are going to start with a first iteration and the simplest code possible to have a wiki written with Vert.x. While the next iterations will introduce more elegance into the code base as well as proper testing, we will see that quick prototyping with Vert.x is both a simple and a realistic target.

At this stage the wiki will use server-side rendering of HTML pages and data persistence through a JDBC connection. To do so, we will use the following libraries.

1. Vert.x web as while the Vert.x core library *does* support the creation of HTTP servers, it does not provide elegant APIs to deal with routing, handling of request payloads, etc.
2. Vert.x JDBC client to provide an asynchronous API over JDBC.
3. Apache FreeMarker to render server-side pages as it is an uncomplicated template engine.
4. Txtmark to render Markdown text to HTML, allowing the edition of wiki pages in Markdown.

2.1. Bootstrapping a Maven project

This guide makes the choice of using Apache Maven as the build tool, primarily because it is very well integrated with the major integrated development environments. You can equally use another build tool such as Gradle.

The Vert.x community offers a template project structure from https://github.com/vert-x3/vertx-maven-starter that can be cloned. Since you will likely want to use (Git) version control as well, the fastest route is to clone the repository, delete its `.git/` folder and then create a new Git repository:

```
git clone https://github.com/vert-x3/vertx-maven-starter.git vertx-wiki
cd vertx-wiki
rm -rf .git
git init
```

The project offers a sample verticle as well as a unit test. You can safely delete all `.java` files beneath `src/` to hack on the wiki, but before doing so you may test that the project builds and runs successfully:

```
mvn package exec:java
```

You will notice that the Maven project `pom.xml` does 2 interesting things:

1. it uses the Maven Shade Plugin to create a single Jar archive with all required dependencies, suffixed by `-fat.jar`, also called *"a fat Jar"*, and

2. it uses the Exec Maven Plugin to provide the `exec:java` goal that in turns starts the application through the Vert.x `io.vertx.core.Launcher` class. This is actually equivalent to running using the `vertx` command-line tool that ships in the Vert.x distribution.

Finally, you will notice the presence of the `redeploy.sh` and `redeploy.bat` scripts that you can alternatively use for automatic compilation and redeployment upon code changes. Note that doing so requires ensuring that the `VERTICLE` variable in these scripts matches the main verticle to be used.

Alternatively, the Fabric8 project hosts a Vert.x Maven plugin. It has goals to initialize, build, package and run a Vert.x project.

To generate a similar project as by cloning the Git starter repository:

```
mkdir vertx-wiki
cd vertx-wiki
mvn io.fabric8:vertx-maven-plugin:1.0.7:setup -DvertxVersion=3.5.0
git init
```

2.2. Adding the required dependencies

The first batch of dependencies to add to the Maven `pom.xml` file are those for the web processing and rendering:

```
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-web</artifactId>
</dependency>
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-web-templ-freemarker</artifactId>
</dependency>
<dependency>
  <groupId>com.github.rjeschke</groupId>
  <artifactId>txtmark</artifactId>
  <version>0.13</version>
</dependency>
```

As the `vertx-web-templ-freemarker` name suggests, Vert.x web provides pluggable support for popular template engines: Handlebars, Jade, MVEL, Pebble, Thymeleaf and of course Freemarker.

The second set of dependencies are those required for JDBC database access:

```xml
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-jdbc-client</artifactId>
</dependency>
<dependency>
  <groupId>org.hsqldb</groupId>
  <artifactId>hsqldb</artifactId>
  <version>2.3.4</version>
</dependency>
```

The Vert.x JDBC client library provides access to any JDBC-compliant database, but of course our project needs to have a JDBC driver on the *classpath*.

HSQLDB is well-known relational database that is written in Java. It is quite popular when used as an embedded database to avoid the requirement of having a third-party database server running separately. It is also popular for unit and integration testing as it offers a (volatile) in-memory storage.

HSQLDB as an embedded database is a good fit to get us started. It stores data in local files, and since the HSQLDB library Jar provides a JDBC driver the Vert.x JDBC configuration will be straightforward.

Vert.x also offers dedicated MySQL and PostgreSQL client libraries.

Of course you can use the general-purpose Vert.x JDBC client to connect to MySQL or PostgreSQL databases, but these libraries offers better performance by working with these 2 database server network protocols rather than going through the (blocking) JDBC APIs.

Vert.x also provides libraries to deal with the popular non-relational databases MongoDB and Redis. The larger community offers integration with other storage systems like Apache Cassandra, OrientDB or ElasticSearch.

2.3. Anatomy of a verticle

The verticle for our wiki consists of a single `io.vertx.guides.wiki.MainVerticle` Java class. This class extends `io.vertx.core.AbstractVerticle`, the base class for verticles that mainly provides:

1. life-cycle `start` and `stop` methods to override,
2. a *protected* field called `vertx` that references the Vert.x environment where the verticle is being deployed,
3. an accessor to some configuration object that allows passing external configuration to a verticle.

To get started our verticle can just override the `start` method as follows:

```java
public class MainVerticle extends AbstractVerticle {

  @Override
  public void start(Future<Void> startFuture) throws Exception {
    startFuture.complete();
  }
}
```

There are 2 forms of `start` (and `stop`) methods: 1 with no argument and 1 with a *future* object reference. The no-argument variants imply that the verticle initialization or house-keeping phases always succeed unless an exception is being thrown. The variants with a *future* object provide a more fine-grained approach to *eventually* signal that operations succeeded or not. Indeed, some initialization or cleanup code may require asynchronous operations, so reporting via a *future* object naturally fits with asynchronous idioms.

2.4. A word on Vert.x future objects and callbacks

Vert.x futures are not JDK futures: they can be composed and queried in a non-blocking fashion. They shall be used for simple coordination of asynchronous tasks, and especially those of deploying verticles and checking if they were successfully deployed or not.

The Vert.x core APIs are based on callbacks to notify of asynchronous events. The seasoned developer will naturally think that this opens the door to the so-called *"callback hell"* where multiple levels of nested callbacks render the code difficult to comprehend as illustrated by this fictional code:

```java
foo.a(1, res1 -> {
  if (res1.succeeded()) {
    bar.b("abc", 1, res2 -> {
      if (res.succeeded()) {
         baz.c(res3 -> {
           dosomething(res1, res2, res3, res4 -> {
               // (...)
           });
         });
      }
    });
  }
});
```

While the core APIs could have been designed to favor promises and futures, the choice of callbacks is actually interesting since it allows different programming abstractions to be used. Vert.x is a largely un-opinionated project, and callbacks allow the implementation of different models that better cope with asynchronous programming: reactive extensions (via RxJava), promises and futures, fibers (using bytecode instrumentation), etc.

Since all Vert.x APIs are callback-oriented before other abstractions like RxJava can be leveraged,

this guide only uses callbacks in the first sections to ensure that the reader gets familiar with the core concepts in Vert.x. It is also arguably easier to start with callbacks to draw a line between the many sections of asynchronous code. Once it becomes evident in the sample code that callbacks do not always lead to easily readable code, we will introduce the RxJava support to show how the same asynchronous code can be better expressed by thinking in streams of processed events.

2.5. Wiki verticle initialization phases

To get our wiki running, we need to perform a 2-phases initialization:

1. we need to establish a JDBC database connection, and also make sure that the database schema is in place, and
2. we need to start a HTTP server for the web application.

Each phase can fail (e.g., the HTTP server TCP port is already being used), and they should not run in parallel as the web application code first needs the database access to work.

To make our code *cleaner* we will define 1 method per phase, and adopt a pattern of returning a *future / promise* object to notify when each of the phases completes, and whether it did so successfully or not:

```
private Future<Void> prepareDatabase() {
  Future<Void> future = Future.future();
  // (...)
  return future;
}

private Future<Void> startHttpServer() {
  Future<Void> future = Future.future();
  // (...)
  return future;
}
```

By having each method returning a *future* object, the implementation of the `start` method becomes a composition:

```
@Override
public void start(Future<Void> startFuture) throws Exception {
  Future<Void> steps = prepareDatabase().compose(v -> startHttpServer());
  steps.setHandler(startFuture.completer());
}
```

When the *future* of `prepareDatabase` completes successfully, then `startHttpServer` is called and the `steps` future completes depending of the outcome of the future returned by `startHttpServer`. `startHttpServer` is never called if `prepareDatabase` encounters an error, in which case the `steps` *future* is in a *failed* state and becomes completed with the exception describing the error.

Eventually `steps` completes: `setHandler` defines a handler to be called upon completion. In our case we simply want to complete `startFuture` with `steps` and use the `completer` method to obtain a handler. This is equivalent to:

```
Future<Void> steps = prepareDatabase().compose(v -> startHttpServer());
steps.setHandler(ar -> {  ①
  if (ar.succeeded()) {
    startFuture.complete();
  } else {
    startFuture.fail(ar.cause());
  }
});
```

① `ar` is of type `AsyncResult<Void>`. `AsyncResult<T>` is used to pass the result of an asynchronous processing and may either yield a value of type `T` on success or a failure exception is the processing failed.

2.5.1. Database initialization

The wiki database schema consists of a single table `Pages` with the following columns:

Column	Type	Description
`Id`	Integer	Primary key
`Name`	Characters	Name of a wiki page, must be unique
`Content`	Text	Markdown text of a wiki page

The database operations will be typical *create, read, update, delete* operations. To get us started, we simply store the corresponding SQL queries as static fields of the `MainVerticle` class. Note that they are written in a SQL dialect that HSQLDB understands, but that other relational databases may not necessarily support:

```
private static final String SQL_CREATE_PAGES_TABLE = "create table if not exists Pages (Id integer identity primary key,
Name varchar(255) unique, Content clob)";
private static final String SQL_GET_PAGE = "select Id, Content from Pages where Name = ?"; ①
private static final String SQL_CREATE_PAGE = "insert into Pages values (NULL, ?, ?)";
private static final String SQL_SAVE_PAGE = "update Pages set Content = ? where Id = ?";
private static final String SQL_ALL_PAGES = "select Name from Pages";
private static final String SQL_DELETE_PAGE = "delete from Pages where Id = ?";
```

① The `?` in the queries are placeholders to pass data when executing queries, and the Vert.x JDBC client prevents from SQL injections.

The application verticle needs to keep a reference to a `JDBCClient` object (from the `io.vertx.ext.jdbc` package) that serves as the connection to the database. We do so using a field in `MainVerticle`, and we also create a general-purpose logger from the `org.slf4j` package:

```
private JDBCClient dbClient;

private static final Logger LOGGER = LoggerFactory.getLogger(MainVerticle.class);
```

Last but not least, here is the complete implementation of the `prepareDatabase` method. It attempts to obtain a JDBC client connection, then performs a SQL query to create the `Pages` table unless it already exists:

```
private Future<Void> prepareDatabase() {
  Future<Void> future = Future.future();

  dbClient = JDBCClient.createShared(vertx, new JsonObject()  ①
    .put("url", "jdbc:hsqldb:file:db/wiki")   ②
    .put("driver_class", "org.hsqldb.jdbcDriver")   ③
    .put("max_pool_size", 30));   ④

  dbClient.getConnection(ar -> {    ⑤
    if (ar.failed()) {
      LOGGER.error("Could not open a database connection", ar.cause());
      future.fail(ar.cause());    ⑥
    } else {
      SQLConnection connection = ar.result();   ⑦
      connection.execute(SQL_CREATE_PAGES_TABLE, create -> {
        connection.close();   ⑧
        if (create.failed()) {
          LOGGER.error("Database preparation error", create.cause());
          future.fail(create.cause());
        } else {
          future.complete();  ⑨
        }
      });
    }
  });

  return future;
}
```

① `createShared` creates a shared connection to be shared among verticles known to the `vertx` instance, which in general is a good thing.

② The JDBC client connection is made by passing a Vert.x JSON object. Here `url` is the JDBC URL.

③ Just like `url`, `driver_class` is specific to the JDBC driver being used and points to the driver class.

④ `max_pool_size` is the number of concurrent connections. We chose 30 here, but it is just an arbitrary number.

⑤ Getting a connection is an asynchronous operation that gives us an `AsyncResult<SQLConnection>`. It must then be tested to see if the connection could be established or not (`AsyncResult` is actually a super-interface of `Future`).

⑥ If the SQL connection could not be obtained, then the method *future* is completed to fail with the `AsyncResult`-provided exception via the `cause` method.

⑦ The `SQLConnection` is the result of the successful `AsyncResult`. We can use it to perform a SQL query.

⑧ Before checking whether the SQL query succeeded or not, we must release it by calling `close`, otherwise the JDBC client connection pool can eventually drain.

⑨ We complete the method *future* object with a success.

The SQL database modules supported by the Vert.x project do not currently offer anything beyond passing SQL queries (e.g., an object-relationnal mapper) as they focus on providing asynchronous access to databases. However, nothing forbids using more advanced modules from the community, and we especially recommend checking out projects like this jOOq generator for Vert.x or this POJO mapper.

2.5.2. Notes about logging

The previous subsection introduced a logger, and we opted for the SFL4J library. Vert.x is also unopinionated on logging: you can choose any popular Java logging library. We recommend SLF4J since it is a popular logging abstraction and unification library in the Java ecosystem.

We also recommend using Logback as a logger implementation. Integrating both SLF4J and Logback can be done by adding two dependencies, or just `logback-classic` that points to both libraries (incidentally they are from the same author):

```
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>1.2.3</version>
</dependency>
```

By default SLF4J outputs many log events to the console from Vert.x, Netty, C3PO and the wiki application. We can reduce the verbosity by adding the a `src/main/resources/logback.xml` configuration file (see https://logback.qos.ch/ for more details):

```xml
<configuration>

  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>
    </encoder>
  </appender>

  <logger name="com.mchange.v2" level="warn"/>
  <logger name="io.netty" level="warn"/>
  <logger name="io.vertx" level="info"/>
  <logger name="io.vertx.guides.wiki" level="debug"/>

  <root level="debug">
    <appender-ref ref="STDOUT"/>
  </root>

</configuration>
```

Last but not least HSQLDB does not integrate well with loggers when embedded. By default it tries to reconfigure the logging system in place, so we need to disable it by passing a `-Dhsqldb.reconfig_logging=false` property to the Java Virtual Machine when executing applications.

2.5.3. HTTP server initialization

The HTTP server makes use of the `vertx-web` project to easily define *dispatching routes* for incoming HTTP requests. Indeed, the Vert.x core API allows to start HTTP servers and listen for incoming connections, but it does not provide any facility to, say, have different handlers depending on the requested URL or processing request bodies. This is the role of a *router* as it dispatches requests to different processing handlers depending on the URL, the HTTP method, etc.

The initialization consists in setting up a *request router*, then starting the HTTP server:

```java
private Future<Void> startHttpServer() {
  Future<Void> future = Future.future();
  HttpServer server = vertx.createHttpServer();   ①

  Router router = Router.router(vertx);   ②
  router.get("/").handler(this::indexHandler);
  router.get("/wiki/:page").handler(this::pageRenderingHandler); ③
  router.post().handler(BodyHandler.create());  ④
  router.post("/save").handler(this::pageUpdateHandler);
  router.post("/create").handler(this::pageCreateHandler);
  router.post("/delete").handler(this::pageDeletionHandler);

  server
    .requestHandler(router::accept)   ⑤
    .listen(8080, ar -> {   ⑥
      if (ar.succeeded()) {
        LOGGER.info("HTTP server running on port 8080");
        future.complete();
      } else {
        LOGGER.error("Could not start a HTTP server", ar.cause());
        future.fail(ar.cause());
      }
    });

  return future;
}
```

① The `vertx` context object provides methods to create HTTP servers, clients, TCP/UDP servers and clients, etc.

② The `Router` class comes from `vertx-web`: `io.vertx.ext.web.Router`.

③ Routes have their own handlers, and they can be defined by URL and/or by HTTP method. For short handlers a Java lambda is an option, but for more elaborated handlers it is a good idea to reference private methods instead. Note that URLs can be parametric: `/wiki/:page` will match a request like `/wiki/Hello`, in which case a `page` parameter will be available with value `Hello`.

④ This makes all HTTP POST requests got through a first handler, here `io.vertx.ext.web.handler.BodyHandler`. This handler automatically decodes the body from the HTTP requests (e.g., form submissions), which can then be manipulated as Vert.x buffer objects.

⑤ The router object can be used as a HTTP server handler, which then dispatches to other handlers as defined above.

⑥ Starting a HTTP server is an asynchronous operation, so an `AsyncResult<HttpServer>` needs to be checked for success. By the way the `8080` parameter specifies the TCP port to be used by the server.

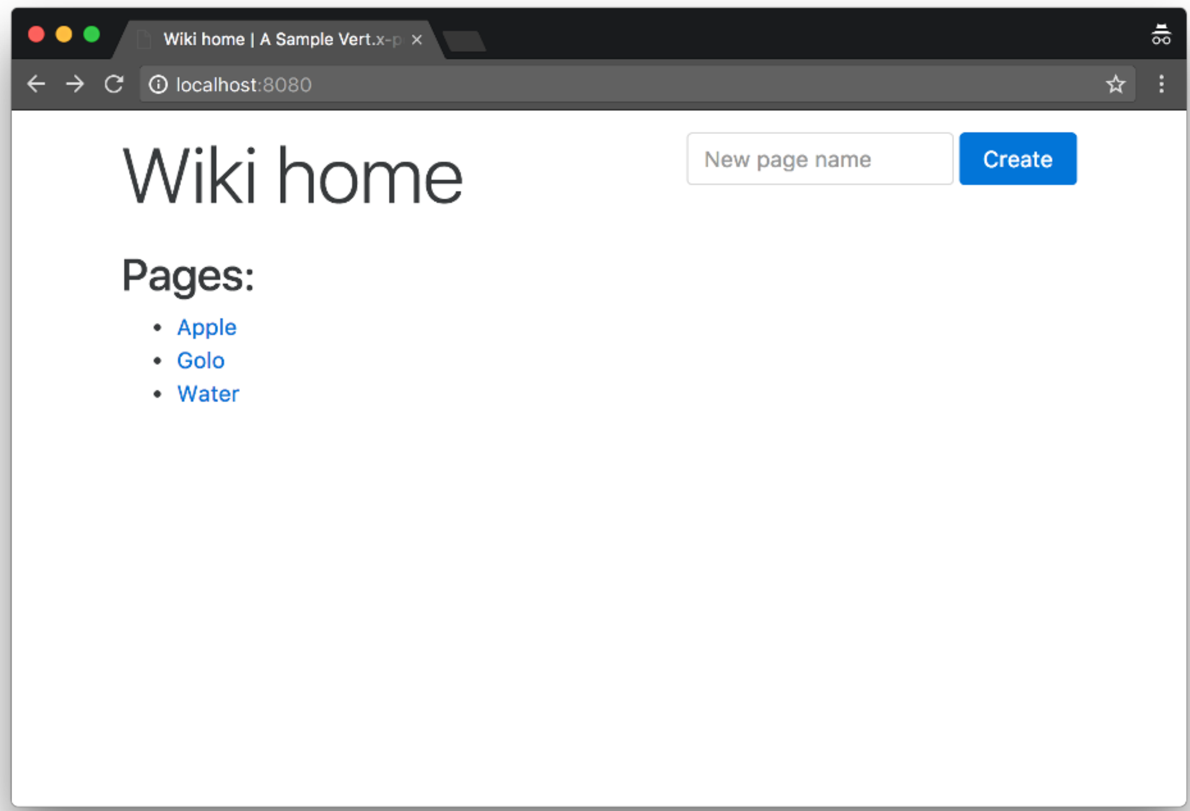
2.6. HTTP router handlers

The HTTP router instance of the `startHttpServer` method points to different handlers based on URL

patterns and HTTP methods. Each handlers deals with a HTTP request, performs a database query, and renders HTML from a FreeMarker template.

2.6.1. Index page handler

The index page provides a list of pointers to all wiki entries and a field to create a new one:



The implementation is a straightforward `select *` SQL query where data is then passed to the FreeMarker engine to render the HTML response.

The `indexHandler` method code is as follows:

```java
private final FreeMarkerTemplateEngine templateEngine = FreeMarkerTemplateEngine.create();

private void indexHandler(RoutingContext context) {
  dbClient.getConnection(car -> {
    if (car.succeeded()) {
      SQLConnection connection = car.result();
      connection.query(SQL_ALL_PAGES, res -> {
        connection.close();

        if (res.succeeded()) {
          List<String> pages = res.result() ①
            .getResults()
            .stream()
            .map(json -> json.getString(0))
            .sorted()
            .collect(Collectors.toList());

          context.put("title", "Wiki home");  ②
          context.put("pages", pages);
          templateEngine.render(context, "templates", "/index.ftl", ar -> {   ③
            if (ar.succeeded()) {
              context.response().putHeader("Content-Type", "text/html");
              context.response().end(ar.result());  ④
            } else {
              context.fail(ar.cause());
            }
          });

        } else {
          context.fail(res.cause());  ⑤
        }
      });
    } else {
      context.fail(car.cause());
    }
  });
}
```

① SQL query results are being returned as instances of `JsonArray` and `JsonObject`.

② The `RoutingContext` instance can be used to put arbitrary key / value data that is then available from templates, or chained router handlers.

③ Rendering a template is an asynchronous operation that leads us to the usual `AsyncResult` handling pattern.

④ The `AsyncResult` contains the template rendering as a `String` in case of success, and we can end the HTTP response stream with the value.

⑤ In case of failure the `fail` method from `RoutingContext` provides a sensible way to return a HTTP 500 error to the HTTP client.

FreeMarker templates shall be placed in the `src/main/resources/templates` folder. The `index.ftl`

template code is as follows:

```
<#include "header.ftl">

<div class="row">

  <div class="col-md-12 mt-1">
    <div class="float-xs-right">
      <form class="form-inline" action="/create" method="post">
        <div class="form-group">
          <input type="text" class="form-control" id="name" name="name" placeholder="New page name">
        </div>
        <button type="submit" class="btn btn-primary">Create</button>
      </form>
    </div>
    <h1 class="display-4">${context.title}</h1>
  </div>

  <div class="col-md-12 mt-1">
  <#list context.pages>
    <h2>Pages:</h2>
    <ul>
      <#items as page>
        <li><a href="/wiki/${page}">${page}</a></li>
      </#items>
    </ul>
  <#else>
    <p>The wiki is currently empty!</p>
  </#list>
  </div>

</div>

<#include "footer.ftl">
```

Key / value data stored in the `RoutingContext` object is made available through the `context` FreeMarker variable.

Since lots of templates have common header and footers, we extracted the following code in `header.ftl` and `footer.ftl`:

header.ftl

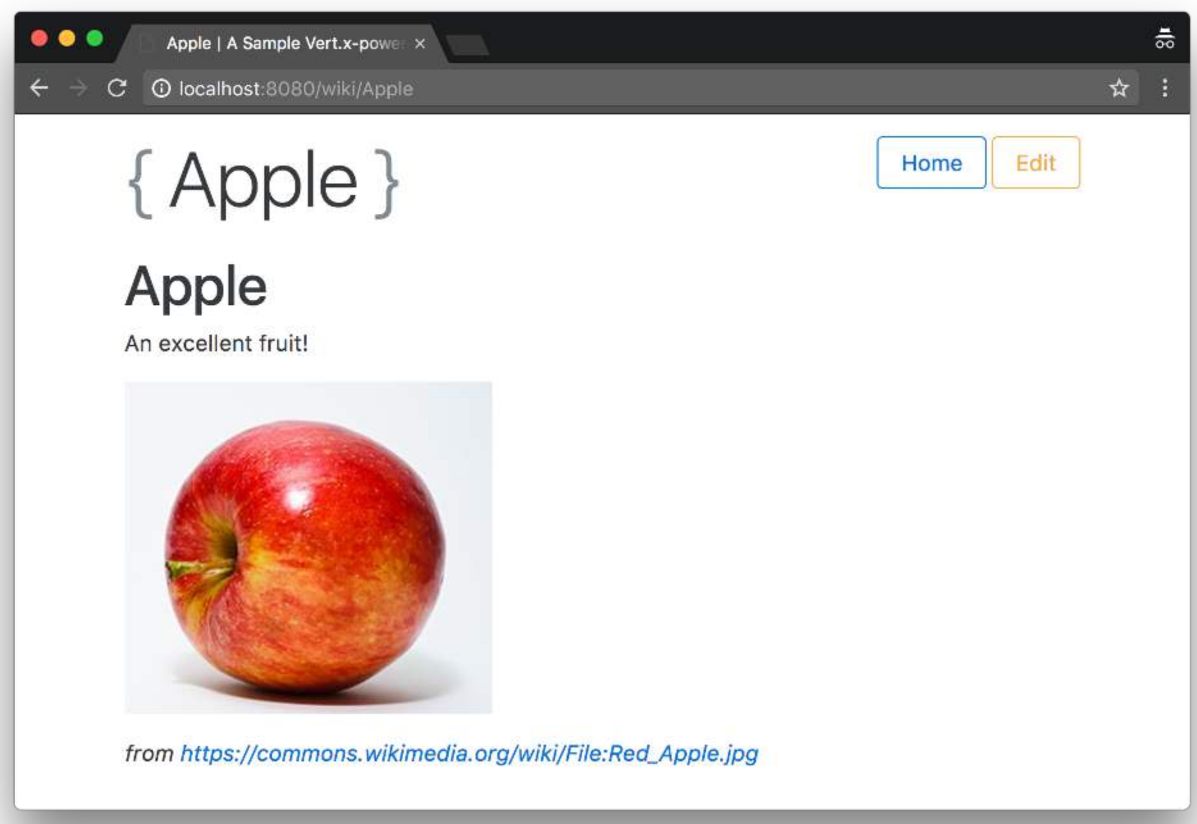
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
  <meta http-equiv="x-ua-compatible" content="ie=edge">
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-alpha.5/css/bootstrap.min.css"
        integrity="sha384-AysaV+vQoT3kOAXZkl02PThvDr8HYKPZhNT5h/CXfBThSRXQ6jW5DO2ekP5ViFdi" crossorigin="anonymous">
  <title>${context.title} | A Sample Vert.x-powered Wiki</title>
</head>
<body>

<div class="container">
```

footer.ftl

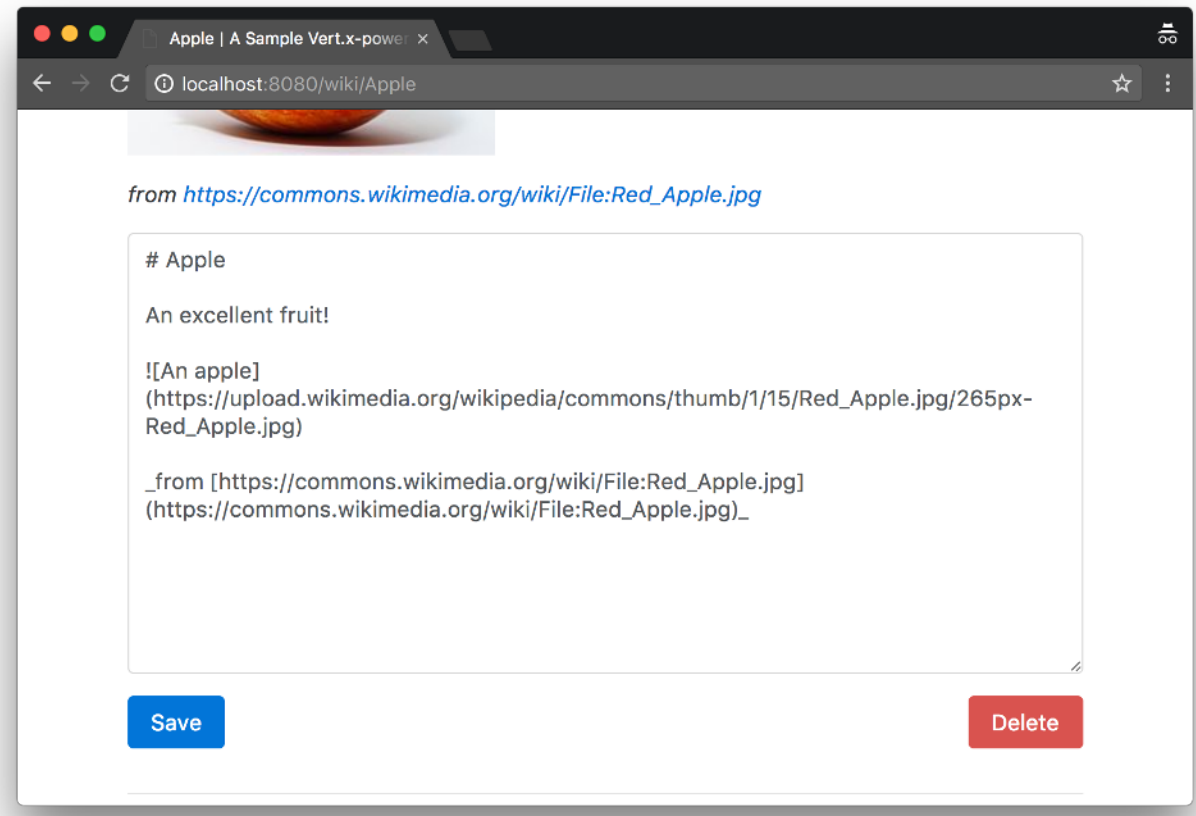
```
</div> <!-- .container -->

<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.1.1/jquery.min.js"
        integrity="sha384-3ceskX3iaEnIogmQchP8opvBy3Mi7Ce34nWjpBIwVTHfGYWQS9jwHDVRnpKKHJg7"
        crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/tether/1.3.7/js/tether.min.js"
        integrity="sha384-XTs3FgkjiBgo8qjEjBk0tGmf3wPrWtA6coPfQDfFEY8AnYJwjalXCiosYRBIBZX8"
        crossorigin="anonymous"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-alpha.5/js/bootstrap.min.js"
        integrity="sha384-BLiI7JTZm+JWlgKa0M0kGRpJbF2J8q+qreVrKBC47e3K6BW78kGLrCkeRX6I9RoK"
        crossorigin="anonymous"></script>

</body>
</html>
```

2.6.2. Wiki page rendering handler

This handler deals with HTTP GET requests to have wiki pages being rendered, as in:



The page also provides a button to edit the content in Markdown. Instead of having a separate handler and template, we simply rely on JavaScript and CSS to toggle the editor on and off when the button is being clicked:



The `pageRenderingHandler` method code is the following:

```java
private static final String EMPTY_PAGE_MARKDOWN =
  "# A new page\n" +
    "\n" +
    "Feel-free to write in Markdown!\n";

private void pageRenderingHandler(RoutingContext context) {
  String page = context.request().getParam("page");   ①

  dbClient.getConnection(car -> {
    if (car.succeeded()) {

      SQLConnection connection = car.result();
      connection.queryWithParams(SQL_GET_PAGE, new JsonArray().add(page), fetch -> {  ②
        connection.close();
        if (fetch.succeeded()) {

          JsonArray row = fetch.result().getResults()
            .stream()
            .findFirst()
            .orElseGet(() -> new JsonArray().add(-1).add(EMPTY_PAGE_MARKDOWN));
          Integer id = row.getInteger(0);
          String rawContent = row.getString(1);

          context.put("title", page);
          context.put("id", id);
          context.put("newPage", fetch.result().getResults().size() == 0 ? "yes" : "no");
          context.put("rawContent", rawContent);
          context.put("content", Processor.process(rawContent));  ③
          context.put("timestamp", new Date().toString());

          templateEngine.render(context, "templates", "/page.ftl", ar -> {
            if (ar.succeeded()) {
              context.response().putHeader("Content-Type", "text/html");
              context.response().end(ar.result());
            } else {
              context.fail(ar.cause());
            }
          });
        } else {
          context.fail(fetch.cause());
        }
      });

    } else {
      context.fail(car.cause());
    }
  });
}
```

① URL parameters (`/wiki/:name` here) can be accessed through the context request object.

② Passing argument values to SQL queries is done using a `JsonArray`, with the elements in order of

the ? symbols in the SQL query.

③ The Processor class comes from the *txtmark* Markdown rendering library that we use.

The page.ftl FreeMarker template code is as follows:

```
<#include "header.ftl">

<div class="row">

  <div class="col-md-12 mt-1">
      <span class="float-xs-right">
        <a class="btn btn-outline-primary" href="/" role="button" aria-pressed="true">Home</a>
        <button class="btn btn-outline-warning" type="button" data-toggle="collapse"
                data-target="#editor" aria-expanded="false" aria-controls="editor">Edit</button>
      </span>
    <h1 class="display-4">
      <span class="text-muted">{</span>
    ${context.title}
      <span class="text-muted">}</span>
    </h1>
  </div>

  <div class="col-md-12 mt-1 clearfix">
  ${context.content}
  </div>

  <div class="col-md-12 collapsable collapse clearfix" id="editor">
    <form action="/save" method="post">
      <div class="form-group">
        <input type="hidden" name="id" value="${context.id}">
        <input type="hidden" name="title" value="${context.title}">
        <input type="hidden" name="newPage" value="${context.newPage}">
        <textarea class="form-control" id="markdown" name="markdown" rows="15">${context.rawContent}</textarea>
      </div>
      <button type="submit" class="btn btn-primary">Save</button>
    <#if context.id != -1>
      <button type="submit" formaction="/delete" class="btn btn-danger float-xs-right">Delete</button>
    </#if>
    </form>
  </div>

  <div class="col-md-12 mt-1">
    <hr class="mt-1">
    <p class="small">Rendered: ${context.timestamp}</p>
  </div>

</div>

<#include "footer.ftl">
```

2.6.3. Page creation handler

The index page offers a field to create a new page, and its surrounding HTML form points to a URL that is being managed by this handler. The strategy isn't actually to create a new entry in the database, but simply to redirect to a wiki page URL with the name to create. Since the wiki page doesn't exist, the pageRenderingHandler method will use a default text for new pages, and a user can eventually create that page by editing then saving it.

The handler is the `pageCreateHandler` method, and its implementation is a redirection through a HTTP 303 status code:

```
private void pageCreateHandler(RoutingContext context) {
  String pageName = context.request().getParam("name");
  String location = "/wiki/" + pageName;
  if (pageName == null || pageName.isEmpty()) {
    location = "/";
  }
  context.response().setStatusCode(303);
  context.response().putHeader("Location", location);
  context.response().end();
}
```

2.6.4. Page saving handler

The `pageUpdateHandler` method deals with HTTP POST requests when saving a wiki page. This happens both when updating an existing page (issuing a SQL `update` query) or saving a new page (issuing a SQL `insert` query):

```java
private void pageUpdateHandler(RoutingContext context) {
  String id = context.request().getParam("id");   ①
  String title = context.request().getParam("title");
  String markdown = context.request().getParam("markdown");
  boolean newPage = "yes".equals(context.request().getParam("newPage"));  ②

  dbClient.getConnection(car -> {
    if (car.succeeded()) {
      SQLConnection connection = car.result();
      String sql = newPage ? SQL_CREATE_PAGE : SQL_SAVE_PAGE;
      JsonArray params = new JsonArray();   ③
      if (newPage) {
        params.add(title).add(markdown);
      } else {
        params.add(markdown).add(id);
      }
      connection.updateWithParams(sql, params, res -> {   ④
        connection.close();
        if (res.succeeded()) {
          context.response().setStatusCode(303);    ⑤
          context.response().putHeader("Location", "/wiki/" + title);
          context.response().end();
        } else {
          context.fail(res.cause());
        }
      });
    } else {
      context.fail(car.cause());
    }
  });
}
```

① Form parameters sent through a HTTP POST request are available from the `RoutingContext` object. Note that without a `BodyHandler` within the `Router` configuration chain these values would not be available, and the form submission payload would need to be manually decoded from the HTTP POST request payload.

② We rely on a form hidden field rendered in the `page.ftl` FreeMarker template to know if we are updating an existing page or saving a new page.

③ Again, preparing the SQL query with parameters uses a `JsonArray` to pass values.

④ The `updateWithParams` method is used for `insert` / `update` / `delete` SQL queries.

⑤ Upon success, we simply redirect to the page that has been edited.

2.6.5. Page deletion handler

The implementation of the `pageDeletionHandler` method is straightforward: given a wiki entry identifier, it issues a `delete` SQL query then redirects to the wiki index page:

```java
private void pageDeletionHandler(RoutingContext context) {
  String id = context.request().getParam("id");
  dbClient.getConnection(car -> {
    if (car.succeeded()) {
      SQLConnection connection = car.result();
      connection.updateWithParams(SQL_DELETE_PAGE, new JsonArray().add(id), res -> {
        connection.close();
        if (res.succeeded()) {
          context.response().setStatusCode(303);
          context.response().putHeader("Location", "/");
          context.response().end();
        } else {
          context.fail(res.cause());
        }
      });
    } else {
      context.fail(car.cause());
    }
  });
}
```

2.7. Running the application

At this stage we have a working, self-contained wiki application.

To run it we first need to build it with Maven:

```
$ mvn clean package
```

Since the build produces a Jar will all required dependencies embedded (including Vert.x and a JDBC database), running the wiki is as simple as:

```
$ java -jar target/wiki-step-1-1.1.0-fat.jar
```

You can then point your favorite web browser to http://localhost:8080/ and enjoy using the wiki.

Chapter 3. Refactoring into independent and reusable verticles

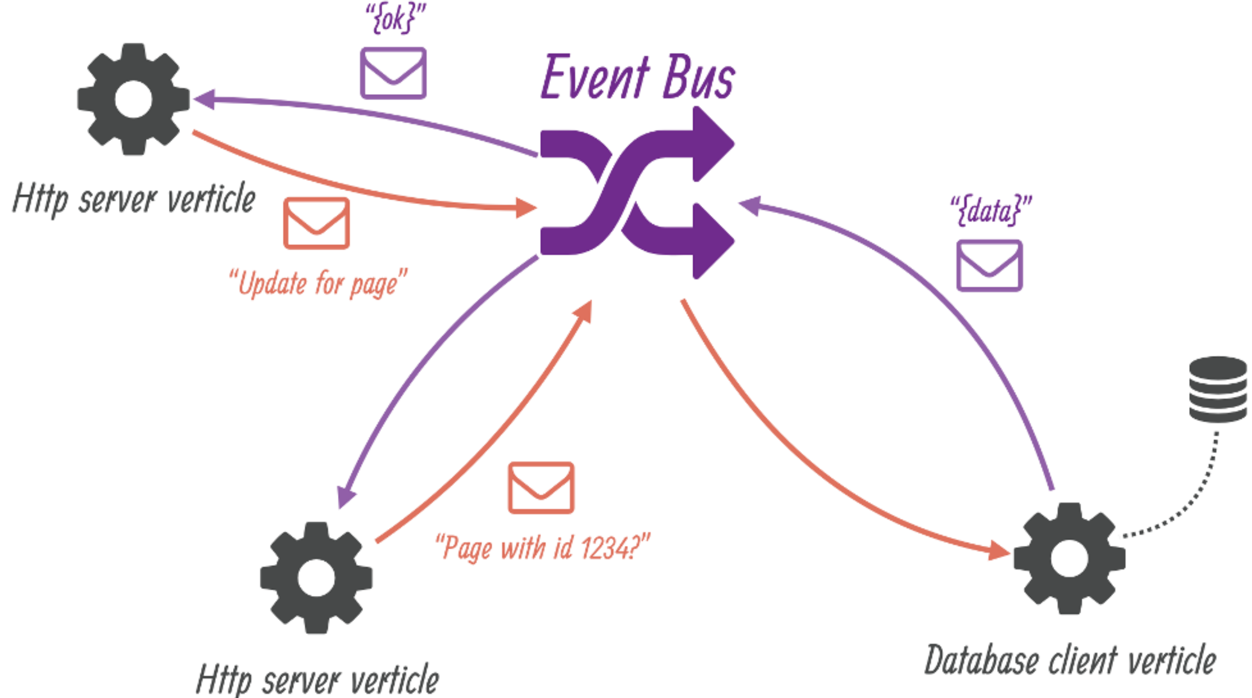
The corresponding source code is in the `step-2` folder of the guide repository.

The first iteration got us a working wiki application. Still, its implementation suffers from the following issues:

1. HTTP requests processing and database access code are interleaved within the same methods, and
2. lots of configuration data (e.g., port numbers, JDBC driver, etc) are hard-coded strings in the code.

3.1. Architecture and technical choices

This second iteration is about refactoring the code into independent and reusable verticles:



We will deploy 2 verticles to deal with HTTP requests, and 1 verticle for encapsulating persistence through the database. The 2 resulting verticles will not have direct references to each other as they will only agree on destination names in the event bus as well as message formats. This provides a simple yet effective decoupling.

The messages sent on the event bus will be encoded in JSON. While Vert.x supports flexible serialization schemes on the event bus for demanding or highly-specific contexts, it is generally a wise choice to go with JSON data. Another advantage of using JSON is that it is a language-agnostic text format. As Vert.x is *polyglot*, JSON is ideal shall verticles written in different languages need to communicate via message passing.

3.2. The HTTP server verticle

The verticle class preamble and `start` method look as follows:

```
public class HttpServerVerticle extends AbstractVerticle {

  private static final Logger LOGGER = LoggerFactory.getLogger(HttpServerVerticle.class);

  public static final String CONFIG_HTTP_SERVER_PORT = "http.server.port";  ①
  public static final String CONFIG_WIKIDB_QUEUE = "wikidb.queue";

  private String wikiDbQueue = "wikidb.queue";

  @Override
  public void start(Future<Void> startFuture) throws Exception {

    wikiDbQueue = config().getString(CONFIG_WIKIDB_QUEUE, "wikidb.queue");  ②

    HttpServer server = vertx.createHttpServer();

    Router router = Router.router(vertx);
    router.get("/").handler(this::indexHandler);
    router.get("/wiki/:page").handler(this::pageRenderingHandler);
    router.post().handler(BodyHandler.create());
    router.post("/save").handler(this::pageUpdateHandler);
    router.post("/create").handler(this::pageCreateHandler);
    router.post("/delete").handler(this::pageDeletionHandler);

    int portNumber = config().getInteger(CONFIG_HTTP_SERVER_PORT, 8080);  ③
    server
      .requestHandler(router::accept)
      .listen(portNumber, ar -> {
        if (ar.succeeded()) {
          LOGGER.info("HTTP server running on port " + portNumber);
          startFuture.complete();
        } else {
          LOGGER.error("Could not start a HTTP server", ar.cause());
          startFuture.fail(ar.cause());
        }
      });
  }

  // (...)
```

① We expose public constants for the verticle configuration parameters: the HTTP port number and the name of the event bus destination to post messages to the database verticle.

② The `AbstractVerticle#config()` method allows accessing the verticle configuration that has been provided. The second parameter is a default value in case no specific value was given.

③ Configuration values can not just be `String` objects but also integers, boolean values, complex

JSON data, etc.

The rest of the class is mostly an extract of the HTTP-only code, with what was previously database code being replaced with event bus messages. Here is the `indexHandler` method code:

```
private final FreeMarkerTemplateEngine templateEngine = FreeMarkerTemplateEngine.create();

private void indexHandler(RoutingContext context) {

  DeliveryOptions options = new DeliveryOptions().addHeader("action", "all-pages"); ②

  vertx.eventBus().send(wikiDbQueue, new JsonObject(), options, reply -> {  ①
    if (reply.succeeded()) {
      JsonObject body = (JsonObject) reply.result().body();   ③
      context.put("title", "Wiki home");
      context.put("pages", body.getJsonArray("pages").getList());
      templateEngine.render(context, "templates", "/index.ftl", ar -> {
        if (ar.succeeded()) {
          context.response().putHeader("Content-Type", "text/html");
          context.response().end(ar.result());
        } else {
          context.fail(ar.cause());
        }
      });
    } else {
      context.fail(reply.cause());
    }
  });
}
```

① The `vertx` object gives access to the event bus, and we send a message to queue for the database vertice.

② Delivery options allow us to specify headers, payload codecs and timeouts.

③ Upon success a reply contains a payload.

As we can see, an event bus message consists of a body, options, and it can optionally expect a reply. In the event that no response is expected there is a variant of the `send` method does not have a handler.

We encode payloads as JSON objects, and we specify which action the database verticle should do through a message header called `action`.

The rest of the verticle code consists in the router handlers that also use the event-bus to fetch and store data:

```
private static final String EMPTY_PAGE_MARKDOWN =
"# A new page\n" +
  "\n" +
  "Feel-free to write in Markdown!\n";
```

```java
private void pageRenderingHandler(RoutingContext context) {

  String requestedPage = context.request().getParam("page");
  JsonObject request = new JsonObject().put("page", requestedPage);

  DeliveryOptions options = new DeliveryOptions().addHeader("action", "get-page");
  vertx.eventBus().send(wikiDbQueue, request, options, reply -> {

    if (reply.succeeded()) {
      JsonObject body = (JsonObject) reply.result().body();

      boolean found = body.getBoolean("found");
      String rawContent = body.getString("rawContent", EMPTY_PAGE_MARKDOWN);
      context.put("title", requestedPage);
      context.put("id", body.getInteger("id", -1));
      context.put("newPage", found ? "no" : "yes");
      context.put("rawContent", rawContent);
      context.put("content", Processor.process(rawContent));
      context.put("timestamp", new Date().toString());

      templateEngine.render(context, "templates","/page.ftl", ar -> {
        if (ar.succeeded()) {
          context.response().putHeader("Content-Type", "text/html");
          context.response().end(ar.result());
        } else {
          context.fail(ar.cause());
        }
      });

    } else {
      context.fail(reply.cause());
    }
  });
}

private void pageUpdateHandler(RoutingContext context) {

  String title = context.request().getParam("title");
  JsonObject request = new JsonObject()
    .put("id", context.request().getParam("id"))
    .put("title", title)
    .put("markdown", context.request().getParam("markdown"));

  DeliveryOptions options = new DeliveryOptions();
  if ("yes".equals(context.request().getParam("newPage"))) {
    options.addHeader("action", "create-page");
  } else {
    options.addHeader("action", "save-page");
  }
```

```
  vertx.eventBus().send(wikiDbQueue, request, options, reply -> {
    if (reply.succeeded()) {
      context.response().setStatusCode(303);
      context.response().putHeader("Location", "/wiki/" + title);
      context.response().end();
    } else {
      context.fail(reply.cause());
    }
  });
}

private void pageCreateHandler(RoutingContext context) {
  String pageName = context.request().getParam("name");
  String location = "/wiki/" + pageName;
  if (pageName == null || pageName.isEmpty()) {
    location = "/";
  }
  context.response().setStatusCode(303);
  context.response().putHeader("Location", location);
  context.response().end();
}

private void pageDeletionHandler(RoutingContext context) {
  String id = context.request().getParam("id");
  JsonObject request = new JsonObject().put("id", id);
  DeliveryOptions options = new DeliveryOptions().addHeader("action", "delete-page");
  vertx.eventBus().send(wikiDbQueue, request, options, reply -> {
    if (reply.succeeded()) {
      context.response().setStatusCode(303);
      context.response().putHeader("Location", "/");
      context.response().end();
    } else {
      context.fail(reply.cause());
    }
  });
}
```

3.3. The database verticle

Connecting to a database using JDBC requires of course a driver and configuration, which we had hard-coded in the first iteration.

3.3.1. Configurable SQL queries

While the verticle will turn the previously hard-coded values to configuration parameters, we will also go a step further by loading the SQL queries from a properties file.

The queries will be loaded from a file passed as a configuration parameter or from a default resource if none is being provided. The advantage of this approach is that the verticle can adapt

both to different JDBC drivers *and* SQL dialects.

The verticle class preamble consists mainly of configuration key definitions:

```
public class WikiDatabaseVerticle extends AbstractVerticle {

  public static final String CONFIG_WIKIDB_JDBC_URL = "wikidb.jdbc.url";
  public static final String CONFIG_WIKIDB_JDBC_DRIVER_CLASS = "wikidb.jdbc.driver_class";
  public static final String CONFIG_WIKIDB_JDBC_MAX_POOL_SIZE = "wikidb.jdbc.max_pool_size";
  public static final String CONFIG_WIKIDB_SQL_QUERIES_RESOURCE_FILE = "wikidb.sqlqueries.resource.file";

  public static final String CONFIG_WIKIDB_QUEUE = "wikidb.queue";

  private static final Logger LOGGER = LoggerFactory.getLogger(WikiDatabaseVerticle.class);

  // (...)
```

SQL queries are being stored in a properties file, with the default ones for HSQLDB being located in `src/main/resources/db-queries.properties`:

```
create-pages-table=create table if not exists Pages (Id integer identity primary key, Name varchar(255) unique, Content
clob)
get-page=select Id, Content from Pages where Name = ?
create-page=insert into Pages values (NULL, ?, ?)
save-page=update Pages set Content = ? where Id = ?
all-pages=select Name from Pages
delete-page=delete from Pages where Id = ?
```

The following code from the `WikiDatabaseVerticle` class loads the SQL queries from a file, and make them available from a map:

```java
private enum SqlQuery {
  CREATE_PAGES_TABLE,
  ALL_PAGES,
  GET_PAGE,
  CREATE_PAGE,
  SAVE_PAGE,
  DELETE_PAGE
}

private final HashMap<SqlQuery, String> sqlQueries = new HashMap<>();

private void loadSqlQueries() throws IOException {

  String queriesFile = config().getString(CONFIG_WIKIDB_SQL_QUERIES_RESOURCE_FILE);
  InputStream queriesInputStream;
  if (queriesFile != null) {
    queriesInputStream = new FileInputStream(queriesFile);
  } else {
    queriesInputStream = getClass().getResourceAsStream("/db-queries.properties");
  }

  Properties queriesProps = new Properties();
  queriesProps.load(queriesInputStream);
  queriesInputStream.close();

  sqlQueries.put(SqlQuery.CREATE_PAGES_TABLE, queriesProps.getProperty("create-pages-table"));
  sqlQueries.put(SqlQuery.ALL_PAGES, queriesProps.getProperty("all-pages"));
  sqlQueries.put(SqlQuery.GET_PAGE, queriesProps.getProperty("get-page"));
  sqlQueries.put(SqlQuery.CREATE_PAGE, queriesProps.getProperty("create-page"));
  sqlQueries.put(SqlQuery.SAVE_PAGE, queriesProps.getProperty("save-page"));
  sqlQueries.put(SqlQuery.DELETE_PAGE, queriesProps.getProperty("delete-page"));
}
```

We use the `SqlQuery` enumeration type to avoid string constants later in the code. The code of the verticle `start` method is the following:

```java
private JDBCClient dbClient;

@Override
public void start(Future<Void> startFuture) throws Exception {

  /*
   * Note: this uses blocking APIs, but data is small...
   */
  loadSqlQueries();  ①

  dbClient = JDBCClient.createShared(vertx, new JsonObject()
    .put("url", config().getString(CONFIG_WIKIDB_JDBC_URL, "jdbc:hsqldb:file:db/wiki"))
    .put("driver_class", config().getString(CONFIG_WIKIDB_JDBC_DRIVER_CLASS, "org.hsqldb.jdbcDriver"))
    .put("max_pool_size", config().getInteger(CONFIG_WIKIDB_JDBC_MAX_POOL_SIZE, 30)));

  dbClient.getConnection(ar -> {
    if (ar.failed()) {
      LOGGER.error("Could not open a database connection", ar.cause());
      startFuture.fail(ar.cause());
    } else {
      SQLConnection connection = ar.result();
      connection.execute(sqlQueries.get(SqlQuery.CREATE_PAGES_TABLE), create -> {   ②
        connection.close();
        if (create.failed()) {
          LOGGER.error("Database preparation error", create.cause());
          startFuture.fail(create.cause());
        } else {
          vertx.eventBus().consumer(config().getString(CONFIG_WIKIDB_QUEUE, "wikidb.queue"), this::onMessage);  ③
          startFuture.complete();
        }
      });
    }
  });
}
```

① Interestingly we break an important principle in Vert.x which is to avoid blocking APIs, but since there are no asynchronous APIs for accessing resources on the classpath our options are limited. We could use the Vert.x `executeBlocking` method to offload the blocking I/O operations from the event loop to a worker thread, but since the data is very small there is no obvious benefit in doing so.

② Here is an example of using SQL queries.

③ The `consumer` method registers an event bus destination handler.

3.3.2. Dispatching requests

The event bus message handler is the `onMessage` method:

```java
public enum ErrorCodes {
  NO_ACTION_SPECIFIED,
  BAD_ACTION,
  DB_ERROR
}

public void onMessage(Message<JsonObject> message) {

  if (!message.headers().contains("action")) {
    LOGGER.error("No action header specified for message with headers {} and body {}",
      message.headers(), message.body().encodePrettily());
    message.fail(ErrorCodes.NO_ACTION_SPECIFIED.ordinal(), "No action header specified");
    return;
  }
  String action = message.headers().get("action");

  switch (action) {
    case "all-pages":
      fetchAllPages(message);
      break;
    case "get-page":
      fetchPage(message);
      break;
    case "create-page":
      createPage(message);
      break;
    case "save-page":
      savePage(message);
      break;
    case "delete-page":
      deletePage(message);
      break;
    default:
      message.fail(ErrorCodes.BAD_ACTION.ordinal(), "Bad action: " + action);
  }
}
```

We defined a `ErrorCodes` enumeration for errors, which we use to report back to the message sender. To do so, the `fail` method of the `Message` class provides a convenient shortcut to reply with an error, and the original message sender gets a failed `AsyncResult`.

3.3.3. Reducing the JDBC client boilerplate

So far we have seen the *complete* interaction to perform a SQL query:

1. retrieve a connection,
2. perform requests,
3. release the connection.

This leads to code where lots of error processing needs to happen for each asynchronous operation, as in:

```
dbClient.getConnection(car -> {
  if (car.succeeded()) {
    SQLConnection connection = car.result();
    connection.query(sqlQueries.get(SqlQuery.ALL_PAGES), res -> {
      connection.close();
      if (res.succeeded()) {
        List<String> pages = res.result()
          .getResults()
          .stream()
          .map(json -> json.getString(0))
          .sorted()
          .collect(Collectors.toList());
        message.reply(new JsonObject().put("pages", new JsonArray(pages)));
      } else {
        reportQueryError(message, res.cause());
      }
    });
  } else {
    reportQueryError(message, car.cause());
  }
});
```

Starting from Vert.x 3.5.0, the JDBC client now supports *one-shot* operations where a connection is being acquired to do a SQL operation, then released internally. The same code as above now reduces to:

```
dbClient.query(sqlQueries.get(SqlQuery.ALL_PAGES), res -> {
  if (res.succeeded()) {
    List<String> pages = res.result()
      .getResults()
      .stream()
      .map(json -> json.getString(0))
      .sorted()
      .collect(Collectors.toList());
    message.reply(new JsonObject().put("pages", new JsonArray(pages)));
  } else {
    reportQueryError(message, res.cause());
  }
});
```

This is very useful for cases where the connection is being acquired for a single operation. Performance-wise it is important to note that re-using a connection for chained SQL operations is better.

The rest of the class consists of private methods called when `onMessage` dispatches incoming

messages:

```
private void fetchAllPages(Message<JsonObject> message) {
  dbClient.query(sqlQueries.get(SqlQuery.ALL_PAGES), res -> {
    if (res.succeeded()) {
      List<String> pages = res.result()
        .getResults()
        .stream()
        .map(json -> json.getString(0))
        .sorted()
        .collect(Collectors.toList());
      message.reply(new JsonObject().put("pages", new JsonArray(pages)));
    } else {
      reportQueryError(message, res.cause());
    }
  });
}

private void fetchPage(Message<JsonObject> message) {
  String requestedPage = message.body().getString("page");
  JsonArray params = new JsonArray().add(requestedPage);

  dbClient.queryWithParams(sqlQueries.get(SqlQuery.GET_PAGE), params, fetch -> {
    if (fetch.succeeded()) {
      JsonObject response = new JsonObject();
      ResultSet resultSet = fetch.result();
      if (resultSet.getNumRows() == 0) {
        response.put("found", false);
      } else {
        response.put("found", true);
        JsonArray row = resultSet.getResults().get(0);
        response.put("id", row.getInteger(0));
        response.put("rawContent", row.getString(1));
      }
      message.reply(response);
    } else {
      reportQueryError(message, fetch.cause());
    }
  });
}

private void createPage(Message<JsonObject> message) {
  JsonObject request = message.body();
  JsonArray data = new JsonArray()
    .add(request.getString("title"))
    .add(request.getString("markdown"));

  dbClient.updateWithParams(sqlQueries.get(SqlQuery.CREATE_PAGE), data, res -> {
    if (res.succeeded()) {
      message.reply("ok");
```

```
    } else {
      reportQueryError(message, res.cause());
    }
  });
}

private void savePage(Message<JsonObject> message) {
  JsonObject request = message.body();
  JsonArray data = new JsonArray()
    .add(request.getString("markdown"))
    .add(request.getString("id"));

  dbClient.updateWithParams(sqlQueries.get(SqlQuery.SAVE_PAGE), data, res -> {
    if (res.succeeded()) {
      message.reply("ok");
    } else {
      reportQueryError(message, res.cause());
    }
  });
}

private void deletePage(Message<JsonObject> message) {
  JsonArray data = new JsonArray().add(message.body().getString("id"));

  dbClient.updateWithParams(sqlQueries.get(SqlQuery.DELETE_PAGE), data, res -> {
    if (res.succeeded()) {
      message.reply("ok");
    } else {
      reportQueryError(message, res.cause());
    }
  });
}

private void reportQueryError(Message<JsonObject> message, Throwable cause) {
  LOGGER.error("Database query error", cause);
  message.fail(ErrorCodes.DB_ERROR.ordinal(), cause.getMessage());
}
```

3.4. Deploying the verticles from a main verticle

We still have a `MainVerticle` class, but instead of containing all the business logic like in the initial iteration, its sole purpose is to bootstrap the application and deploy other verticles.

The code consists in deploying 1 instance of `WikiDatabaseVerticle` and 2 instances of `HttpServerVerticle` :

```java
public class MainVerticle extends AbstractVerticle {

  @Override
  public void start(Future<Void> startFuture) throws Exception {

    Future<String> dbVerticleDeployment = Future.future();  ①
    vertx.deployVerticle(new WikiDatabaseVerticle(), dbVerticleDeployment.completer());  ②

    dbVerticleDeployment.compose(id -> {  ③

      Future<String> httpVerticleDeployment = Future.future();
      vertx.deployVerticle(
        "io.vertx.guides.wiki.HttpServerVerticle",  ④
        new DeploymentOptions().setInstances(2),    ⑤
        httpVerticleDeployment.completer());

      return httpVerticleDeployment;  ⑥

    }).setHandler(ar -> {   ⑦
      if (ar.succeeded()) {
        startFuture.complete();
      } else {
        startFuture.fail(ar.cause());
      }
    });
  }
}
```

① Deploying a verticle is an asynchronous operation, so we need a `Future` for that. The `String` parametric type is because a verticle gets an identifier when successfully deployed.

② One option is to create a verticle instance with `new`, and pass the object reference to the `deploy` method. The `completer` return value is a handler that simply completes its future.

③ Sequential composition with `compose` allows to run one asynchronous operation after the other. When the initial future completes successfully, the composition function is invoked.

④ A class name as a string is also an option to specify a verticle to deploy. For other JVM languages string-based conventions allow a module / script to be specified.

⑤ The `DeploymentOption` class allows to specify a number of parameters and especially the number of instances to deploy.

⑥ The composition function returns the next future. Its completion will trigger the completion of the composite operation.

⑦ We define a handler that eventually completes the `MainVerticle` start future.

The astute reader will probably wonder how we can deploy the code for a HTTP server on the same TCP port twice and not expect any error for either of the instances, since the TCP port will already be in use. With many web frameworks we would need to choose different TCP ports, and have a frontal HTTP proxy to perform load balancing between the ports.

There is no need to do that with Vert.x as multiple verticles can share the same TCP ports. Incoming

connections are simply distributed in a round-robin fashion from accepting threads.

Chapter 4. Refactoring to Vert.x services

The corresponding source code is in the `step-3` folder of the guide repository.

The previous refactoring was a big step forward compared to our initial implementation, as we extracted independent and configurable verticles connected using asynchronous messages on the event bus. We also saw that we could deploy several instances of a given verticle to better cope with load and to better leverage CPU cores.

In this section we see how to design and use Vert.x services. The main advantage of a service is that it defines an interface for doing certain operations that a verticle exposes. We also leverage code generation for all the event bus messaging plumbing, instead of crafting it ourselves like we did in the previous section.

We are also going to refactor the code into different Java packages:

```
step-3/src/main/java/
 └── io
     └── vertx
         └── guides
             └── wiki
                 ├── MainVerticle.java
                 ├── database
                 │   ├── ErrorCodes.java
                 │   ├── SqlQuery.java
                 │   ├── WikiDatabaseService.java
                 │   ├── WikiDatabaseServiceImpl.java
                 │   ├── WikiDatabaseVerticle.java
                 │   └── package-info.java
                 └── http
                     └── HttpServerVerticle.java
```

`io.vertx.guides.wiki` will now contain the main verticle, `io.vertx.guides.wiki.database` the database verticle and service, and `io.vertx.guides.wiki.http` the HTTP server verticle.

4.1. Maven configuration changes

First, we need to add the following 2 dependencies to our project. Obviously we need the `vertx-service-proxy` APIs:

```
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-service-proxy</artifactId>
</dependency>
```

We need the Vert.x code generation module as a compilation-time only dependency (hence the

`provided` scope):

```
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-codegen</artifactId>
  <scope>provided</scope>
</dependency>
```

Next we have to tweak the `maven-compiler-plugin` configuration to use code generation, which is done via a `javac` annotation processor:

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.5.1</version>
  <configuration>
    <source>1.8</source>
    <target>1.8</target>
    <useIncrementalCompilation>false</useIncrementalCompilation>

    <annotationProcessors>
      <annotationProcessor>io.vertx.codegen.CodeGenProcessor</annotationProcessor>
    </annotationProcessors>
    <generatedSourcesDirectory>${project.basedir}/src/main/generated</generatedSourcesDirectory>
    <compilerArgs>
      <arg>-AoutputDirectory=${project.basedir}/src/main</arg>
    </compilerArgs>

  </configuration>
</plugin>
```

Note that the generated code is put in `src/main/generated`, which some integrated development environments like IntelliJ IDEA will automatically pick up on the classpath.

It is also a good idea to update the `maven-clean-plugin` to remove those generated files:

```
<plugin>
  <artifactId>maven-clean-plugin</artifactId>
  <version>3.0.0</version>
  <configuration>
    <filesets>
      <fileset>
        <directory>${project.basedir}/src/main/generated</directory>
      </fileset>
    </filesets>
  </configuration>
</plugin>
```

The full documentation on Vert.x services is available at http://vertx.io/docs/vertx-service-proxy/java/

4.2. Database service interface

Defining a service interface is as simple as defining a Java interface, except that there are certain rules to respect for code generation to work and also to ensure inter-operability with other code in Vert.x.

The beginning of the interface definition is:

```
@ProxyGen
public interface WikiDatabaseService {

  @Fluent
  WikiDatabaseService fetchAllPages(Handler<AsyncResult<JsonArray>> resultHandler);

  @Fluent
  WikiDatabaseService fetchPage(String name, Handler<AsyncResult<JsonObject>> resultHandler);

  @Fluent
  WikiDatabaseService createPage(String title, String markdown, Handler<AsyncResult<Void>> resultHandler);

  @Fluent
  WikiDatabaseService savePage(int id, String markdown, Handler<AsyncResult<Void>> resultHandler);

  @Fluent
  WikiDatabaseService deletePage(int id, Handler<AsyncResult<Void>> resultHandler);

  // (...)
```

1. The `ProxyGen` annotation is used to trigger the code generation of a proxy for clients of that service.
2. The `Fluent` annotation is optional, but allows *fluent* interfaces where operations can be chained by returning the service instance. This is mostly useful for the code generator when the service shall be consumed from other JVM languages.
3. Parameter types need to be strings, Java primitive types, JSON objects or arrays, any enumeration type or a `java.util` collection (`List` / `Set` / `Map`) of the previous types. The only way to support arbitrary Java classes is to have them as Vert.x data objects, annotated with `@DataObject`. The last opportunity to pass other types is service reference types.
4. Since services provide asynchronous results, the last argument of a service method needs to be a `Handler<AsyncResult<T>>` where `T` is any of the types suitable for code generation as described above.

It is a good practice that service interfaces provide static methods to create instances of both the actual service implementation and proxy for client code over the event bus.

We define `create` as simply delegating to the implementation class and its constructor:

```
static WikiDatabaseService create(JDBCClient dbClient, HashMap<SqlQuery, String> sqlQueries, Handler<AsyncResult
<WikiDatabaseService>> readyHandler) {
  return new WikiDatabaseServiceImpl(dbClient, sqlQueries, readyHandler);
}
```

The Vert.x code generator creates the proxy class and names it by suffixing with `VertxEBProxy`. Constructors of these proxy classes need a reference to the Vert.x context as well as a destination address on the event bus:

```
static WikiDatabaseService createProxy(Vertx vertx, String address) {
  return new WikiDatabaseServiceVertxEBProxy(vertx, address);
}
```

The `SqlQuery` and `ErrorCodes` enumeration types that were inner classes in the previous iteration have been extracted to package-protected types, see `SqlQuery.java` and `ErrorCodes.java`.

4.3. Database service implementation

The service implementation is a straightforward port of the previous `WikiDatabaseVerticle` class code. The essential difference is the support of the asynchronous result handler in the constructor (to report the initialization outcome) and in the service methods (to report the operation success).

The class code is the following:

```
class WikiDatabaseServiceImpl implements WikiDatabaseService {

  private static final Logger LOGGER = LoggerFactory.getLogger(WikiDatabaseServiceImpl.class);

  private final HashMap<SqlQuery, String> sqlQueries;
  private final JDBCClient dbClient;

  WikiDatabaseServiceImpl(JDBCClient dbClient, HashMap<SqlQuery, String> sqlQueries, Handler<AsyncResult
<WikiDatabaseService>> readyHandler) {
    this.dbClient = dbClient;
    this.sqlQueries = sqlQueries;

    dbClient.getConnection(ar -> {
      if (ar.failed()) {
        LOGGER.error("Could not open a database connection", ar.cause());
        readyHandler.handle(Future.failedFuture(ar.cause()));
      } else {
        SQLConnection connection = ar.result();
        connection.execute(sqlQueries.get(SqlQuery.CREATE_PAGES_TABLE), create -> {
          connection.close();
          if (create.failed()) {
            LOGGER.error("Database preparation error", create.cause());
            readyHandler.handle(Future.failedFuture(create.cause()));
          } else {
            readyHandler.handle(Future.succeededFuture(this));
          }
        });
      }
    });
```

```
  }

  @Override
  public WikiDatabaseService fetchAllPages(Handler<AsyncResult<JsonArray>> resultHandler) {
    dbClient.query(sqlQueries.get(SqlQuery.ALL_PAGES), res -> {
      if (res.succeeded()) {
        JsonArray pages = new JsonArray(res.result()
          .getResults()
          .stream()
          .map(json -> json.getString(0))
          .sorted()
          .collect(Collectors.toList()));
        resultHandler.handle(Future.succeededFuture(pages));
      } else {
        LOGGER.error("Database query error", res.cause());
        resultHandler.handle(Future.failedFuture(res.cause()));
      }
    });
    return this;
  }

  @Override
  public WikiDatabaseService fetchPage(String name, Handler<AsyncResult<JsonObject>> resultHandler) {
    dbClient.queryWithParams(sqlQueries.get(SqlQuery.GET_PAGE), new JsonArray().add(name), fetch -> {
      if (fetch.succeeded()) {
        JsonObject response = new JsonObject();
        ResultSet resultSet = fetch.result();
        if (resultSet.getNumRows() == 0) {
          response.put("found", false);
        } else {
          response.put("found", true);
          JsonArray row = resultSet.getResults().get(0);
          response.put("id", row.getInteger(0));
          response.put("rawContent", row.getString(1));
        }
        resultHandler.handle(Future.succeededFuture(response));
      } else {
        LOGGER.error("Database query error", fetch.cause());
        resultHandler.handle(Future.failedFuture(fetch.cause()));
      }
    });
    return this;
  }

  @Override
  public WikiDatabaseService createPage(String title, String markdown, Handler<AsyncResult<Void>> resultHandler) {
    JsonArray data = new JsonArray().add(title).add(markdown);
    dbClient.updateWithParams(sqlQueries.get(SqlQuery.CREATE_PAGE), data, res -> {
      if (res.succeeded()) {
        resultHandler.handle(Future.succeededFuture());
      } else {
        LOGGER.error("Database query error", res.cause());
        resultHandler.handle(Future.failedFuture(res.cause()));
      }
    });
    return this;
  }

  @Override
  public WikiDatabaseService savePage(int id, String markdown, Handler<AsyncResult<Void>> resultHandler) {
    JsonArray data = new JsonArray().add(markdown).add(id);
    dbClient.updateWithParams(sqlQueries.get(SqlQuery.SAVE_PAGE), data, res -> {
      if (res.succeeded()) {
        resultHandler.handle(Future.succeededFuture());
      } else {
        LOGGER.error("Database query error", res.cause());
        resultHandler.handle(Future.failedFuture(res.cause()));
```

```java
      }
    });
    return this;
  }

  @Override
  public WikiDatabaseService deletePage(int id, Handler<AsyncResult<Void>> resultHandler) {
    JsonArray data = new JsonArray().add(id);
    dbClient.updateWithParams(sqlQueries.get(SqlQuery.DELETE_PAGE), data, res -> {
      if (res.succeeded()) {
        resultHandler.handle(Future.succeededFuture());
      } else {
        LOGGER.error("Database query error", res.cause());
        resultHandler.handle(Future.failedFuture(res.cause()));
      }
    });
    return this;
  }
}
```

There is one last step required before the proxy code generation works: the service package needs to have a `package-info.java` annotated to define a Vert.x module:

```java
@ModuleGen(groupPackage = "io.vertx.guides.wiki.database", name = "wiki-database")
package io.vertx.guides.wiki.database;

import io.vertx.codegen.annotations.ModuleGen;
```

4.4. Exposing the database service from the database verticle

As most of the database handling code has been moved to `WikiDatabaseServiceImpl`, the `WikiDatabaseVerticle` class now consists of 2 methods: the `start` method to register the service and a utility method to load SQL queries:

```java
public class WikiDatabaseVerticle extends AbstractVerticle {

  public static final String CONFIG_WIKIDB_JDBC_URL = "wikidb.jdbc.url";
  public static final String CONFIG_WIKIDB_JDBC_DRIVER_CLASS = "wikidb.jdbc.driver_class";
  public static final String CONFIG_WIKIDB_JDBC_MAX_POOL_SIZE = "wikidb.jdbc.max_pool_size";
  public static final String CONFIG_WIKIDB_SQL_QUERIES_RESOURCE_FILE = "wikidb.sqlqueries.resource.file";
  public static final String CONFIG_WIKIDB_QUEUE = "wikidb.queue";

  @Override
  public void start(Future<Void> startFuture) throws Exception {

    HashMap<SqlQuery, String> sqlQueries = loadSqlQueries();

    JDBCClient dbClient = JDBCClient.createShared(vertx, new JsonObject()
      .put("url", config().getString(CONFIG_WIKIDB_JDBC_URL, "jdbc:hsqldb:file:db/wiki"))
      .put("driver_class", config().getString(CONFIG_WIKIDB_JDBC_DRIVER_CLASS, "org.hsqldb.jdbcDriver"))
      .put("max_pool_size", config().getInteger(CONFIG_WIKIDB_JDBC_MAX_POOL_SIZE, 30)));

    WikiDatabaseService.create(dbClient, sqlQueries, ready -> {
      if (ready.succeeded()) {
        ProxyHelper.registerService(WikiDatabaseService.class, vertx, ready.result(), CONFIG_WIKIDB_QUEUE); ①
        startFuture.complete();
      } else {
        startFuture.fail(ready.cause());
      }
    });
  }

  /*
   * Note: this uses blocking APIs, but data is small...
   */
  private HashMap<SqlQuery, String> loadSqlQueries() throws IOException {

    String queriesFile = config().getString(CONFIG_WIKIDB_SQL_QUERIES_RESOURCE_FILE);
    InputStream queriesInputStream;
    if (queriesFile != null) {
      queriesInputStream = new FileInputStream(queriesFile);
    } else {
      queriesInputStream = getClass().getResourceAsStream("/db-queries.properties");
    }

    Properties queriesProps = new Properties();
    queriesProps.load(queriesInputStream);
    queriesInputStream.close();

    HashMap<SqlQuery, String> sqlQueries = new HashMap<>();
    sqlQueries.put(SqlQuery.CREATE_PAGES_TABLE, queriesProps.getProperty("create-pages-table"));
    sqlQueries.put(SqlQuery.ALL_PAGES, queriesProps.getProperty("all-pages"));
    sqlQueries.put(SqlQuery.GET_PAGE, queriesProps.getProperty("get-page"));
    sqlQueries.put(SqlQuery.CREATE_PAGE, queriesProps.getProperty("create-page"));
    sqlQueries.put(SqlQuery.SAVE_PAGE, queriesProps.getProperty("save-page"));
    sqlQueries.put(SqlQuery.DELETE_PAGE, queriesProps.getProperty("delete-page"));
    return sqlQueries;
  }
}
```

① We register the service here.

Registering a service requires an interface class, a Vert.x context, an implementation and an event

bus destination.

The `WikiDatabaseServiceVertxEBProxy` generated class handles receiving messages on the event bus and then dispatching them to the `WikiDatabaseServiceImpl`. What it does is actually very close to what we did in the previous section: messages are being sent with a `action` header to specify which method to invoke, and parameters are encoded in JSON.

4.5. Obtaining a database service proxy

The final steps to refactoring to Vert.x services is to adapt the HTTP server verticle to obtain a proxy to the database service and use it in the handlers instead of the event bus.

First, we need to create the proxy when the verticle starts:

```
private WikiDatabaseService dbService;

@Override
public void start(Future<Void> startFuture) throws Exception {

  String wikiDbQueue = config().getString(CONFIG_WIKIDB_QUEUE, "wikidb.queue"); ①
  dbService = WikiDatabaseService.createProxy(vertx, wikiDbQueue);

  HttpServer server = vertx.createHttpServer();
  // (...)
```

① We just need to make sure to use the same event bus destination as the service that was published by `WikiDatabaseVerticle`.

Then, we need to replace calls to the event bus with calls to the database service:

```
private void indexHandler(RoutingContext context) {
  dbService.fetchAllPages(reply -> {
    if (reply.succeeded()) {
      context.put("title", "Wiki home");
      context.put("pages", reply.result().getList());
      templateEngine.render(context, "templates", "/index.ftl", ar -> {
        if (ar.succeeded()) {
          context.response().putHeader("Content-Type", "text/html");
          context.response().end(ar.result());
        } else {
          context.fail(ar.cause());
        }
      });
    } else {
      context.fail(reply.cause());
    }
  });
}

private void pageRenderingHandler(RoutingContext context) {
```

```java
  String requestedPage = context.request().getParam("page");
  dbService.fetchPage(requestedPage, reply -> {
    if (reply.succeeded()) {

      JsonObject payLoad = reply.result();
      boolean found = payLoad.getBoolean("found");
      String rawContent = payLoad.getString("rawContent", EMPTY_PAGE_MARKDOWN);
      context.put("title", requestedPage);
      context.put("id", payLoad.getInteger("id", -1));
      context.put("newPage", found ? "no" : "yes");
      context.put("rawContent", rawContent);
      context.put("content", Processor.process(rawContent));
      context.put("timestamp", new Date().toString());

      templateEngine.render(context, "templates", "/page.ftl", ar -> {
        if (ar.succeeded()) {
          context.response().putHeader("Content-Type", "text/html");
          context.response().end(ar.result());
        } else {
          context.fail(ar.cause());
        }
      });

    } else {
      context.fail(reply.cause());
    }
  });
}

private void pageUpdateHandler(RoutingContext context) {
  String title = context.request().getParam("title");

  Handler<AsyncResult<Void>> handler = reply -> {
    if (reply.succeeded()) {
      context.response().setStatusCode(303);
      context.response().putHeader("Location", "/wiki/" + title);
      context.response().end();
    } else {
      context.fail(reply.cause());
    }
  };

  String markdown = context.request().getParam("markdown");
  if ("yes".equals(context.request().getParam("newPage"))) {
    dbService.createPage(title, markdown, handler);
  } else {
    dbService.savePage(Integer.valueOf(context.request().getParam("id")), markdown, handler);
  }
}

private void pageCreateHandler(RoutingContext context) {
  String pageName = context.request().getParam("name");
  String location = "/wiki/" + pageName;
  if (pageName == null || pageName.isEmpty()) {
```

```
    location = "/";
  }
  context.response().setStatusCode(303);
  context.response().putHeader("Location", location);
  context.response().end();
}

private void pageDeletionHandler(RoutingContext context) {
  dbService.deletePage(Integer.valueOf(context.request().getParam("id")), reply -> {
    if (reply.succeeded()) {
      context.response().setStatusCode(303);
      context.response().putHeader("Location", "/");
      context.response().end();
    } else {
      context.fail(reply.cause());
    }
  });
}
```

The `WikiDatabaseServiceVertxProxyHandler` generated class deals with forwarding calls as event bus messages.

It is still perfectly possible to consume a Vert.x service directly via event bus messages since this is what generated proxys do.

Chapter 5. Testing Vert.x code

The corresponding source code is in the step-4 folder of the guide repository.

Up to this point we have been developing the wiki implementation without testing. This is of course not a good practice, so let us see how to write tests for Vert.x code.

5.1. Getting started

The vertx-unit module provides utilities to test asynchronous operations in Vert.x. Aside from that, you can use your testing framework of choice like JUnit.

With JUnit, the required Maven dependencies are the following:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-unit</artifactId>
  <scope>test</scope>
</dependency>
```

JUnit tests need to be annotated with the VertxUnitRunner runner to use the vertx-unit features:

```
@RunWith(VertxUnitRunner.class)
public class SomeTest {
  // (...)
}
```

With that runner, JUnit test and life-cycle methods accept a TestContext argument. This object provides access to basic assertions, a context to store data, and several async-oriented helpers that we will see in this section.

To illustrate that, let us consider an asynchronous scenario where we want to check that a timer task has been called once, and that a periodic task has been called 3 times. Since that code is asynchronous, the test method exits before the test completes, so making that test pass or fail also needs to be done in an asynchronous fashion:

```java
@Test /*(timeout=5000)*/  ⑧
public void async_behavior(TestContext context) { ①
  Vertx vertx = Vertx.vertx();  ②
  context.assertEquals("foo", "foo");  ③
  Async a1 = context.async();   ④
  Async a2 = context.async(3);  ⑤
  vertx.setTimer(100, n -> a1.complete());  ⑥
  vertx.setPeriodic(100, n -> a2.countDown());  ⑦
}
```

① `TestContext` is a parameter provided by the runner.

② Since we are in unit tests, we need to create a Vert.x context.

③ Here is an example of a basic `TestContext` assertion.

④ We get a first `Async` object that can later be completed (or failed).

⑤ This `Async` object works as a countdown that completes successfully after 3 calls.

⑥ We complete when the timer fires.

⑦ Each periodic task tick triggers a countdown. The test passes when all `Async` objects have completed.

⑧ There is a default (long) timeout for asynchronous test cases, but it can be overridden through the JUnit `@Test` annotation.

5.2. Testing database operations

The database service is a good fit for writing tests.

We first need to deploy the database verticle. We will configure the JDBC connection to be HSQLDB with an in-memory storage, and upon success we will fetch a service proxy for our test cases.

Since these operations are involving, we leverage the JUnit *before* / *after* life-cycle methods:

```java
private Vertx vertx;
private WikiDatabaseService service;

@Before
public void prepare(TestContext context) throws InterruptedException {
  vertx = Vertx.vertx();

  JsonObject conf = new JsonObject()  ①
    .put(WikiDatabaseVerticle.CONFIG_WIKIDB_JDBC_URL, "jdbc:hsqldb:mem:testdb;shutdown=true")
    .put(WikiDatabaseVerticle.CONFIG_WIKIDB_JDBC_MAX_POOL_SIZE, 4);

  vertx.deployVerticle(new WikiDatabaseVerticle(), new DeploymentOptions().setConfig(conf),
    context.asyncAssertSuccess(id ->  ②
      service = WikiDatabaseService.createProxy(vertx, WikiDatabaseVerticle.CONFIG_WIKIDB_QUEUE)));
}
```

① We only override some of the verticle settings, the others will have default values.

② `asyncAssertSuccess` is useful to provide a handler that checks for the success of an asynchronous operation. There is a variant with no arguments, and a variant like this one where we can chain the result to another handler.

Cleaning up the Vert.x context is straightforward, and again we use `asyncAssertSuccess` to ensure that no error was encountered:

```
@After
public void finish(TestContext context) {
  vertx.close(context.asyncAssertSuccess());
}
```

The service operations are essentially CRUD operations, so a JUnit test case combining all of them is a fine way to test:

```
@Test
public void crud_operations(TestContext context) {
  Async async = context.async();

  service.createPage("Test", "Some content", context.asyncAssertSuccess(v1 -> {

    service.fetchPage("Test", context.asyncAssertSuccess(json1 -> {
      context.assertTrue(json1.getBoolean("found"));
      context.assertTrue(json1.containsKey("id"));
      context.assertEquals("Some content", json1.getString("rawContent"));

      service.savePage(json1.getInteger("id"), "Yo!", context.asyncAssertSuccess(v2 -> {

        service.fetchAllPages(context.asyncAssertSuccess(array1 -> {
          context.assertEquals(1, array1.size());

          service.fetchPage("Test", context.asyncAssertSuccess(json2 -> {
            context.assertEquals("Yo!", json2.getString("rawContent"));

            service.deletePage(json1.getInteger("id"), v3 -> {

              service.fetchAllPages(context.asyncAssertSuccess(array2 -> {
                context.assertTrue(array2.isEmpty());
                async.complete();  ①
              }));
            });
          }));
        }));
      }));
    }));
  }));
  async.awaitSuccess(5000); ②
}
```

① This is where the sole `Async` eventually completes.

② This is an alternative to exiting the test case method and relying on a JUnit timeout. Here the execution on the test case thread waits until either the `Async` completes or the timeout period elapses.

Chapter 6. Integrating with a 3rd-party web service

The corresponding source code is in the `step-5` folder of the guide repository.

Modern applications rarely live on a separated island as they need to integrate with other (distributed) application and services. Integration is very often achieved using APIs exposed over the versatile HTTP protocol.

This section shows how to integrate with a 3rd-party web service using the HTTP client APIs of Vert.x.

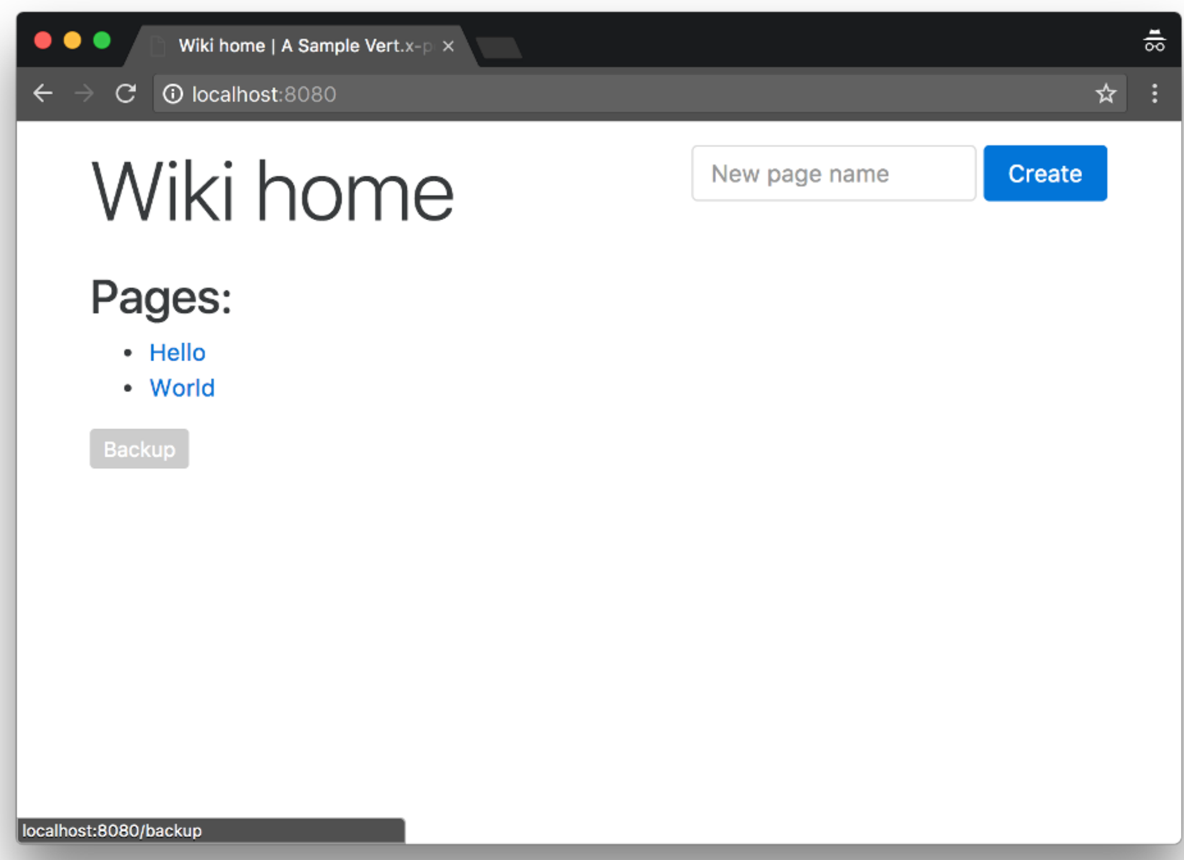
6.1. Scenario: backing up to GitHub Gist

The GitHub Gist service is popular for sharing code snippets to the world. Other services can use it, an example being the Medium publishing platform where links to Gists allow code snippets to be embedded inside publications.

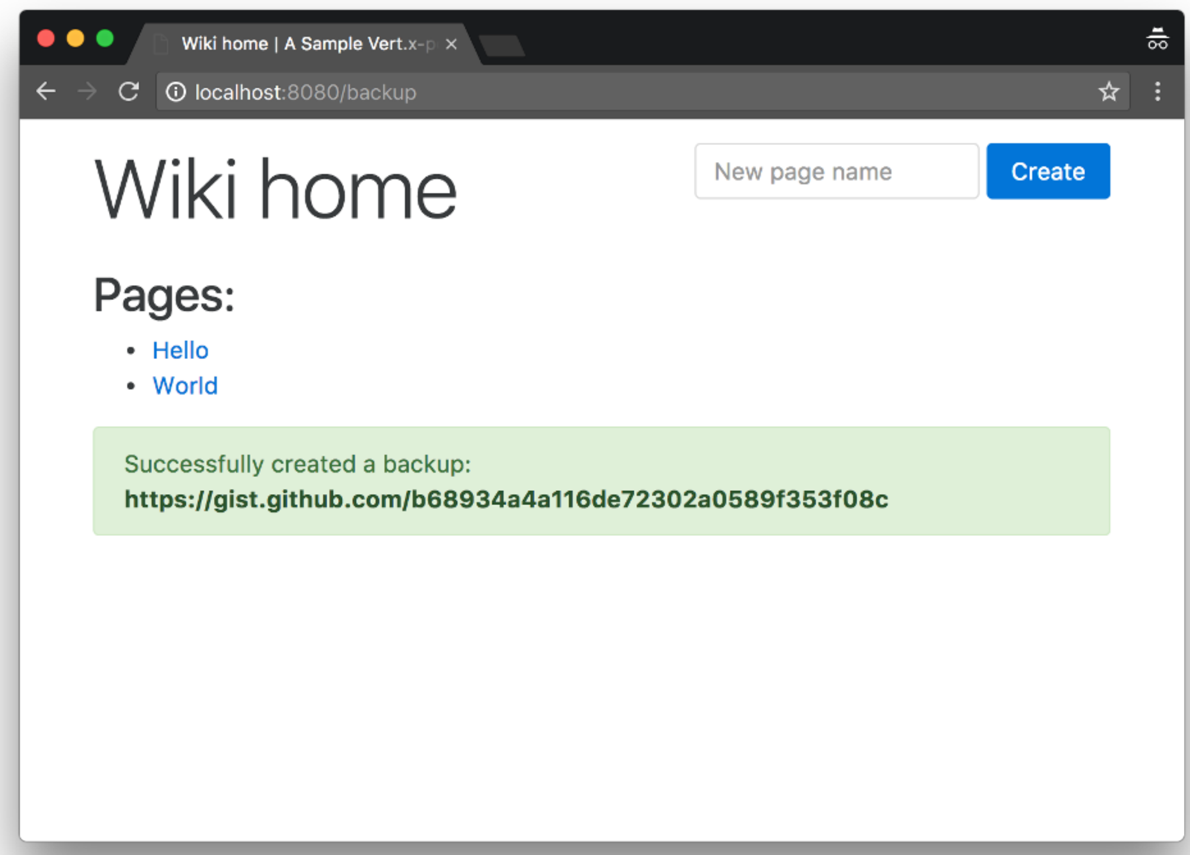
GitHub exposes a detailed API for fetching, creating, updating and deleting Gists. The API uses HTTPS endpoints starting at https://api.github.com/ and JSON payloads.

While many operations require authorization from the client using OAuth authentication, creating a Gist is possible while being anonymous. We are going to leverage this feature to backup our wiki pages as Gists.

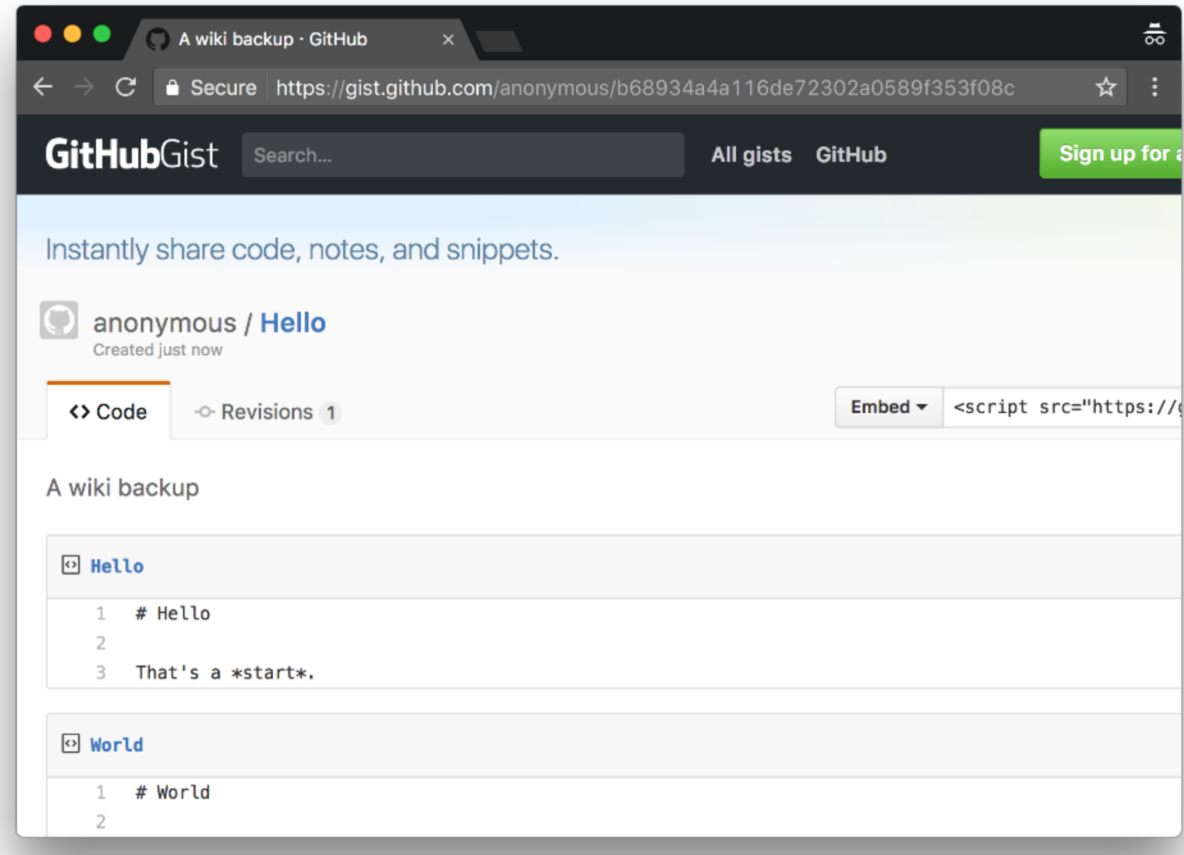
A new button is going to be added on the wiki index page:



Clicking the *backup* button will trigger the creating of a Gist:



A backup Gist consists of 1 file per wiki page, with the content being the raw Markdown text:



6.2. Updating the database service

Before we dive into the web client API and perform HTTP requests to another service, we need to update the database service API to fetch all the wiki pages data in one pass. This corresponds to the following SQL query to add to `db-queries.properties`:

```
all-pages-data=select * from Pages
```

A new method is added to the `WikiDatabaseService` interface:

```
@Fluent
WikiDatabaseService fetchAllPagesData(Handler<AsyncResult<List<JsonObject>>> resultHandler);
```

The implementation in `WikiDatabaseServiceImpl` is the following:

```java
@Override
public WikiDatabaseService fetchAllPagesData(Handler<AsyncResult<List<JsonObject>>> resultHandler) {
  dbClient.query(sqlQueries.get(SqlQuery.ALL_PAGES_DATA), queryResult -> {
    if (queryResult.succeeded()) {
      resultHandler.handle(Future.succeededFuture(queryResult.result().getRows()));
    } else {
      LOGGER.error("Database query error", queryResult.cause());
      resultHandler.handle(Future.failedFuture(queryResult.cause()));
    }
  });
  return this;
}
```

6.3. The web client API

The Vert.x core library offers a `createHttpClient` method from the `vertx` context object. Instances of `io.vertx.core.http.HttpClient` provides low-level methods for doing all kinds of HTTP requests with a fine-grained control over the protocol and the stream of events.

The web client API provides a simpler facade, especially for simplifying the payload data (un)marshaling. This API comes in the form of a new dependency:

```xml
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-web-client</artifactId>
  <version>${vertx.version}</version>
</dependency>
```

The following is a sample usage from a unit test. The test starts a HTTP server and then it does a HTTP GET request with the web client API to check that the request to the server succeeded:

```java
@Test
public void start_http_server(TestContext context) {
  Async async = context.async();

  vertx.createHttpServer().requestHandler(req ->
    req.response().putHeader("Content-Type", "text/plain").end("Ok"))
  .listen(8080, context.asyncAssertSuccess(server -> {

    WebClient webClient = WebClient.create(vertx);

      webClient.get(8080, "localhost", "/").send(ar -> {
        if (ar.succeeded()) {
          HttpResponse<Buffer> response = ar.result();
          context.assertTrue(response.headers().contains("Content-Type"));
          context.assertEquals("text/plain", response.getHeader("Content-Type"));
          context.assertEquals("Ok", response.body().toString());
          webClient.close();
          async.complete();
        } else {
          async.resolve(Future.failedFuture(ar.cause()));
        }
      });
    }));
}
```

6.4. Creating anonymous Gists

We first need a web client object to issue HTTP requests to the Gist API:

```java
webClient = WebClient.create(vertx, new WebClientOptions()
  .setSsl(true)
  .setUserAgent("vert-x3"));
```

Since requests are made using HTTPS, we need to configure the web client with SSL support.

The GitHub API requires a valid `User-Agent` header and requests a GitHub account or organization identifier. We override the default user agent with `vert-x3` but you may opt to use your own value instead.

We then modify the web router configuration in the `HttpServerVerticle` class to add a new route for triggering backups:

```java
router.get("/backup").handler(this::backupHandler);
```

The code for this handler is the following:

```
private void backupHandler(RoutingContext context) {
  dbService.fetchAllPagesData(reply -> {
    if (reply.succeeded()) {

      JsonObject filesObject = new JsonObject();
      JsonObject gistPayload = new JsonObject() ①
        .put("files", filesObject)
        .put("description", "A wiki backup")
        .put("public", true);

      reply
        .result()
        .forEach(page -> {
          JsonObject fileObject = new JsonObject(); ②
          filesObject.put(page.getString("NAME"), fileObject);
          fileObject.put("content", page.getString("CONTENT"));
        });

      webClient.post(443, "api.github.com", "/gists") ③
        .putHeader("Accept", "application/vnd.github.v3+json") ④
        .putHeader("Content-Type", "application/json")
        .as(BodyCodec.jsonObject()) ⑤
        .sendJsonObject(gistPayload, ar -> {  ⑥
          if (ar.succeeded()) {
            HttpResponse<JsonObject> response = ar.result();
            if (response.statusCode() == 201) {
              context.put("backup_gist_url", response.body().getString("html_url"));  ⑦
              indexHandler(context);
            } else {
              StringBuilder message = new StringBuilder()
                .append("Could not backup the wiki: ")
                .append(response.statusMessage());
              JsonObject body = response.body();
              if (body != null) {
                message.append(System.getProperty("line.separator"))
                  .append(body.encodePrettily());
              }
              LOGGER.error(message.toString());
              context.fail(502);
            }
          } else {
            Throwable err = ar.cause();
            LOGGER.error("HTTP Client error", err);
            context.fail(err);
          }
      });

    } else {
      context.fail(reply.cause());
```

```
    }
  });
}
```

① The Gist creation request payload is a JSON document as outlined in the GitHub API documentation.

② Each file is an entry under the `files` object of the payload, where the title is the key and the value is the text.

③ The web client needs to issue a `POST` request on port 443 (HTTPS), and the path must be `/gists`.

④ It is mandatory to have a `Accept` header in the request with the `application/vnd.github.v3+json` MIME type, otherwise the request fails. It is also important to specify that the payload is a JSON object at the next line.

⑤ The `BodyCodec` class provides a helper to specify that the response will be directly converted to a Vert.x `JsonObject` instance. It is also possible to use `BodyCodec#json(Class<T>)` and the JSON data will be mapped to a Java object of type `T` (this uses Jackson data mapping under the hood).

⑥ `sendJsonObject` is a helper for triggering the HTTP request with a JSON payload.

⑦ Upon success we can traverse the JSON data (`html_url` key) to get the user-friendly URL of the newly created Gist.

Chapter 7. Exposing a web API

The corresponding source code is in the `step-6` folder of the guide repository.

Exposing a web HTTP/JSON API is very straightforward using what we have already covered from the `vertx-web` module. We are going to expose a web API with the following URL scheme:

- `GET /api/pages` gives a document will all wiki page names and identifiers,
- `POST /api/pages` creates a new wiki page from a document,
- `PUT /api/pages/:id` updates a wiki page from a document,
- `DELETE /api/pages/:id` deletes a wiki page.

Here is a screenshot of interacting with the API using the HTTPie command-line tool:

```
monza:step-6 jponge$ http POST localhost:8080/api/pages name=Hello markdown="# Hello, world!"
HTTP/1.1 201 Created
Content-Length: 16
Content-Type: application/json

{
    "success": true
}

monza:step-6 jponge$ http localhost:8080/api/pages
HTTP/1.1 200 OK
Content-Length: 50
Content-Type: application/json

{
    "pages": [
        {
            "id": 0,
            "name": "Hello"
        }
    ],
    "success": true
}

monza:step-6 jponge$ http localhost:8080/api/pages/0
HTTP/1.1 200 OK
Content-Length: 110
Content-Type: application/json

{
    "page": {
        "html": "<h1>Hello, world!</h1>\n",
        "id": 0,
        "markdown": "# Hello, world!",
        "name": "Hello"
    },
    "success": true
}

monza:step-6 jponge$
```

7.1. Web sub-routers

We are going to add new route handlers to the `HttpServerVerticle`. While we could just add handlers to the existing router, we can also take advantage of *sub-routers*. They allow a router to be mounted as a sub-router of another one, which can be useful for organizing and/or re-using handlers.

Here is the code for the API router:

```
Router apiRouter = Router.router(vertx);
apiRouter.get("/pages").handler(this::apiRoot);
apiRouter.get("/pages/:id").handler(this::apiGetPage);
apiRouter.post().handler(BodyHandler.create());
apiRouter.post("/pages").handler(this::apiCreatePage);
apiRouter.put().handler(BodyHandler.create());
apiRouter.put("/pages/:id").handler(this::apiUpdatePage);
apiRouter.delete("/pages/:id").handler(this::apiDeletePage);
router.mountSubRouter("/api", apiRouter); ①
```

① This is where we mount our router, so requests starting with the `/api` path will be directed to `apiRouter`.

7.2. Handlers

Here is the code for the different API router handlers.

7.2.1. Root resource

```
private void apiRoot(RoutingContext context) {
  dbService.fetchAllPagesData(reply -> {
    JsonObject response = new JsonObject();
    if (reply.succeeded()) {
      List<JsonObject> pages = reply.result()
        .stream()
        .map(obj -> new JsonObject()
          .put("id", obj.getInteger("ID"))  ①
          .put("name", obj.getString("NAME")))
        .collect(Collectors.toList());
      response
        .put("success", true)
        .put("pages", pages); ②
      context.response().setStatusCode(200);
      context.response().putHeader("Content-Type", "application/json");
      context.response().end(response.encode()); ③
    } else {
      response
        .put("success", false)
        .put("error", reply.cause().getMessage());
      context.response().setStatusCode(500);
      context.response().putHeader("Content-Type", "application/json");
      context.response().end(response.encode());
    }
  });
}
```

① We just remap database results in page information entry objects.

② The resulting JSON array becomes the value for the `pages` key in the response payload.

③ `JsonObject#encode()` gives a compact `String` representation of the JSON data.

7.2.2. Getting a page

```
private void apiGetPage(RoutingContext context) {
  int id = Integer.valueOf(context.request().getParam("id"));
  dbService.fetchPageById(id, reply -> {
    JsonObject response = new JsonObject();
    if (reply.succeeded()) {
      JsonObject dbObject = reply.result();
      if (dbObject.getBoolean("found")) {
        JsonObject payload = new JsonObject()
          .put("name", dbObject.getString("name"))
          .put("id", dbObject.getInteger("id"))
          .put("markdown", dbObject.getString("content"))
          .put("html", Processor.process(dbObject.getString("content")));
        response
          .put("success", true)
          .put("page", payload);
        context.response().setStatusCode(200);
      } else {
        context.response().setStatusCode(404);
        response
          .put("success", false)
          .put("error", "There is no page with ID " + id);
      }
    } else {
      response
        .put("success", false)
        .put("error", reply.cause().getMessage());
      context.response().setStatusCode(500);
    }
    context.response().putHeader("Content-Type", "application/json");
    context.response().end(response.encode());
  });
}
```

7.2.3. Creating a page

```java
private void apiCreatePage(RoutingContext context) {
  JsonObject page = context.getBodyAsJson();
  if (!validateJsonPageDocument(context, page, "name", "markdown")) {
    return;
  }
  dbService.createPage(page.getString("name"), page.getString("markdown"), reply -> {
    if (reply.succeeded()) {
      context.response().setStatusCode(201);
      context.response().putHeader("Content-Type", "application/json");
      context.response().end(new JsonObject().put("success", true).encode());
    } else {
      context.response().setStatusCode(500);
      context.response().putHeader("Content-Type", "application/json");
      context.response().end(new JsonObject()
        .put("success", false)
        .put("error", reply.cause().getMessage()).encode());
    }
  });
}
```

This handler and other handlers need to deal with incoming JSON documents. The following `validateJsonPageDocument` method is a helper for doing validation and early error reporting, so that the remainder of the processing assume the presence of certain JSON entries:

```java
private boolean validateJsonPageDocument(RoutingContext context, JsonObject page, String... expectedKeys) {
  if (!Arrays.stream(expectedKeys).allMatch(page::containsKey)) {
    LOGGER.error("Bad page creation JSON payload: " + page.encodePrettily() + " from " + context.request().
remoteAddress());
    context.response().setStatusCode(400);
    context.response().putHeader("Content-Type", "application/json");
    context.response().end(new JsonObject()
      .put("success", false)
      .put("error", "Bad request payload").encode());
    return false;
  }
  return true;
}
```

7.2.4. Updating a page

```java
private void apiUpdatePage(RoutingContext context) {
  int id = Integer.valueOf(context.request().getParam("id"));
  JsonObject page = context.getBodyAsJson();
  if (!validateJsonPageDocument(context, page, "markdown")) {
    return;
  }
  dbService.savePage(id, page.getString("markdown"), reply -> {
    handleSimpleDbReply(context, reply);
  });
}
```

The `handleSimpleDbReply` method is a helper for finishing the request processing:

```java
private void handleSimpleDbReply(RoutingContext context, AsyncResult<Void> reply) {
  if (reply.succeeded()) {
    context.response().setStatusCode(200);
    context.response().putHeader("Content-Type", "application/json");
    context.response().end(new JsonObject().put("success", true).encode());
  } else {
    context.response().setStatusCode(500);
    context.response().putHeader("Content-Type", "application/json");
    context.response().end(new JsonObject()
      .put("success", false)
      .put("error", reply.cause().getMessage()).encode());
  }
}
```

7.2.5. Deleting a page

```java
private void apiDeletePage(RoutingContext context) {
  int id = Integer.valueOf(context.request().getParam("id"));
  dbService.deletePage(id, reply -> {
    handleSimpleDbReply(context, reply);
  });
}
```

7.3. Unit testing the API

We write a basic test case in the `io.vertx.guides.wiki.http.ApiTest` class.

The preamble consists in preparing the test environment. The HTTP server verticle needs the database verticle to be running, so we need to deploy both in our test Vert.x context:

```java
@RunWith(VertxUnitRunner.class)
public class ApiTest {

  private Vertx vertx;
  private WebClient webClient;

  @Before
  public void prepare(TestContext context) {
    vertx = Vertx.vertx();

    JsonObject dbConf = new JsonObject()
      .put(WikiDatabaseVerticle.CONFIG_WIKIDB_JDBC_URL, "jdbc:hsqldb:mem:testdb;shutdown=true") ①
      .put(WikiDatabaseVerticle.CONFIG_WIKIDB_JDBC_MAX_POOL_SIZE, 4);

    vertx.deployVerticle(new WikiDatabaseVerticle(),
      new DeploymentOptions().setConfig(dbConf), context.asyncAssertSuccess());

    vertx.deployVerticle(new HttpServerVerticle(), context.asyncAssertSuccess());

    webClient = WebClient.create(vertx, new WebClientOptions()
      .setDefaultHost("localhost")
      .setDefaultPort(8080));
  }

  @After
  public void finish(TestContext context) {
    vertx.close(context.asyncAssertSuccess());
  }

  // (...)
```

① We use a different JDBC URL to use an in-memory database for the tests.

The proper test case is a simple scenario where all types of requests are being made. It creates a page, fetches it, updates it then deletes it:

```java
@Test
public void play_with_api(TestContext context) {
  Async async = context.async();

  JsonObject page = new JsonObject()
    .put("name", "Sample")
    .put("markdown", "# A page");

  Future<JsonObject> postRequest = Future.future();
  webClient.post("/api/pages")
    .as(BodyCodec.jsonObject())
    .sendJsonObject(page, ar -> {
      if (ar.succeeded()) {
        HttpResponse<JsonObject> postResponse = ar.result();
        postRequest.complete(postResponse.body());
      } else {
```

```
        context.fail(ar.cause());
      }
    });

  Future<JsonObject> getRequest = Future.future();
  postRequest.compose(h -> {
    webClient.get("/api/pages")
      .as(BodyCodec.jsonObject())
      .send(ar -> {
        if (ar.succeeded()) {
          HttpResponse<JsonObject> getResponse = ar.result();
          getRequest.complete(getResponse.body());
        } else {
          context.fail(ar.cause());
        }
      });
  }, getRequest);

  Future<JsonObject> putRequest = Future.future();
  getRequest.compose(response -> {
    JsonArray array = response.getJsonArray("pages");
    context.assertEquals(1, array.size());
    context.assertEquals(0, array.getJsonObject(0).getInteger("id"));
    webClient.put("/api/pages/0")
      .as(BodyCodec.jsonObject())
      .sendJsonObject(new JsonObject()
        .put("id", 0)
        .put("markdown", "Oh Yeah!"), ar -> {
        if (ar.succeeded()) {
          HttpResponse<JsonObject> putResponse = ar.result();
          putRequest.complete(putResponse.body());
        } else {
          context.fail(ar.cause());
        }
      });
  }, putRequest);

  Future<JsonObject> deleteRequest = Future.future();
  putRequest.compose(response -> {
    context.assertTrue(response.getBoolean("success"));
    webClient.delete("/api/pages/0")
      .as(BodyCodec.jsonObject())
      .send(ar -> {
        if (ar.succeeded()) {
          HttpResponse<JsonObject> delResponse = ar.result();
          deleteRequest.complete(delResponse.body());
        } else {
          context.fail(ar.cause());
        }
      });
  }, deleteRequest);
```

```
  deleteRequest.compose(response -> {
    context.assertTrue(response.getBoolean("success"));
    async.complete();
  }, Future.failedFuture("Oh?"));
}
```

The test uses `Future` objects composition rather than nested callbacks; the last composition must complete the async future or the test will eventually time out.

Chapter 8. Securing and controlling access

The corresponding source code is in the `step-7` folder of the guide repository.

Securing and controlling access is easy to do with Vert.x. In this section, we will:

1. move from HTTP to HTTPS, and
2. add user authentication with group-based privileges to the web application, and
3. control access to the web API using *JSON web tokens* (JWT).

8.1. HTTPS support in Vert.x

Vert.x provides support for SSL-encrypted network connections. It is common to expose HTTP servers in production through a front HTTP server / proxy like Nginx, and have it use HTTPS for incoming connections. Vert.x can also expose HTTPS by itself, so as to provide end-to-end encryption.

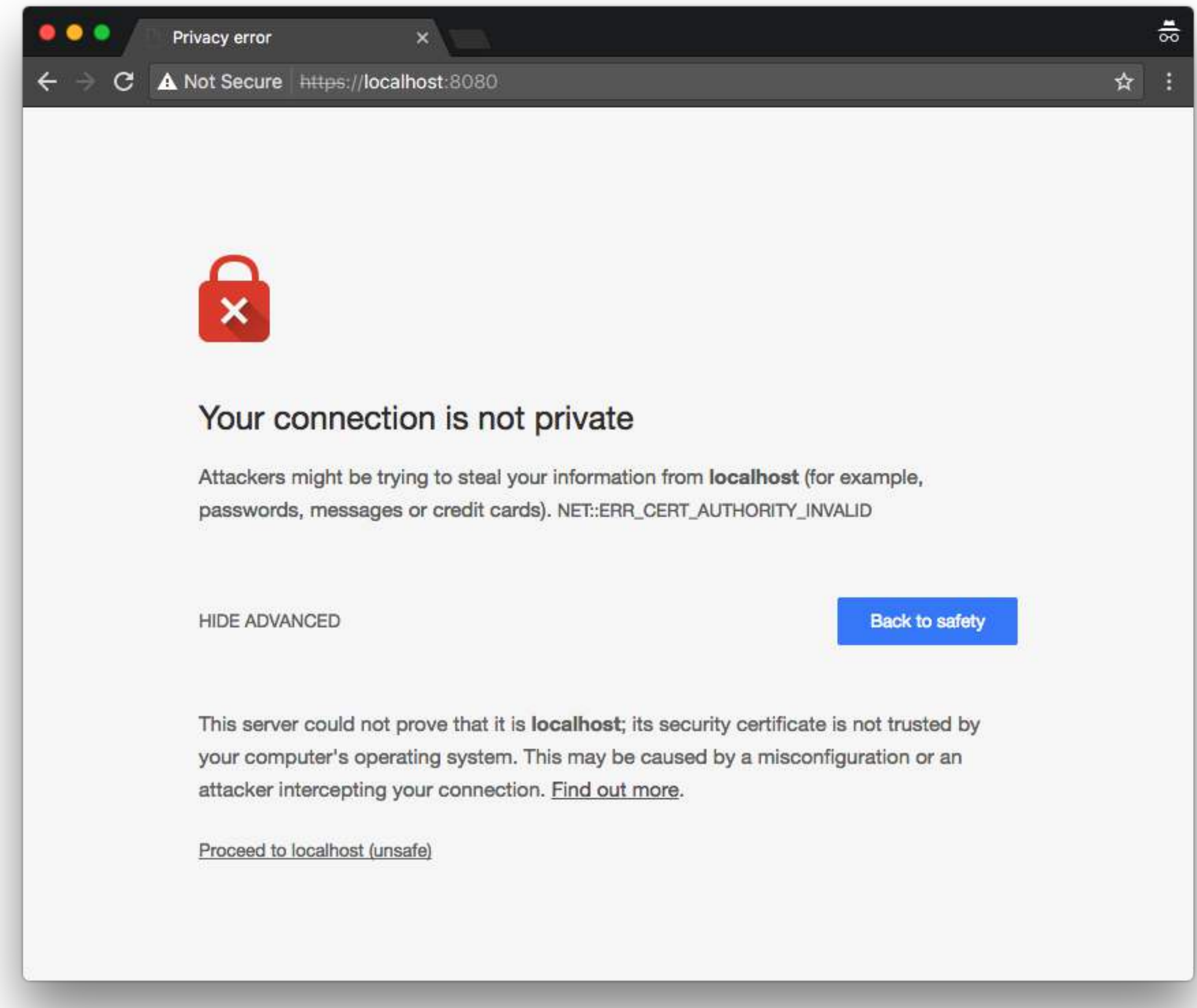
Certificates can be stored in Java *KeyStore* files. You will likely need a *self-signed certificate* for testing purposes, and here is how to create one in a `server-keystore.jks` KeyStore where the password is `secret`:

```
keytool -genkey \
  -alias test \
  -keyalg RSA \
  -keystore server-keystore.jks \
  -keysize 2048 \
  -validity 360 \
  -dname CN=localhost \
  -keypass secret \
  -storepass secret
```

We can then change the HTTP server creation to pass a `HttpServerOptions` object to specify that we want SSL, and to point to our KeyStore file:

```
HttpServer server = vertx.createHttpServer(new HttpServerOptions()
  .setSsl(true)
  .setKeyStoreOptions(new JksOptions()
    .setPath("server-keystore.jks")
    .setPassword("secret")));
```

We can point a web browser to https://localhost:8080/, but since the certificate is a self-signed one any good browser will rightfully yield a security warning:



Last but not least, we need to update the test case in `ApiTest` since the original code was made for issuing HTTP requests with the web client:

```
webClient = WebClient.create(vertx, new WebClientOptions()
  .setDefaultHost("localhost")
  .setDefaultPort(8080)
  .setSsl(true) ①
  .setTrustOptions(new JksOptions().setPath("server-keystore.jks").setPassword("secret"))); ②
```

① Ensures SSL.

② Since the certificate is self-signed, we need to explicitly trust it otherwise the web client connections will fail just like a web browser would.

8.2. Access control and authentication

Vert.x provides a wide range of options for doing authentication and authorization. The officially supported modules cover JDBC, MongoDB, Apache Shiro, OAuth2 with well-known providers and JWT (JSON web tokens).

While the next section will cover JWT, this one focuses on using Apache Shiro which is especially

interesting when authentication must be backed by a LDAP or Active Directory server. In our case we will simply store the credentials in a properties file to keep things simple, but the API usage against a LDAP server remains the same.

The goal is to require users to authenticate for using the wiki, and have role-based permissions:

- having no role only allows reading pages,
- having a *writer* role allows editing pages,
- having an *editor* role allows creating, editing and deleting pages,
- having an *admin* role is equivalent to having all possible roles.

The Vert.x Shiro integration has some issues due to the inner workings of Apache Shiro. Some parts are blocking which can hinder performance, and some data is stored using thread local state. You should not try abusing the exposed internal state APIs either.

8.2.1. Adding Apache Shiro authentication to routes

The first step is to add the `vertx-auth-shiro` module to the Maven dependencies list:

```
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-auth-shiro</artifactId>
</dependency>
```

The properties file definition that we use is the following, located in `src/main/resources/wiki-users.properties`:

```
user.root=w00t,admin
user.foo=bar,editor,writer
user.bar=baz,writer
user.baz=baz

role.admin=*
role.editor=create,delete,update
role.writer=update
```

A `user`-prefixed entry is a user account, where the first value entry is the password possibly followed by roles. In this example user `bar` has password `baz`, is a `writer`, and a `writer` has `update` permission.

Back to the `HttpServerVerticle` class code, we create an authentication provider using Apache Shiro:

```
AuthProvider auth = ShiroAuth.create(vertx, new ShiroAuthOptions()
  .setType(ShiroAuthRealmType.PROPERTIES)
  .setConfig(new JsonObject()
    .put("properties_path", "classpath:wiki-users.properties")));
```

The `ShiroAuth` object instance is then used to deal with server-side user sessions:

```
Router router = Router.router(vertx);

router.route().handler(CookieHandler.create());
router.route().handler(BodyHandler.create());
router.route().handler(SessionHandler.create(LocalSessionStore.create(vertx)));
router.route().handler(UserSessionHandler.create(auth));  ①

AuthHandler authHandler = RedirectAuthHandler.create(auth, "/login"); ②
router.route("/").handler(authHandler);  ③
router.route("/wiki/*").handler(authHandler);
router.route("/action/*").handler(authHandler);

router.get("/").handler(this::indexHandler);
router.get("/wiki/:page").handler(this::pageRenderingHandler);
router.post("/action/save").handler(this::pageUpdateHandler);
router.post("/action/create").handler(this::pageCreateHandler);
router.get("/action/backup").handler(this::backupHandler);
router.post("/action/delete").handler(this::pageDeletionHandler);
```

① We install a user session handler for all routes.

② This automatically redirects requests to `/login` when there is no user session for the request.

③ We install `authHandler` for all routes where authentication is required.

Finally, we need to create 3 routes for displaying a login form, handling login form submissions and logging out users:

```
router.get("/login").handler(this::loginHandler);
router.post("/login-auth").handler(FormLoginHandler.create(auth));  ①

router.get("/logout").handler(context -> {
  context.clearUser();  ②
  context.response()
    .setStatusCode(302)
    .putHeader("Location", "/")
    .end();
});
```

① `FormLoginHandler` is a helper for processing login submission requests. By default it expects the HTTP POST request to have: `username` as the login, `password` as the password, and `return_url` as the URL to redirect to upon success.

② Logging out a user is a simple as clearing it from the current `RoutingContext`.

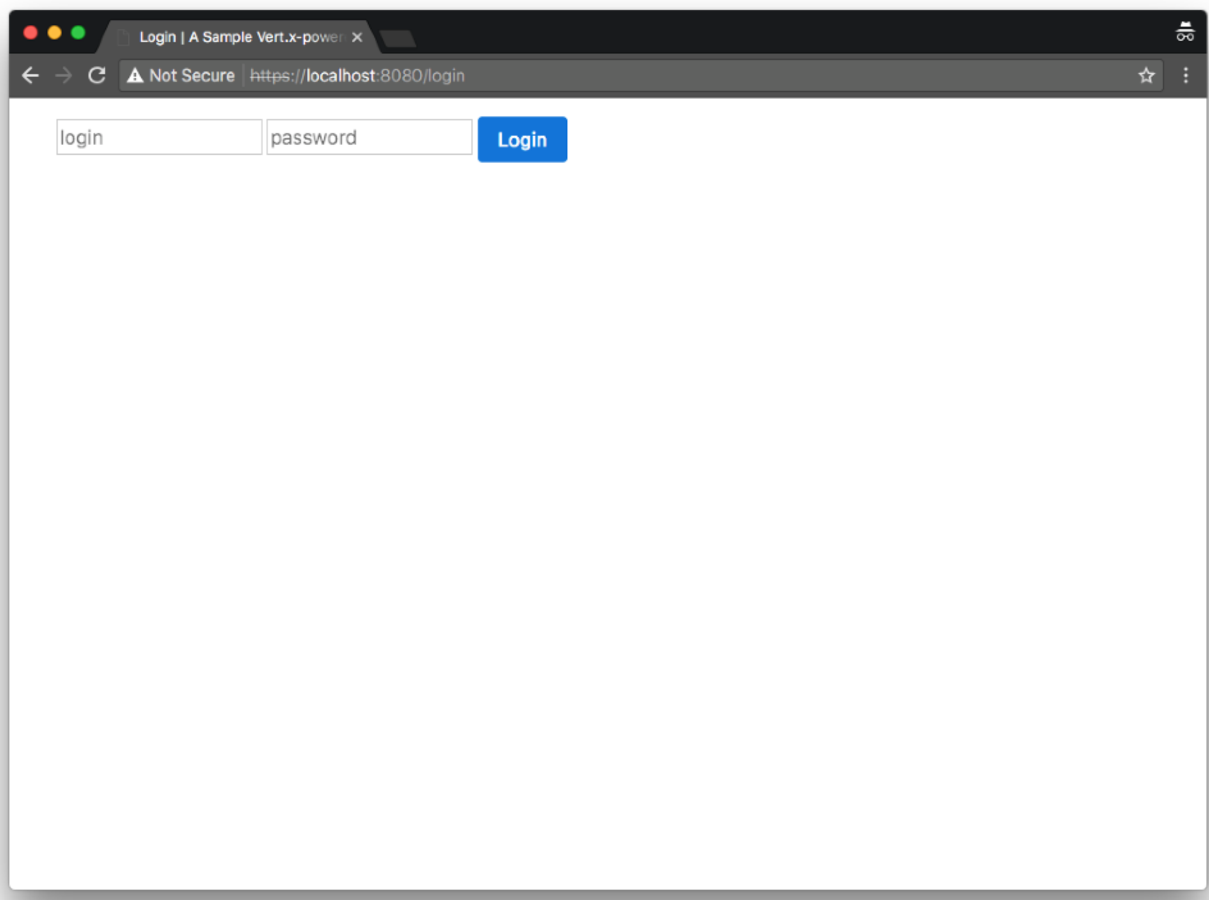
The code for the `loginHandler` method is:

```java
private void loginHandler(RoutingContext context) {
  context.put("title", "Login");
  templateEngine.render(context, "templates", "/login.ftl", ar -> {
    if (ar.succeeded()) {
      context.response().putHeader("Content-Type", "text/html");
      context.response().end(ar.result());
    } else {
      context.fail(ar.cause());
    }
  });
}
```

The HTML template is placed in `src/main/resources/templates/login.ftl`:

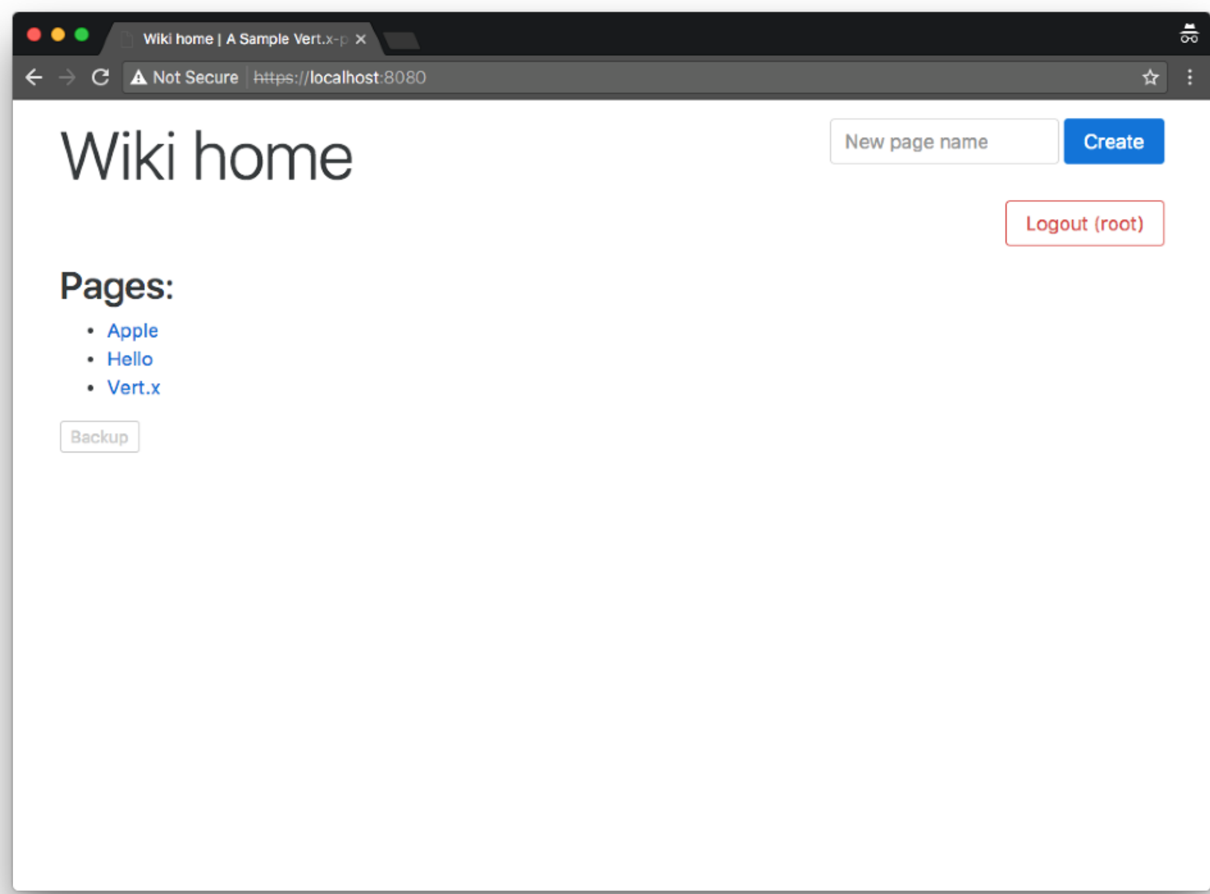
```html
<#include "header.ftl">

<div class="row">

  <div class="col-md-12 mt-1">
    <form action="/login-auth" method="POST">
      <div class="form-group">
        <input type="text" name="username" placeholder="login">
        <input type="password" name="password" placeholder="password">
        <input type="hidden" name="return_url" value="/">
        <button type="submit" class="btn btn-primary">Login</button>
      </div>
    </form>
  </div>

</div>

<#include "footer.ftl">
```

The login page looks as follows:

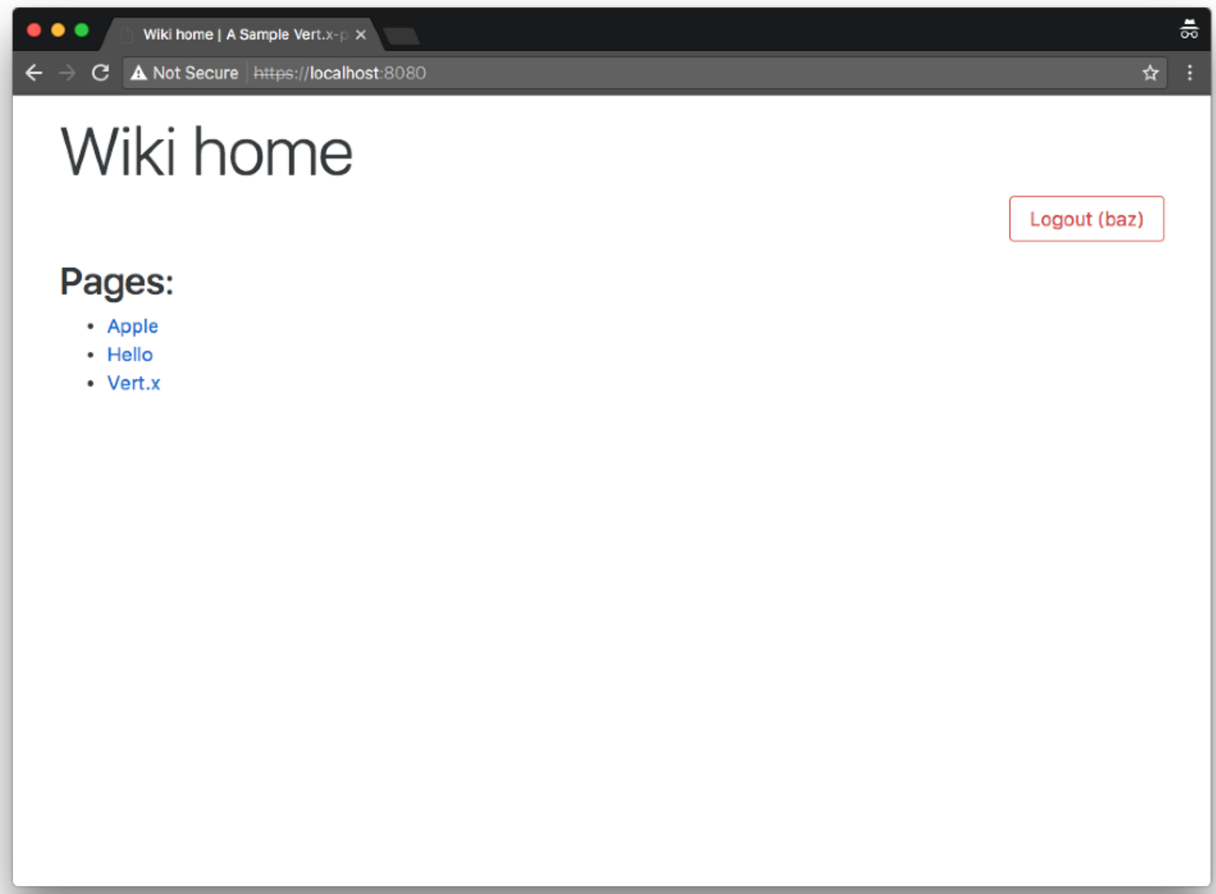


8.2.2. Supporting features based on roles

Features need to be activated only if the user has sufficient permissions. In the following screenshot an administrator can create a page and perform a backup:



By contrast a user with no role cannot perform these actions:



To do that, we can access the `RoutingContext` user reference, and query for permissions. Here is how this is implemented for the `indexHandler` handler method:

```java
private void indexHandler(RoutingContext context) {
  context.user().isAuthorised("create", res -> {  ①
    boolean canCreatePage = res.succeeded() && res.result();  ②
    dbService.fetchAllPages(reply -> {
      if (reply.succeeded()) {
        context.put("title", "Wiki home");
        context.put("pages", reply.result().getList());
        context.put("canCreatePage", canCreatePage);  ③
        context.put("username", context.user().principal().getString("username"));  ④
        templateEngine.render(context, "templates", "/index.ftl", ar -> {
          if (ar.succeeded()) {
            context.response().putHeader("Content-Type", "text/html");
            context.response().end(ar.result());
          } else {
            context.fail(ar.cause());
          }
        });
      } else {
        context.fail(reply.cause());
      }
    });
  });
}
```

① This is how a permission query is made. Note that this is an asynchronous operation.

② We use the result to…

③ …leverage it in the HTML template.

④ We also have access to the user login.

The template code has been modified to only render certain fragments based on the value of `canCreatePage`:

```
<#include "header.ftl">

<div class="row">

  <div class="col-md-12 mt-1">
  <#if context.canCreatePage>
    <div class="float-xs-right">
      <form class="form-inline" action="/action/create" method="post">
        <div class="form-group">
          <input type="text" class="form-control" id="name" name="name" placeholder="New page name">
        </div>
        <button type="submit" class="btn btn-primary">Create</button>
      </form>
    </div>
  </#if>
    <h1 class="display-4">${context.title}</h1>
    <div class="float-xs-right">
      <a class="btn btn-outline-danger" href="/logout" role="button" aria-pressed="true">Logout (${context.username})</a>
    </div>
  </div>

  <div class="col-md-12 mt-1">
  <#list context.pages>
    <h2>Pages:</h2>
    <ul>
      <#items as page>
        <li><a href="/wiki/${page}">${page}</a></li>
      </#items>
    </ul>
  <#else>
    <p>The wiki is currently empty!</p>
  </#list>

  <#if context.canCreatePage>
    <#if context.backup_gist_url?has_content>
      <div class="alert alert-success" role="alert">
        Successfully created a backup:
        <a href="${context.backup_gist_url}" class="alert-link">${context.backup_gist_url}</a>
      </div>
    <#else>
      <p>
        <a class="btn btn-outline-secondary btn-sm" href="/action/backup" role="button" aria-pressed="true">Backup</a>
      </p>
    </#if>
  </#if>
  </div>

</div>

<#include "footer.ftl">
```

The code is similar for ensuring that updating or deleting a page is restricted to certain roles and is available from the guide Git repository.

It is important to ensure that checks are also being done on HTTP POST request handlers and not just when rendering HTML pages. Indeed, malicious attackers could still craft requests and perform actions while not being authenticated. Here is how to protect page deletions by wrapping the `pageDeletionHandler` code inside a topmost permission check:

```java
private void pageDeletionHandler(RoutingContext context) {
  context.user().isAuthorised("delete", res -> {
    if (res.succeeded() && res.result()) {

      // Original code:
      dbService.deletePage(Integer.valueOf(context.request().getParam("id")), reply -> {
        if (reply.succeeded()) {
          context.response().setStatusCode(303);
          context.response().putHeader("Location", "/");
          context.response().end();
        } else {
          context.fail(reply.cause());
        }
      });

    } else {
      context.response().setStatusCode(403).end();
    }
  });
}
```

8.3. Authenticating web API requests with JWT

JSON Web Tokens (RFC 7519) is a standard for issuing JSON-encoded tokens containing *claims*, typically identifying a user and permissions, although claims can be just about anything.

A token is issued by a server and it is signed with the server key. A client can send a token back along with subsequent requests: both the client and the server can check that a token is authentic and unaltered.

While a JWT token is signed, its content is not encrypted. It must be transported over a secure channel (e.g., HTTPS) and it should never have sensitive data as a claim (e.g., passwords, private API keys, etc).

8.3.1. Adding JWT support

We start by adding the `vertx-auth-jwt` module to the Maven dependencies:

```xml
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-auth-jwt</artifactId>
</dependency>
```

We will have a JCEKS keystore to hold the keys for our tests. Here is how to generate a `keystore.jceks` with the suitable keys of various lengths:

```
keytool -genseckey -keystore keystore.jceks -storetype jceks -storepass secret -keyalg HMacSHA256 -keysize 2048 -alias
HS256 -keypass secret
keytool -genseckey -keystore keystore.jceks -storetype jceks -storepass secret -keyalg HMacSHA384 -keysize 2048 -alias
HS384 -keypass secret
keytool -genseckey -keystore keystore.jceks -storetype jceks -storepass secret -keyalg HMacSHA512 -keysize 2048 -alias
HS512 -keypass secret
keytool -genkey -keystore keystore.jceks -storetype jceks -storepass secret -keyalg RSA -keysize 2048 -alias RS256
-keypass secret -sigalg SHA256withRSA -dname "CN=,OU=,O=,L=,ST=,C=" -validity 360
keytool -genkey -keystore keystore.jceks -storetype jceks -storepass secret -keyalg RSA -keysize 2048 -alias RS384
-keypass secret -sigalg SHA384withRSA -dname "CN=,OU=,O=,L=,ST=,C=" -validity 360
keytool -genkey -keystore keystore.jceks -storetype jceks -storepass secret -keyalg RSA -keysize 2048 -alias RS512
-keypass secret -sigalg SHA512withRSA -dname "CN=,OU=,O=,L=,ST=,C=" -validity 360
keytool -genkeypair -keystore keystore.jceks -storetype jceks -storepass secret -keyalg EC -keysize 256 -alias ES256
-keypass secret -sigalg SHA256withECDSA -dname "CN=,OU=,O=,L=,ST=,C=" -validity 360
keytool -genkeypair -keystore keystore.jceks -storetype jceks -storepass secret -keyalg EC -keysize 256 -alias ES384
-keypass secret -sigalg SHA384withECDSA -dname "CN=,OU=,O=,L=,ST=,C=" -validity 360
keytool -genkeypair -keystore keystore.jceks -storetype jceks -storepass secret -keyalg EC -keysize 256 -alias ES512
-keypass secret -sigalg SHA512withECDSA -dname "CN=,OU=,O=,L=,ST=,C=" -validity 360
```

We need to install a JWT token handler on API routes:

```
Router apiRouter = Router.router(vertx);

JWTAuth jwtAuth = JWTAuth.create(vertx, new JsonObject()
  .put("keyStore", new JsonObject()
    .put("path", "keystore.jceks")
    .put("type", "jceks")
    .put("password", "secret")));

apiRouter.route().handler(JWTAuthHandler.create(jwtAuth, "/api/token"));
```

We pass `/api/token` as a parameter for the `JWTAuthHandler` object creation to specify that this URL shall be ignored. Indeed, this URL is being used to generate new JWT tokens:

```java
apiRouter.get("/token").handler(context -> {

  JsonObject creds = new JsonObject()
    .put("username", context.request().getHeader("login"))
    .put("password", context.request().getHeader("password"));
  auth.authenticate(creds, authResult -> {  ①

    if (authResult.succeeded()) {
      User user = authResult.result();
      user.isAuthorised("create", canCreate -> {  ②
        user.isAuthorised("delete", canDelete -> {
          user.isAuthorised("update", canUpdate -> {

            String token = jwtAuth.generateToken( ③
              new JsonObject()
                .put("username", context.request().getHeader("login"))
                .put("canCreate", canCreate.succeeded() && canCreate.result())
                .put("canDelete", canDelete.succeeded() && canDelete.result())
                .put("canUpdate", canUpdate.succeeded() && canUpdate.result()),
              new JWTOptions()
                .setSubject("Wiki API")
                .setIssuer("Vert.x"));
            context.response().putHeader("Content-Type", "text/plain").end(token);
          });
        });
      });
    } else {
      context.fail(401);
    }
  });
});
```

① We expect login and password information to have been passed through HTTP request headers, and we authenticate using the Apache Shiro authentication provider of the previous section.

② Upon success we can query for roles.

③ We generate a token with `username`, `canCreate`, `canDelete` and `canUpdate` claims.

Each API handler method can now query the current user principal and claims. Here is how the `apiDeletePage` does it:

```
private void apiDeletePage(RoutingContext context) {
  if (context.user().principal().getBoolean("canDelete", false)) {
    int id = Integer.valueOf(context.request().getParam("id"));
    dbService.deletePage(id, reply -> {
      handleSimpleDbReply(context, reply);
    });
  } else {
    context.fail(401);
  }
}
```

8.3.2. Using JWT tokens

To illustrate how to work with JWT tokens, let’s create a new one for the `root` user:

```
$ http --verbose --verify no GET https://localhost:8080/api/token login:root password:w00t
GET /api/token HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8080
User-Agent: HTTPie/0.9.8
login: root
password: w00t



HTTP/1.1 200 OK
Content-Length: 242
Content-Type: text/plain
Set-Cookie: vertx-web.session=8cbb38ac4ce96737bfe31cc0ceaae2b9; Path=/

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VybmFtZSI6InJvb3QiLCJjYW5DcmVhdGUiOnRydWUsImNhbkRlbGV0ZSI6dHJ1ZSwiY2FuVXBkYXRl
Ijp0cnVlLCJpYXQiOjE0ODk0NDE1OTAsImlzcyI6IlZlcnQueCIsInN1YiI6Ildpa2kgQVBJIn0=.RmtJb81QKVUFreXL-ajZ8ktLGasoKEqG8GSQncRWrN8=
```

The response text is the token value and shall be retained.

We can check that performing an API request without the token results in a denial of access:

```
$ http --verbose --verify no GET https://localhost:8080/api/pages
GET /api/pages HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8080
User-Agent: HTTPie/0.9.8



HTTP/1.1 401 Unauthorized
Content-Length: 12

Unauthorized
```

Sending a JWT token along with a request is done using a `Authorization` HTTP request header where the value must be `Bearer <token value>`. Here is how to fix the API request above by passing the JWT token that had been issued to us:

```
$ http --verbose --verify no GET https://localhost:8080/api/pages 'Authorization: Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VybmFtZSI6InJvb3QiLCJjYW5DcmVhdGUiOnRydWUsImNhbkRlbGV0ZSI6dHJ1ZSwiY2FuVXBkYXRl
Ijp0cnVlLCJpYXQiOjE0ODk0NDE1OTAsImlzcyI6IlZlcnQueCIsInN1YiI6Ildpa2kgQVBJIn0=.RmtJb81QKVUFreXL-
ajZ8ktLGasoKEqG8GSQncRWrN8='
GET /api/pages HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Authorization: Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VybmFtZSI6InJvb3QiLCJjYW5DcmVhdGUiOnRydWUsImNhbkRlbGV0ZSI6dHJ1ZSwiY2FuVXBkYXRl
Ijp0cnVlLCJpYXQiOjE0ODk0NDE1OTAsImlzcyI6IlZlcnQueCIsInN1YiI6Ildpa2kgQVBJIn0=.RmtJb81QKVUFreXL-ajZ8ktLGasoKEqG8GSQncRWrN8=
Connection: keep-alive
Host: localhost:8080
User-Agent: HTTPie/0.9.8



HTTP/1.1 200 OK
Content-Length: 99
Content-Type: application/json
Set-Cookie: vertx-web.session=0598697483371c7f3cb434fbe35f15e4; Path=/

{
    "pages": [
        {
            "id": 0,
            "name": "Hello"
        },
        {
            "id": 1,
            "name": "Apple"
        },
        {
            "id": 2,
            "name": "Vert.x"
        }
    ],
    "success": true
}
```

8.3.3. Adapting the API test fixture

The `ApiTest` class needs to be updated to support JWT tokens.

We add a new field for retrieving the token value to be used in test cases:

```
private String jwtTokenHeaderValue;
```

We add first step to retrieve a JTW token authenticated as user `foo`:

```
@Test
public void play_with_api(TestContext context) {
  Async async = context.async();

  Future<String> tokenRequest = Future.future();
  webClient.get("/api/token")
    .putHeader("login", "foo")  ①
    .putHeader("password", "bar")
    .as(BodyCodec.string()) ②
    .send(ar -> {
      if (ar.succeeded()) {
        tokenRequest.complete(ar.result().body());  ③
      } else {
        context.fail(ar.cause());
      }
    });
    // (...)
```

① Credentials are passed as headers.

② The response payload is of `text/plain` type, so we use that for the body decoding codec.

③ Upon success we complete the `tokenRequest` future with the token value.

Using the JWT token is now a matter of passing it back as a header to HTTP requests:

```
Future<JsonObject> postRequest = Future.future();
tokenRequest.compose(token -> {
  jwtTokenHeaderValue = "Bearer " + token;  ①
  webClient.post("/api/pages")
    .putHeader("Authorization", jwtTokenHeaderValue)  ②
    .as(BodyCodec.jsonObject())
    .sendJsonObject(page, ar -> {
      if (ar.succeeded()) {
        HttpResponse<JsonObject> postResponse = ar.result();
        postRequest.complete(postResponse.body());
      } else {
        context.fail(ar.cause());
      }
    });
}, postRequest);

Future<JsonObject> getRequest = Future.future();
postRequest.compose(h -> {
  webClient.get("/api/pages")
    .putHeader("Authorization", jwtTokenHeaderValue)
    .as(BodyCodec.jsonObject())
    .send(ar -> {
      if (ar.succeeded()) {
        HttpResponse<JsonObject> getResponse = ar.result();
        getRequest.complete(getResponse.body());
      } else {
        context.fail(ar.cause());
      }
    });
}, getRequest);
// (...)
```

① We store the token with the `Bearer` prefix to the field for the next requests.

② We pass the token as a header.

Chapter 9. Reactive programming with RxJava

The corresponding source code is in the `step-8` folder of the guide repository.

So far we have explored various areas of the Vert.x stack, using the callback-based APIs. It just works and this programming model is well-known from developers in many languages. However, it can become a bit tedious, especially when you combine several sources of events, or deal with complex data flows.

This is exactly where RxJava shines, and Vert.x integrates with it seamlessly.

9.1. Enabling the RxJava APIs

In addition to the callback-based API, the Vert.x modules offer an *"Rxified"* API. To enable it, start by adding the `vertx-rx-java` module to the Maven POM file:

```
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-rx-java</artifactId>
</dependency>
```

Verticles then have to be modified so that they extend `io.vertx.rxjava.core.AbstractVerticle` instead of `io.vertx.core.AbstractVerticle`. How is this different? The former class extends the latter and exposes a `io.vertx.rxjava.core.Vertx` field.

`io.vertx.rxjava.core.Vertx` defines extra `rxSomething(…)` methods that are equivalent to their callback-based counterparts.

Let's take a look at the `MainVerticle` to get a better idea of how it works in practice:

```
Single<String> dbVerticleDeployment = vertx.rxDeployVerticle(
  "io.vertx.guides.wiki.database.WikiDatabaseVerticle");
```

The `rxDeploy` method does not take a `Handler<AsyncResult<String>>` as final parameter. Instead, it returns a `Single<String>`.

Besides, the operation does not start when the method is called. It starts when you subscribe to the `Single`. When the operation completes, it emits the deployment `id` or signals the cause of the problem with a `Throwable`.

9.2. Deploying verticles in order

To finalize the `MainVerticle` refactoring, we must make sure the deployment operations get triggered and happen in order:

```java
dbVerticleDeployment
  .flatMap(id -> { ①

    Single<String> httpVerticleDeployment = vertx.rxDeployVerticle(
      "io.vertx.guides.wiki.http.HttpServerVerticle",
      new DeploymentOptions().setInstances(2));

    return httpVerticleDeployment;
  })
  .subscribe(id -> startFuture.complete(), startFuture::fail); ②
```

① The `flatMap` operator applies the function to the result of `dbVerticleDeployment`. Here it schedules the deployment of the `HttpServerVerticle`.

② Operations start when subscribing. On success or on error, the `MainVerticle` start future is either completed or failed.

9.3. Partially *"Rxifying"* `HttpServerVerticle`

If you follow the guide in sequence, editing the code as you go, your `HttpServerVerticle` class is still using the callback-based API. Before you can use the RxJava API to execute asynchronous operations *naturally*, i.e. ***concurrently***, you need to refactor `HttpServerVerticle`.

9.3.1. Import RxJava versions of Vert.x classes

```java
import io.vertx.rxjava.core.AbstractVerticle;
import io.vertx.rxjava.core.http.HttpServer;
import io.vertx.rxjava.ext.auth.AuthProvider;
import io.vertx.rxjava.ext.auth.User;
import io.vertx.rxjava.ext.auth.jwt.JWTAuth;
import io.vertx.rxjava.ext.auth.shiro.ShiroAuth;
import io.vertx.rxjava.ext.web.Router;
import io.vertx.rxjava.ext.web.RoutingContext;
import io.vertx.rxjava.ext.web.client.WebClient;
import io.vertx.rxjava.ext.web.client.HttpResponse; ①
import io.vertx.rxjava.ext.web.codec.BodyCodec;
import io.vertx.rxjava.ext.web.handler.*;
import io.vertx.rxjava.ext.web.sstore.LocalSessionStore;
import io.vertx.rxjava.ext.web.templ.FreeMarkerTemplateEngine;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import rx.Observable;
import rx.Single;
```

① Our `backupHandler()` method still uses `HttpResponse` class, so it must be imported. It turns out that the RxJava version of `HttpResponse` provided by Vert.x can serve as a drop-in replacement in this specific case. The *"Rxified"* code in `step-8` folder of the guide repository does not import this class as the response type is inferred by the lambda expression.

9.3.2. Use delegate on a *"Rxified"* `vertx` instance

To call a method expecting a `io.vertx.core.Vertx` instance when you have a `io.vertx.rxjava.core.Vertx` one, call the `getDelegate()` method. Verticle's `start()` method needs to be adjusted when creating an instance of `WikiDatabaseService`:

```
@Override
public void start(Future<Void> startFuture) throws Exception {

  String wikiDbQueue = config().getString(CONFIG_WIKIDB_QUEUE, "wikidb.queue");
  dbService = io.vertx.guides.wiki.database.WikiDatabaseService.createProxy(vertx.getDelegate(), wikiDbQueue);
```

9.4. Executing authorization queries concurrently

In the previous example, we saw how to use RxJava operators and the Rxified Vert.x API to execute asynchronous operations in order. But sometimes this guarantee is not required, or you simply want them to run concurrently for performance reasons.

The JWT token generation process in the `HttpServerVerticle` is a good example of such a situation. To create a token, we need all authorization query results to complete, but queries are independent from each other:

```
auth.rxAuthenticate(creds).flatMap(user -> {
  Single<Boolean> create = user.rxIsAuthorised("create"); ①
  Single<Boolean> delete = user.rxIsAuthorised("delete");
  Single<Boolean> update = user.rxIsAuthorised("update");

  return Single.zip(create, delete, update, (canCreate, canDelete, canUpdate) -> { ②
    return jwtAuth.generateToken(
      new JsonObject()
        .put("username", context.request().getHeader("login"))
        .put("canCreate", canCreate)
        .put("canDelete", canDelete)
        .put("canUpdate", canUpdate),
      new JWTOptions()
        .setSubject("Wiki API")
        .setIssuer("Vert.x"));
  });
}).subscribe(token -> {
  context.response().putHeader("Content-Type", "text/plain").end(token);
}, t -> context.fail(401));
```

① Three `Single` objects are created, representing the different authorization queries.

② When the three operations complete successfully, the `zip` operator callback is invoked with the results.

9.5. Working with database connections

To acquire a database connection from the pool, all you have to do is calling `rxGetConnection` on the `JDBCClient`:

```
Single<SQLConnection> connection = dbClient.rxGetConnection();
```

The method returns a `Single<Connection>` which you can easily transform with `flatMap` to execute SQL queries:

```
Single<ResultSet> resultSet = dbClient.rxQueryWithParams(
  sqlQueries.get(SqlQuery.GET_PAGE_BY_ID), new JsonArray().add(id));
```

But how can we release the connection if the `SQLConnection` reference is no longer reachable? A simple and convenient way to do this is to invoke `close` when the `Single<SQLConnection>` is unsubscribed:

```
private Single<SQLConnection> getConnection() {
  return dbClient.rxGetConnection().flatMap(conn -> {
    Single<SQLConnection> connectionSingle = Single.just(conn); ①
    return connectionSingle.doOnUnsubscribe(conn::close); ②
  });
}
```

① After the connection is acquired we wrap it into a `Single`

② The `Single` is modified to invoke `close` when unsubscribed

Now we shall use `getConnection` anytime we need to execute SQL queries in our database verticle.

9.6. Bridging the gap between callbacks and RxJava

At times, you may have to mix your RxJava code with a callback-based API. For example, service proxy interfaces can only be defined with callbacks, but the implementation uses the Vert.x Rxified API.

In this case, the `io.vertx.rx.java.RxHelper` class can adapt a `Handler<AsyncResult<T>>` to an RxJava `Subscriber<T>`:

```
@Override
public WikiDatabaseService fetchAllPagesData(Handler<AsyncResult<List<JsonObject>>> resultHandler) { ①
  dbClient.rxQuery(sqlQueries.get(SqlQuery.ALL_PAGES_DATA))
    .map(ResultSet::getRows)
    .subscribe(RxHelper.toSubscriber(resultHandler));  ②
  return this;
}
```

① `fetchAllPagesData` is an asynchronous service proxy operation, defined with a `Handler<AsyncResult<List<JsonObject>>>` callback.

② The `toSubscriber` method adapts the `resultHandler` to a `Subscriber<List<JsonObject>>`, so that the handler is invoked when the list of rows is emitted.

9.7. Data flows

RxJava is not only great at combining different sources of events, it is also very helpful with data flows. Unlike a Vert.x or JDK future, an `Observable` emits a stream of events, not just a single one. And it comes with an extensive set of data manipulation operators.

We can use a few of them to refactor the `fetchAllPages` database verticle method:

```
public WikiDatabaseService fetchAllPages(Handler<AsyncResult<JsonArray>> resultHandler) {
  dbClient.rxQuery(sqlQueries.get(SqlQuery.ALL_PAGES))
    .flatMapObservable(res -> {  ①
      List<JsonArray> results = res.getResults();
      return Observable.from(results); ②
    })
    .map(json -> json.getString(0)) ③
    .sorted() ④
    .collect(JsonArray::new, JsonArray::add) ⑤
    .subscribe(RxHelper.toSubscriber(resultHandler));
  return this;
}
```

① With `flatMapObservable` we will create an `Observable` from the item emitted by the `Single<Result>`.

② `from` converts the database `results` iterable into an `Observable` emitting the database row items.

③ Since we only need the page name we can `map` each `JsonObject` row to the first column.

④ The client expects the data to be `sorted` in alphabetical order.

⑤ The event bus service reply consists in a single `JsonArray`. `collect` creates a new one with `JsonArray::new` and later adds items as they are emitted with `JsonArray::add`.

Chapter 10. Client-side web application using AngularJS

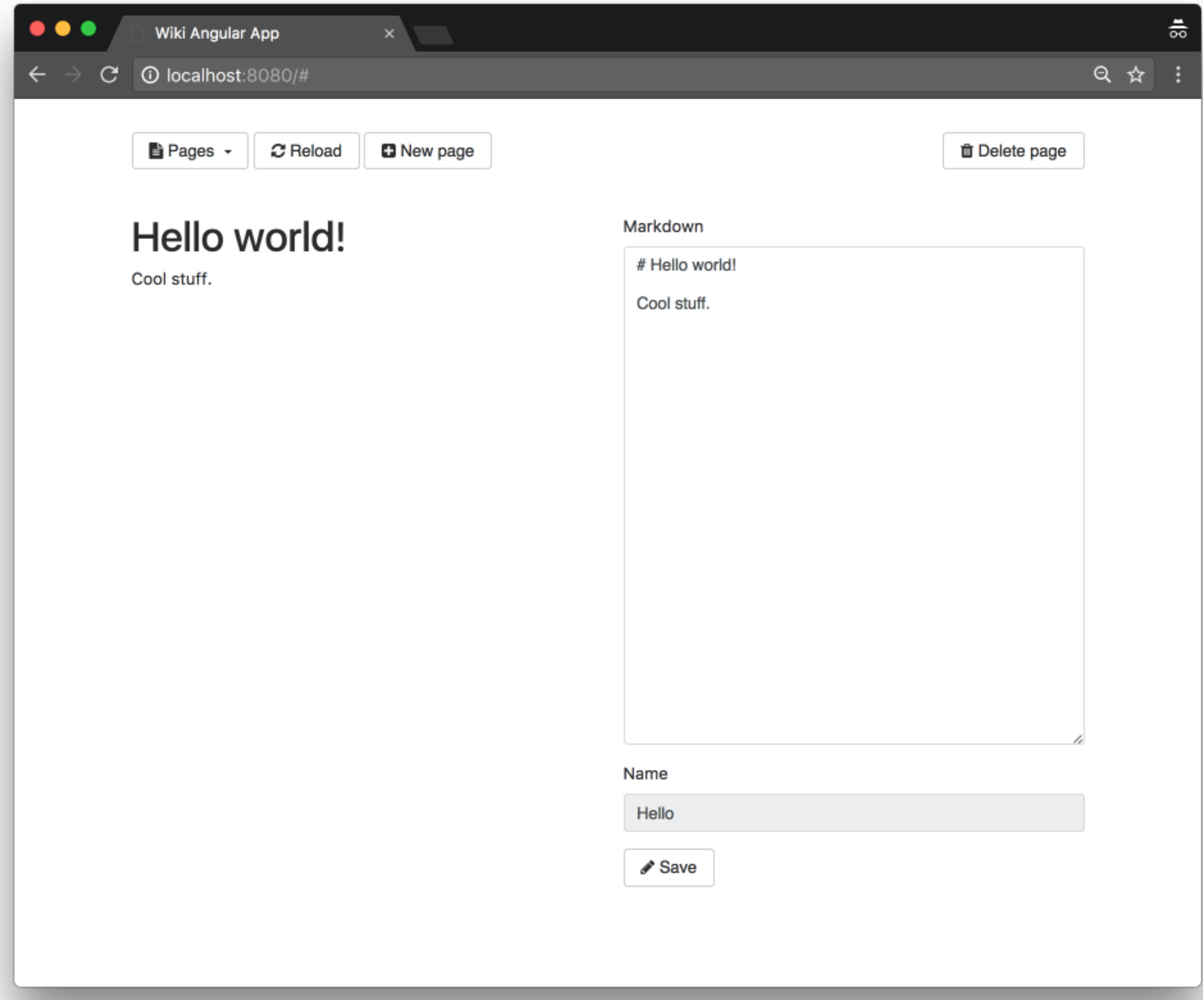
The corresponding source code is in the `step-9` folder of the guide repository.

So far our web interface was using traditional server-side rendering of HTML content. Some types of applications can take advantage of client-side rendering to improve user experience by avoiding full page reloads, and approaching the experience of native applications.

Many popular frameworks exist for that purpose. We chose the popular AngularJS framework for this guide, but one could equally choose React, Vue.js, Riot or another framework/library without any loss of generality.

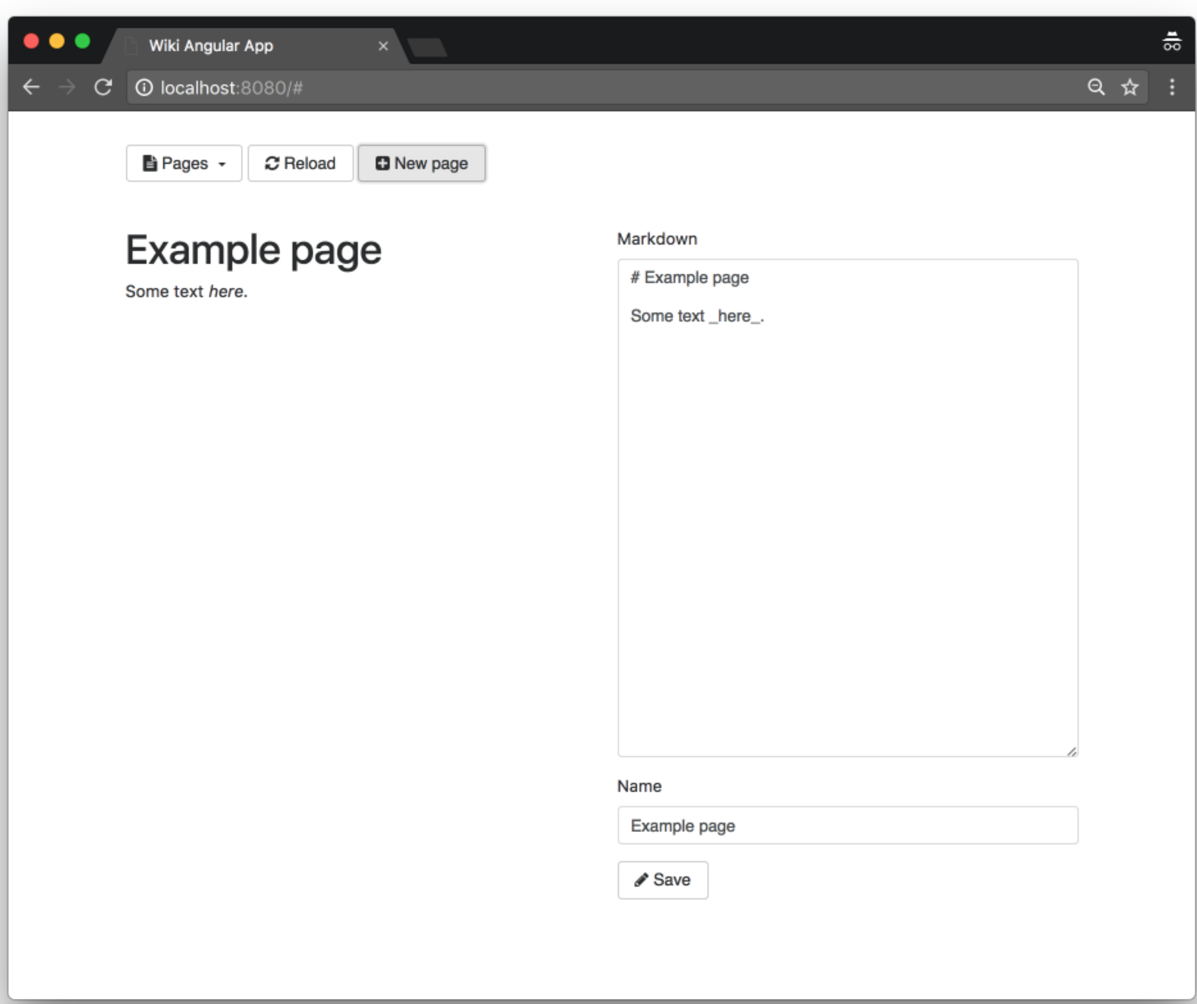
10.1. Single page application

The wiki editing application that we are building allows to select a page, and edit it with the first half of the screen being a HTML preview, and the other half being the Markdown editor:



The HTML preview is being rendered by calling a new endpoint in our backend. Rendering is triggered when the Markdown editor text changes. To avoid overloading the backend with unnecessary requests when the user is busy typing Markdown, a delay is being introduced so as to only trigger the rendering when no change has been made during that delay.

The application interface is also dynamic, as new pages make the deletion button disappear:



10.2. Vert.x Backend

10.2.1. Simplifying the HTTP verticle code

A client-side application needs a backend that exposes:

1. the static HTML, CSS and JavaScript content to bootstrap applications in web browsers, and
2. a web API, typically a HTTP/JSON service.

We simplified the HTTP verticle implementation to *just* cover what is needed. Starting from the RxJava version from *step #8*, we removed all server-side rendering code as well as the authentication and JWT token issuing code to expose a plain open HTTP/JSON interface.

Of course building a version that leverages JWT tokens and authentication makes sense for a real-world deployments, but now that we have covered these features we would rather focus on the essential bits in this part of the guide.

As an example, the `apiUpdatePage` method implementation code is now:

```
private void apiUpdatePage(RoutingContext context) {
  int id = Integer.valueOf(context.request().getParam("id"));
  JsonObject page = context.getBodyAsJson();
  if (!validateJsonPageDocument(context, page, "markdown")) {
    return;
  }
  dbService.rxSavePage(id, page.getString("markdown")).subscribe(
    v -> apiResponse(context, 200, null, null),
    t -> apiFailure(context, t));
}
```

10.2.2. Exposed routes

The HTTP/JSON API is exposed through the sames routes as in the previous steps:

```
router.get("/api/pages").handler(this::apiRoot);
router.get("/api/pages/:id").handler(this::apiGetPage);
router.post().handler(BodyHandler.create());
router.post("/api/pages").handler(this::apiCreatePage);
router.put().handler(BodyHandler.create());
router.put("/api/pages/:id").handler(this::apiUpdatePage);
router.delete("/api/pages/:id").handler(this::apiDeletePage);
```

The front application static assets are being served from `/app`, and we redirect requests to `/` to the `/app/index.html` static file:

```
router.get("/app/*").handler(StaticHandler.create().setCachingEnabled(false)); ① ②
router.get("/").handler(context -> context.reroute("/app/index.html"));
```

① Disabling caching is useful in development.

② By default the files are expected to be in the `webroot` package on the *classpath*, so the files shall be placed under `src/main/resources/webroot` in a Maven or Gradle project.

Last but not least, we anticipate that the application will need the backend to render Markdown to HTML, so we offer a HTTP POST endpoint for this purpose:

```
router.post("/app/markdown").handler(context -> {
  String html = Processor.process(context.getBodyAsString());
  context.response()
    .putHeader("Content-Type", "text/html")
    .setStatusCode(200)
    .end(html);
});
```

10.3. AngularJS frontend

This guide is not a proper introduction to AngularJS (see the official tutorial instead), we assume some familiarity with the framework from the reader.

10.3.1. Application view

The interface fits in a single HTML file located at `src/main/resources/webroot/index.html`. The `head` section is:

```
<html lang="en" ng-app="wikiApp"> ①
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
  <title>Wiki Angular App</title>
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-alpha.6/css/bootstrap.min.css"
        integrity="sha384-rwoIResjU2yc3z8GV/NPeZWAv56rSmLldC3R/AZzGRnGxQQKnKkoFVhFQhNUwEyJ" crossorigin="anonymous">
  <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/css/font-awesome.css">
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.4/angular.min.js"></script>
  <script src="https://cdn.jsdelivr.net/lodash/4.17.4/lodash.min.js"></script>
  <script src="/app/wiki.js"></script>  ②
  <style>
    body {
      padding-top: 2rem;
      padding-bottom: 2rem;
    }
  </style>
</head>
<body>
```

① The AngularJS module is named `wikiApp`.

② `wiki.js` holds the code for our AngularJS module and controller.

As you can see beyond AngularJS we are using the following dependencies from external CDNs:

- Boostrap to style our interface,
- Font Awesome to provide icons,
- Lodash to help with some functional idioms in our JavaScript code.

Bootstrap requires some further scripts that can be loaded at the end of the document for performance reasons:

```
<script src="https://code.jquery.com/jquery-3.1.1.slim.min.js"
        integrity="sha384-A7FZj7v+d/sdmMqp/nOQwliLvUsJfDHW+k9Omg/a/EheAdgtzNs3hpfag6Ed950n"
        crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/tether/1.4.0/js/tether.min.js"
        integrity="sha384-DztdAPBWPRXSA/3eYEEUWrWCy7G5KFbe8fFjk5JAIxUYHKkDx6Qin1DkWx51bBrb"
        crossorigin="anonymous"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-alpha.6/js/bootstrap.min.js"
        integrity="sha384-vBWWzlZJ8ea9aCX4pEW3rVHjgjt7zpkNpZk+02D9phzyeVkE+jo0ieGizqPLForn"
        crossorigin="anonymous"></script>

</body>
</html>
```

Our AngularJS controller is called `WikiController` and it is bound to a `div` which is also a Bootstrap container:

```
<div class="container" ng-controller="WikiController">
  <!-- (...) -->
```

The buttons on top of the interface consist of the following elements:

```
<div class="row">

  <div class="col-md-12">
    <span class="dropdown">
      <button class="btn btn-secondary dropdown-toggle" type="button" id="pageDropdownButton" data-toggle="dropdown"
              aria-haspopup="true" aria-expanded="false">
        <i class="fa fa-file-text" aria-hidden="true"></i> Pages
      </button>
      <div class="dropdown-menu" aria-labelledby="pageDropdownButton">
        <a ng-repeat="page in pages track by page.id" class="dropdown-item" ng-click="load(page.id)" href="#"
>{{page.name}}</a> ①
      </div>
    </span>
    <span>
      <button type="button" class="btn btn-secondary" ng-click="reload()"><i class="fa fa-refresh"
                                                                             aria-hidden="true"></i> Reload</button> ②
    </span>
    <span>
      <button type="button" class="btn btn-secondary" ng-click="newPage()"><i class="fa fa-plus-square"
                                                                              aria-hidden="true"></i> New page</button>
    </span>
    <span class="float-right">
      <button type="button" class="btn btn-secondary" ng-click="delete()" ng-show="pageExists()"><i class="fa fa-trash"
                                                                                                     aria-hidden=
"true"></i> Delete page</button> ③
    </span>
  </div>

  <div class="col-md-12"> ④
    <div class="invisible alert" role="alert" id="alertMessage">
      {{alertMessage}}
    </div>
  </div>

</div>
```

① For each wiki page name we generate an element using `ng-repeat` and `ng-click` to define the controller action (`load`) when it is being clicked.

② The refresh button is bound to the `reload` controller action. All other buttons work the same way.

③ The `ng-show` directive allows us to show or hide the element depending on the controller `pageExists` method value.

④ This `div` is used to display notifications of success or failures.

The Markdown preview and editor elements are the following:

```
<div class="row">

  <div class="col-md-6" id="rendering"></div>

  <div class="col-md-6">
    <form>
      <div class="form-group">
        <label for="markdown">Markdown</label>
        <textarea id="markdown" class="form-control" rows="25" ng-model="pageMarkdown"></textarea> ①
      </div>
      <div class="form-group">
        <label for="pageName">Name</label>
        <input class="form-control" type="text" value="" id="pageName" ng-model="pageName" ng-disabled="pageExists()">
      </div>
      <button type="button" class="btn btn-secondary" ng-click="save()"><i class="fa fa-pencil" aria-hidden="true"></i>
Save</button>
    </form>
  </div>

</div>
```

① `ng-model` binds the `textarea` content to the `pageMarkdown` property of the controller.

10.3.2. Application controller

The `wiki.js` JavaScript starts with an AngularJS module declaration:

```
'use strict';

angular.module("wikiApp", [])
  .controller("WikiController", ["$scope", "$http", "$timeout", function ($scope, $http, $timeout) {

    var DEFAULT_PAGENAME = "Example page";
    var DEFAULT_MARKDOWN = "# Example page\n\nSome text _here_.\n";

    // (...)
```

The `wikiApp` module has no plugin dependency, and declares a single `WikiController` controller. The controller requires dependency injection of the following objects:

- `$scope` to provide DOM scoping to the controller, and
- `$http` to perform asynchronous HTTP requests to the backend, and
- `$timeout` to trigger actions after a given delay while staying tied to the AngularJS life-cycle (e.g., to ensure that any state modification triggers view changes, which is not the case when using the classic setTimeout function).

Controller methods are being tied to the `$scope` object. Let us start with 3 simple methods:

```
$scope.newPage = function() {
  $scope.pageId = undefined;
  $scope.pageName = DEFAULT_PAGENAME;
  $scope.pageMarkdown = DEFAULT_MARKDOWN;
};

$scope.reload = function () {
  $http.get("/api/pages").then(function (response) {
    $scope.pages = response.data.pages;
  });
};

$scope.pageExists = function() {
  return $scope.pageId !== undefined;
};
```

Creating a new page consists in initializing controller properties that are attached to the `$scope` object. Reloading the pages objects from the backend is a matter of performing a HTTP GET request (note that the `$http` request methods return promises). The `pageExists` method is being used to show / hide elements in the interface.

Loading the content of the page is also a matter of performing a HTTP GET request, and updating the preview a DOM manipulation:

```
$scope.load = function (id) {
  $http.get("/api/pages/" + id).then(function(response) {
    var page = response.data.page;
    $scope.pageId = page.id;
    $scope.pageName = page.name;
    $scope.pageMarkdown = page.markdown;
    $scope.updateRendering(page.html);
  });
};

$scope.updateRendering = function(html) {
  document.getElementById("rendering").innerHTML = html;
};
```

The next methods support saving / updating and deleting pages. For these operations we used the full `then` promise method with the first argument being called on success, and the second one being called on error. We also introduce the `success` and `error` helper methods to display notifications (3 seconds on success, 5 seconds on error):

```
$scope.save = function() {
  var payload;
  if ($scope.pageId === undefined) {
    payload = {
```

```
      "name": $scope.pageName,
      "markdown": $scope.pageMarkdown
    };
    $http.post("/api/pages", payload).then(function(ok) {
      $scope.reload();
      $scope.success("Page created");
      var guessMaxId = _.maxBy($scope.pages, function(page) { return page.id; });
      $scope.load(guessMaxId.id || 0);
    }, function(err) {
      $scope.error(err.data.error);
    });
  } else {
    var payload = {
      "markdown": $scope.pageMarkdown
    };
    $http.put("/api/pages/" + $scope.pageId, payload).then(function(ok) {
      $scope.success("Page saved");
    }, function(err) {
      $scope.error(err.data.error);
    });
  }
};

$scope.delete = function() {
  $http.delete("/api/pages/" + $scope.pageId).then(function(ok) {
    $scope.reload();
    $scope.newPage();
    $scope.success("Page deleted");
  }, function(err) {
    $scope.error(err.data.error);
  });
};

$scope.success = function(message) {
  $scope.alertMessage = message;
  var alert = document.getElementById("alertMessage");
  alert.classList.add("alert-success");
  alert.classList.remove("invisible");
  $timeout(function() {
    alert.classList.add("invisible");
    alert.classList.remove("alert-success");
  }, 3000);
};

$scope.error = function(message) {
  $scope.alertMessage = message;
  var alert = document.getElementById("alertMessage");
  alert.classList.add("alert-danger");
  alert.classList.remove("invisible");
  $timeout(function() {
    alert.classList.add("invisible");
```

```
    alert.classList.remove("alert-danger");
  }, 5000);
};
```

initializing the application state and views is done by fetching the pages list, and starting with a blank new page editor:

```
$scope.reload();
$scope.newPage();
```

Finally here is how we perform live rendering of Markdown text:

```
var markdownRenderingPromise = null;
$scope.$watch("pageMarkdown", function(text) {  ①
  if (markdownRenderingPromise !== null) {
    $timeout.cancel(markdownRenderingPromise);  ③
  }
  markdownRenderingPromise = $timeout(function() {
    markdownRenderingPromise = null;
    $http.post("/app/markdown", text).then(function(response) { ④
      $scope.updateRendering(response.data);
    });
  }, 300); ②
});
```

① `$scope.$watch` allows being notified of state changes. Here we monitor changes on the `pageMarkdown` property that is bound to the editor `textarea`.

② 300 milliseconds is a *fine* delay to trigger rendering if nothing has changed in the editor.

③ Timeouts are promise, so if the state has changed we cancel the previous one and create a new one. This is how we delay rendering instead of doing it on every keystroke.

④ We ask the backend to render the editor text into some HTML, then refresh the preview.

Chapter 11. Real-time web features using cross-border messaging over the event bus

The corresponding source code is in the `step-10` folder of the guide repository.

Earlier in this guide we saw the event bus in action for verticles to communicate using message-passing inside Vert.x applications. The developer only has to register a consumer to receive messages and sending / publishing messages.

The SockJS event bus bridge extends these capabilities to the client-side, in the web browser. It creates a distributed event bus which not only spans multiple Vert.x instances in a cluster, but also includes client-side JavaScript running in (many) browsers. We can therefore create a huge distributed event bus encompassing many browsers and servers, resulting in a consistent message-based programming model across all constituents of a distributed application.

In this chapter, we will modify the code from `step-9` so that:

- the Markdown content to be rendered is sent to the server without creating new HTTP requests, and
- the page shows a warning when a user is editing a page which has just been modified by another user.

11.1. Setting up the SockJS event bus bridge

11.1.1. On the server

SockJS is a client-side JavaScript library and protocol that provides a simple WebSocket-like interface for making connections to SockJS servers, irrespective of whether the actual browser or network will allow real WebSockets. It does this by supporting various different transports between browser and server, and choosing one at runtime according to their capabilities.

As a first step, we need to setup the `SockJSHandler` that is provided by the `vertx-web` project:

```
SockJSHandler sockJSHandler = SockJSHandler.create(vertx); ①
BridgeOptions bridgeOptions = new BridgeOptions()
  .addInboundPermitted(new PermittedOptions().setAddress("app.markdown"))  ②
  .addOutboundPermitted(new PermittedOptions().setAddress("page.saved")); ③
sockJSHandler.bridge(bridgeOptions); ④
router.route("/eventbus/*").handler(sockJSHandler); ⑤
```

① Create a new `SockJSHandler` for this `vertx` instance.

② Allow delivery of messages coming from the browser on the `app.markdown` address. We will use this address to get the server process the Markdown content as we edit a wiki page.

③ Allow sending messages going to the browser on the `page.saved` address. We will use this address

to notify browsers that a wiki page has been modified.

④ Configure the handler to bridge SockJS traffic to the event bus.

⑤ Handle all requests under the `/eventbus` path with the SockJS handler.

For most applications you probably do not want client side JavaScript to be able to send just any message to any handler on the server side, or to all other browsers. For example:

- you may have a service on the event bus that allows data to be accessed or deleted, and clearly we do not want badly behaved or malicious clients to be able to delete all the data in your database,
- we do not necessarily want any client to be able to listen on any event bus address.

To deal with this, a SockJS bridge will by default refuse any messages through. This is why it is up to you to tell the bridge what messages are ok for it to pass through (as an exception reply messages are always allowed to pass through).

11.1.2. On the client

Now that the server is ready to accept messages, we shall configure the client.

First, the SockJS library and the Vert.x event bus JavaScript client must be loaded. The easiest way to get started is to get the files from a public content delivery network:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/sockjs-client/1.1.4/sockjs.min.js"
        integrity="sha256-KWJavOowudybFMUCd547Wvd/u8vUg/2g0uSWYU5Ae+w="
        crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vertx/3.4.1/vertx-eventbus.min.js"
        integrity="sha256-EX8Kk2SwcSUXBJ4RORlETHNwHWEw+57C/YDLnbiIl3U="
        crossorigin="anonymous"></script>
```

The event bus client can be downloaded beforehand and bundled with the application. It is published on `Maven`, `npm`, `bower` and even `webjars` repositories.

Then, we create a new instance of the `EventBus` Javascript object:

```
var eb = new EventBus(window.location.protocol + "//" + window.location.host + "/eventbus");
```

11.2. Sending the Markdown content over the event bus for processing

The SockJS bridge is now up and running. In order to process the Markdown content on the server side, we need to register a consumer. The consumer handles messages sent to the `app.markdown`

address:

```
vertx.eventBus().<String>consumer("app.markdown", msg -> {
  String html = Processor.process(msg.body());
  msg.reply(html);
});
```

There is nothing new here, we have already been creating event bus consumers before. Now let's turn to what happens in the client code:

```
eb.send("app.markdown", text, function (err, reply) { ①
  if (err === null) {
    $scope.$apply(function () { ②
      $scope.updateRendering(reply.body); ③
    });
  } else {
    console.warn("Error rendering Markdown content: " + JSON.stringify(err));
  }
});
```

① The *reply handler* is a function taking two parameters: an error (if any) and the `reply` object. The `reply` object content is nested inside the `body` property.

② Since the event bus client is not managed by AngularJS, `$scope.$apply` wraps the callback to perform proper scope life-cycle.

③ As we did when working with `$http`, we invoke `updateRendering` with the HTML result.

Admittedly, the code is very similar to its HTTP endpoint equivalent. However the benefit here does not lie in the number of lines of code.

Indeed, if you communicate with the server over the event bus, the bridge transparently distributes incoming messages among registered consumers. Consequently, when Vert.x runs in cluster mode, the browser is not tied to a single server for processing (apart from the SockJS connection). What's more the connection to the server is never closed, so with HTTP/1.1 this saves establishing a TCP connection for each request, which may be useful if you have lots of exchanges between servers and clients.

11.3. Warning the user when the page is modified

In many applications, the *last commit wins* principle is how conflicts are being resolved: when two users edit the same resource at the same time, the last one to press the *save* button overwrites any previous changes.

There are ways around this issue, like entity versioning or extensive literature on the topic of distributed consensus. Nevertheless, let's stick to a simple solution and see how we can notify the user when a change has been made so that at the very least (s)he can get a chance to deal with the situation. As soon as the content has been changed in the database, the user can decide what is the best course of action to take: overwrite or reload.

To start with, we need to add an `alert alert-warning` message `div` in the page. But we want it to show-up only when the `pageModified` scope variable is set to `true`.

```
<div class="col-md-12">
  <div class="alert alert-warning ng-class:{'invisible': !pageModified};" role="alert">
    The page has been modified by another user.
    <a href="#" ng-click="load(pageId)">Reload</a>
  </div>
</div>
```

Now, `pageModified` must be set to `true` when this page is saved. Let's register an event bus handler for the `page.saved` address:

```
var clientUuid = generateUUID(); ①
eb.onopen = function () {
  eb.registerHandler("page.saved", function (error, message) { ②
    if (message.body ③
      && $scope.pageId === message.body.id ④
      && clientUuid !== message.body.client) { ⑤
      $scope.$apply(function () { ⑥
        $scope.pageModified = true; ⑦
      });
    }
  });
};
```

① We do not want to print the warning if we modified the content ourselves so we need a client identifier.

② The callback will be invoked when a message is received on the `page.saved` address.

③ Check that the body is not empty.

④ Make sure this event is related to the current wiki page.

⑤ Check that we are not the origin of the changes.

⑥ Since the event bus client is not managed by AngularJS, `$scope.$apply` wraps the callback to perform proper scope life cycle.

⑦ Set `pageModified` to true.

Eventually we have to push messages when the content of a page is saved in the database:

```
dbService.rxSavePage(id, page.getString("markdown"))
  .doOnSuccess(v -> { ①
    JsonObject event = new JsonObject()
      .put("id", id) ②
      .put("client", page.getString("client")); ③
    vertx.eventBus().publish("page.saved", event); ④
  })
  .subscribe(v -> apiResponse(context, 200, null, null), t -> apiFailure(context, t));
```

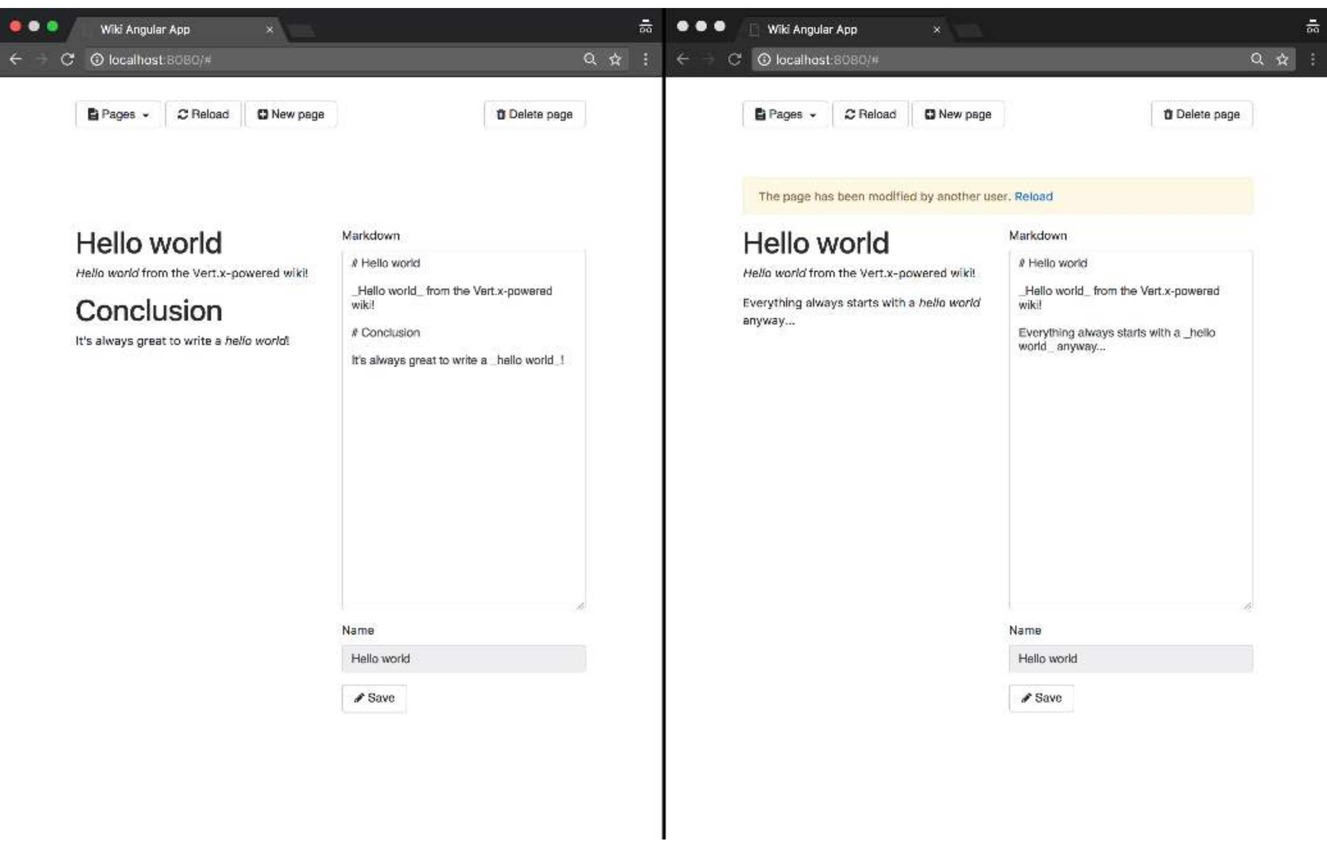
① `rxSavePage` returns a `Single<Void>` object. On success (i.e. no database failure) we publish an event.

② The message contains the page identifier.

③ The message contains the client identifier.

④ The event is published on the `page.saved` address.

If we open the application in two tabs inside the same browser (or different browsers), select the same page on both, and update the content in one, the warning message is printed:



We could easily leverage the SockJS bridge for other purposes, like showing how many users are currently on a given page, supporting live comments in a chat box, etc. The key point is that both the server and the client sides share the same programming model by message passing over the event-bus.

Chapter 12. Conclusion

This is the end of this guide. Let us take a moment to recapitulate the important takeaways from the previous sections, and then conclude by pointing to further useful resources.

12.1. Summary

We started this guide by building a wiki web application with Vert.x. While the first iteration was typical of *"quick and dirty rapid prototyping"* it showed that one could quickly and easily build such an application with server-side rendering of web templates and persistence with a relational database.

The next steps showed how to improve the design through successive refactoring: first separating each technical concern as a standalone verticle, then extracting Vert.x services for API cleanliness and finally introducing JUnit tests for asynchronous code.

We then ventured into consuming third-party HTTP/JSON services using the web client API that further simplifies the usage of the HTTP client from Vert.x core. Conversely, we also saw how to expose HTTP/JSON web APIs with the elegant Vert.x web module.

From there it is very easy to extend the approach to build API gateways for providing facades to many services. If you were to build such gateways, we suggest that you take advantage of:

- the Vert.x RxJava support to describe service consumption data flow streams, and
- the Vert.x circuit-breaker module to deal consistently with the potential failure of services.

Access control, authentication and security are often neglected or come as an aftermath. We saw that Vert.x provides a simple pluggable authentication mechanism to leverage databases, files or LDAP directories. SSL network encryption is very easy to setup for a server, or for a client to use. Finally Vert.x supports JWT tokens, a very useful and decentralized authentication scheme for web APIs.

The Vert.x core API relies on callbacks as it is the most generic way of processing asynchronous events. Vert.x provides a simple promise / future API. While Vert.x futures are composable, they shall be confined to limited usages such as dealing with verticle deployments and initialization. We saw how RxJava is supported in Vert.x, and we encourage you to use it for your own verticles. What’s more RxJava is the most popular reactive programming library on the JVM, so you will easily find third-party libraries to integrate consistently in your end-to-end reactive applications.

Modern web applications tend to have the server expose *just* HTTP/JSON APIs, and rely on client-side web frameworks for the user interface. We saw how to do that with AngularJS so as to turn our wiki into a single-page application.

Finally, we saw how elegant it was to extend the event bus of an application to allow web applications to send and receive events from the browser using the SockJS bridge. While it may initially seem like to be a small feature, in practice it turns out to greatly simplify the development of *real-time* web application features. The SockJS bridge can actually be useful also in cases where one would have used an HTTP endpoint: sending a message then getting a response over the event

bus can sometimes be simpler than doing an HTTP call, having the server process the HTTP request, forwarding it to a verticle on the event bus and eventually terminating the HTTP exchange by encoding a JSON response.

12.2. Going further

The Vert.x website is of course the authority on all things Vert.x.

There are many features and modules that we haven't covered in this guide, such as:

- clustering using Hazelcast, Infinispan, Apache Ignite or Apache Zookeeper,
- how the code looks like with other supported languages,
- exposing and consuming over HTTP/2, possibly (but not necessarily) with gRPC
- using NoSQL databases such as MongoDB or Redis,
- sending emails over SMTP,
- messaging with AMQP, Stomp, Kafka, MQTT or RabbitMQ,
- using OAuth2 authentication from custom and popular providers,
- Vert.x sync for writing blocking-style code that is later turned into *fibers* non-blocking code at runtime,
- publishing and discovering micro-services from registries, for instance when deploying on cloud environments like OpenShift,
- exposing metrics and health checks.

This list is not exhaustive: Vert.x is a toolkit so you are the one to decide what ingredients are required for your project, big or small.

You may also find it useful to browse the Vert.x awesome curated list of community projects as it goes beyond what is being supported by the project.

If you are developing micro-services, we suggest reading the "Building Reactive Microservices in Java" book by Clément Escoffier.

12.3. That's all folks!

We hope that you enjoyed reading this guide, and that it turned out to be useful in your journey towards asynchronous programming with Vert.x.

Feel-free to get in touch with the authors, either directly by email or through the Vert.x project user group. Of course we appreciate praise, but we appreciate as much any constructive feedback that can improve this content.

Thank you very much!