Table of Contents

vertx-guide-for-java-devs-chinese-translation

这是 Vert.x 官方文档 A gentle guide to asynchronous programming with Eclipse Vert.x for Java developers 非官方中文翻译。

其英文原版在 GitHub 上的位置： https://github.com/vert-x3/vertx-guide-for-java-devs

翻译不当之处，欢迎指正！

目录

介绍

这本手册是关于使用 Vert.x 异步编程的易读介绍，针对那些熟悉主流“非异步” web 开发框架或者库（例如 Java EE、Spring）的开发者。

关于此手册

我们假定读者熟悉 Java 编程语言及相关生态。

我们将从一个 wiki web 程序开始（其使用关系型数据库并在服务端渲染网页），然后通过一步步的改进，使其最终成长为一个拥有“实时” web 特性的现代单页应用。在这个过程中，你将会学到：

1. 设计一个 web 程序，其在服务端通过模板渲染网页，并使用关系型数据库来持久化（存储）数据。
2. 清晰的抽离出每个技术组件，以作为可重复使用的事件处理单元（被称作 verticle）。
3. 不同的 verticle 之间（这些 verticle 使用同一个 JVM 进程或处于同一个集群下不同节点）可以互相无缝地通信交流，通过提取出 Vert.x 服务来优化这些 verticle 的设计。
4. 通过异步操作来测试代码。
5. 将暴露了 HTTP/JSON web API 的第三方服务融入进项目之中。
6. 实现一个 HTTP/JSON web API。
7. 使用 HTTPS，为浏览器会话产生的用户认证，为第三方应用访问提供的 JWT token，来实现对资源的保护及访问控制。
8. 借助流行的 RxJava 库和它在 Vert.x 中的集成来重构部分代码，以实现响应式编程。
9. 客户端采用 AngularJS 来实现单页应用。
10. 使用统一的集成于 SockJS 之上的 Vert.x event bus 机制来实现实时 web 项目。
11. 注意：

    所有的文档及代码示例都可以在这里找到：https://github.com/vert-x3/vertx-guide-for-java-devs 我们欢迎提供任何 issue reports，反馈及 pull-request！

什么是 Vert.x?

> Eclipse Vert.x 是一个可以在 JVM 上构建响应式应用的工具包。
>
> — Vert.x 官网

Eclipse Vert.x（以下简称 Vert.x） 是 Eclipse 基金会门下的一个开源项目，其最初是由 Tim Fox 在 2012 年发起的。

Vert.x 是一个工具包集合而不是一个框架：核心库为编写异步网络应用定义了基本的 API，你可以（自由地）为你的项目选择有用的模块（例如：数据库连接、监控、身份认证、日志、服务发现、集群支持等等）。Vert.x 基于 Netty 项目，Netty 是一个为 JVM 设计的高性能异步网络库。通常来说，使用 Vert.x 提供的高层次 API 会更有利于你（编写代码），相较原生 Netty 来说，也毫无性能损失。但如果你有所需要， Vert.x 同样允许你访问 Netty 的内部。

Vert.x 并不强制要求任何包或者构建环境。因为 Vert.x core 自身就是一个常规的 Jar（每一个 Jar 包含所有依赖）库，所以它可以作为 Jar 嵌入应用之中，甚至被部署到流行的组件和应用容器内。

Vert.x 被设计用于异步通信，因此相较于 Java servlets 或 java.net socket classes 这些同步 API 来说，可以使用较少的线程来解决更多的并发网络连接（问题）。Vert.x 对于多种类型的应用都非常有用：高性能消息/事件处理、微服务、API 网关、为移动应用设计的 HTTP API 等等。Vert.x 及其相关生态为构建端到端响应式应用提供了多种多样的技术工具。

虽然可能听起来 Vert.x 仅仅应用于高性能应用，但这篇导引手册可以证明 Vert.x 对于传统 web 应用同样非常有用。代码仍将保持简洁易理解，即便遇到突然的流量高峰，已经采用异步事件处理方式编写的代码可以轻松处理。

同时值得一提的是，Vert.x 支持多种流行的 JVM 语言：Java、Groovy、Scala、Kotlin、JavaScript、Ruby和Ceylon。Vert.x 支持多种语言的目标不只是提供各种语言 API 的访问，而是确保每一种语言都可以采用其特有的语言特性、以符合语言习惯的方式来调用 API（例如：使用 Scala 的 future 替换 Vert.x 的 future）。Vert.x 也可以很好的支持在一个应用之中，采用不同的 JVM 语言来开发不同的技术模块。

Vert.x 核心概念

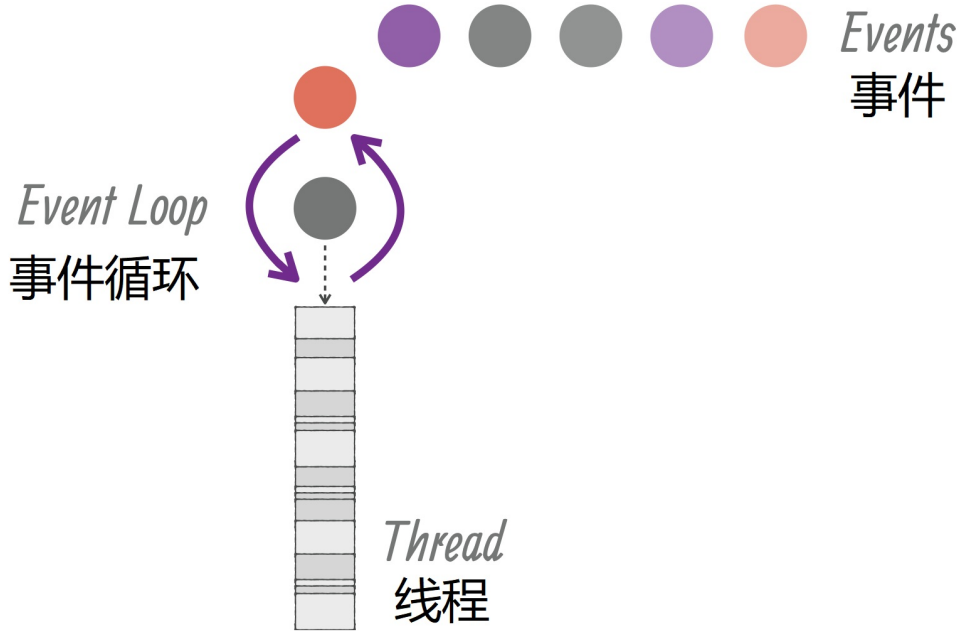
Vert.x 中有两个关键的概念：

1. 什么是 verticle
2. event bus 如何使不同的 verticle 之间通信

线程与编程模型

很多网络库和框架依赖于一种简单的线程策略：为每一个客户端都分配一个线程用于连接，并且直到断开连接前，每个线程处理不同客户端的业务。Servlet 或者由 java.io 和 java.net 包编写的网络程序都是如此。虽然“同步 I/O”线程模型对于保持简单与易理解来说很有优势，但线程资源并不便宜，并且在高负载的情况下，操作系统会花费大量的时间浪费在线程调度管理上，因此对于大量并发请求来说，此模型扩展性较差。

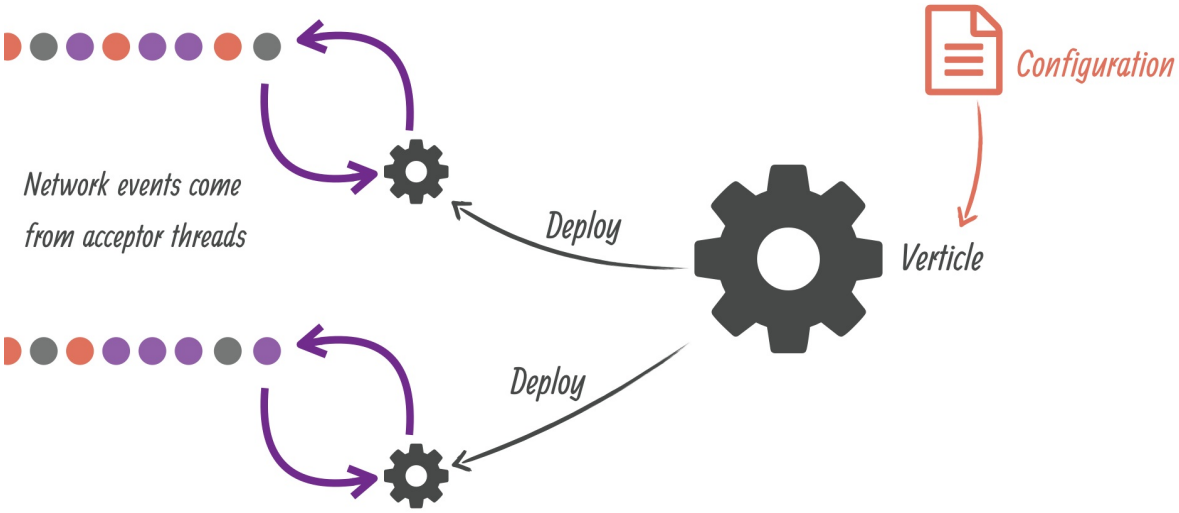
Vert.x 中的可被部署的单位（或单元）被称作 Verticle。Verticle 基于「事件循环」来处理传入的事件，事件可以是接受网络缓冲、定时事件或者来自其他 verticle 的消息。在异步编程模型中，「事件循环」非常典型：



每一个事件都应在合理的时间内处理完成，不应让其阻塞「事件循环」。这意味着可能阻塞线程的操作不应当在「事件循环」中被执行，就像在图形界面处理事件时（不应当）卡住 Java 或者 Swing 界面去做一个很慢的网络请求一样。在本手册的后面将会看到，Vert.x 提供了一种在「事件循环」外处理阻塞操作的机制。当「事件循环」在处理一个事件耗时过长时，Vert.x 总会在日志中发出警告。为了匹配应用的需求（例如：运行环境是相对较慢的物联网 ARM 主板），这个特性同样也是可以被配置。

每一个「事件循环」都运行在线程上。默认情况下，每一个 CPU 核心线程运行 2 个「事件循环」。所以正常情况下，verticle 将始终在同一个线程上处理事件，因此无需使用线程协调机制来调整 verticle 的状态（例如：Java class fields）。

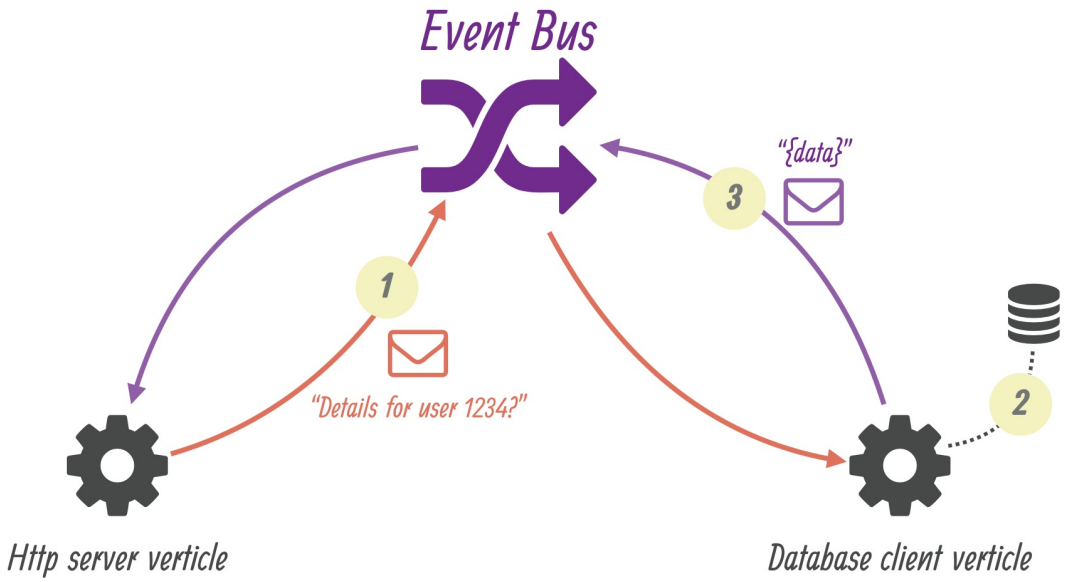
verticle 可以进行一些配置(configuration)（例如：证书、网络地址等等），并且可以被部署多次：



传入的网络数据将会被接收线程收到，并作为事件交由相应的 verticle 处理。当一个 verticle 开启了网络服务器并被部署多次时，事件将会采用轮询形式被分发给 verticle 实例，这有利于处理大量并发网络请求时最大化 CPU 利用率。Verticle 的生命周期只有简单的开启与结束，并且 verticle 可以 deploy 其他 verticle。

Event bus

在 Vert.x 中，verticle 将代码组织成可部署的单元。Vert.x event bus 是不同 verticle 之间通过异步消息传递进行通信的主要工具。例如假设存在一个 verticle 负责 HTTP 请求，另一个 verticle 负责管理数据库访问。Event bus 将允许 HTTP verticle 发送请求给数据库 verticle 来执行 SQL 查询，并返回结果给 HTTP verticle。



因为 JSON 可以被各种语言所编写的 verticle 用来通信，同时也是非常流行的通用型半结构化数据编组格式，所以其被推荐用来作为信息的交换格式，但 event bus 本身并不限制传递的数据类型。

消息可以发送到可接收字符串的目的地。 Event bus 支持以下沟通模式：

1. 点对点消息
2. 请求-响应消息
3. 发布-订阅消息（广播）

即使不在同一个 JVM 进程中，event bus 也可以使 verticle 之间透明的沟通：

- 当网络集群被激活时，event bus 即是分布式的，因此消息可以被发送到运行于另一个应用节点的 verticle 上
- 为了与其他第三方应用沟通（通信），可以通过简单的 TCP 协议访问 event bus
- Event bus 也可以被设置为通用的消息桥梁（例如：AMQP、Stomp）
- SockJS bridge 允许 web 应用就像其他 verticle 一样，基于 event bus 在运行于浏览器中的 JavaScript 无缝地接收和发布消息

使用 Vert.x 实现一个最小可实施的 wiki 系统

> 提示：
>
> 相关的源代码可以在本手册（注：英文原版）的仓库 step-1 目录下找到。

我们将从第一个迭代版本开始，以尽可能简单的代码，使用 Vert.x 实现一个 wiki 系统。在之后的迭代版本，代码将会更加优雅，同时引入适当的测试，我们将可以看到使用 Vert.x 快速构建原型兼具简单性与实际性。

在当前阶段，这个 wiki 系统将会采用服务端渲染 HTML 的方式，通过 JDBC 连接实现数据持久化。为了实现这些，我们将会使用到以下库：

1. Vert.x web 同 Vert.x core 库（但其并未提供 API 处理路由、处理请求负载等等）一样支持创建 HTTP server。
2. Vert.x JDBC 客户端用来提供基于 JDBC 的异步 API。
3. Apache FreeMarker 是一个简单的模板引擎，用来在服务端渲染页面。
4. Txtmark 用来将 Markdown 文本渲染为 HTML，可以实现使用 Markdown 编辑 wiki 页面。

开始一个 Maven 项目

本手册使用 Apache Maven 作为构建工具，主要是因为其比较好的集成在了大多数集成开发环境。你也可以选择使用其他的构建工具（例如：Gradle）。

Vert.x 社区提供了一个项目结构模板，可以在这里获取。如果你选择使用 Git 作为版本控制系统的话，（搭建起项目的）最快方式就是克隆这个仓库，删掉它的 `.git/` 目录，重新创建为一个新的 Git 仓库。

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

这个项目提供了一个示例 verticle 和一个单元测试。删除 `src/` 目录下所有的 `.java` 文件来自定义（hack） wiki 项目是安全的，但在此之间，可以先尝试构建项目，并测试是否可以运行：

```
mvn package exec:java
```

Maven 项目的 `pom.xml` 做了两件有趣的事：

1. 它使用 Maven Shade 插件创建一个包含所有需要的依赖的 Jar 打包文件（也被叫做“a fat Jar”），其后缀为 `-fat.jar`
2. 它使用 Exec Maven 插件来提供 `exec:java` 以用于通过 Vert.x `io.vertx.core.Launcher` 类来依次启动应用。这等价于通过 `vertx` 命令行工具（在 Vert.x 分布节点中传送命令）来运行（项目）。

在代码变更之后，你可以选择使用 `redeploy.sh` 和 `redeploy.bat` 脚本来自动编译和重新部署（项目）。但要注意，使用这些脚本需要确保脚本中的 `VERTICLE` 变量与 main verticle 中实际用到的一样。

> 注意：
>
> 另外，Fabric8 项目下有一个 Vert.x Maven plugin。它用于初始化、构建、打包并运行一个 Vert.x 项目。 生成一个与克隆自 Git starter 仓库相似的项目：
>
> ```
> mkdir vertx-wiki
> cd vertx-wiki
> mvn io.fabric8:vertx-maven-plugin:1.0.7:setup -DvertxVersion=3.5.0
> git init
> ```

添加需要的依赖

首先在 Maven `pom.xml` 文件中添加用于 web 处理和渲染的依赖：

```
<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>
```

> 提示：
>
> 正如 `vertx-web-templ-freemarker` 名字所表示的那样，对于流行的模板引擎，Vert.x web 提供了插件式的支持：Handlebars、Jade、MVEL、Pebble、Thymeleaf 以及 Freemarker。

然后添加 JDBC 数据访问相关的依赖：

```
<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>
```

Vert.x JDBC client 库可以支持任何 JDBC-兼容 数据库的访问。自然而然，在我们的项目中，classpath 中需要有 JDBC driver。

HSQLDB 是一个非常知名的关系型数据（使用 Java 编写）。它广泛作为嵌入型数据库（译者注：嵌入你的程序之中）使用，因为这样可以避免依赖第三方数据库服务器而独立运行。在单元和集成测试时，它也常被用作提供易失性内存存储。

在开始阶段，HSQLDB 作为嵌入型数据库非常适合（我们的项目）。它在本地存储文件，并且 HSQLDB library Jar 提供了 JDBC driver，因此 Vert.x JDBC 的配置将会非常简单。

> 提示：
>
> Vert.x 也提供了 MySQL 和 PostgreSQL client 专用的库。
>
> 当然你也可以使用通用的 Vert.x JDBC client 来连接 MySQL 或者 PostgreSQL 数据库，但上面的库使用这两种数据库的网络协议，而不是通过阻塞式的 JDBC API，因此会提供更好的性能
>
> 提示：
>
> Vert.x 也提供了处理流行的非关系型数据库 MongoDB 和 Redis 的库。社区里也提供了其他存储系统的集成，例如 Apache Cassandra、OrientDB 和 ElasticSearch。

剖析 verticle

我们的 wiki 系统只有 `io.vertx.guides.wiki.MainVerticle` 这一个 verticle Java 类。这个类扩展自 `io.vertx.core.AbstractVerticle`，作为 verticle 的基类，其主要：

1. 提供生命周期 `start` 与 `stop` 方法来重写
2. 提供一个名为 `vertx` 的受保护（protected）字段：verticle 被部署所在的 Vert.x 环境的引用
3. 提供一个对某些配置对象的访问器，用来传递一些外部配置给 verticle

为了启动我们的 verticle，只需重写如下的 `start` 方法：

```
public class MainVerticle extends AbstractVerticle {

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

`start` 和 `stop` 方法有 2 种形式：一种没有参数，另一种带有一个 future 对象引用。无参情况表示 verticle 初始化或者 house-keeping phases 总是执行成功，除非有异常被抛出。带有 future 对象参数时，提供了一种更加细粒度的访问，来表明操作成功与否（译者注：通过 future 对象参数回掉判断）。实际上，一些初始化或者 cleanup 代码可能会要求异步操作，因此通过一个 future 对象给出结果理所当然，这符合异步的惯用表现形式。

A word on Vert.x future objects and callbacks

Vert.x future 并不是 JDK 的 future：Vert.x future 可以在非阻塞式程序中被组织并检查结果。他们应被用于异步任务的简单协调，尤其是在部署 verticle 时检查部署成功与否。

Vert.x core API 基于回掉来实现异步事件的通知。富有经验的开发者自然会认为这开启了“回掉地狱”之门，如同下面这个虚构的例子一样，多层的异步嵌套会使得代码难以理解：

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

尽管 core API 在设计上就更加偏向（使用） promise 和 future，但回掉允许不同的编程概念（一起）被使用，因此使用回掉实际上也有道理。Vert.x 并不是一个固执己见的项目，许多异步编程模型的实现都可以使用回掉：reactive extensions (via RxJava)、promises 和 futures、fibers (using bytecode instrumentation) 等等。

既然在像 RxJava 这样的概念发挥影响力之前，所有的 Vert.x API 都是“回掉导向型”的，那么本手册在最开始时将**只使用**回掉，以确保读者可以熟悉 Vert.x 中的核心概念。可以说在刚刚开始时，在许多部分的异步代码块之中，使用回掉来画出一条线来更加容易。但一旦在示例代码中回掉开始让代码变得不再易读，我们就将会引入 RxJava 来展示同样的异步代码如果以处理事件流（streams of processed events）来考虑，将可以表示得更加优雅。

Wiki verticle 初始化的步骤

为了让我们的 wiki 运行起来，需要分 2 步进行初始化：

1. 我们需要建立 JDBC 数据库连接，同时确保数据库模式的存在
2. 同时需要为我们的 web 应用来开启一个 HTTP 服务器

每个步骤都有失败的可能性（例如，HTTP 服务器需要的 TCP 端口可能已经被占用），并且他们不应当并行进行，web 应用的代码首先需要的是数据库可以正常访问。

为了使我们的代码更加简洁，我们为每个步骤定义一个方法，并且采用返回一个 future / promise 对象的形式来告知我们的步骤执行成功与否：

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

```
}

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

由于每个方法返回一个 future 对象，那么 `start` 方法的实现就可成为一个 composition：

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

当 `prepareDatabase` 的 future 成功完成，然后 `startHttpServer` 就被调用，而 `steps` future 完成情况取决于 `startHttpServer` future 的结果。 如果 `prepareDatabase` 遇到了错误， `startHttpServer` 则不会被调用，在这种情况下， `steps` future 将以一个失败的状态完成，并携带一个描述错误的异常。

最终 `steps` 完成： `setHandler` 定义了一个 hander，以供完成时调用。在上面的例子中，我们只是想使用 `steps` 来完成 `setHandler` ，并且通过 `completer` 方法来获得一个 handler。也可以写成：

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

注： `ar` 的类型是 `AsyncResult<Void>` 。 `AsyncResult<T>` 被用来传递异步处理的结果，当成功时可能会有一个 `T` 类型的结果，当失败时传递一个失败异常。

数据库初始化

Wiki 数据库模式由一张表 `pages` 构成，其字段信息如下：

列名	类型	描述
Id	Integer	主键
Name	Characters	Wiki 页的名字，必须是唯一的
Content	Text	Wiki 页 Markdown 内容

数据库操作是典型的“查、插、删、改”操作。在最开始，我们简单的将 SQL 语句以静态常量的形式存储在 `MainVerticle` 类中。请注意，他们是 HSQLDB 可解析的特定 SQL，有可能在其他关系型数据库中并不支持：

```
private static final String SQL_CREATE_PAGES_TABLE = "create table if not exists Pa
ges (Id integer identity primary key, Name varchar(255) unique, Content clob)";
private static final String SQL_GET_PAGE = "select Id, Content from Pages where Nam
e = ?"; // 注
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 = ?";
```

注：语句中的 `?` 是在执行时传递数据的占位符，因此 Vert.x JDBC client 可以防止 SQL 注入。

我们的应用 verticle 需要保持一个 `JDBCClient` 对象（来自 `io.vertx.ext.jdbc` 包）的引用来提供数据库的连接。我们在 `MainVerticle` 中声明了 `dbClient`，并且创建了一个来自 `org.slf4j` 包的通用日志记录器。

```
private JDBCClient dbClient;

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

下面是一个 `prepareDatabase` 方法的完整实现。它尝试获取一个 JDBC client 连接，然后执行 SQL，在 `Pages` 表不存在的情况下来创建表：

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

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

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

```
          future.complete();  // 注 9
        }
      });
    }
  });

  return future;
}
```

注：

1. `createShared` 创建一个共享的连接，其在 `vertx` 实例已知的 verticle 之间共享，通常来说这是一件好事。
2. 通过传递一个 JSON 对象来创建 JDBC client 连接。其中 `url` 指的是 JDBC url。
3. 使用 `url` 、 `driver_class` 等等来配置 JDBC driver 并且指出 JDBC driver 类。
4. `max_pool_size` 指的是并发连接数。这里设为 30 是武断决定，任意选择了一个数字。 5) 获取一个连接是异步操作，其提供给我们一个 `AsyncResult<SQLConnection>` 。它在使用之前必须检测是否可以建立连接（ `AsyncResult` 实际上是 `Future` 的超类接口）。
5. 如果不能得到 SQL 连接，future 就以失败为结果完成，并返回提供了异常（通过 `cause` 方法得到）的 `AsyncResult` 。
6. `SQLConnection` 是成功的 `AsyncResult` 的结果。我们可以使用它来执行 SQL 查询。
7. 在我们检查 SQL 查询执行成功与否之前，我们必须先通过调用 `close` 方法来释放连接，否则 JDBC client 连接池最终将干涸（无连接可用）。
8. 我们成功完成 future 操作。

> 提示：
>
> Vert.x 项目支持的 SQL 数据库模块目前主要关注提供对数据库的异步访问，除了传递 SQL 查询外，并没有提供其余更多（例如，对象-关系映射）。然而，完全可以使用来自社区的更先进的模块，我们尤其推荐了解一下像 jOOq generator for Vert.x 或 POJO mapper 这样的项目。

关于日志

上面引入了一个日志记录器，这里选择使用 SFL4J library。关于日志记录，Vert.x 也不是固执己见的：你可以选择任何流行的 Java 日志库。这里之所以推荐 SLF4J 是因为在 Java 生态之中，它是非常流行的日志抽象和统一库。

我们同样推荐使用 Logback 来作为日志记录器的实现。可以通过添加两个依赖，来同时集成 SLF4J 和 Logback，或者仅仅添加 `logback-classic` ，它将同时添加两个库的依赖（顺便一提，他们来自同一个作者）。

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

默认情况下，SLF4J 将会输出很多来自 Vert.x、Netty、C3PO 以及 wiki 应用的日志事件到控制台。我们可以通过添加一个 `src/main/resources/logback.xml` 配置文件来减少冗杂（查看 https://logback.qos.ch/ 这里获得更多信息）。

```
<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>
```

最后但同样重要，HSQLDB 在内嵌时，与日志记录器集成得并不太好。默认情况下，它将尝试重新配置日志系统，所以我们在执行应用时，需要通过加 `Dhsqldb.reconfig_logging=false` 属性给 Java 虚拟机来禁用它这一点。

HTTP 服务器初始化

HTTP 服务器通过使用 `vertx-web` 项目，来比较容易得为传入的 HTTP 请求来定义分发路由。实际上，Vert.x core API 允许开启 HTTP 服务器并监听传入的连接，但它没有提供任何机制来根据请求 URL 不同来提供不同的 handler。根据 URL、HTTP 方法等等来分发请求到不同的处理 handler，这便是 router 的作用。

初始化过程由设置请求路由，开启 HTTP 服务器组成：

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

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

  server
```

```
      .requestHandler(router::accept)   // 注 5
      .listen(8080, ar -> {   // 注 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;
  }
```

注：

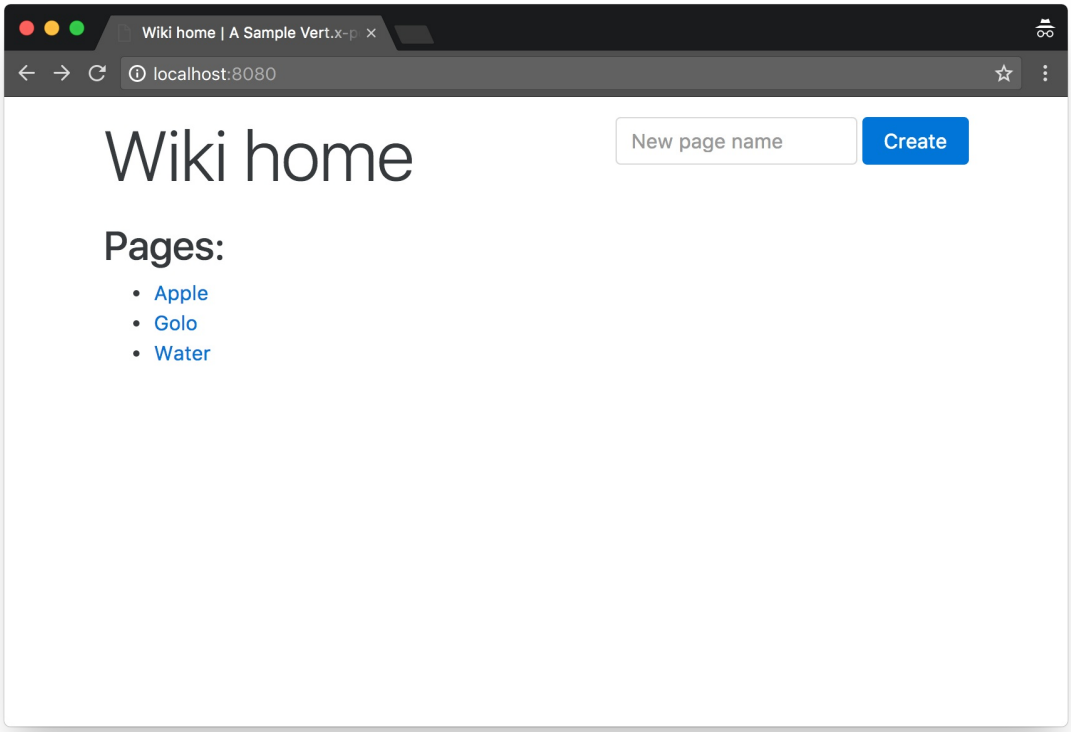
1. `vertx` 上下文对象提供了方法来创建 HTTP 服务器、客户端，TCP/UDP 服务器、客户端等等。
2. `Router` 类来自 `vertx-web`：`io.vertx.ext.web.Router` 。
3. 路由拥有自己的 handler，它们可以根据 URL 或 HTTP 方法来被定义。对于较短的 handler，可以采用 Java lambda 表达式的形式，但对于更复杂的 handler 来说，引用一个私有方法则更佳。注意 URL 可以带有参数，例如 `/wiki/:page` 将匹配类似 `/wiki/Hello` 这样的请求，这样 `page` 参数将被设置为 `Hello` 。
4. 这将会使所有 HTTP POST 请求通过 `io.vertx.ext.web.handler.BodyHandler` 这个 handler。它将自动的解码来自 HTTP 请求（例如，表单提交）（它们可以被作为 Vert.x buffer 对象来使用）的 body 体。
5. router 对象可以被用来作为 HTTP 服务器的 handler，然后分发请求给之前定义的其他 handler。
6. 开启一个 HTTP 服务器是异步操作，因此需要 `AsyncResult<HttpServer>` 来检查操作是否成功。`8080` 参数具体指定了服务器的 TCP 端口。

HTTP router handlers

`startHttpServer` 方法的 HTTP 路由实例根据 URL 模式及 HTTP 方法的不同指向不同的 handler。每一个 handler 处理一个 HTTP 请求，执行数据库查询，以及使用 FreeMarker 模板来渲染 HTML 页面。

索引页（主页） handler

主页提供了所有 wiki 页面的入口及一个创建新 wiki 的区域。



通过一个简单的 `select *` SQL 查询，并将数据交由 FreeMarker 引擎来渲染得到 HTML 响应。

`indexHandler` 方法代码如下所示：

```
private final FreeMarkerTemplateEngine templateEngine = FreeMarkerTemplateEngine.cr
eate();

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() // 注 1
            .getResults()
            .stream()
            .map(json -> json.getString(0))
            .sorted()
            .collect(Collectors.toList());

          context.put("title", "Wiki home");  // 注 2
          context.put("pages", pages);
          templateEngine.render(context, "templates", "/index.ftl", ar -> {   // 注
 3
```

```
            if (ar.succeeded()) {
              context.response().putHeader("Content-Type", "text/html");
              context.response().end(ar.result());  // 注 4
            } else {
              context.fail(ar.cause());
            }
          });

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

注：

1. SQL 查询结果以 `JsonArray` 和 `JsonObject` 实例的形式返回。
2. `RoutingContext` 实例可以放置任意内容的键值对，可以供之后的模板或路由 handler 使用。
3. 渲染一个模板同样是异步操作，也使用 `AsyncResult` 处理方式。
4. 当成功时， `AsyncResult` 包含的是渲染后的内容（以 `String` 形式），因此我们可以使用它来结束 HTTP 响应流。
5. 当失败时， `RoutingContext` 的 `fail` 方法提供了一个合理的途径返回 HTTP 500 error 给 HTTP 客户端。

FreeMarker 模板应当被放置在 `src/main/resources/templates` 目录。 `index.ftl` 模板代码如下所示：

```
<#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">
```

通过 FreeMarker 变量 `context`，可以使用存储在 `RoutingContext` 对象之中的键值对数据。

因为很多模板都有着共同的页头与页脚，所以我们将其分离为 `header.ftl` 和 `footer.ftl`：

```
```html
<!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/CXfBThSRXQ6jW5D
O2ekP5ViFdi" crossorigin="anonymous">
  <title>${context.title} | A Sample Vert.x-powered Wiki</title>
</head>
<body>

<div class="container">
```
```

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

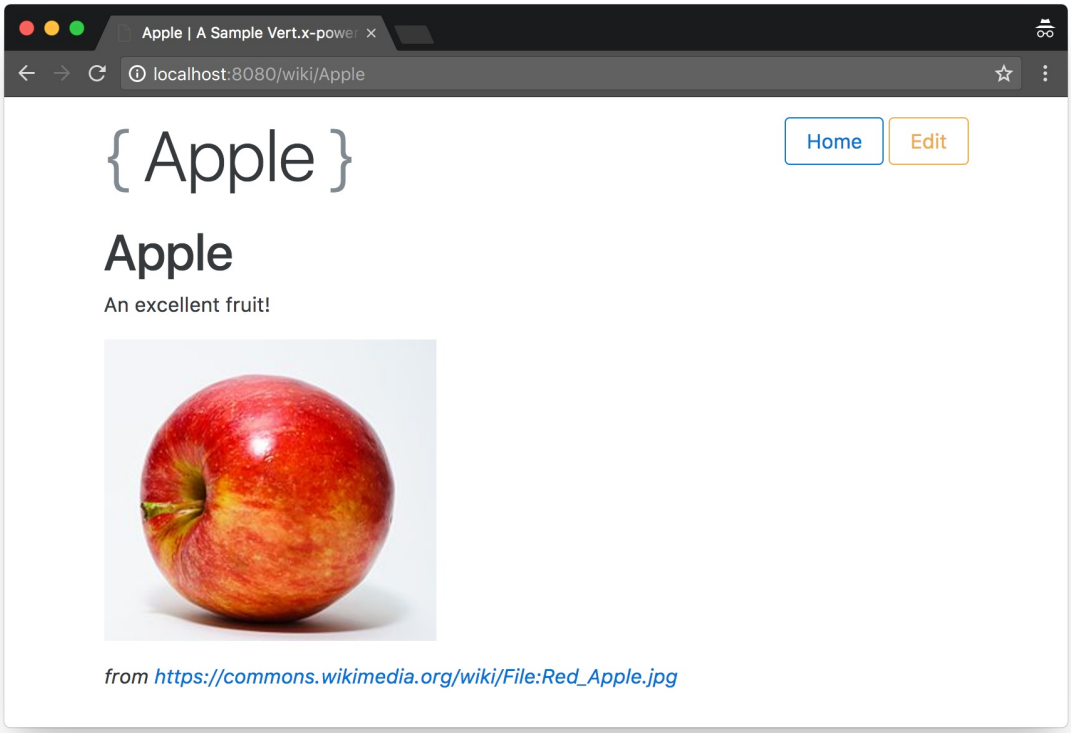
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.1.1/jquery.min.js"
        integrity="sha384-3ceskX3iaEnIogmQchP8opvBy3Mi7Ce34nWjpBIwVTHfGYWQS9jwH
DVRnpKKHJg7"
        crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/tether/1.3.7/js/tether.min.
js"
        integrity="sha384-XTs3FgkjiBgo8qjEjBk0tGmf3wPrWtA6coPfQDfFEY8AnYJwjalXC
```
```

```
iosYRBIBZX8"
        crossorigin="anonymous"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-alpha.5/js/bootstr
ap.min.js"
        integrity="sha384-BLiI7JTZm+JWlgKa0M0kGRpJbF2J8q+qreVrKBC47e3K6BW78kGLr
CkeRX6I9RoK"
        crossorigin="anonymous"></script>

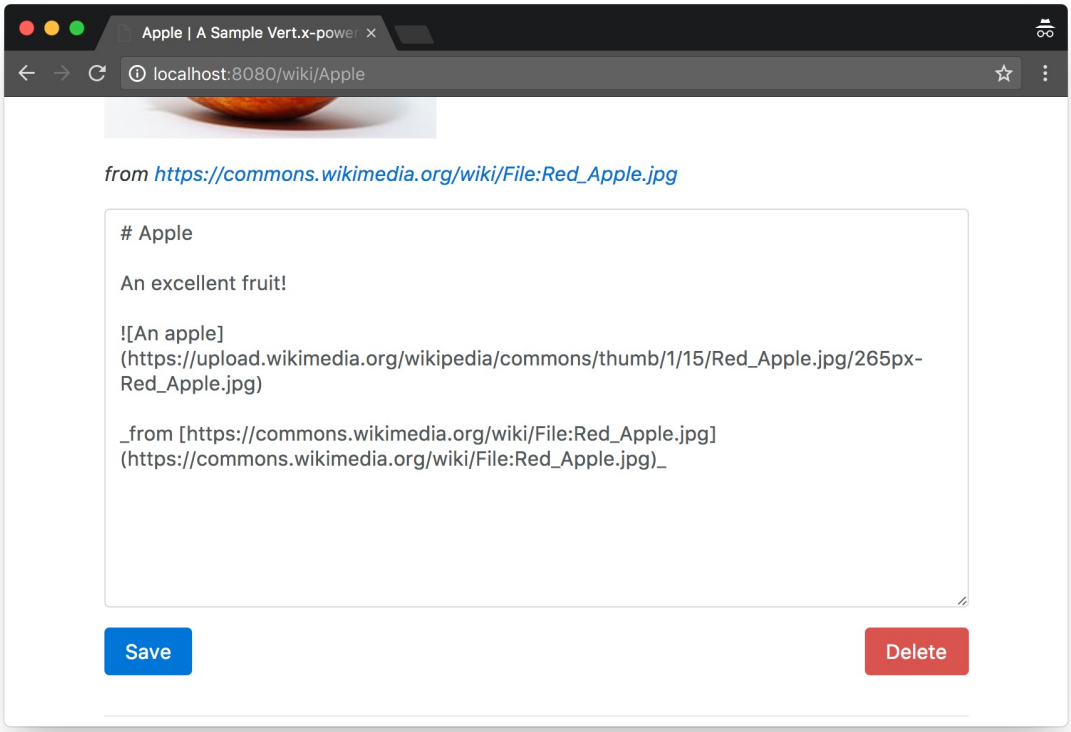
</body>
</html>
```

Wiki 页面渲染 handler

此 handler 处理 HTTP GET 请求，生成一个渲染过的 Wiki 页面，就像下图一样：



此页面同样提供了一个编辑按钮来以 Markdown 形式编辑内容。当按钮被点击时，不需要使用不同的 handler 与模板，只需简单的使用 JavaScript 和 CSS 来切换编辑器的开与关即可。



`pageRenderingHandler` 方法代码如下：

```
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");   // 注 1

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

      SQLConnection connection = car.result();
      connection.queryWithParams(SQL_GET_PAGE, new JsonArray().add(page), fetch ->
{  // 注 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));  // 注 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());
    }
  });
}
```

注：

1. URL 参数（`/wiki/:name`）可以通过 context request 对象取得。
2. 通过 `JsonArray` 按照 `?` 的顺序，来传递参数给 SQL 查询。
3. `Processor` 类来自我们使用的 *txtmark* Markdown 渲染库。

`page.ftl` FreeMarker 模板代码如下所示：

```
<#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="tru
e">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="${context.newPage}">
        <textarea class="form-control" id="markdown" name="markdown" rows="15">${co
ntext.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-rig
ht">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">
```

创建页面 handler

首页提供了一个区域来创建新页面，内容部分的页面的处理由此 handler 负责。此 handler 实际上并没有在数据库中新增一条记录，而是简单的重定向到 Wiki 页面（以名字为 URL 参数）。因为这个页面不存在，所以 `pageRenderingHandler` 方法将在新页面使用默认文本，只有在编辑结束保存时，才最终创建页面。

`pageRenderingHandler` 方法通过 HTTP 303 状态码重定向来实现：

```
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();
  }
```

保存页面 handler

`pageUpdateHandler` 方法处理保存 Wiki 页面时的 HTTP POST 请求。可能有两种情况：一是更新已有的页面（使用 SQL `update`）；或者保存一个新页面（使用 SQL `insert`）

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

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

注：

1. 通过 HTTP POST 请求的表单参数可以通过 `RoutingContext` 取得。注意：如果没有提供 `BodyHandler`，这些参数就无法直接取得，需要从 HTTP POST 请求 payload 中手动解码得到表单数据。
2. 我们需要从 `page.ftl` FreeMarker 模板渲染的页面中取得表单的一个隐藏字段，来得知是更新

页面还是新增一个页面。

3. 同样是采用 `JsonArray` 来传递数据给预编译的 SQL。
4. `updateWithParams` 方法被用来执行 `insert` / `update` / `delete` SQL 操作。
5. 成功后，我们简单地重定向到被编辑后的页面。

删除页面 handler

`pageDeletionHandler` 方法的实现很简单：给定 Wiki 页面的唯一标识，执行 `delete` SQL 操作，然后重定向到 Wiki 首页：

```
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());
    }
  });
}
```

运行我们的应用

至此，我们的 Wiki 应用可以正常工作并且功能完备。

在运行之前，首先要使用 Maven 构建我们的项目：

```
$ mvn clean package
```

因为最终得到的 Jar 文件包含了所有需要的依赖（包括 Vert.x 和 JDBC 数据库），所以运行起我们的应用非常简单：

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

然后就可以使用浏览器访问 http://localhost:8080/ 来享用我们的 Wiki 应用啦~

重构：独立可重用的 Verticle

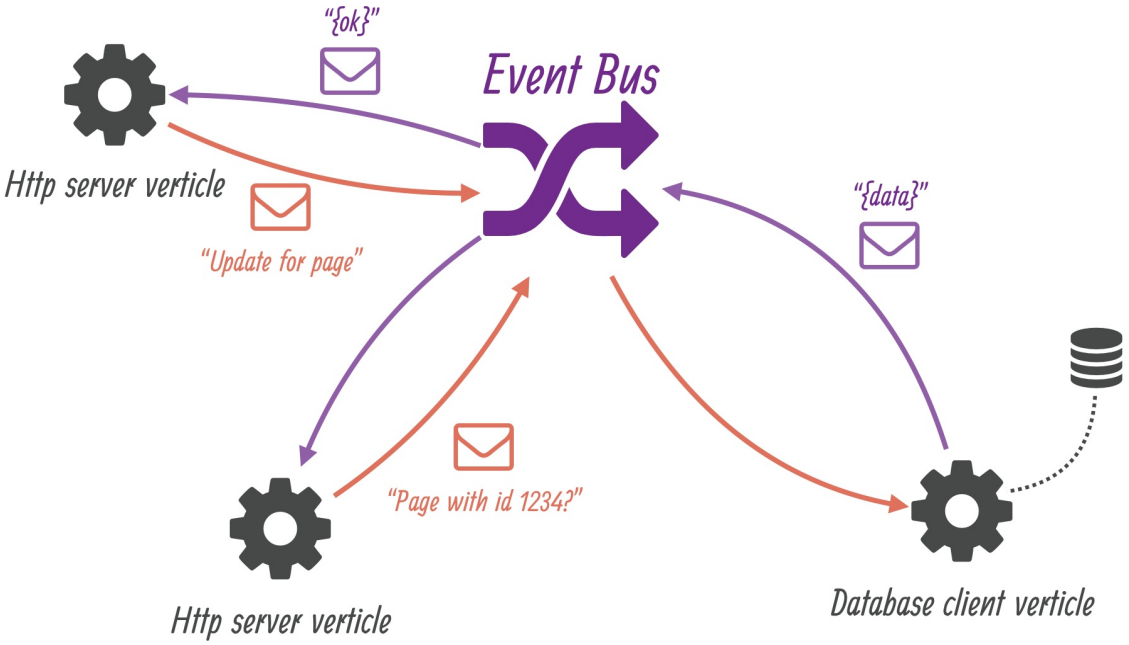
> 提示：
>
> 相关的源代码可以在本手册（注：英文原版）的仓库 step-2 目录下找到。

经过第一次迭代，我们得到了一个可以使用的 Wiki 应用。但它的实现之中仍有一些问题：

1. 处理 HTTP 请求的代码与访问数据库的代码交织在同一个方法之中
2. 许多配置数据（例如：端口号、JDBC 驱动等等）是以字符串的形式硬编码在代码之中

架构与技术选择

迭代的第二个版本设法重构代码，以实现 Verticle 的独立与可重用：



我们将部署 2 个 verticle 来分别处理 HTTP 请求与数据持久化。这 2 个 verticle 之间并不会直接互相引用，它们仅仅通过 event bus 中声明的名字及消息格式来通信。这是一种简单但有效的解耦。

在 event bus 上传递的消息采用 JSON 格式编码。尽管 Vert.x 对于要求高或非常特定的上下文，支持各种灵活的序列化方案，但通常意义上来说，JSON 是一个不错的选择。JSON 的另一个优点就是它是语言无关的文本格式。对于支持多语言的 Vert.x 来说，JSON 就是完美之选，其可以在不同语言编写的 verticle 之间传递消息。

HTTP 服务器 verticle

Verticle 类前半部分和 `start` 方法如下所示：

```
public class HttpServerVerticle extends AbstractVerticle {
```

```
  private static final Logger LOGGER = LoggerFactory.getLogger(HttpServerVerticle.c
lass);

  public static final String CONFIG_HTTP_SERVER_PORT = "http.server.port";  // 注 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");  // 注 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);  // 注 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());
        }
      });
  }

  // (...)
```

注：

1. 我们暴露以下 verticle 配置参数常量为 public：HTTP 端口和 event bus 中接收消息的终点名称（用于存储消息到数据库 verticle）。
2. `AbstractVerticle#config()` 方法允许访问已有的 verticle 配置。在没有配置的情况下，将使用第二个参数。
3. 配置值不只限于 `String` 类型，整形、布尔值、复杂的 JSON 数据等等都可以。

类中其他部分大多数是 HTTP 相关的代码，将之前数据库代码替换为 event bus 消息。下面是 `indexHandler` 方法：

```
private final FreeMarkerTemplateEngine templateEngine = FreeMarkerTemplateEngine.cr
eate();

private void indexHandler(RoutingContext context) {

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

  vertx.eventBus().send(wikiDbQueue, new JsonObject(), options, reply -> {  // 注 1
    if (reply.succeeded()) {
      JsonObject body = (JsonObject) reply.result().body();   // 注 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());
    }
  });
}
```

注：

1. 可以从 `vertx` 对象中取得 event bus，我们为数据库 verticle 向队列发送消息。
2. 传送选项（delivery option）允许我们指定 headers、payload codecs 和 timeouts。
3. 成功时，回应中包含 payload。

可以看到，一个 event bus 消息包含了 body、option 以及预期的回复。当预期没有回复时，可以使用没有 handler 的 `send` 方法。

我们将 payload 编码为 JSON 对象，通过名为 `action` 的消息 header 来指定数据库 verticle 应当执行哪一个操作（`action`）。

verticle 中剩下的代码里路由 handler 同样采用 event-bus 来获取、存储数据：

```
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 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());
    }
  });
}
```

数据库 verticle

通过 JDBC 连接数据库自然需要用到 driver 及其配置，在之前的版本中，其被我们硬编码到代码之中。

可配置的 SQL 查询

虽然 verticle 可以使用先前硬编码的值作为配置参数，但我们可以更进一步，从 properties 文件中加载 SQL 查询语句。

查询语句会从文件中加载来作为配置参数，当不存在时会从默认资源中取得。这种方法的优势在于可以适应不同的 JDBC driver 和 SQL 方言。

Verticle 类前半部分主要是配置键的定义：

```
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_po
ol_size";
  public static final String CONFIG_WIKIDB_SQL_QUERIES_RESOURCE_FILE = "wikidb.sqlq
ueries.resource.file";

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

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

  // (...)
```

SQL 查询被存储在一个 properties 文件，对于 HSQLDB 默认情况下，存储于 `src/main/resources/db-queries.properties`：

```
create-pages-table=create table if not exists Pages (Id integer identity primary ke
y, 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 = ?
```

下面是 `WikiDatabaseVerticle` 类从文件中加载 SQL 查询，并存入一个 map 的代码：

```
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-page
s-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"));
}
```

我们使用 `SqlQuery` 枚举类型来避免之后在代码中使用字符串常量。此 verticle 的 `start` 方法如下所示：

```
private JDBCClient dbClient;

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

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

  dbClient = JDBCClient.createShared(vertx, new JsonObject()
    .put("url", config().getString(CONFIG_WIKIDB_JDBC_URL, "jdbc:hsqldb:file:db/wik
i"))
    .put("driver_class", config().getString(CONFIG_WIKIDB_JDBC_DRIVER_CLASS, "org.h
sqldb.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 -> {
// 注 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);  // 注 3
          startFuture.complete();
        }
      });
    }
  });
}
```

注：

1. 有趣的是我们打破了 Vert.x 中非常重要的一个原则 —— 避免阻塞式 API，但访问 classpath 的资源并没有异步式的接口，因此我们选择非常有限。我们可以使用 Vert.x `executeBlocking` 方法来将阻塞式 I/O 操作从 event loop 中拆解到其他线程（a worker thread），但因为数据非常小，这样做并不会有明显的好处。
2. 这是一个使用 SQL 语句的例子。
3. `consumer` 方法注册一个 event bus 终点 handler。（The `consumer` method registers an event bus destination handler.）

分发请求

event bus 消息的 handler 就是 `onMessage` 方法：

```
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 specif
ied");
    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);
    }
  }
```

我们为各种错误定义了一个 `ErrorCodes` 枚举，其可以被用来报告错误给消息发送者。`Message` 类的 `fail` 方法 提供了一个快捷方便得回复错误的方式，原始的消息发送者会得到一个失败的 `AsyncResult` 。

减少 JDBC 客户端构建代码

到目前为止，可以看到执行 SQL 查询的完整交互：

1. 取得连接
2. 执行请求
3. 释放连接

就像下面的代码一样，这会导致每个异步操作，都需要大量的错误处理代码：

```
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());
  }
});
```

从 Vert.x 3.5.0 开始，JDBC 连接开始支持一步到位的操作，其可以提供一个连接来执行 SQL 操作，然后自己释放掉。与上面代码功能一致，精简过后的代码如下所示：

```
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());
  }
});
```

这对于需要取得连接，只执行一个操作的场景非常有用。当然对于一连串的 SQL 操作来说，重复使用一个连接会使得性能更好。

类中剩下的代码包含 `onMessage` 分发传入的消息后需要被调用的私有方法：

```
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());
  }
```

在 main verticle 中部署 verticles

我们仍然有 `MainVerticle` 类，但不同于第一版包含所有业务逻辑代码，它的作用只是启动应用，部署其他verticle。

下面的代码部署了一个 `WikiDatabaseVerticle` 实例，两个 `HttpServerVerticle` 实例：

```
public class MainVerticle extends AbstractVerticle {

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

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

    dbVerticleDeployment.compose(id -> {  // 注 3

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

      return httpVerticleDeployment;  // 注 6

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

注：

1. 部署一个 verticle 是异步操作，所以我们需要一个 `Future` 。参数类型是 `String` 的原因是在成功部署之后，会得到一个 verticle 的标识符。
2. 部署的一种方式是使用 `new` 来创建一个 verticle 实例，然后将其对象引用传递给 `deploy` 方

法。`completer` 返回的值是一个 handler，其用来简单的完成这个 future。
3. 使用 `compose` 按顺序的组合可以实现在其后执行异步操作。当前面的 future 成功完成，组合函数就被激活。
4. 可以通过类名作为字符串来指定一个 verticle 来部署。对于其他 JVM 语言来说，基于字符串的允许一个 module / script 被指定。
5. `DeploymentOption` 类允许指定部署的实例个数。
6. 组合函数返回下一个 future。它的完成将触发组合操作的完成。
7. 我们定义了一个 handler 来最终完成 `MainVerticle` start future。

聪明的你可能会奇怪为什么 HTTP server 可以被部署到同一个 TCP 端口两次，而不会出现因为端口占用的错误。因为对于大多数的 web 框架来说，我们需要选择不同的 TCP 端口，并且需要一个前置的 HTTP 代理来实现端口间的负载均衡。

而 Vert.x 的 verticle 可以实现多个 verticle 共享一个 TCP 端口。来自接收线程（accepting threads）传入的网络连接会简单得基于轮询模式分配。

再重构：Vert.x 服务

> 提示：
>
> 相关的源代码可以在本手册（注：英文原版）的仓库 step-3 目录下找到。

相较于最初的实现来说，之前的重构已经是一个巨大的进步：其基于 event bus，独立可配置的 verticle 通过异步消息来连接；同时，我们部署的几个 verticle 实例可以更好的应对负载以及更好的利用 CPU 核心。

在下面这个部分之中，我们将看到如何设计并使用 Vert.x 服务（services）。服务的优势在于，它为 verticle 需要做的具体操作定义了接口。同时，不再像之前那样自己去处理消息，而是利用代码生成来使用 event bus 消息。

Java 代码将被重构为以下几个包：

```
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` 现在包含 main verticle
- `io.vertx.guides.wiki.database` 包含数据库 verticle 以及 service
- `io.vertx.guides.wiki.http` 包含 HTTP 服务器 verticle

调整 Maven 配置

首先，我们的项目需要添加以下两个依赖。第一个是 `vertx-service-proxy` API：

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

第二，我们需要 Vert.x 代码生成模块，其只在编译时依赖（因此 scope 被设置为 `provided`）：

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

另外为了能够生成代码，我们还需要稍微调整一下 `maven-compiler-plugin` 配置，通过一个 `javac` 注解处理器来实现：

```
<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</annotationProce
ssor>
        </annotationProcessors>
        <generatedSourcesDirectory>${project.basedir}/src/main/generated</generated
SourcesDirectory>
        <compilerArgs>
            <arg>-AoutputDirectory=${project.basedir}/src/main</arg>
        </compilerArgs>

    </configuration>
</plugin>
```

注意：生成的代码放置于 `src/main/generated`，像 IntelliJ IDEA 这类 IDE 会自动将其加入 classpath。

为了移除生成的多余文件，更新 `maven-clean-plugin` 如下：

```
<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>
```

数据库服务接口

定义一个服务接口就像定义 Java 接口一样简单，除了必须要遵守一些特定规则，以生成代码及保证 Vert.x 中的其他代码可以与之相互操作。

接口最开始定义为如下形式：

```
@ProxyGen
public interface WikiDatabaseService {

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

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

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

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

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

    // (...)
```

1. 通过 `ProxyGen` 注解的使用来触发此 service 的客户端代理代码的生成。
2. `Fluent` 注解是可选的，它意味着接口的方法可以链式调用。当 service 可能会被其他 JVM 语言使用的时候，这对于代码生成器来说很有用。
3. 参数类型默认只能是字符串、Java 基础类型、JSON 对象或数组、枚举类型、或者之前提到的类型的集合类型（ `java.util` ）（ `List` 、 `Set` 、 `Map` ）。若想使用其他任意的 Java 类作为 Vert.x 数据对象，需要为它们添加 `@DataObject` 注解。另外，还可以传递 service 引用类型变量。
4. 因为 service 提供了异步结果，所以最后一个参数定义为 `Handler<AsyncResult<T>>` ，在代码生成时， `T` 是上面描述的任意一种适当的类型。

service 接口提供一个静态方法来为所有实际的实现提供实例对象，以及提供一个底层基于 event bus 的客户端代理，这样是比较好的实践。

我们简单得通过实现类的构造方法来将任务委派给实现类，将其定义为 `create` ：

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

```
}
```

Vert.x 代码生成器创建代理类，并使用类名加上 `VertxEBProxy` 后缀作为命名。代理类的构造方法需要 Vert.x 上下文引用以及 event bus 的目标地址。

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

> 注意：在上一版中，我们将 `SqlQuery` 和 `ErrorCodes` 枚举类型定义为了内部类，而这一版中，它们分别定义在 `SqlQuery.java` 和 `ErrorCodes.java` 之中。（译者注：可以直接去最前面提到的地址中获取代码）

数据库服务实现

数据库服务的实现就是上一版 `WikiDatabaseVerticle` 的简易版本。最主要的区别就是构造函数提供异步结果 handler（来报告初始化结果），以及方法也同样提供了异步结果（来告知操作是否成功）。

类代码如下所示：

```
class WikiDatabaseServiceImpl implements WikiDatabaseService {

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

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

    WikiDatabaseServiceImpl(JDBCClient dbClient, HashMap<SqlQuery, String> sqlQueri
es, 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>> result
Handler) {
        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<JsonObjec
t>> 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<As
yncResult<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<AsyncResul
t<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()));
        }
        });
        return this;
    }

    @Override
    public WikiDatabaseService deletePage(int id, Handler<AsyncResult<Void>> result
Handler) {
        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;
    }
}
```

为了能够生成代理代码，还需要做一件事：在 service 包下增加一个 `package-info.java` 注释以用来定义一个 Vert.x 模块：

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

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

在数据库 verticle 中暴露数据库服务

因为大部分数据库处理代码都被转移到了 `WikiDatabaseServiceImpl` 之中，所以 `WikiDatabaseVerticle` 类现在只包含两个方法： `start` 方法用于注册服务；另一个功用方法用于加载 SQL 语句：

```
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_po
ol_size";
  public static final String CONFIG_WIKIDB_SQL_QUERIES_RESOURCE_FILE = "wikidb.sqlq
ueries.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/w
iki"))
      .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-pa
ges-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;
  }
}
```

- 注：
- 我们在此处注册服务。

注册一个服务需要：一个接口类、Vert.x 上下文对象、实现类以及 event bus 目标地址。

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.

`WikiDatabaseServiceVertxEBProxy` 生成的类将处理来自 event bus 的消息，并将他们分发给 `WikiDatabaseServiceImpl` 去处理。这与之前我们编写的代码所做的事情非常接近：在上一节中，发送的消息中携带着一个名为 `action` 的头部，其指定了调用哪一个方法，参数采用 JSON 的形式。

使用一个数据库服务代理

将项目重构为使用 Vert.x 服务的最后一步就是改写 HTTP server verticle，其代码的 handler 中不再直接使用 event bus，转而使用数据库服务代理。

首先，我们需要在 verticle 启动时创建一个代理：

```
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();
  // (...)
```

- 注：
- 我们只需要保证此处的 event bus 目标地址与我们在 `WikiDatabaseVerticle` 发布服务时的地址一致即可。

然后，我们需要将代码中 event bus 的调用改为数据库服务：

```
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) {
  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());
    }
  });
}
```

- `WikiDatabaseServiceVertxProxyHandler` 生成的类来处理转发的调用（就像之前的 event bus 消息一样）。

> 提示：
>
> 虽然生成了代理类来做这件事，但依然可以通过 event bus 消息来直接使用 Vert.x 服务。

测试 Vert.x 代码

> 提示：
>
> 相关的源代码可以在本手册（注：英文原版）的仓库 step-4 目录下找到。

截至目前，我们开发 wiki 系统过程中并没有进行测试。这当然不是一个好习惯，所以让我们看看怎么编写测试代码。

开始

在 Vert.x 中，`vertx-unit` 模块提供了测试异步操作的工具。除此之外，你还可以使用像是 JUint 这样的测试框架。

使用 JUnit 进行测试的话，Maven 依赖应当如下所示：

```
<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 测试代码需要添加注解 `VertxUnitRunner`，以使用 `vertx-unit` 的特性：

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

引入这个 runner 后，JUnit 测试方法及生命周期方法都可以接受一个名为 `TestContext` 参数。其提供了基础断言、存储数据了的上下文，以及一些异步的工具。

让我们假设一个异步场景，我们想要检查一个 timer 任务是否被调用了一次，一个 periodic 任务是否被调用了三次。因为代码是异步的，所以测试方法会在测试完成之前就结束执行。因此我们的测试也需要以异步形式编写。

```
@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` 是 runner 提供的参数。
2. 我们需要创建一个 Vert.x 上下文来完成这个测试。
3. 这是一个基础 `TestContext` 断言的例子。
4. 我们得到一个 `Async` 对象，其可以稍后被标记为完成或失败。（译者注：它是一个测试的异步退出点）
5. 这个 `Async` 对象就如同一个计数器一样，在 3 次调用后，会被标记为完成。
6. 当 timer 执行的时候，测试正确完成。
7. 周期任务每次执行都会触发计数器。当 `Async` 对象完成的时候，测试通过。
8. 对于异步测试来说，有一个默认的超时限制，但可以在 JUnit `@Test` 注解中覆盖默认设定。

测试数据库操作

数据库服务非常适合编写测试代码。

首先我们先部署数据库 verticle。我们将其配置为 JDBC 连接 HSQLDB（内存存储数据库），当成功后我们取得一个“服务代理”来进行我们的测试。

因为这些操作过于细节（involving），所以我们引入 JUnit `before`，`after` 生命周期方法：

```
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;shutd
own=true")
    .put(WikiDatabaseVerticle.CONFIG_WIKIDB_JDBC_MAX_POOL_SIZE, 4);

  vertx.deployVerticle(new WikiDatabaseVerticle(), new DeploymentOptions().setConfi
g(conf),
    context.asyncAssertSuccess(id ->  // (2)
      service = WikiDatabaseService.createProxy(vertx, WikiDatabaseVerticle.CONFIG_
WIKIDB_QUEUE)));
}
```

1. 我们只覆盖一部分 verticle 的设置，其他的会采用默认值。
2. `asyncAssertSuccess` 提供了一个 handler 来检查异步操作的结果。它还有一个无参形式的重载，但对于有参形式来讲（如此处所示），我们可以将结果链接到另一个 handler。

清理Vert.x上下文非常简单，另外此处我们还是使用 `asyncAssertSuccess` 来检查是否有错误发生。

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

service的操作本质上是查插删改，所以 JUnit 最好将他们组合起来进行测试：

```
@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. 这是 `Async` 结束完成的唯一退出点。
2. 还可以通过通过 JUnit 超时来退出测试。此处的测试线程会等待 `Async` 完成或者时间超时。

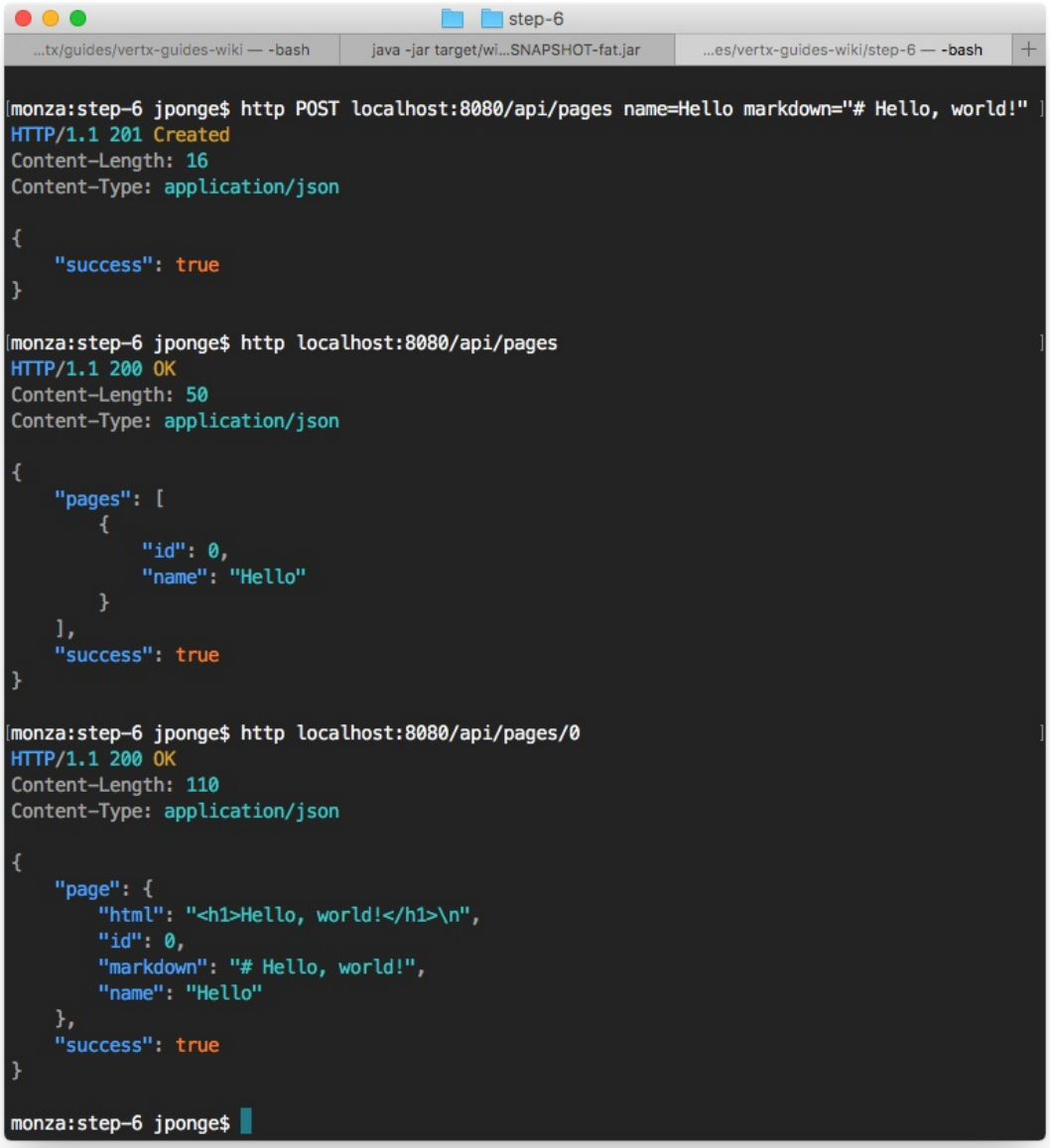
实现 web API

> 提示：
>
> 相关的源代码可以在本手册（注：英文原版）的仓库 step-6 目录下找到。

通过我们已经使用过的 `vertx-web` 模块，非常容易地就可以实现 HTTP/JSON 形式的 web API。我们接下来会使用如下 URL 来暴露我们的 web API：

- `GET /api/pages` 提供所有页面名字及 ID 的描述
- `POST /api/pages` 根据请求创建一个新的 wiki 页面
- `PUT /api/pages/:id` 根据请求更新一个 wiki 页面
- `DELETE /api/pages/:id` 删除一个 wiki 页面

下面是使用 HTTPie（一个命令行工具）与我们的 API 交互的截图：



Web 子路由

我们将向 `HttpServerVerticle` 添加一些新的路由hanlder。虽然我们可以直接向已经存在的 router 之中添加，但更好的方式是使用子路由（sub-routers）。一个路由可以作为子路由，挂载到另外一个上面，这有利于组织或重利用 handler。

以下是 API 路由的代码：

```
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)
```

- 注：
- 我们在此处挂载我们的路由，因此所有以 `/api` 为路径的请求都会被定向到 `apiRouter` 之中。

Handlers

以下是不同 API 的路由hanlder 的代码。

根资源

```
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());
    }
  });
}
```

- 注：
- 此处只是简单的将数据库查询结果中的信息记录对象重新映射。
- 填充返回的 payload 中的 `pages` 的值。
- `JsonObject#encode()` 方法提供 JSON 数据紧凑的（译者注：无换行等） `String` 形式数据。

获取一个页面

```
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());
  });
}
```

创建一个页面

```
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());
    }
  });
}
```

此 handler 需要处理传入的 JSON 数据。使用 `validateJsonPageDocument` 方法可以校验数据，并尽早的报告出错误，这样后面的处理就可以假定所需的数据记录是存在的：

```
private boolean validateJsonPageDocument(RoutingContext context, JsonObject page, S
tring... expectedKeys) {
  if (!Arrays.stream(expectedKeys).allMatch(page::containsKey)) {
    LOGGER.error("Bad page creation JSON payload: " + page.encodePrettily() + " fro
m " + 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;
}
```

更新一个页面

```
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);
  });
}
```

`handleSimpleDbReply` 方法是为了处理请求的一个帮助方法：

```
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());
  }
}
```

删除一个页面

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

对 API 进行单元测试

我们在 `io.vertx.guides.wiki.http.ApiTest` 类中编写了一个基本的测试用例。

我们首先要准备好测试环境。HTTP服务器verticle需要数据库verticle，因此我们将他们一起部署到我们的测试Vert.x上下文中。

```
@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;shu
tdown=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());
  }

  // (...)
```

- 注：
- 对于测试，我们使用一个不同的 JDBC URL，将数据存储在一个内存数据库中。

一个完整的测试用例就是一个所有类型请求都会执行的场景。此处，它创建页面，获取它，更新然后删除它：

```
@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?"));
}
```

> 提示：
>
> 测试之中使用了 `Future` 对象，而不是嵌套的回调；最后一个部分必须将 `async` 标记为完成，否则的话测试最终只能超时结束。

安全保证及访问控制

> 提示：
>
> 相关的源代码可以在本手册（注：英文原版）的仓库 step-7 目录下找到。

使用 Vert.x，很容易就可以实现安全保证及访问控制。在本小节中，我们将会：

1. 使用 HTTPS 代替 HTTP
2. 对 Web 应用增加基于组权限的用户认证
3. 使用 JSON web tokens (JWT) 控制对 web API 的访问。

Vert.x 中使用 HTTPS

Vert.x 提供了 SSL-加密 网络连接的支持。在生产环境中，将服务通过 HTTP 协议暴露给像是 Nginx 这样的前置代理服务器，再由其使用 HTTPS 协议与客户端连接的做法很常见。虽然如此，但 Vert.x 也可以自身使用 HTTPS 协议来提供服务端与客户端之间的加密。

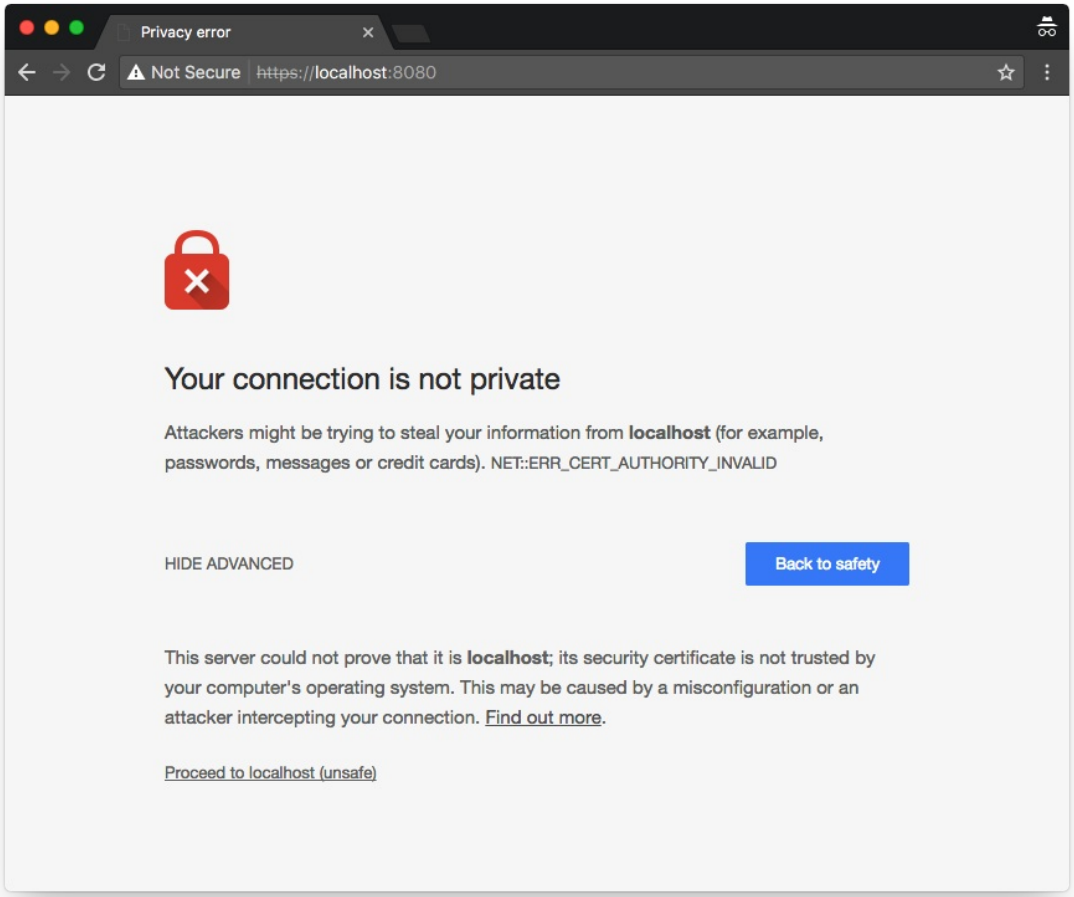
证书可以使用 Java KeyStore 文件的形式。可以自己签发一个证书来进行测试，以下命令可以创建一个名为 `server-keystore.jks` 的 KeyStore 文件，其密码为 `secret` 。

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

我们可以在 HTTP 服务器创建时，传入一个 `HttpServerOptions` 对象来指明使用 SSL，并指出 KeyStore 文件位置。

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

我们可以用浏览器访问 https://localhost:8080/ ，但证书是自签发的，因此正常的浏览器都会给与安全警告来阻止访问：



最后但也值得注意的是，因为 `ApiTest` 中的 Web 客户端只是处理 HTTP 请求，所以我们也需要更新测试用例代码：

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

1. 确保使用 SSL。
2. 因为证书是自签发的，所以我们需要明确指出信任该证书，否则连接就会像浏览器那样失败。