Community Experience Distilled

Real-time Web Application Development using Vert.x 2.0

An intuitive guide to building applications for the real-time web with the Vert.x platform

Tero Parviainen

[PACKT] PUBLISHING open source* community experience distilled

Real-time Web Application Development using Vert.x 2.0

An intuitive guide to building applications for the real-time web with the Vert.x platform

Tero Parviainen

BIRMINGHAM - MUMBAI

Real-time Web Application Development using Vert.x 2.0

First published: September 2013

Production Reference: 1170913

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-795-2

www.packtpub.com

Cover Image by Abhishek Pandey (abhishek.pandey1210@gmail.com)

Credits

Author
Tero Parviainen

Reviewers
Marko Klemetti
Mikko Nylén
Fredrik Sandell

Acquisition Editor
Vinay Argekar

Commissioning Editor
Sharvari Tawde

Technical Editors
Sandeep Madnaik
Shali Sasidharan

Project Coordinator
Joel Goveya

Proofreader
Clyde Jenkins

Indexer
Tejal Soni

Graphics
Sheetal Aute
Valentina Dsilva

Production Coordinator
Nitesh Thakur
Kirtee Shingan

Cover Work
Kirtee Shingan

About the Author

Tero Parviainen has been building software professionally for about 12 years, mostly for the web and most of it with Java, Ruby, JavaScript, and Clojure. From large enterprise back-office systems to consumer mobile applications, he has worked in a variety of different environments.

He currently works as an independent software maker, focusing mainly on software development contracting and training for several customers. He can be found on Twitter and GitHub as `@teropa`.

I'd like to thank my colleagues at Deveo and Eficode for all their support during the writing process.

I'd also like to thank the reviewers for this book: Marko Klemetti, Mikko Nylén, and Fredrik Sandell. The content and flow of the book was improved immensely based on their feedback and ideas.

Finally, I'd like to thank all the people from Packt Publishing who have helped me through the book authoring process: Ekta Patel, Ameya Sawant, Joel Goveya, and Sharvari Tawde.

About the Reviewers

Marko Klemetti (`@mrako`) is a father of three, a leader, and a developer. He is currently the head of the leading Finnish Devops Unit in Eficode (`http://www.eficode.com`). With his team, he changes the way Finnish and multinational organizations create and purchase software. He is also the founder and architect of Trail (`http://www.entertrail.com`), an internationally successful solution for Social Asset Management in Performing Arts.

He has specialized in bringing efficiency to large software production environments by applying modern software development practices and tools, such as **Continuous Delivery** (**CD**) and **Acceptance Test Driven Development** (**ATDD**). With his two decades of software development experience, he is able to engage both the executives and the developers in process change. He is passionate in making programming both fun and productive at the same time.

Mikko Nylén is an enthusiastic software developer who has multiple years of experience in developing large-scale web applications and services using JVM, node.js, and Ruby for some of the top companies in Finland.

Fredrik Sandell is a 26-year-old software developer with a Masters degree in Computer Science from Chalmers University of Technology. He loves software development and is especially interested in JVM technologies and web development. He has developed software privately and professionally for more than 10 years and has been a part of development projects from some of the largest telecom companies in Sweden.

ww.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

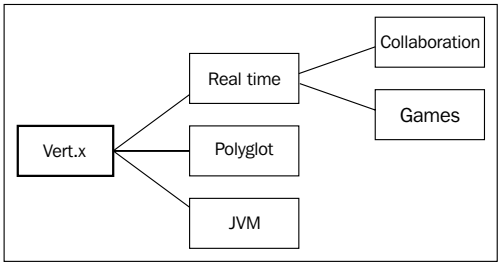
The real-time web is here. We've come a long way from the collection of linked, static pages that started it all; from the very first web applications in the 1990s, through the Ajax revolution about seven years ago, to the latest crop of highly dynamic web properties.

The web applications of today are alive. Facebook, Gmail, corporate financial dashboards, chat services, and stock tickers all give us the information we want, when we want it. This is made possible by new web standards, increasingly high-speed networks, and the multitude of new categories of devices from smartphones to smart televisions.

To build these new kinds of applications, we also need new kind of technology. The web frameworks that powered much of the page-based web aren't necessarily cut out to serve the real-time web. This is where Vert.x comes in. Vert.x has been designed from the ground up to enable you to build scalable, real-time web applications.

Real-Time Web Application Development using Vert.x 2.0 will show you how to build a real-time web application with Vert.x. During the course of the book, you will go from the very first "Hello, World" to publishing a fully featured application on the Internet.

The book presents one extended example application: a real-time mind map editor. Mind maps are a popular technique for capturing ideas and information in a connected tree of words and concepts.



We will build an editor with which people can create mind maps like this. In addition, we will allow people to do it collaboratively. Multiple people will be able to make changes to the same mind map simultaneously from their web browsers, and see each other's changes happen in real time.

Why Vert.x?

There are many techniques and technologies one could employ to develop a real-time web application. From Node.js and Meteor to Ruby's EventMachine, and the WebSocket support in Java EE 7, application developers have an abundance of choices. However, Vert.x provides a unique combination of qualities that makes it a very attractive choice.

Simple

The design of Vert.x is remarkably coherent and simple. This enables you, the application developer, to build substantial systems without getting tangled in complexity. A Vert.x application consists of one more loosely coupled components, called Verticles, running concurrently. However, the application developers never need to think about concurrency, and the co-ordination difficulties it entails. Instead, we write our code as if it was single threaded, and communicate with other parts of the system asynchronously. In this sense, Vert.x is similar to actor-based systems, such as Akka (`http://akka.io/`), though there are some key differences that will become clear during the course of the book.

Polyglot

Vert.x is a so-called polyglot application platform. This means that Vert.x isn't built for a specific programming language, but actually supports several. The languages currently supported are JavaScript, Java, Ruby, Python, CoffeeScript, and Groovy. At the time of writing, there are also ongoing efforts to add support for Clojure and Scala.

This means whatever your programming background might be, you are likely to be right at home on the Vert.x platform. You can even mix and match languages within a single application, always choosing the one that matches a particular problem best.

The Java platform

Vert.x runs on the Java Virtual Machine. That means your application will benefit from the thousands of engineering years that have gone into making the JVM a performing, stable, and robust platform that it is now. You can also tap into the vast amount of Java technologies that are available out there.

General purpose

Vert.x is not just a web framework. Strictly speaking, it is a platform for all kinds of scalable, networked applications. In this sense, it is not unlike Node.js, for example. In addition to web applications, you can use Vert.x to build anything from BitTorrent clients to electronic trading systems.

In this book, however, we will concentrate on building a real-time web application. As you build the application, you will see how Vert.x is really not your typical web framework, but a versatile platform suitable for many purposes. Once you're done, you will be well equipped to build your own applications on Vert.x, whatever they might be.

What this book covers

Chapter 1, *Getting Started with Vert.x*, guides you through the installation of the Vert.x 2.0 platform and its prerequisites. In this chapter, you'll also write your very first Vert.x application: the web equivalent of "Hello, World".

Chapter 2, *Developing a Vert.x Web Application*, covers the development of a full-fledged Vert.x web application, including both the server and browser components. You will become familiar with the architecture of a typical Vert.x application.

Chapter 3, *Integrating with a Database*, extends the web application from the previous chapter by adding support for persisting data in a MongoDB database, using one of the available open source Vert.x modules: the MongoDB Persistor.

Chapter 4, *Real-time Communication*, builds on everything you've learned so far to deliver the secret sauce: real-time communication. You will develop a real-time, collaborative, browser-based mind map editor.

Chapter 5, *Polyglot Development and Modules*, presents some of the polyglot features of Vert.x, as well as the development of reusable and distributable modules, by creating a Java module that is used to save mind maps as PNG images.

Chapter 6, *Deploying and Scaling Vert.x*, shows how to deploy your Vert.x application on Internet, by setting up a Linux server with continuous deployment. Finally, we discuss the basics of scaling Vert.x for growing amounts of users and data.

What you need for this book

You will need a computer with an operating system that is supported by both Vert.x and Java. For this purpose, any reasonably recent version of Mac OS X, Windows, or Linux is fine.

You'll need an editor or an IDE, with which you can comfortably work with JavaScript and Java code. We assume you have a preference, but if you don't, you can use the text editor that comes with your operating system. If you're looking for a good code editor, I recommend Sublime Text (http://www.sublimetext.com/).

You will need some additional software, such as Java, MongoDB, and Vert.x itself. For all of these, I'll let you know when and how to obtain them during the course of the book.

Who this book is for

The book is aimed at web developers who want to take their skills to the next level and start building real-time web applications. In order to be able to follow along, you should have a working knowledge of the common technologies used in web development, particularly JavaScript and HTML.

Since Vert.x runs on the Java Virtual Machine, familiarity with Java will help you understand some of the underlying mechanics. You will also write some Java code. However, you do not need to be a Java developer to be able to follow along. Vert.x welcomes developers with all kinds of backgrounds, including Ruby and Python. In this book, I want to do the same.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "The `vertx` command has multiple uses, but the main one is to launch a Vert.x instance, which is something you'll be doing a lot."

A block of code is set as follows:

```
eventBus.registerHandler('mindMaps.list', function(args,
  responder) {
  responder({"mindMaps": Object.keys(mindMaps).map(function(key) {
    return mindMaps[key];
  })});
});
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
var container = require("vertx/container");
container.deployModule("io.vertx~mod-web-server~2.0.0-final", {
  port: 8080,
  host: "localhost",
  bridge: true,
  inbound_permitted: [
    { address: 'mindMaps.list' },
    { address: 'mindMaps.save' },
    { address: 'mindMaps.delete' }
  ]
});
vertx.deployVerticle('mindmaps.js');
```

Any command-line input or output is written as follows:

```
$ vertx run app.js
mindmaps.js deployed
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "In Chrome, navigate to **View | Developer | JavaScript Console**".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Getting Started with Vert.x

The web frameworks that powered much of the page-based web aren't necessarily cut out to serve the real-time web. This is where Vert.x excels. Vert.x has been designed from the ground up to enable you to build scalable and real-time web applications.

In this chapter, you will take your very first steps in Vert.x development by installing Vert.x, and running it from the command line.

You will also get familiar with some key Vert.x concepts and put them into use by creating a web server for serving static files.

Installing Vert.x

To install Vert.x, you simply download and unpack a ZIP file. A Vert.x installation lives in a self-contained directory structure, which can be located anywhere on your machine.

It's also recommended that you include the path to the Vert.x executable in your `PATH` environment variable. This will make working with the Vert.x command line easier.

In the following pages, we'll go through these steps, after which you'll have everything up and running.

Prerequisite – Java

Vert.x is written in Java, and needs the Java JDK 7 to run the program. Older versions of Java are not supported.

Checking your version of Java

If you're not sure whether you already have Java or you're not sure about its version, launch a terminal or a command prompt and check the Java version:

```
java -version
```

In the output, you should see a version number beginning with 1.7. If not, you'll need to obtain the JDK before you can start working with Vert.x.

If you are on Windows, it may be the case that Java is installed but the java command still isn't available in the command prompt. If you suspect this is the case, check if the directory `C:\Program Files\Java` exists. If it has a subdirectory beginning with `jdk1.7.0` you're good to go.

Obtaining Java

If you're running Mac OS X or Windows, I recommend you to install Oracle's Java SE Version 7 or newer. You will find it from Oracle's website at `http://www.oracle.com` (be sure to select the full JDK or the **for Developers** option, and not just the JRE or the **for Consumers** option). Follow the instructions provided by the web page and the installer to complete the installation.

If you're running Linux, I recommend you install OpenJDK 7 from your package manager, if available. On Ubuntu and Debian, it will be in the `openjdk-7-jdk` package:

```
apt-get install openjdk-7-jdk
```

Alternatively, Oracle's Java SE is also available for Linux.

Obtaining a Vert.x distribution

Because Vert.x is built in Java, the distribution package is the same regardless of your operating system or hardware. Head to `http://vertx.io/`, and download the latest version of Vert.x.

In the following instructions, the Vert.x version is marked as `x.x.x`. Substitute it with the version of Vert.x that you have downloaded.

The next steps will be different for different operating systems. You'll need to unpack the downloaded ZIP file and add the `bin` directory of the distribution to your `PATH` environment variable. This will allow you to easily launch Vert.x from anywhere on your machine.

OS X and Linux

If you are using the Homebrew package manager on OS X, you can just install the Vert.x package and skip these steps.

Other package manager integrations may exist, but obtaining the distribution from the Vert.x website is still the most common installation method.

1. Start a command line shell (such as the terminal applications on OS X and Ubuntu), and navigate to a directory into which you want to install Vert.x. For example, if you have one named `dev` within your `Home` directory:

   ```
   $ cd ~/dev
   ```

2. Unzip the Vert.x distribution package you downloaded earlier. This will create a folder named `vert.x-x.x.x.final`, from where we will run Vert.x.

   ```
   $ unzip ~/Downloads/vertx-x.x.x.final.zip
   ```

3. Open your `.bash_profile` file in a text editor.

   ```
   $ pico ~/.bash_profile
   ```

4. Add the path to the `bin` subdirectory of the Vert.x distribution to the `PATH` environment variable, by adding the following line:

   ```
   export PATH=$PATH:$HOME/dev/vert.x-x.x.x.final/bin
   ```

5. Save and close the editor.
6. The Vert.x executable will now be in your `PATH` for all the future terminal windows you open. You can also apply this change immediately to your current terminal window by typing:

   ```
   $ source ~/.bash_profile
   ```

You can now proceed to *Running Vert.x* section to test your installation.

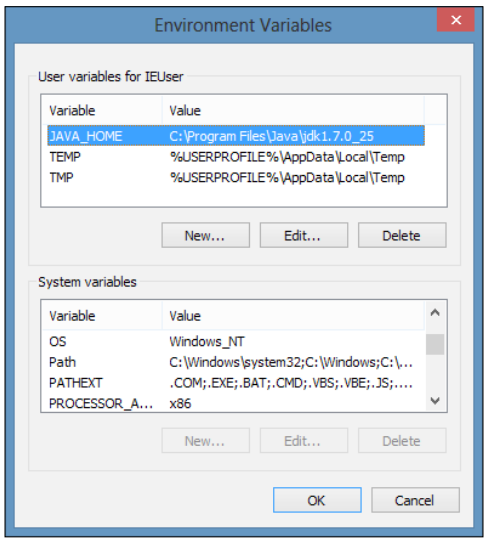
Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Windows

To install Vert.x on Windows 7 or Windows 8, perform with the following steps:

1. Open **File Explorer** (in Windows 8) or **Windows Explorer** (in Windows 7) and find the downloaded Vert.x ZIP file. Right-click on the file and select **Extract All…**.
2. Select the folder into which you want to install Vert.x (for example, one named `dev` within your `Home` directory). After selecting the directory, click on **Extract**.
3. Open **Control Panel**. Navigate to **System and Security** | **System** | **Advanced system settings** | **Environment Variables**.
4. In the **System variables** listbox, find the variable **Path**. Select it and click on **Edit**.
5. At the end of the existing variable value, add a semicolon followed by the path to the `bin` folder of the extracted Vert.x distribution.
   ```
   ;C:\Users\YourUsername\dev\vert.x-x.x.x.final\bin
   ```
6. If you do not already have an environment variable named `JAVA_HOME` in either the **User variables** or the **System variables** listbox, you will need to add it, so that Vert.x will know where to find Java.
   - Below the **System variables** listbox, click on **New**. Set the name as `JAVA_HOME` and the value as the path to your Java JDK 7 installation folder (for example, `C:\Program Files\Java\jdk1.7.0_10`).



You can now proceed to the *Running Vert.x* section to verify your installation.

Running Vert.x

Let's verify that the Vert.x installation was successful by trying to run it. Go to a terminal (on OS X/Linux) or a Command Prompt (on Windows). Try running the `vertx` command:

```
vertx version
```

This should simply output the version of your Vert.x distribution. If you see the version, you have successfully installed Vert.x!

The `vertx` command has multiple uses, but the main one is to launch a Vert.x instance, which is something you'll be doing a lot.

A **Vert.x instance** is the container in which you run the Vert.x applications. It is a single Java virtual machine, which is hosting the Vert.x runtime and its thread pools, classloaders, and other infrastructure.

If you want to see the different things that you can do with the `vertx` command, just run it without any arguments, and it will print out a description of all the different use cases:

```
vertx
```

Embedding Vert.x

As an alternative to using the `vertx` command, it is also possible to launch an **embedded Vert.x instance** from within an existing Java (or other JVM language) application. This is useful when you have an existing application and want to integrate the Vert.x framework in to it.

To embed Vert.x, you need to add the **Vert.x JARs** to your application's classpath. The JARs are available in the Vert.x installation directory, and also as Maven dependencies. After this, you can instantiate a Vert.x instance programmatically.

We won't be using embedded Vert.x instances in this book, but you can find more information about it in the *Embedding the Vert.x platform* section of the Vert.x documentation available at `http://vertx.io`.

Your first verticle – Hello world

Now that you have Vert.x installed and are able to run it, you're all set to write your first Vert.x application. This application will consist of a single **verticle** that prints out the classic "Hello world" message.

A verticle is the fundamental building block of the Vert.x applications. You can think of a verticle as a component of an application, which typically consists of one or a few code files and is focused on a specific task.

Verticles are similar to packages in Java or namespaces in C#; they are used to organize different parts of a system. However, as opposed to packages or namespaces, verticles are also a runtime construct with some interesting properties when it comes to concurrency. We will discuss them in *Chapter 2*, *Developing a Vert.x Web Application*.

Verticles can be implemented in any of the supported Vert.x languages (JavaScript, CoffeeScript, Java, Ruby, Python, or Groovy). For this one, let's use JavaScript.

Create a file named `hello.js` (it doesn't matter where you put it). Open the file in an editor and add the following contents:

```
var console = require("vertx/console");
console.log("Hello world");
```

That's all the code you need for this simple application.

Now, let's launch a Vert.x instance and run `hello.js` in it as a **verticle**:

```
vertx run hello.js
```

This should print out the message. Even though there's nothing else to do, the Vert.x instance will keep running until we explicitly shut it down. Use *Ctrl + C* to shut down the Vert.x instance when you're done.

So, what just happened?

- You wrote some code that loads the Vert.x console library and then uses it to log a message to the screen
- You fired up a Vert.x instance using the `vertx` command
- In that instance, you deployed the code as a verticle

In JavaScript verticles, we will always use the `require` function to load the code from within the Vert.x framework or from other JavaScript files. The `require` function is defined by the CommonJS modules/1.1 standard, which Vert.x implements. You can find more information about it at `http://wiki.commonjs.org/wiki/Modules/1.1`.

You have just gone through the basic process of writing and running a Vert.x application. Now, let's turn to something a bit more useful.

A web server

Our application is going to need a web server, which is used to serve all the HTML, CSS, and JavaScript files to web browsers.

Vert.x includes all the building blocks for setting up a web server, including the HTTP networking and filesystem access. However, there is also something more high-level we can use, that is, a publicly available web server module, which does file serving over HTTP for us out of the box.

Vert.x modules are a solution for packaging and distributing the Vert.x applications or pieces of application functionality for reuse.

A module can include one or more verticles, and an application can make use of any number of modules written in different programming languages.

Vert.x has a growing public-module registry (`http://modulereg.vertx.io/`), from which you can get a variety of open source modules to your applications. The web server module is one of them, and we will install some more later in the book.

In addition to public modules, it is also possible and highly encouraged to package your own applications and libraries as modules. You will learn how to do this in *Chapter 5*, *Polyglot Development and Modules*.

First create a folder for the application we will be building for the duration of this book. You can just call it `mindmap`, and put it within your `Home` directory:

```
cd
mkdir mindmap
cd mindmap
```

In this folder, create a file named `app.js`. This will be the **deployment verticle** of our application. We will use it to deploy all the other verticles and modules that our application needs. As far as the Vert.x itself is concerned, there is nothing special about a deployment verticle; it is just a code organization practice.

We are going to deploy the **mod-web-server** module. The latest version of the module at the time of writing is 2.0.0-final, which we will be using here. Alternatively, you can look for the latest version in the Vert.x module registry.

Add the following code to `app.js`:

```
var container = require("vertx/container");
container.deployModule("io.vertx~mod-web-server~2.0.0-final", {
  port: 8080,
  host: "localhost"
});
```

Let's go through the contents of this file:

- On the first line, we have used the `require` function again; it loads into the Vert.x container. It represents the runtime in which the current verticle is running, and can be used to deploy and undeploy other verticles and modules.
- Next, we called the `deployModule` function of the container to deploy the web server module. We gave two arguments to the function:
    - The fully qualified name and version of the module to deploy
    - A module-specific configuration object. Within the configuration object, we passed two entries to the module: the host name and port to which to bind the server

Now you can run this code as a verticle:

```
vertx run app.js
```

The first time a new module is deployed, Vert.x will automatically download and install it from the public module registry. You will see a subfolder named `mods` appearing in the `project` folder, and within it the contents of the installed modules. (In this case, the web server module.)

If you now point your web browser at `http://localhost:8080/`, you will see a **404 Resource not found** message. This means that the web server is running, but there is nothing to serve. Let's fix that.

Create a subfolder named `web` in the `project` folder. By default, this is where the web server module looks for static files to serve the browsers (this folder is configurable through the web server's module configuration object).

```
mkdir web
```

In this folder, add a simple HTML document in a file named `index.html`, with the following contents:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
</head>
<body>
  Hello!
</body>
</html>
```

Now, as you point your browser at `http://localhost:8080/`, you will see the **Hello!** message. You are running a web server in Vert.x!

Summary

In this chapter, you have installed Vert.x and taken the very first steps towards building a real-time web application by configuring and running a web server.

You have already covered a lot of ground:

- Obtaining and installing Vert.x
- Launching Vert.x instances and running verticles from the command line
- Embedding a Vert.x instance to an existing application
- Installing and deploying the Vert.x modules
- Some key concepts, such as Vert.x instances, verticles, and modules
- Accessing the Vert.x core API from JavaScript
- Running a web server
- In the next chapter, we'll build on our simple web server by adding the very first actual features to our mind map application.

2
Developing a Vert.x Web Application

After installing and configuring Vert.x, let's get started with developing Vert.x in full swing. In this chapter:

- You'll develop a web application for adding and removing mind maps using Vert.x and the jQuery JavaScript library
- You'll get to know the Event bus, the central nervous system of Vert.x
- You'll see how to integrate client-side JavaScript code to the server-side Vert.x application

Adding a new verticle for mind map management

We are going to add the very first features to our mind map application:

- Listing existing mind maps
- Creating a mind map
- Deleting a mind map

The result for these basic operations will be a simple CRUD-style web interface.

Let's begin with the server-side implementation. All Vert.x code is organized into verticles, and each verticle will contain one cohesive unit of functionality. These three operations form one such unit, so let's create a new verticle for them.

In the application's root directory, create a file named `mindmaps.js`. For now, just set its contents to a simple console printout for checking that it has been deployed:

```
var console = require('vertx/console');
console.log('mindmaps.js deployed');
```

Next, deploy this new verticle by adding a line for it in the deployment verticle we created in the previous chapter, `app.js`:

```
var container = require('vertx/container');
container.deployModule('io.vertx~mod-web-server~2.0.0-final', {
  port: 8080,
  host: "localhost"
});
container.deployVerticle('mindmaps.js');
```

In the last chapter, we used the `deployModule` function to deploy the web server module from the public module registry. This time we use a function named `deployVerticle` to deploy a local file as a verticle. These are the two functions for deploying code in Vert.x. In this case, the `deployVerticle` function takes just one argument, the relative path of the file to deploy.

When you now launch a Vert.x instance from the command line, you will see the message printed from the `mindmaps.js` verticle:

```
$ vertx run app.js
mindmaps.js deployed
```

If you wanted to only run the `mindmaps.js` verticle without the web server or anything else, you could just as well give `mindmaps.js` directly as the argument to the `vertx run` command.

Verticles don't care whether you deploy them independently or as part of a larger application. They do not even know whether something else apart from their own code is running in the application or not. This allows for very loosely coupled and modular designs.

Implementing server-side mind map management

Our mind map verticle will provide support for the following three operations outlined earlier:

- Obtaining the list of all existing mind maps
- Saving a mind map
- Deleting a mind map

In most web frameworks, we would write something like a controller or a service object with a method for each of these operations. Vert.x provides a slightly different solution. Verticles are fundamentally loosely coupled, and they never call each other directly. Direct calls are, in fact, made impossible by the Vert.x runtime. Instead, the verticles communicate via the Vert.x **Event bus**.

All communication between verticles happens through the event bus. It is the central nervous system of Vert.x.

Verticles can talk to other verticles by publishing events on addresses of the event bus, and listen to what other verticles have to say by registering event handlers that listen to events on addresses of the event bus. When publishing an event, a verticle does not know who, if anyone, is going to receive it. When receiving an event, a verticle does not know from where it is coming.

Each event has an associated data payload, such as a JSON object, a string, a number, or a raw byte buffer. JSON is the payload type recommended for most cases, and we will use it exclusively in this book.

The event bus pattern is neither a new invention, nor is it exclusive to Vert.x. Buses have been used to decouple system components for a long time, from hardware buses, such as PCI, to data buses in many object-oriented systems (`http://c2.com/cgi/wiki?DataBusPattern`) and Enterprise service buses in large software systems.

Our mind map management verticle is going to listen to mind map management events on the event bus. When someone sends such an event, our verticle will receive it, do some work, and then respond to the sender with some results. This means that all of these handlers follow the **request-reply** communication pattern.

The Vert.x event bus provides three fundamental communication patterns:

Publish/subscribe: When an event is published, all handlers listening on the address will receive it

Point-to-point: When an event is published, at most one handler listening on the address will receive it. If there are multiple handlers, incoming events are distributed among them in a round robin fashion

Request-reply: An extension of point-to-point, where the sender can attach a response handler, which the receiver can call to respond to the original sender

Our three mind map operations map to the following event bus addresses, requests, and responses. We are going to follow an address naming scheme where each operation is prefixed by our application name, to prevent any name clashes with other code running in the same Vert.x instance.

Address	Request data payload	Response data payload
`mindMaps.list`	An empty JSON object	A JSON object with one key: `mindMaps`, whose value is an array of mind map objects
`mindMaps.save`	A JSON object representing the mind map to save	A JSON object representing the saved mind map. Most notably, it will have the `_id` attribute assigned
`mindMaps.delete`	A JSON object with one key: `_id`, whose value is the identifier of the mind map to delete	An empty JSON object

Let's add the code for these operations. First, we need to obtain the event bus object. Replace the contents of `mindmaps.js` with:

```
var eventBus = require('vertx/event_bus');
var mindMaps = {};
```

We first require the `vertx/event_bus` CommonJS module, just like we did with `vertx/console` in the previous chapter. This object provides access to the Vert.x event bus.

We also initialize the object for existing mind maps into the local `mindMaps` variable. For the duration of this chapter, this object will represent our mind map database. It will contain a simple mapping of mind map identifiers to the corresponding mind map objects. This faux database will be replaced with real database access in *Chapter 3*, *Integrating with a Database*.

Listing mind maps

Add the event bus handler for listing mind maps to the file:

```
eventBus.registerHandler('mindMaps.list', function(args,
  responder) {
  responder({"mindMaps": Object.keys(mindMaps).map(function(key) {
    return mindMaps[key];
  })});
});
```

This code registers an event handler to the `mindMaps.list` address on the event bus. The event handler is a plain JavaScript function, which in this case takes two arguments:

- **Event arguments**: In this, case it is actually ignored completely by our handler (because listing doesn't need any arguments). Callers will typically supply an empty object as the value of the arguments.
- **The responder function**: This is a function that our handler will call with a response to the original event. The event bus will route it back to the sender of that event.

Our handler constructs a JSON object with one key: `mindMaps`, with an array of mind maps as the value. The array is constructed by iterating over the keys in the `mindMaps` object and getting the value associated with each key. The handler then calls the responder function, providing the JSON object as an argument. This invocation sends the object back to the sender.

Saving a mind map

The second handler, saving a mind map, has slightly more to it.

```
eventBus.registerHandler('mindMaps.save', function(mindMap,
  responder) {
  if (!mindMap._id) {
    mindMap._id = Math.random();
  }
  mindMaps[mindMap._id] = mindMap;
  responder(mindMap);
});
```

This handler function also takes two arguments. The first argument carries the mind map object to create. The second argument is the responder function, as before.

The handler assigns an identifier to the `_id` attribute of the mind map if there isn't one already, and assigns the mind map to the object of mind maps. It then responds to the original sender with this modified mind map object.

For now, we just assign a random number as the identifier. In *Chapter 3*, *Integrating with a Database*, we will replace this with a real identifier.

Deleting a mind map

Finally, let's add the deletion handler.

```
eventBus.registerHandler('mindMaps.delete', function(args,
  responder) {
  delete mindMaps[args.id];
  responder({});
});
```

This handler takes an object that is expected to have the `_id` key, and a replier function.

The handler deletes the mind map using the JavaScript `delete` operator.

The handler then calls the `responder` with an empty object. This will let the sender know that the delete operation has been executed.

The resulting code

After adding the three mind map management operations, the complete code for `mindmaps.js` will look like this:

```
var eventBus = require('vertx/event_bus');
var mindMaps = {};
eventBus.registerHandler('mindMaps.list', function(args,
  responder) {
  responder({"mindMaps": Object.keys(mindMaps).map(function(key) {
    return mindMaps[key];
  })});
});
eventBus.registerHandler('mindMaps.save', function(mindMap,
  responder) {
  if (!mindMap._id) {
    mindMap._id = Math.random();
  }
  mindMaps[mindMap._id] = mindMap;
  responder(mindMap);
});
eventBus.registerHandler('mindMaps.delete', function(args,
  responder) {
  delete mindMaps[args.id];
  responder({});
});
```

Although we don't have a user interface yet, you may want to check for typos by restarting Vert.x now, and see that the code deploys OK (no error messages are shown).

Bridging the event bus to clients

When `mindmaps.js` is now deployed as a verticle, the three event bus handlers will be made available to all other verticles in the same Vert.x instance.

Because we are writing a web application, what we actually want to do is access these handlers from the web browser. Vert.x makes this possible by providing an **Event bus bridge** to browsers. The event bus bridge is an extension to the event bus, which makes the event bus available to the browser that has been connected to the Vert.x instance.

The bridge is built using the `SockJS` library, which provides real-time full duplex client-server communication built on HTML5 WebSockets. It also has fallback mechanisms for older browsers that don't support WebSockets. We won't be interfacing with `SockJS` directly in this book, but if you want to learn more about it, you can take a look at the Vert.x documentation for `SockJS` servers and clients, and the `SockJS` website at `http://sockjs.org`.

The event bus bridge is by no means the only solution for connecting a browser application to a Vert.x application. You could just as well build a plain REST-style HTTP interface. Vert.x provides a full set of APIs for implementing HTTP(S) APIs and routing requests to them, and these APIs are well-documented in the Vert.x reference documentation.

The reason we use the event bus bridge instead is for the real-time aspect of our application. WebSockets are an ideal transport technology for real-time applications on the web, and the Vert.x event bus bridge provides a very compelling architectural pattern that builds on raw WebSockets. We will discuss this in more detail in *Chapter 4*, *Real-time Communication*.

To enable the event bus bridge, we need to do some setup both on the server side and on the client side.

The server

Vert.x ships with a `SockJS` server, which also contains functionality for connecting to the server-side event bus. The web server module that we have deployed can provide this functionality for us, but it is disabled by default.

We need to tweak the module configuration object of our web server module deployment to do a couple of things:

- Enable the event bus bridge
- Define which messages we allow to pass to and from web browser clients

By default, the event bus bridge does not allow any messages to be passed between the server and the browser clients. We must explicitly define a whitelist of the events we want to expose. The reason for this whitelisting policy is that we do not want to expose event handlers accidentally that should only be accessible from within the Vert.x instance.

Open the deployment verticle (`app.js`) and add the bridging configuration:

```
var container = require("vertx/container");
container.deployModule("io.vertx~mod-web-server~2.0.0-final", {
  port: 8080,
  host: "localhost",
  bridge: true,
  inbound_permitted: [
    { address: 'mindMaps.list' },
    { address: 'mindMaps.save' },
    { address: 'mindMaps.delete' }
  ]
});
vertx.deployVerticle('mindmaps.js');
```

We first set the `bridge` configuration entry to `true`, which will let the web server know that it should enable the event bus bridge.

We then configure the events that are allowed to pass in to our Vert.x instance through the bridge.

The value of the `inbound_permitted` entry is an array, with an item for each of the events we allow to pass. The items have addresses that match the addresses we used with the `registerHandler` function calls in `mindmaps.js`.

The bridge is documented at `http://vertx.io/core_manual_js.html#sockjs-eventbus-bridge` and its configuration for the web server module is at `https://github.com/vert-x/mod-web-server`.

The client

On the client, we will need to bring in a couple of JavaScript libraries:

- The `SockJS` client library
- The client-side event bus integration

We also need to write some JavaScript code of our own to bring everything together.

Edit the `web/index.html` file and add a few script tags just before the closing body tag:

```
<!DOCTYPE html>
<html>
<body>
  Hello!
```

```
    <script src="//cdnjs.cloudflare.com/ajax/libs/sockjs-
      client/0.3.4/sockjs.min.js"></script>
    <script src="//cdnjs.cloudflare.com/ajax/libs/vertx/2.0.0/
      vertxbus.min.js"></script>
    <script src="/client.js"></script>
  </body>
  </html>
```

We first include the two required JavaScript libraries, and then our own `client.js` file, which we'll create in a moment.

All third-party JavaScript libraries used in this book are loaded from CloudFlare's **JavaScript Content Distribution Network** (**CDNJS**). See the web portal at `http://cdnjs.com/` for the latest versions of the libraries.

If you prefer not to use outside services for hosting JavaScript, you can also place them under the `web` subdirectory and include them directly from there.

The `SockJS` client file is available at `http://sockjs.org/`.

The event bus bridge file is available in the `client` subdirectory in the Vert.x distribution you installed in *Chapter 1*, *Getting Started with Vert.x*.

Next, create the `web/client.js` file and add the code for connecting the event bus:

```
var eb = new vertx.EventBus(window.location.protocol + '//' +
  window.location.hostname + ':' +
    window.location.port + '/eventbus');
eb.onopen = function() {
  console.log("Event bus connected");
};
```

First, we created a `vertx.EventBus` object (provided by the `vertxbus.js` library). Its constructor function takes the URL of the server-side event bus bridge. The web-server module deploys the bridge to the path `/eventbus` in the same host and port in which the web server itself is running.

We need to wait until the event bus is connected before we can send or receive any events. For this purpose, we attach a function to the `eb.onopen` property. This function will be called when the event bus connection has been established.

Finally, we just output a log message. Check from your browser's JavaScript console that this message is actually printed. If it isn't, see the browser's **Network** tab, and the terminal window with the running Vert.x for any error messages.

Please note that the Vert.x instance needs to be killed and restarted after all the changes have been made to server-side code.

Testing the bridge

Even though we don't have a user interface yet, you can test the bridge by using the JavaScript console in your browser.

In Chrome, navigate to **View** | **Developer** | **JavaScript Console**
In Firefox, navigate to **Tools** | **Web Developer** | **Web Console**

In the console, you can send the `mindMaps.save` event to test mind map creation:

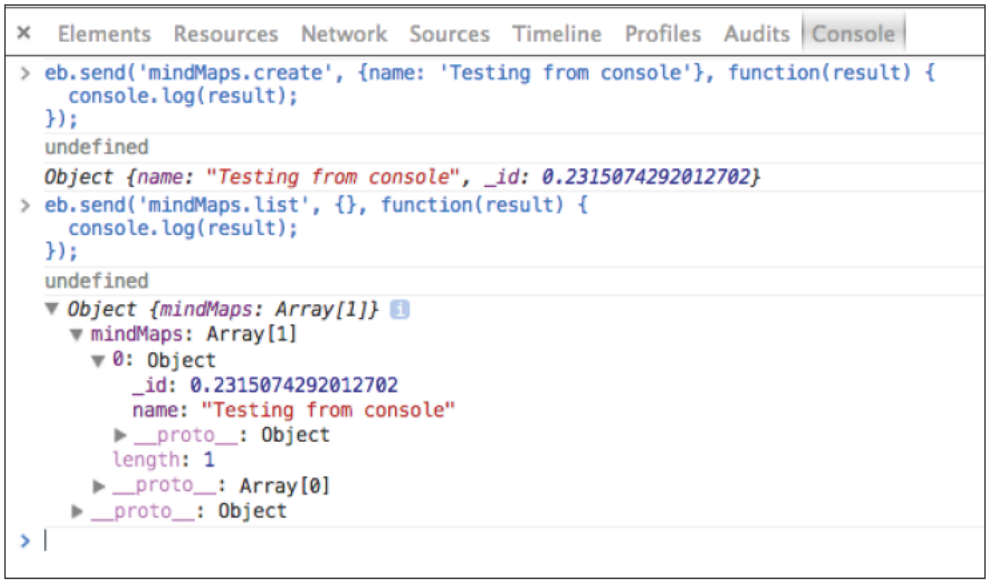
```
eb.send('mindMaps.save', {name: 'Testing from console'},
  function(result) {
  console.log(result);
});
```

This should print out the created mind map with an `_id` attribute attached.

You can also send the `mindMaps.list` event by typing:

```
eb.send('mindMaps.list', {}, function(result) {
  console.log(result);
});
```

This will print out the response to the console once it arrives. The response should include an array with the mind map you had just created.



The event bus uses HTML5 WebSockets as the transport mechanism wherever it's available, which is in most modern browsers. To get some more visibility into what is going on with the WebSocket, it can be useful to look at the Google Chrome development tools.

In Chrome, navigate to **View | Developer | Developer Tools | Network**. Reload the page. From the request list on the left, find and select the event bus connection (a request beginning with `/eventbus`). Finally, select the **Frames** tab. It will display all data transferred on the event bus bridge.

Adding the user interface

We have all the event handlers set up, and the client connected. The final piece of the puzzle is the user interface for listing, creating, and deleting mind maps.

We are going to use the **jQuery** JavaScript library for the UI, because it is familiar to most of the web developers, and it fills our needs relatively well.

Vert.x itself does not require jQuery, and integrating the event bus bridge to other JavaScript libraries or frameworks should be straightforward.

First, include the jQuery library in `web/index.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
</head>
<body>
  Hello!
  <script src="//cdnjs.cloudflare.com/ajax/libs/sockjs-
    client/0.3.4/sockjs.min.js"></script>
  <script src="//cdnjs.cloudflare.com/ajax/libs/vertx/2.0.0/
    vertxbus.min.js"></script>
  <script src=" //cdnjs.cloudflare.com/ajax/libs/jquery/2.0.3/
    jquery.min.js "></script>
  <script src="/client.js"></script>
</body>
</html>
```

Listing the mind maps

We are now ready to implement our user interface. The first thing our application will need to do is to get the list of existing mind maps from the server:

```
var eb = new vertx.EventBus(window.location.protocol + '//' +
  window.location.hostname + ':' +
    window.location.port + '/eventbus');
eb.onopen = function() {
  eb.send('mindMaps.list', {}, function(res) {
    console.log(res);
  });
};
```

We set everything up inside the `eb.onopen` callback, so that we know the event bus will be connected when we begin.

We send the `mindMaps.list` event to the event bus. We give it one argument, an empty object, and attach a response handler function. In the response handler, we just print out the result to the console.

The event bus bridge will send this event to the server, and the event bus on the server will route it to the handler defined in `mindmaps.js`. When the server-side handler calls its responder function with the result, it is routed back to the callback function supplied here. All of this happens transparently, and we don't need to care about serialization, routing, or other transport implementation details.

If you now reload the page, there should be no errors, and you should see the returned array of mind maps in the browser's JavaScript console.

All we need now is to show the results on the page. Let's add a placeholder to the HTML markup for them. In `web/index.html`, add an empty `<ul>` element:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
</head>
<body>
  <ul class="mind-maps">
  </ul>
  <script src="//cdnjs.cloudflare.com/ajax/libs/sockjs-
    client/0.3.4/sockjs.min.js"></script>
  <script src="//cdnjs.cloudflare.com/ajax/libs/vertx/2.0.0/
    vertxbus.min.js"></script>
```

```
    <script src="//cdnjs.cloudflare.com/ajax/libs/
      jquery/2.0.3/jquery.min.js"></script>
    <script src="/client.js"></script>
  </body>
  </html>
```

The page now contains an unordered list with the CSS class `mind-maps`. We can use this CSS class to grab the element in our JavaScript code.

In the JavaScript file `web/client.js`, add the highlighted code:

```
var eb = new vertx.EventBus(window.location.protocol + '//' +
  window.location.hostname + ':' +
    window.location.port + '/eventbus');
eb.onopen = function() {
  var renderListItem = function(mindMap) {
    var li = $('<li>');
    $('<span>').text(mindMap.name).appendTo(li);
    li.appendTo('.mind-maps');
  };
  eb.send('mindMaps.list', {}, function(res) {
    $.each(res.mindMaps, function() {
      renderListItem(this);
    });
  })
};
```

The new `renderListItem` function uses jQuery to create a list item (`<li>`) HTML element. It also creates a `<span>` element containing the name of the mind map and adds it to the list item. Finally, it adds the list item to the `mind-maps` list.

The modified event bus response handler calls this new function once for each returned mind map.

The result is a list of mind maps, with .the name of each mind map displayed.

When you now reload the page, you will see a list with one item: the test mind map you added in *Testing the bridge* section (Unless you have restarted Vert.x since, in which case the list will be empty). To see some results, proceed to the next section.

Creating a mind map

Listing the mind maps isn't very useful by itself. We need to be able to create some as well.

Let's add the HTML markup first. In `web/index.html`, add:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
</head>
<body>
  <ul class="mind-maps">
  </ul>
  <h2>Create a Mind Map</h2>
  <form class="create-form">
    <input type="text" name="name">
    <input type="submit" value="Create">
  </form>
  <script src="//cdnjs.cloudflare.com/ajax/libs/sockjs-
    client/0.3.4/sockjs.min.js"></script>
  <script src="//cdnjs.cloudflare.com/ajax/libs/vertx/2.0.0/
    vertxbus.min.js"></script>
  <script src="//cdnjs.cloudflare.com/ajax/libs/jquery/2.0.3/
    jquery.min.js"></script>
  <script src="/client.js"></script>
</body>
</html>
```

We have added a `<form>` tag with the `create-form` CSS class.

Inside the form, we have a text input field and a standard form submit button.

Next, add a submit handler for the new form in `web/client.js`:

```
var eb = new vertx.EventBus(window.location.protocol + '//' +
  window.location.hostname + ':' +
    window.location.port + '/eventbus');
eb.onopen = function() {
  var renderListItem = function(mindMap) {
    var li = $('<li>');
```

```
      $('<span>').text(mindMap.name).appendTo(li);
      li.appendTo('.mind-maps');
    };
    $('.create-form').submit(function() {
      var nameInput = $('[name=name]', this);
      eb.send('mindMaps.save', {name: nameInput.val()}, function(result)
{
        renderListItem(result);
        nameInput.val('');
      });
      return false;
    });
    eb.send('mindMaps.list', {}, function(res) {
      $.each(res.mindMaps, function() {
        renderListItem(this);
      })
    })
};
```

The handler function grabs the name input from the form and assigns it to the local `nameInput` variable. It then sends a message to the `mindMaps.save` address on the event bus, giving it a JSON object as the payload. The object just contains the mind map name, which is set as the current value of the `nameInput` field.

When a response is received from the event bus, the new mind map is rendered using the same function as we used before. The value of the `nameInput` field is also cleared, so that the form is ready for adding the next mind map.

Finally, the handler returns `false`, which lets jQuery know that it should stop the submission event from propagating further.

When you now reload the page, you will be able to create mind maps using the form, and they will appear in the list when the form is submitted.

Deleting a mind map

The final operation that we still need to add is deletion.

In the list of mind maps, we want to have a delete button next to each mind map. When that button is clicked, we want to let the server know that a mind map should be deleted.

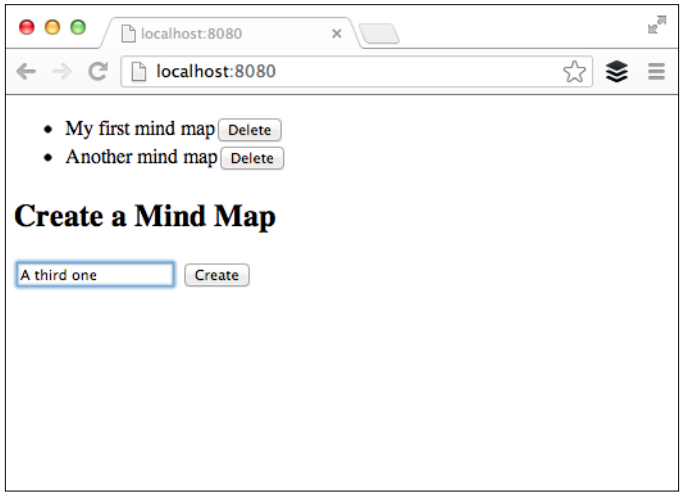
Add the highlighted code to `web/client.js` in order to handle this operation:

```
var eb = new vertx.EventBus(window.location.protocol + '//' +
  window.location.hostname + ':' +
    window.location.port + '/eventbus');
eb.onopen = function() {
  var renderListItem = function(mindMap) {
    var li = $('<li>');
    var deleteMindMap = function() {
      eb.send('mindMaps.delete', {id: mindMap._id}, function() {
        li.remove();
      });
      return false;
    };
    $('<span>').text(mindMap.name).appendTo(li);
    $('<button>').text('Delete').on('click',
      deleteMindMap).appendTo(li);
    li.appendTo('.mind-maps');
  };
  $('.create-form').submit(function() {
    var nameInput = $('[name=name]', this);
    eb.send('mindMaps.save', {name: nameInput.val()},
      function(result) {
      renderListItem(result);
      nameInput.val('');
    });
    return false;
  });
  eb.send('mindMaps.list', {}, function(res) {
    $.each(res.mindMaps, function() {
      renderListItem(this);
    })
  })
};
```

We have a new function named `deleteMindMap`, created for each mind map as a local variable in `renderListItem`. The function sends the `mindMaps.delete` event to the server, giving it an object specifying the `_id` of the mind map. In the response callback, we remove the mind map list item, so that the mind map disappears from the screen.

For each mind map, we add a `<button>` element next to the name. We wire the click event for this button to invoke the `deleteMindMap` function.

We now have a complete implementation for listing, creating, and deleting mind maps, with server-side event handlers and a client-side user interface, connected via the Vert.x event bus and the event bus bridge!



Verticles and concurrency

To conclude this chapter, let's discuss concurrency in Vert.x a bit further. We have now deployed two verticles: the deployment verticle, and the `mindmaps.js` verticle. We will add more during the course of the book. It is important to understand how Vert.x manages these verticles.

Vert.x has been built for the concurrent world from the ground up. On the one hand, Vert.x enables concurrent code execution. On the other hand, it manages to hide many of the complexities of concurrent programming from application developers. It does this by providing a simple and safe verticle programming model.

When a Vert.x instance is launched, it sets up a thread pool. The size of this thread pool is calibrated based on the number of CPU cores on your machine. When a verticle needs to execute some code, it will do so in a thread from this thread pool.

For each verticle running in the thread pool, Vert.x guarantees two things that are very significant from a concurrency perspective:

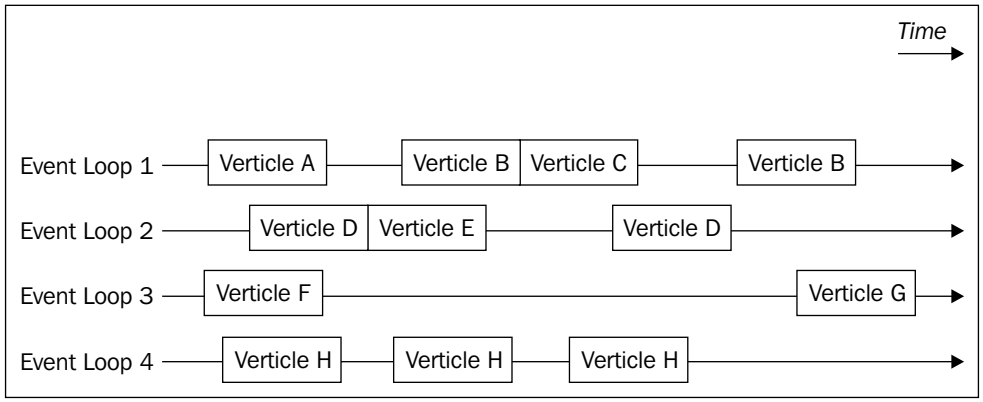
1. Each verticle is assigned its own Java classloader, which means it is impossible for verticles to access each other's state, even if it was defined in static Java variables or by other programming mechanisms that usually allow for global state to be shared.

2. A verticle instance will always run in the same thread (though that thread may also run other verticles at other times). This eliminates the need to do any state synchronization within a verticle. You can basically write your verticles as if your program was single-threaded.

A great majority of Vert.x code is written in an asynchronous fashion, Event loop threads sit and wait for events to appear. When an event is received, some code is executed in the context of a verticle, and then the event loop starts waiting for the next event. For example, the `mindmaps.js` verticle waits for events to appear from the event bus and executes one of its handler functions when an event appears. We have seen this pattern in use with the event bus, but it also applies to everything else in Vert.x. When we invoke a filesystem or a network operation, we have to wait for a response event before we know the operation has been executed.

This is an implementation of the reactor pattern (`http://en.wikipedia.org/wiki/Reactor_pattern`), which is the same pattern that platforms such as `Node.js` and Ruby's `EventMachine` implement. A single Vert.x verticle is similar to a `Node.js` application it is running in an asynchronous event loop in a single thread. A Vert.x instance with multiple verticles implements a so-called multireactor pattern, because there may be multiple event loops running in parallel.

An example of how a Vert.x application might run on a quad-core machine is shown in the following diagram. Each core is running an event loop thread (**Event Loop 1-4**). The application has eight verticles running in total (**Verticle A-H**). Each verticle will always run on the same event loop (Verticle A will never move away from Event Loop 1), but a single event loop may run different verticles at different times (Event Loop 1 is running Verticles A through C).



Because the thread pool is bounded based on the number of CPU cores, it is not a good idea to run CPU intensive operations, blocking IO operations, or other long running tasks in event loops. This kind of work should be done in worker verticles, which are the verticles that run outside the main thread pool, and have their own concurrency characteristics. We will get familiar with the worker verticles in *Chapter 5*, *Polyglot Development and Modules*.

It is also possible to deploy multiple instances of a single verticle, so that multiple handlers for the same task can execute in parallel. We will see how to do this in *Chapter 6*, *Deploying and Scaling Vert.x*.

Summary

We have covered a lot of ground in this chapter. We have actually written a simple, fully functional web application in Vert.x.

Building web application features using a distributed event bus pattern has a distinctly different feel to it than using traditional RESTful calls over HTTP. The separation between server and client is somewhat faded, because both sides can participate in event bus communication exactly the same way. Both can initiate and receive events. You can think of a Vert.x application as a large interconnected system with all the server nodes and web browsers communicating with each other through a single event bus.

We have covered:

- The concepts of the Vert.x event bus and the event bus bridge
- Deploying verticles programmatically
- Registering event handlers on the event bus
- Using the request-response pattern on the event bus
- Enabling the event bus bridge on the Vert.x web-server module and exposing events to the bridge by whitelisting them
- Sending events to the event bus from a bridged web browser
- Building an HTML/JavaScript user interface with jQuery
- The basics of the Vert.x concurrency model

3
Integrating with a Database

Most applications need a persistent data store, and our mind map application is no exception. In Vert.x applications, you can use persistence services that are based on the infrastructure you have already seen: modules and the event bus.

In this chapter, you will connect the application to a MongoDB database using the Vert.x Mongo Persistor module. You will see how to execute database operations over the event bus, and how to handle their results. In the process, you will also learn more about the module system and the public module registry.

MongoDB

MongoDB is an open source document-oriented NoSQL database, which balances ease of use with performance and scalability in a compelling manner.

NoSQL is a term commonly used for the new breed of database systems that has cropped up in the last few years.

Common to most NoSQL systems is the fact that they are non-relational and do not use SQL as the query language. This is a departure from the approach most databases have taken since the 1970s. Different NoSQL databases have different uses, but most of them emphasize scalability and developer-friendliness.

For more information about NoSQL in general, see Martin Fowler's NoSQL guide at `http://martinfowler.com/nosql.html`.

MongoDB is a very good match specifically for Vert.x development, because with it you can persist and query JSON objects transparently. In addition to the data itself, MongoDB database queries are also represented in JSON. As we discussed in the previous chapter, JSON is the data format of choice for communication over the Vert.x event bus. MongoDB fits into this picture naturally.

Our use of MongoDB will be quite simple, and all the operations will be explained as we go. You should be able to follow along without spending much time learning MongoDB itself. To get a deeper understanding of MongoDB, I recommend the official documentation at `http://www.mongodb.org/` as a good starting point.

MongoDB Alternatives

MongoDB is far from the only persistence option in Vert.x application development.

There is a publicly available JDBC persistor, which allows you to access a relational database. It supports all databases that can be used through Java's JDBC API.

There is also a Redis client, which provides access to the Redis key-value store. Both of these modules are available in the Vert.x module registry. We will get to know it later in this chapter.

For other databases you can write your own database access code, as long as the database has a Java driver or connector available. Just wrap the database access code to a Vert.x worker module. We will discuss worker modules in detail in *Chapter 5*, *Polyglot Development and Modules*.

Installing MongoDB

Before being able to connect to a MongoDB database, you'll need to have MongoDB installed, and its server daemon process up and running.

MongoDB installation is straightforward, and there are several following options:

- With Homebrew or MacPorts for OS X
- With APT for Ubuntu or Debian Linux
- With RPM for Red Hat, CentOS, or Fedora Linux
- With a precompiled and packaged download for OS X, Linux, Windows, or Solaris.

There are official installation guides for all of these platforms at `http://docs.mongodb.org/`. Find the one for your platform, and follow it to get MongoDB set up.

For this chapter, we assume that you have installed and configured MongoDB with the default options, and the following settings configured:

- Running at `localhost`, port 27017
- Authentication disabled

Once you are done with the installation, you should be able to connect to the local database by just launching a MongoDB shell with the `mongo` command that comes with the distribution, as follows:

```
mongo
MongoDB shell version: 2.2.0
connecting to: test
>
```

Use *Ctrl*+*C* to exit the shell.

There are also cloud services such as MongoHQ (`https://www.mongohq.com/`), which can host your MongoDB database for you. If you decide to use a cloud service, there is no need to install MongoDB. Just point the Vert.x Mongo Persistor configuration to the hosted database address.

Installing the Vert.x Mongo Persistor module

Now that you have MongoDB installed and running, you can connect to it from Vert.x. This can be done with the Mongo Persistor module, available in the public module registry. While we're at it, let's also look at the process of finding and installing a Vert.x module from the registry.

We have already used a publicly available Vert.x module: the Web Server module, deployed in *Chapter 1*, *Getting Started with Vert.x*. We deployed it using the `container.deployModule` function in the deployment verticle, `app.js`. You'll find the module installed in the `mods` subdirectory of the project.

The Mongo Persistor module can be installed in exactly the same way. Just deploy the module from the deployment verticle. Vert.x will find it, install it, and deploy it. The same applies to all modules in the public module registry.

Open the page: `http://modulereg.vertx.io/` in your web browser. This is a web page presenting the contents of the Vert.x module registry.

In the listing, you should see the entry **io.vertx~mod-web-server~2.0.0-final**. This is the Web Server module—the same one we already installed in *Chapter 1*, *Getting Started with Vert.x*.

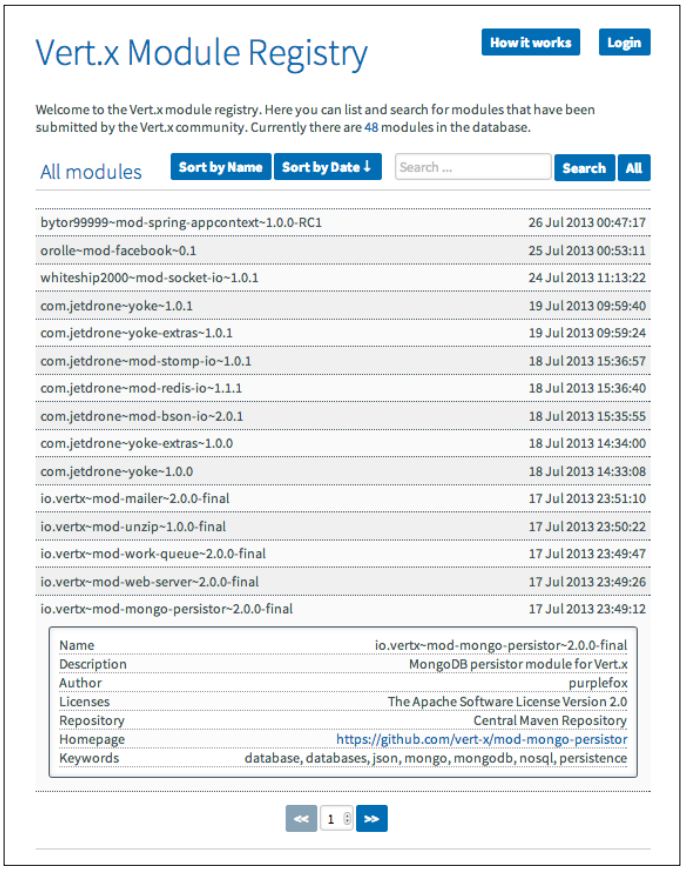
In the listing, you should also find the Mongo Persistor module. It is the one beginning with **io.vertx~mod-mongo-persistor**. Find the newest version (**io.vertx~mod-mongo-persistor~2.0.0-final** when this was written) and click it. This opens an information box, with a link to the Mongo Persistor documentation. By following the link, you will find information on how to configure and use the persistor.

Note that the module registry is just a central location serving information about modules that are publicly available, and it does not actually store the module packages themselves. The actual module packages are stored and deployed in various Maven repositories on the Internet and within company firewalls, as well as in Bintray, an open source package hosting service. For example, the Mongo Persistor is stored in the Maven Central Repository, which you can see from its registry entry.

Maven is a widely used Java build tool. Among other things, it provides tools and infrastructure for distributing software packages. Vert.x leverages this infrastructure for its module distribution. However, all of this happens transparently and you don't need to know anything about Maven to be able to use not any specific modules Vert.x modules.

When you deploy a module that has not been installed locally yet, Vert.x will look for it in Bintray, in the Central Maven Repository, in the Sonatype Snapshots Maven Repository, and in your local Maven repository.

You can also specify where you want Vert.x to look for modules. For example, you could set up an internal Maven repository for module distribution within your company. Just configure the URL of the alternative repository in the `repos.txt` file contained in the `conf` directory of your Vert.x installation.



Let's proceed to installing the module. You'll need to invoke `container.deployModule` from our deployment verticle, `app.js`, just like we've done for the Web Server. This is shown in the following listing:

```
var container = require("vertx/container");

container.deployModule("io.vertx~mod-web-server~2.0.0-final", {
  port: 8080,
  host: "localhost",
  bridge: true,
  inbound_permitted: [
```

```
      { address: 'mindMaps.list' },
      { address: 'mindMaps.save' },
      { address: 'mindMaps.delete' }
    ]
});
container.deployModule("io.vertx~mod-mongo-persistor~2.0.0-final",
  {
    address: "mindMaps.persistor",
    db_name: "mind_maps"
  });

container.deployVerticle('mindmaps.js');
```

The first argument to `deployModule` is the name and version of the module (which matches the coordinates specified in the module registry). The second argument is the module's configuration object. We specify the following two things in the configuration:

- An event bus address for the module. This is where the module will register its event handler function on the event bus. It is a good convention to add a namespace to the persistor, so that it won't get mixed up with any other persistor that might be running in the same Vert.x instance. Here the namespace we use is `mindMaps`—matching the namespace of the handler functions in `mindmaps.js`.
- The name of the database we want this persistor to use. We will use a database named `mind_maps`. MongoDB will automatically create the database when it is accessed for the first time.

If you installed MongoDB to a non-default port or on another machine, you will additionally need to provide the configuration options `host` and `port`.

Restart Vert.x once you have added the configuration. It will fetch the module and install it to the `mods` subdirectory of the project.

```
$ vertx run app.js
```

Implementing database integration for mind map management

We are now all set to integrate the existing database management features in `mindmaps.js` with MongoDB. Instead of using the JavaScript array from *Chapter 2, Developing a Vert.x Web Application,* the verticle will reach into the MongoDB database. All the mind maps will be stored in a collection named `mindMaps` in the database.

All the changes we make will be contained within `mindmaps.js`. Client-side code is not affected by these changes, and should function as before.

If you've written database access code before, you've probably used a direct API: a class or module with methods for finding, saving, and deleting things, and all the other operations one might do with a database. MongoDB also provides such APIs. In Vert.x, however, you usually don't want to use direct APIs directly from the application code.

Instead, the preferred way in Vert.x is to use the event bus. With the Mongo Persistor module, you send events on the event bus in order to execute MongoDB database operations. The Mongo Persistor is going to be listening for these events if it has been deployed. There are no special classes or methods to call in your application code, there's just the event bus, addresses, data structures, and callback functions.

Since database operations are just events on the bus, you might wonder if it's possible to actually initiate them right from the browser, and transfer them to the server over the event bus bridge. This is, in fact, possible as long as you add the database operations to the bridge's `inbound_permitted` list.

However, I do not recommend you do this, except perhaps in prototypes and other disposable applications. Firstly, there are obvious security issues with letting database operations initiate in web browsers, because a browser should always be considered as an untrusted environment. Secondly, in most applications, it will make architecturally more sense to have browser code that doesn't know much, if anything, about databases or database queries.

Requiring the Vert.x console

We are going to need to print some log output to the server console if database errors occur. Before we can do this, we need to add the console `CommonJS` module to the mind maps verticle. Add the following line in the beginning of `mindmaps.js`:

```
var console = require('vertx/console');
```

Listing mind maps

To get a list of mind maps, we send an event to the event bus. We are going to send it to the Mongo Persistor's address: `mindMaps.persistor`. We will also attach a callback function which will be called with the result of the query once it is received from the database.

Replace the contents of the existing `mindMaps.list` event handler in `mindmaps.js` with the following code:

```
eventBus.registerHandler('mindMaps.list', function(args, responder) {
  eventBus.send(
    'mindMaps.persistor',
    {action: "find", collection: "mindMaps", matcher: {}},
    function(reply) {
      if (reply.status === "ok") {
        responder({mindMaps: reply.results});
      } else {
        console.log(reply.message);
      }
    }
  );
});
```

Let's go through this code step-by-step:

- We send a JSON object to `mindMaps.persistor` address with the following three keys:
    - `action`: It specifies what action we want to execute with the database. In this case, we want to find things.
    - `collection`: It specifies the MongoDB collection on which we are executing the operation.
    - `matcher`: It specifies the parameters of the find query: what kind of mind maps we want to find. This is roughly the MongoDB equivalent of an SQL query. In this case, we want to match everything in the collection, so we specify an empty matcher.

- The second argument is the callback, to be called with the reply of the query. We expect the reply to have a `status` key, with a value indicating if the operation was successful or not.
    - In the case of a successful operation (the value of status being `ok`), we send a response back to the responder of the original event (in our case, the client code in the web browser). The responder is handed the `mindMaps` result from the database, which can be found in the results key of the reply object.
    - In the case of a failed operation, we just print the error message to the server console. In a more sophisticated implementation, we would also let the user know about the error by responding to the original function with an error.

Finding a specific mind map

Let's also add another mind map finding handler: One for finding a specific mind map by its identifier. We don't have such a handler yet, because we didn't need it, but we will need it in the next chapter when we implement the mind map editor.

The code for finding a specific mind map is very similar to the code that finds all of them. We send an event to the same address as before. With Mongo Persistor, all operations are sent to one single address. The attribute that distinguishes different types of operations is `action` within the payload object. This time, instead of using the `find` action that returns an array of mind maps, we use the `findone` action that just returns a single mind map—the first one it finds. We also specify the `_id` of the mind map to it. The following listing shows the `findone` action:

```
eventBus.registerHandler('mindMaps.find', function(args,
  responder) {
    eventBus.send(
      'mindMaps.persistor',
      {action: "findone", collection: "mindMaps", matcher: {_id:
        args._id}},
      function(reply) {
        if (reply.status === "ok") {
          responder({mindMap: reply.result});
        } else {
            console.log(reply.message);
        }
      }
    );
});
```

Saving a mind map

When saving mind map, we again send an event to the same address as before. This is shown in the following listing:

```
eventBus.registerHandler('mindMaps.save', function(mindMap,
  responder) {
    eventBus.send(
      'mindMaps.persistor',
      {action: "save", collection: "mindMaps", document: mindMap},
      function(reply) {
        if (reply.status === "ok") {
          mindMap._id = reply._id;
          responder(mindMap);
        } else {
          console.log(reply.message);
        }
      }
    );
});
```

The event payload for saving is slightly different from what we did before:

- We use the `save` action, and define the MongoDB collection we want to save into `mindMaps`, as well as the document we want to save. The document is just the payload of the incoming `mindMaps.save` event.
- If the operation was successful, we find the `_id` from the reply. It contains the identifier assigned to the document, which can later be used to refer to this mind map. We attach the identifier to the `mindMap` document and respond it back with the responder function.
- If the operation was not successful, we print out the error, just like with the listing function.

If you're familiar with MongoDB, you'll know that by default it assigns a so-called ObjectId instance as the `_id` of saved documents. This can be awkward when passing persistent documents to other code or web browsers, because the ObjectId type is not supported by JSON or other common serialization formats.

For this reason, the Vert.x Mongo Persistor assigns a string identifier (a random unique identifier) to new documents, so that we don't have to deal with type conversion.

Deleting a mind map

Deletion follows the same pattern as the other operations:

```
eventBus.registerHandler('mindMaps.delete', function(args,
  responder) {
    eventBus.send(
      'mindMaps.persistor',
      {action: "delete", collection: "mindMaps", matcher: {_id:
        args.id}},
      function(reply) {
        if (reply.status === "ok") {
          responder({});
        } else {
          console.log(reply.message);
        }
      }
    );
});
```

The event payload should look familiar by now:

- The action we want to execute is `delete`, and again we define the `mindMaps` collection as the one with which we want to work. Like with `find`, MongoDB expects a matcher object specifying which documents should be deleted. Just like we did with `findone`, we specify the `_id` of the specific mind map that should be deleted. The `_id` is in the incoming `args` object.
- If the deletion is successful, we respond with an empty object. (The value used here is not significant. The only reason it's defined is that the event bus bridge requires some value to be included.)
- If the deletion is not successful, we once again print the error in the server console.

Refactoring to remove duplication

If you now look at the three handler functions in `mindmaps.js`, you will see that there's some duplicate logic shared between them:

- They all send an event to the `mindMaps.persistor` address on the event bus
- They all check if the operation is successful or not
- If there is an error, they all print the error message to the console

This is all duplication that can be avoided, if we introduce a helper function which all of the three handler functions can use.

JavaScript verticles are regular JavaScript files, and you can define functions in them like you would in any other JavaScript environment. In this case, what we want is a function that does the following things:

- Takes an object defining a Mongo Persistor command and a callback function.
- Sends the command object to the persistor's address on the event bus.
- When the reply arrives, checks if it's OK. If it is, invokes the callback function with the reply.
- If the reply is not OK, prints the error message to the server output. In this case, the callback will not be invoked at all.

The following is an example of such function:

```
function sendPersistorEvent(command, callback) {
  eventBus.send('mindMaps.persistor', command, function(reply) {
    if (reply.status === "ok") {
      callback(reply);
    } else {
      console.log(reply.message);
  }
};
```

Now, we can change our existing handler functions so that they use this new function, thus eliminating the duplication.

The following listing is the resulting `mindmaps.js` content in full:

```
var eventBus = require('vertx/event_bus');
var console  = require('vertx/console');
function sendPersistorEvent(command, callback) {
  eventBus.send('mindMaps.persistor', command, function(reply) {
    if (reply.status === "ok") {
      callback(reply);
    } else {
      console.log(reply.message);
    }
  };
```

```
  eventBus.registerHandler('mindMaps.list', function(args,
    responder) {
    sendPersistorEvent(
      {action: "find", collection: "mindMaps", matcher: {}},
      function(reply) {
        responder({mindMaps: reply.results});
      }
    );
  });
  eventBus.registerHandler('mindMaps.find', function(args,
    responder) {
    sendPersistorEvent(
      {action: "findone", collection: "mindMaps", matcher: {_id:
        args._id}},
      function(reply) {
        responder({mindMap: reply.result});
      }
    );
  });

  eventBus.registerHandler('mindMaps.save', function(mindMap,
    responder) {
    sendPersistorEvent(
      {action: "save", collection: "mindMaps", document: mindMap},
      function(reply) {
        mindMap._id = reply._id;
        responder(mindMap);
      }
    );
  });

  eventBus.registerHandler('mindMaps.delete', function(args,
    responder) {
    sendPersistorEvent(
      {action: "delete", collection: "mindMaps", matcher: {_id:
        args.id}},
      function(reply) {
        responder({});
      }
    );
  });]
}
```

Notice that we've also removed the `mindMaps` object from the file. We no longer need it, because all mind maps are in the MongoDB database.

When you restart Vert.x now, you should be able to use the browser UI just like before. The difference is that now the mind maps you create will actually go to the MongoDB `mind_maps` database and will, for example, survive between Vert.x restarts. You can also examine and manipulate the data via the MongoDB shell.

If the UI doesn't seem to be functional, check the server output. (Recall that we're printing persistor errors there!) There might be a MongoDB connection issue or some other database related problem.

Summary

In this chapter, you've learned to integrate Vert.x with MongoDB. The integration is slightly different from what you may be familiar with, because instead of a direct API, you used Vert.x event bus based integration.

This is how you do pretty much anything in Vert.x, everything goes through the event bus. Instead of specialized APIs, everything is just data. This is occasionally slightly more verbose than direct API calls, but in many ways preferable, especially once we start mixing different programming languages in *Chapter 5*, *Polyglot Development and Modules*.

Following are the things covered in this chapter:

- Installing MongoDB
- Finding and using modules from the public module registry
- Installing and configuring the Vert.x Mongo Persistor module
- Finding, inserting, and deleting data using the Vert.x Mongo Persistor module over the event bus
- Introducing helper functions in JavaScript verticles to reduce duplication

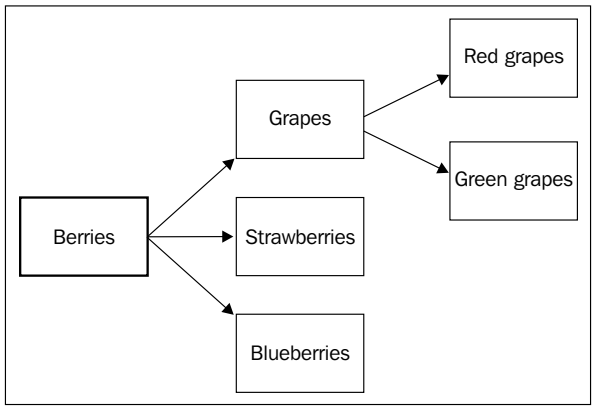
4
Real-time Communication

Let's now delve into the magical world of real-time web applications! In this chapter, we will expand our application with an editor with which you can collaboratively create and manipulate mind maps, and see changes made by each other in real time, when they happen.

To build the editor user interface itself, we will use the D3 JavaScript library, which is a powerful tool for creating data visualizations.

The mind map structure

What does a mind map look like? You may have seen many variations, but what tends to be common to all is that there is a central word or concept, from which other words or concepts spring, forming a tree-like structure. A mind map can indeed be represented as a tree:



This is the kind of mind map structure we need to represent. If we think about our JSON data structure, for each node in the mind map we need the following information:

- The **name** of the node (what the user sees on the screen)
- A unique **key** for the node, so that we can refer to it while making real-time changes. Because the name of a node may change during its lifetime, we need this separate attribute that never changes
- An optional array of **children,** which includes all the other nodes that spring from this one

We should not have any problem while working with a data structure similar to this, because all components in our stack (Vert.x, web browsers, and MongoDB) are well equipped to deal with nested JSON data structures.

Here is a JSON representation of the mind map shown earlier. We will use the mind map itself as the root node, so that the name of the mind map is also its central concept:

```
{
  "_id": "1234-5678-9012-3456",
  "name": "Berries",
  "children": [
    {
      "key": "1",
      "name": "Grapes",
      "children": [
        {
          "key": "2",
          "name": "Red grapes"
        },
        {
          "key": "3",
          "name": "Green grapes"
        }
      ]
    },
    {
      "key": "4",
      "name": "Strawberries"
    },
    {
      "key": "5",
      "name": "Blueberries"
    }
  ]
}
```

Real-time interaction

Now, we know what our data will look like; let's discuss how we can deal with the real-time aspects of the editor.

There are three different operations that our simple mind map editor will support:

- Adding a node to the mind map
- Deleting a node from the mind map
- Renaming a node in the mind map

In a traditional application, we may approach the task of implementing these operations by building request-response type APIs for these operations. When a user adds a node, a change request is sent to the server, and the server responds with a result describing whether the change was successful or not.

In a real-time setting, this doesn't work. We must take into account the changes made by everyone to the mind map at the same time. One relatively simple way to accomplish this is to separate the requests and responses into two different concepts, which we will call as **events** and **commands**.

Events

The changes that happen to a mind map can be thought of as a stream of events. Based on the operations defined, there are three kinds of events which we will implement: **node added**, **node deleted**, and **node renamed**.

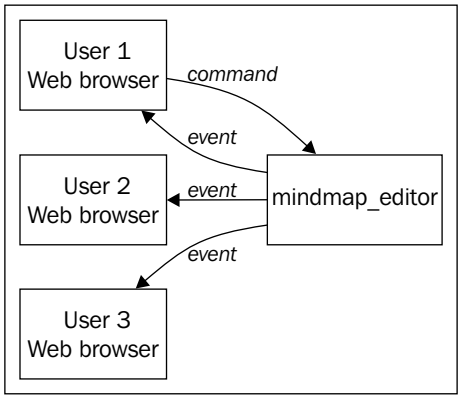
When a user opens a mind map in his or her browser, the browser subscribes to receive all events that are related to that mind map. We will write handlers for all of these events. The responsibility of these handlers will be to update the mind map representation that is shown on the screen.

Commands

When a user wants to make any change to the mind map (add a node, delete a node, or rename a node), they will initiate a command. This command will be sent over the Vert.x event bus (introduced in *Chapter 2*, *Developing a Vert.x Web Application*). On the server we will have a verticle that handles the incoming commands.

Note that when a command is issued, there will be no direct response to it. Instead, the effects of the command will be made visible via an event at a later time. (Or the command might fail; in this case, no such event will happen).

The effect of using this event-command separation pattern is that when the user opens a mind map on their screen, they will subscribe to the real-time event stream of that mind map, and will immediately start seeing all changes made to it. If they want to make changes by themselves, commands will be sent to effect those changes, and the results will be a part of the event stream. So the event stream includes all changes to the mind map, both made by the user and by everyone else.



In a situation where three users are collaboratively editing a mind map, the information flow looks similar to the previous diagram. When **User 1** makes a change, a command is issued and sent over the event bus bridge to `mindmap_editor.js`. It does the backend processing of the command, and finally publishes an event describing the change. All connected users receive the same event.

The editor verticle

The first thing we'll do is to create another verticle, the one that will handle mind map editing commands and send out the mind map change events.

Create a file named `mindmap_editor.js` in the root of the project, and add a basic structure for the command handlers:

```
var eventBus = require('vertx/event_bus');
eventBus.registerHandler('mindMaps.editor.addNode',
  function(args) {
});
eventBus.registerHandler('mindMaps.editor.renameNode',
  function(args) {
});
eventBus.registerHandler('mindMaps.editor.deleteNode',
  function(args) {
});
```

The helper functions

Before we implement the command handlers, there are a few utility functions we can add to make our life much easier. The first one is a function that is able to find a node from a mind map by its unique key. Add this to the top of the `mindmap_editor.js` file which we had just created:

```
function findNodeByKey(root, key) {
  if (root.key === key) {
    return root;
  } else if (root.children) {
    for (var i=0 ; i<root.children.length ; i++) {
      var match = findNodeByKey(root.children[i], key);
      if (match) return match;
    }
  }
}
```

The function implements a depth-first search of the mind map tree. It takes a root node to start the search, and the key to find. It checks whether the root node is the matching node, and returns the value if it matches. Otherwise it will look for the key from the root's children, making a recursive call to itself for each child until a match is found.

A second function is needed for generating node keys. As we discussed earlier, we want each node in the mind map to have a unique key, which can be used to refer the node. The `findNodeByKey` function does so. The database or the Vert.x Mongo persistor will not assign node identifiers for us, because nodes are not first class database objects. They are just nested objects in our data structure. So, let's define our own unique key function:

```
function newNodeKey() {
  return java.util.UUID.randomUUID().toString();
}
```

This function returns a random **UUID** (**Universally Unique Identifier**) string. We exploit the fact that we're running on the JVM by just tapping into the `java.util.UUID` API that comes with Java (refer to `http://docs.oracle.com/javase/7/docs/api/java/util/UUID.html`). In Vert.x you always have the full power and reach of the Java platform underneath the surface.

The third helper function we need at this point is the one for publishing a mind map event on the event bus. Let's add that:

```
function publishMindMapEvent(mindMap, event) {
  eventBus.publish('mindMaps.events.'+mindMap._id, event);
}
```

Each mind map will have their own address on the event bus, on which the mind map's change events are published. We construct an address for the events using the mind map's identifier. The addresses will be in the form `mindMaps.events.1234-5678-9012-3456`.

One additional thing to note here is that we call the `publish` function of the event bus, whereas earlier we used to call the `send` function. The difference is that with `send`, the event is received by at most one handler (point-to-point messaging). With `publish`, all handlers that might be listening will receive it (publish-subscribe messaging). This is exactly what we wanted. All users who happen to have the mind map open in their browser will receive the event. The Vert.x event bus makes both of these communication patterns available by providing these two functions for us.

The add node command handler

All of our command handlers will have a similar structure:

1. Find the mind map and node(s) on which we're operating.
2. Manipulate the mind map data structure.
3. Save the mind map back into the database.
4. Publish an event letting everyone know what just happened.

Let's think about adding a node. The two pieces of information needed here are the identifier of the mind map being edited, and the key of the node that will be the parent of the new node. Optionally, a name for the new node can also be supplied, but if it's omitted we can use a default name. Here's what an add node command could look like

```
{
  "mindMapId": "1234-5678-9012-3456",
  "parentKey": "3",
  "name": "A new node"
}
```

The handler should find the mind map from the database, find the parent node, attach the new node to the parent, and save the changed mind map back into the database. Finally, it should publish an event on that mind map's address, letting everyone know that a change has been occurred.

Expand the command handler stub so it looks like this:

```
eventBus.registerHandler('mindMaps.editor.addNode',
  function(args) {
```

```
    eventBus.send('mindMaps.find', {_id: args.mindMapId},
      function(res) {
      if (res.mindMap) {
        var mindMap = res.mindMap;
        var parent  = findNodeByKey(res.mindMap, args.parentKey);
        var newNode = {key: newNodeKey()};
        if (args.name) {
          newNode.name = args.name;
        } else {
          newNode.name = 'Click to edit';
        }
        if (!parent.children) {
          parent.children = [];
        }
        parent.children.push(newNode);
        eventBus.send('mindMaps.save', mindMap, function() {
          publishMindMapEvent(mindMap, {event: 'nodeAdded',
            parentKey: args.parentKey, node: newNode});
        });
      }
    });
  });
```

The handler makes use of the three helper functions defined earlier. Let's go through the code step-by-step:

- We send an event to the `mindMaps.find` address to load the mind map specified in the command. This event is received by the handler we added to the `mindmaps.js` verticle in *Chapter 3*, *Integrating with a Database*.
- When the response arrives, we first check if the mind map was actually found, and then find the parent node of the new node. We also construct the object for our new node.
- We make sure that the `children` array of the parent node is initialized, and push our new node into it.
- We send the `mindMaps.save` event and attach our updated `mindMap` object. It will again be received by the `mindmaps.js` verticle. Once the save operation is finished, we publish an event about this change. This will let everyone, including the user that sent the command, know that a new node is added.

The rename node command handler

The rename node command will include the identifier of the mind map, the key of the node to rename, and the new name for the node:

```
{
  "mindMapId": "1234-5678-9012-3456",
  "key": "1",
  "newName": "a new name"
}
```

To handle this command, we need to find the node in question and set its name attribute to the new value. Expand the rename node command handler with the following code:

```
eventBus.registerHandler('mindMaps.editor.renameNode',
  function(args) {
  eventBus.send('mindMaps.find', {_id: args.mindMapId},
    function(res) {
    if (res.mindMap) {
      var mindMap = res.mindMap;
      var node    = findNodeByKey(mindMap, args.key);
      if (node) {
        node.name = args.newName;
        eventBus.send('mindMaps.save', mindMap, function(reply) {
          publishMindMapEvent(mindMap, {event: 'nodeRenamed', key:
            args.key, newName: args.newName});
        });
      }
    }
  });
});
```

Here, again we first need to find the mind map in question from the database. If the mind map is found, we proceed to find the node that will be renamed. If the node is found, we set its name to the new value, and save it back in the database. Once it is saved, we publish a rename event to let everyone know that the rename has been happened.

The delete node command handler

The delete node command needs two pieces of information, the identifier of the mind map and the key of the node to delete. For convenience, we will also include the key of the parent node. This isn't strictly necessary, but it eases the implementation.

```
{
  "mindMapId": "1234-5678-9012-3456",
  "parentKey": "1",
  "key": "2"
}
```

The implementation will first find the parent node. It will then remove the node we want to delete from its children. Expand the delete node command handler so that it looks like this:

```
eventBus.registerHandler('mindMaps.editor.deleteNode',
  function(args) {
  eventBus.send('mindMaps.find', {_id: args.mindMapId},
    function(res) {
    if (res.mindMap) {
      var mindMap = res.mindMap;
      var parent  = findNodeByKey(mindMap, args.parentKey);
      parent.children.forEach(function(child, index) {
        if (child.key === args.key) {
          parent.children.splice(index, 1);
          eventBus.send('mindMaps.save', mindMap,
            function(reply) {
            publishMindMapEvent(mindMap, {event: 'nodeDeleted',
              parentKey: args.parentKey, key: args.key});
          });
        }
      });
    }
  });
});
```

Once again, we first find the mind map in question. If the mind map is found, we find the parent node using the `findNodeByKey` function. We iterate through the parent's children using the JavaScript `Array.forEach` function. When the node to be deleted is encountered, we remove it from the array using the `Array.splice` function. We then save the mind map back in the database, and publish an event about what just happened.

Thus we are done with the server-side implementation of the editor!

Deploying the editor verticle

We need to deploy our new editor verticle, and add the operations used by it to the event bus bridge whitelist, so that they can be accessed from browsers.

The new inbound events we permit are `mindMaps.editor.addNode`, `mindMaps.editor.renameNode`, and `mindMaps.editor.deleteNode`. We also have our first permitted outbound events: the ones we publish for mind map changes.

At the time of deployment we don't know exactly what the addresses of the outbound events will be, because they're based on the identifiers of the mind maps. In such cases, instead of specifying the exact address, we can specify a regular expression matcher using the `address_re` attribute. Let's use this for both commands and events. Edit the deployment verticle `app.js`, so that it has the following content:

```
var container = require("vertx/container");
container.deployModule("io.vertx~mod-web-server~2.0.0-final", {
  port: 8080,
  host: "localhost",
  bridge: true,
  inbound_permitted: [
    { address: 'mindMaps.list' },
    { address: 'mindMaps.save' },
    { address: 'mindMaps.delete' },
    { address_re: 'mindMaps\\.editor\\..+' }
  ],
  outbound_permitted: [
    { address_re: 'mindMaps\\.events\\..+' }
  ]
});
container.deployModule("io.vertx~mod-mongo-
  persistor~2.0.0-final", {
  address: "mindMaps.persistor",
  db_name: "mind_maps"
});
container.deployVerticle('mindmaps.js');
container.deployVerticle('mindmap_editor.js');
```

You can now start or restart Vert.x to see if everything deploys:

```
vertx run app.js
```

The client

Let's focus on the client-side implementation of the editor. We want the user to be able to click on any existing mind map from the list and see the mind map's contents. The contents should update in real time, based on the changes that people are making. There should also be facilities for performing the three editing operations: adding nodes, renaming nodes, and deleting nodes.

To visualize the mind map, we will use D3, a popular JavaScript data visualization library. You don't need to be familiar with the library to be able to follow along. The most relevant thing to understand is that D3 is based on a very similar idea as jQuery. You can select certain parts of the document and then manipulate that selection by changing the attributes, adding and removing elements, and attaching event handlers.

The crucial difference between D3 and jQuery is that D3 supports data binding. This means that you can attach some JavaScript data to a selection. For example, you can attach a JavaScript array to a HTML list. Then, by using enter selection, you can specify how items in the array that are not yet on the screen should be rendered.

There's a lot more to D3 than this, and we'll only scratch the surface here. If you want to customize the code or just want to know more about D3, you can read the documentation at `http://d3js.org/`.

The mind map editor file

To keep our code clean, let's put the client-side editing features in a separate JavaScript file. We will implement a `MindMapEditor` object, which handles all the editing. This object will be constructed when the user opens a mind map.

Create a file named `editor.js` in the `web` subdirectory of the project. Add the following contents to it:

```
function MindMapEditor(mindMap, eventBus) {
  this.mindMap = mindMap;
  this.eventBus = eventBus;
  alert("Editor constructed");
}
```

This is the constructor function of the mind map editor. It will be called with two arguments: the mind map to be edited and a reference to the Vert.x event bus.

For now, we just issue a JavaScript alert dialog box from the constructor so that we can test that it has been initialized properly.

Updating the HTML

In the `index.html` page, following three changes are needed:

- We need to include the D3 JavaScript library. We can load it from the CloudFlare JavaScript CDN, as we did with jQuery and the event bus client in *Chapter 2*, *Developing a Vert.x Web Application*
- We need to include our own `editor.js` file
- We need an element in the HTML to which the editor will be placed

Let's add these things to `index.html` and edit it, so that it looks like this:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
</head>
<body>
  <ul class="mind-maps">
  </ul>
  <h2>Create a Mind Map</h2>
  <form class="create-form">
    <input type="text" name="name">
    <input type="submit" value="Create">
  </form>
  <section class="editor">
  </section>
  <script src="//cdnjs.cloudflare.com/ajax/libs/sockjs-
    client/0.3.4/sockjs.min.js"></script>
  <script src="//cdnjs.cloudflare.com/ajax/libs/vertx/
    2.0.0/vertxbus.min.js"></script>
  <script src="//cdnjs.cloudflare.com/ajax/libs/
    jquery/2.0.3/jquery.min.js"></script>
  <script src="//cdnjs.cloudflare.com/ajax/libs/d3/3.0.1/
    d3.v3.min.js"></script>
  <script src="/editor.js"></script>
  <script src="/client.js"></script>
</body>
</html>
```

Opening the editor

Next, let's implement the editor link for each mind map. The editor will be opened by clicking on the name of a mind map from the list. In *Chapter 2*, *Developing a Vert.x Web Application*, we wrapped the mind map names inside the `<span>` elements. Let's replace those with the `<a>` elements, and also attach the click handlers that will open the editor.

Change the contents of the `renderListItem` function in `web/client.js` so that it looks like this:

```
  var renderListItem = function(mindMap) {
    var li = $('<li>');
    var openMindMap = function() {
      new MindMapEditor(mindMap, eb);
      return false;
    };
    var deleteMindMap = function() {
      eb.send('mindMaps.delete', {id: mindMap._id}, function() {
        li.remove();
      });
      return false;
    };
    $('<a>').text(mindMap.name).attr('href', '#').on('click',
openMindMap).appendTo(li);
    $('<button>').text('Delete').on('click', deleteMindMap).
appendTo(li);
    li.appendTo('.mind-maps');
  };
```

When the name of a mind map is clicked, we call the `MindMapEditor` constructor function, passing it the mind map and the event bus object.

This is actually the only change we need to make to `client.js`. All of the editing features will be contained in `editor.js`.

If you now open the application in your browser, you should be able to click on the items in the mind map list, and see the alert dialog open for each item.

Sending the commands

Before digging into D3, let's do some preparation work. From the editor, we need to send the mind map editing commands to the event bus. For this purpose, add the following convenient functions to the `MindMapEditor` object's prototype in `web/editor.js`:

```
MindMapEditor.prototype.addNode = function(parentNode) {
  this.eventBus.send('mindMaps.editor.addNode', {
    mindMapId: this.mindMap._id,
    parentKey: parentNode.key
  });
}
MindMapEditor.prototype.renameNode = function(node, newName) {
  this.eventBus.send('mindMaps.editor.renameNode', {
    mindMapId: this.mindMap._id,
    key: node.key,
    newName: newName
  });
}
MindMapEditor.prototype.deleteNode = function(parentNode,
  childNode) {
  this.eventBus.send('mindMaps.editor.deleteNode', {
    mindMapId: this.mindMap._id,
    parentKey: parentNode.key,
    key: childNode.key
  });
}
```

The code here is straightforward. All we do is construct the command object for each operation and send it over the event bus.

Handling events

The other side of the equation is handling the incoming events. We need to register to the mind map's address on the event bus, and make changes to the mind map data structure so it matches the one on the server.

First, change the `MindMapEditor` constructor, replacing the `alert` call with a call to the `registerEventHandlers` function:

```
function MindMapEditor(mindMap, eventBus) {
  this.mindMap = mindMap;
  this.eventBus = eventBus;
  this.registerEventHandlers();
}
```

Next, implement the `registerEventHandlers` function. In this function, we register a handler function to the event bus, and look at the type of event with which we're dealing. Based on the event type, different handler functions are called:

```
MindMapEditor.prototype.registerEventHandlers = function() {
  var self = this;
  this.eventBus.registerHandler
    ('mindMaps.events.'+self.mindMap._id, function(event) {
    switch (event.event) {
      case 'nodeAdded':   self.onNodeAdded(event);   break;
      case 'nodeRenamed': self.onNodeRenamed(event); break;
      case 'nodeDeleted': self.onNodeDeleted(event); break;
    }
  });
}
```

The three handler functions closely mirror the implementations we have on the server. We just apply the change (add, rename, or delete) to the local mind map data structure:

```
MindMapEditor.prototype.onNodeAdded = function(event) {
  var parent = findNodeByKey(this.mindMap, event.parentKey);
  if (parent) {
    if (!parent.children) {
      parent.children = [];
    }
    parent.children.push(event.node);
  }
}
MindMapEditor.prototype.onNodeRenamed = function(event) {
  var node = findNodeByKey(this.mindMap, event.key);
  if (node) {
    node.name = event.newName;
  }
}
MindMapEditor.prototype.onNodeDeleted = function(event) {
  var parent = findNodeByKey(this.mindMap, event.parentKey);
  if (parent) {
    for (var i=0 ; i<parent.children.length ; i++) {
      if (parent.children[i].key === event.key) {
        parent.children.splice(i, 1);
        return;
      }
    }
  }
}
```

Sharing the findNodeByKey function

You may have noticed the event handler functions are using the `findNodeByKey` function. We have this function implemented on the server in the `mindmap_editor.js` verticle, but not yet in the browser code.

We could copy and paste the function to `web/editor.js` and be done with it, but we can do it better by sharing the code. This is a definite advantage of using the same programming language on the server and in the browser.

Sharing JavaScript code in Vert.x is very simple. All we need is a JavaScript file with the code that works both on the server and the client (that is, it shouldn't use any browser-only APIs, such as HTML manipulation or any Vert.x server-side APIs). The `findNodeByKey` function fits this description.

Vert.x uses the `CommonJS` module standard in JavaScript code, and that standard defines a specific mechanism for sharing code between JavaScript files (or modules). In every `CommonJS` module, there is an object named `exports`. Whenever we want to expose a function or a variable to someone outside the module, we just attach it to the `exports` object.

The problem is that web browsers do not currently implement this standard. For this purpose, we need a workaround so that the file will also load correctly in the browsers. We need to check whether the `exports` object is present, and if it is not, assume we're in a browser and assign the global `window` object to it.

Create a new file `web/mindmap_utils.js`, and move the `findNodeByKey` function from `mindmap_editor.js` to it. Then edit the code so that it looks like this:

```
if (typeof exports === 'undefined') {
  var exports = window;
}
exports.findNodeByKey = function(root, key) {
  if (root.key === key) {
    return root;
  } else if (root.children) {
    for (var i=0 ; i<root.children.length ; i++) {
      var match = exports.findNodeByKey(root.children[i], key);
      if (match) {
        return match;
      }
    }
  }
}
```

On the top of the `mindmap_editor.js` verticle, we need to load this new module, just like we have loaded the `vertx/event_bus` module:

```
var eventBus = require('vertx/event_bus');
var mindMapUtils = require('web/mindmap_utils');
```

Also change all the references to `findNodeByKey` in `mindmap_editor.js` to the fully qualified name `mindMapUtils.findNodeByKey`.

On the client, we can load the file by adding a `script` tag to `index.html`, before the `editor.js` `script` tag:

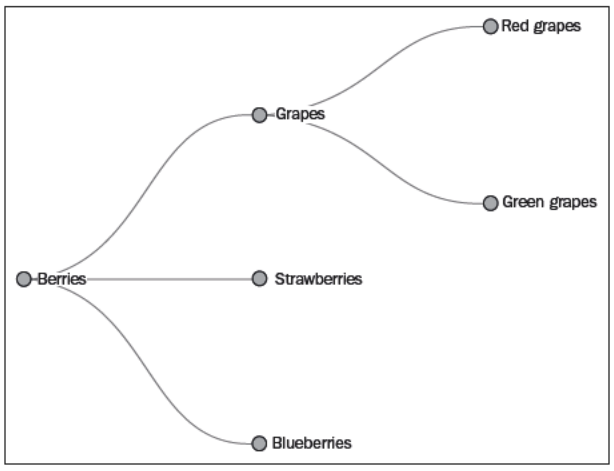
```
<script src="/mindmap_utils.js"></script>
```

And now, we have the same function implementation available both on the server and in the browser.

Notice that we now have four JavaScript files loaded by Vert.x on the server side: `app.js`, `mindmaps.js`, `mindmap_editor.js`, and `web/mindmap_utils.js`. However, only the first three of these are running as **verticles**. The last one is just a library code that we have loaded to the `mindmap_editor.js` verticle using the `require` function. We never deploy it using `container.deployVerticle`. The number of JavaScript files in a Vert.x application doesn't necessarily match the number of verticles exactly.

Initializing the visualization

Now we are all set to implement the mind map visualization in D3. What we end up with will look like this:



To construct the visualization, we will use the tree layout algorithm that comes with D3. It is suitable for visualizing tree-like data, such as our mind maps. Refer to `https://github.com/mbostock/d3/wiki/Tree-Layout` for a description of the algorithm.

With D3 you can create either HTML or SVG visualizations. We are going to use SVG because it makes it easy to create different kinds of visual elements.

SVG (Scalable Vector Graphics) is a standard for representing vector graphics as XML markup. It supports the most common graphical primitives (such as lines, circles, ellipses, polygons, and polylines), and also grouping primitives into more complex groups. SVG can be stored in standalone XML documents, but SVG elements can also be embedded into HTML documents, which is what we will do. All modern web browsers are capable of rendering SVG elements embedded in HTML. It is also possible to apply CSS styles to SVG elements.

For drawing the nodes and links, we use the following visual elements:

- SVG circle elements for showing the nodes
- SVG text elements for showing the node names
- D3 diagonals (backed by SVG path elements) for connecting the nodes. Refer to `https://github.com/mbostock/d3/wiki/SVG-Shapes#wiki-diagonal`

In `web/editor.js`, add a few common attributes to the `MindMapEditor` object:

```
MindMapEditor.width = 1280;
MindMapEditor.height = 800;
MindMapEditor.levelWidth = 150;
MindMapEditor.treeLayout = d3.layout.tree().size
  ([MindMapEditor.height, MindMapEditor.width]);
MindMapEditor.diagonalGenerator = d3.svg.diagonal().projection
  (function(d) { return [d.y, d.x]; });
```

The `width` and `height` attributes control the size of the visualization. The `levelWidth` attribute controls how wide one level in the tree is, that is, a node's child will be drawn `150` pixels to the right of it.

The `treeLayout` attribute contains an instance of D3's tree layout algorithm for our defined size. We will use it to calculate the positions of nodes and links. The layout implementation is stateless and is not bound to any specific mind map, which means we can use a shared instance of it for all mind maps.

The `diagonalGenerator` attribute contains a D3 diagonal generator. It is an object that can produce curved lines between two points, and we will use it to draw each line that links two nodes.

Next, let's define a function to initialize the visualization using the D3 API:

```
MindMapEditor.prototype.initVisualization = function() {
  this.vis = d3.select(".editor").html('').append("svg:svg")
.attr("width", MindMapEditor.width)
.attr("height", MindMapEditor.height)
.append("svg:g")
.attr("transform", "translate(10,0)");
}
```

We select the container section element from the HTML, by using D3's `select` function. Notice the similarity to jQuery element selection. We then clear its contents by calling the `html` function with an empty string.

Next, we create an `svg` element, and set its width and height to our defined size. The `svg` element is appended to the `editor` section in the HTML document. Within the `svg` element, we define an SVG `g` (group) element, and translate it slightly to the right, so that there will be some space on the left side of the visualization for the root node's name. This `g` element will be the container for our visualization. We store it in the `vis` attribute of the `MindMapEditor` object.

Next, update the `MindMapEditor` constructor function created earlier in this chapter so that it calls the `initVisualization` function:

```
function MindMapEditor(mindMap, eventBus) {
  this.mindMap = mindMap;
  this.eventBus = eventBus;
  this.registerEventHandlers();
  this.initVisualization();
}
```

Rendering the visualization

The final piece of the puzzle is the actual rendering of the visualization and the editing operations. D3 makes this easy for us, but there are still quite a few steps to it.

We will do all the work in a new function named `renderVisualization`. Add the following contents to it:

```
MindMapEditor.prototype.renderVisualization = function() {
  var self = this;
```

```
    var nodes = MindMapEditor.treeLayout.
      nodes(this.mindMap).reverse();
    nodes.forEach(function(d) { d.y = d.depth *
      MindMapEditor.levelWidth; });
    var node = this.vis.selectAll("g.node")
      .data(nodes, function(d) { return d.key; });
    var nodeEnter = node.enter().append("svg:g")
      .attr("class", "node")
      .attr("opacity", "0")
      .attr("transform", function(d) { return "translate(" + d.y +
      "," + d.x + ")"; });
    nodeEnter.append("svg:circle").attr("r", 4.5)
      .style("fill", "lightsteelblue")
      .on("click", function(c) { self.addNode(c); });
    nodeEnter.append("svg:text").attr("x", 10)
      .attr("dy", ".35em").text(function(d) { return d.name; })
      .on("click", function(d) {
        var text = prompt('Enter the name of this node', d.name);
        if (text) {
          self.renameNode(d, text);
        }
      });
    node.transition().attr("opacity", "1")
      .attr("transform", function(d) { return "translate(" + d.y +
        "," + d.x + ")"; })
      .select("text")
        .text(function(d) { return d.name; });
    node.exit().remove();
    var link = this.vis.selectAll("path.link")
      .data(MindMapEditor.treeLayout.links(nodes), function(d) {
        return d.target.key; });
    link.enter().insert("svg:path", "g")
      .attr("class", "link")
      .attr("opacity", "0")
      .attr("d", MindMapEditor.diagonalGenerator)
      .on('click', function(l) {
        self.deleteNode(l.source, l.target);
      });
    link.transition()
      .attr("d", MindMapEditor.diagonalGenerator)
      .attr("opacity", "1");
    link.exit().remove();
  }
```

There's a lot going on here! Let's walk through the code statement by statement.

First, we just grab the `this` value (the `MindMapEditor` object) to a local variable, so that we can use it in the inner functions:

```
var self = this;
```

Next, we construct the nodes to display it in the mind map by invoking the D3 tree layout algorithm. By default, it expects an object graph, with an array called `children` in each node. This exactly matches what we have in our mind map data structure:

```
var nodes =
  MindMapEditor.treeLayout.nodes(this.mindMap).reverse();
```

We then adjust the horizontal position of each node to match the level width in our mind map. One thing that may seem peculiar here is the use of the `y` value for controlling the horizontal position. This is because the tree layout algorithm presumes a top to bottom visualization, whereas we have a left to right visualization. Because of this, we need to flip the coordinates:

```
nodes.forEach(function(d) { d.y = d.depth *
  MindMapEditor.levelWidth; });
```

Each node in the graph will be a SVG `g` element with the CSS class `node`. Let's bind these elements to our data by selecting all of them from the document and calling the `data` function with our node data. The second argument to the `data` function is a key function, which will be used by D3 so that it can have an unique key to handle each node. We just use the `key` attribute of the mind map node.

```
var node = this.vis.selectAll("g.node")
    .data(nodes, function(d) { return d.key; });
```

For new nodes, we need to create those SVG `g` elements. To accomplish this, we use the D3 `enter` function and chain to it the transformations we want to apply for the new elements. In this case, we just append a `g` element with the `node` CSS class, a position as defined by the layout algorithm, and an opacity of `0`. Each node begins life as completely transparent, and we will shortly animate the value to opaque.

```
var nodeEnter = node.enter().append("svg:g")
  .attr("class", "node")
  .attr("transform", function(d) { return "translate(" + d.y +
  "," + d.x + ")"; }).attr("opacity", "0");
```

Inside the node group (`g` element), we need to define the node's contents. The first one is a circle that represents the node in the graph. It will be an SVG `circle` element with a radius of `4.5`.

The user can add a child for a node by clicking on the node. We attach an event handler for the click event to the circle. The handler calls the `addNode` function, which we had defined earlier.

```
nodeEnter.append("svg:circle")
  .attr("r", 4.5)
  .on("click", function(d) { self.addNode(d); });
```

The second piece of content each node has is the text label for the name. This is defined as a SVG `text` element, with a `x` attribute of `10` (counted from the left edge of the node group), a line height of `.35em`, and the name of the node as the content. The user can change the name of the node by clicking the text label. The click results in a prompt for the new name. The result of the prompt is sent to the `renameNode` function, along with the node itself:

```
nodeEnter.append("svg:text")
  .attr("x", 10)
  .attr("dy", ".35em")
  .text(function(d) { return d.name; })
  .on("click", function(d) {
    var text = prompt('Enter a name for this node', d.name);
    if (text) {
      self.renameNode(d, text);
    }
  });
```

We want the nodes to animate into existence. This is done using the D3 `transition` function, which is similar to the `selectAll` function that we used before, but instead of applying the final values immediately, it will animate the change over time. We animate both the opacity and the location of the node. We also update the name of the nodes with the transition, so that when name changes during the lifetime of the node, it will be displayed.

```
node.transition()
  .attr("opacity", "1")
  .attr("transform", function(d) { return "translate(" + d.y + ","
    + d.x + ")"; })
  .select("text")
    .text(function(d) { return d.name; });
```

The final operation for nodes is to define what happens when they are removed. We just want to remove the `node` element completely:

```
node.exit().remove();
```

The second part of visualization is the links connecting the nodes. Each link connects exactly two nodes, the source (the parent in the mind map) and the target (the child in the mind map). We will render the links as SVG `path` elements with the CSS class `link`. The initialization is similar to that of the nodes. We can get the links from the layout algorithm, and bind them to the SVG elements using the `data` function. We use the target node key as the link key:

```
var link = this.vis.selectAll("path.link")
  .data(MindMapEditor.treeLayout.links(nodes), function(d) {
    return d.target.key; });
```

For new links, we create the `path` element, set its CSS class to `link`, and make it initially transparent by setting its `opacity` to `0`. The points of `path` are defined by the `diagonalGenerator` object.

The user can delete a node by clicking on the link that connects to it. We add this behavior by attaching a click handler to the link, and calling the `deleteNode` function with the source node (the parent) and the target node (the node to delete).

```
link.enter().insert("svg:path", "g")
  .attr("class", "link")
  .attr("opacity", "0")
  .attr("d", MindMapEditor.diagonalGenerator)
  .on('click', function(l) {
    self.deleteNode(l.source, l.target);
  });
```

Similar to nodes, the positions and opacities of links are animated, so that transitions are smooth:

```
  link.transition()
    .attr("d", MindMapEditor.diagonalGenerator)
    .attr("opacity", "1");
```

Also, removed links are completely removed from the visualization:

```
link.exit().remove();
```

To gain a better understanding about the SVG structure that the D3 code produces, see the following markup. It's generated from a simple mind map with a root node that has two child nodes (with comments for clarity):

```
<!-- The root SVG element with the dimensions we've configured -->
<svg width="1280" height="800">
  <!-- All content wrapped in a simple g (group), moved
    slightly to the right -->
  <g transform="translate(10,0)">
```

```
    <!-- The links between the root and the two children, as
      SVG paths with CSS class link. The coordinates of the
        path generated by the diagonal generator -->
      <path class="link" opacity="1" d="M0,400C75,400 75,200
        150,200"></path>
      <path class="link" opacity="1" d="M0,400C75,400 75,600
        150,600"></path>
      <!-- The nodes themselves. Each one is a g (group) with CSS
        class node. The group is translated to the position
        of the node (calculated by the tree layout). In each
        groups there's the node circle and the text label.
        Note that elements in a group use coordinates relative
        to the group parent, not the whole SVG element -->
      <g class="node" transform="translate(150,600)" opacity="1">
        <circle r="4.5" style="fill: #b0c4de;"></circle>
        <text x="10" dy=".35em">Strawberries</text>
      </g>
      <g class="node" transform="translate(150,200)" opacity="1">
        <circle r="4.5" style="fill: #b0c4de;"></circle>
        <text x="10" dy=".35em">Grapes</text>
      </g>
      <g class="node" transform="translate(0,400)" opacity="1">
        <circle r="4.5" style="fill: #b0c4de;"></circle>
        <text x="10" dy=".35em">Berries</text>
      </g>
    </g>
  </svg>
```

Calling the render function

Finally, we need to add some code that calls the `renderVisualization` function.

We need to render the visualization when the editor is first opened. That can be achieved by invoking the `MindMapEditor` constructor:

```
function MindMapEditor(mindMap, eventBus) {
  this.mindMap = mindMap;
  this.eventBus = eventBus;
  this.registerEventHandlers();
  this.initVisualization();
  this.renderVisualization();
}
```

We also need to re-render the visualization whenever a change caused by an event has been applied. We can do that by calling the rendering function from the event handler function:

```
MindMapEditor.prototype.registerEventHandlers = function() {
  var self = this;
  this.eventBus.registerHandler
    ('mindMaps.events.'+self.mindMap._id, function(event) {
    switch (event.event) {
      case 'nodeAdded':   self.onNodeAdded(event);   break;
      case 'nodeRenamed': self.onNodeRenamed(event); break;
      case 'nodeDeleted': self.onNodeDeleted(event); break;
    }
    self.renderVisualization();
  });
}
```

Styling the visualization

If you take a look at the visualization in the browser, you'll notice that it doesn't look very good. The default SVG styling doesn't really do justice to our data. We can remedy this situation by adding some CSS styling.

Create a file named `style.css` in the `web` subdirectory. Add the following contents to it:

```
.node circle {
  cursor: pointer;
  fill: lightsteelblue;
  stroke: steelblue;
  stroke-width: 1.5px;
}
.node text {
  font-size: 11px;
}
path.link {
  cursor: pointer;
  fill: none;
  stroke: #ccc;
  stroke-width: 1.5px;
}
```

We style the node circles as blue circles, the node labels with a `11px` font size, and the links as grey lines with a width of `1.5px`. We also change the mouse cursor to a pointer when the user hovers over the nodes or links, to indicate click-ability.

Include the CSS file into the application by adding the following line in the `head` section of `web/index`:

```
<link rel="stylesheet" href="style.css"/>
```

Testing the editor

We are now all done with the mind map editor! You should be able to use it to create and edit mind maps.

To get a feel of the real-time features, open multiple browser windows with the same mind map open and see the updates in action. You can also try it out with some friends to really see how the real-time editing feels like.

Summary

This chapter has been quite a ride! We've taken our project from a simple CRUD interface to a full-blown real-time web application. The amount of code we have is more than doubled, but it still isn't that large if you consider what it can do. It shows the power of the Vert.x platform (with some help from jQuery and D3, of course).

In this chapter, we have covered:

- The command-event separation pattern for real-time applications
- Tapping into Java APIs from Vert.x JavaScript code
- Using the publish-subscribe pattern on the event bus
- Defining permitted events on the event bus bridge with regular expressions
- Sharing JavaScript code between the server and the client
- Using D3 to visualize a tree structure

In the next chapter, we will dive into the polyglot features of Vert.x by adding a Java verticle and integrating it with our existing code. We'll also discuss modules and how they help us in organizing and distributing code.

5
Polyglot Development and Modules

Our application is fully functional and people can use it for collaborative mind mapping. However, there is one crucial feature that we're missing: there is no way to get a finished mind map out of the application. It would be useful to have a capability to save mind maps as images, so that they could be sent over e-mail, printed, or uploaded to the websites. This is what we will implement in this chapter.

We will build a service that receives the SVG representation of the mind map that we created in the previous chapter and returns a PNG image of that mind map. We will use the Apache Batik library to do the heavy lifting of SVG generation. Along the way, we will see how Java libraries can be used in the Vert.x applications. We will implement the exporter verticle in Java instead of JavaScript, and package it as a separate Vert.x module, as a **worker verticle**.

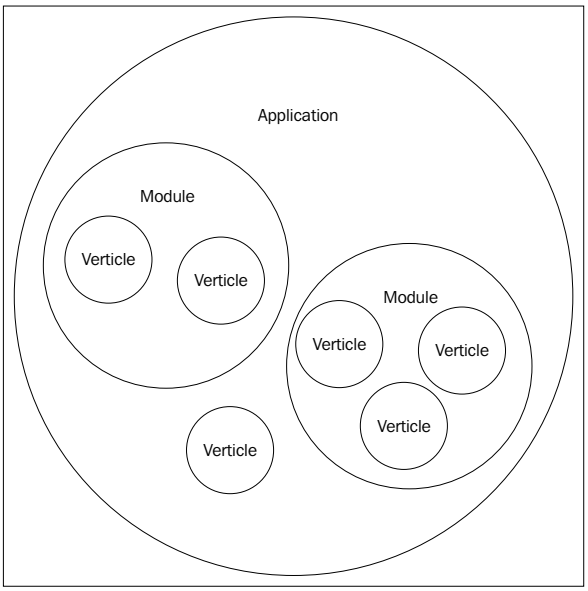
Vert.x modules and reuse

As we have seen, the different components of a Vert.x application are organized into verticles. The verticles are cohesive and decoupled runtime units that do not communicate with each other directly.

The next step is to discuss how these verticles can be distributed across projects and developed as independent units.

Because verticles are plain source files, you could certainly reuse the verticles by just copying them over to different projects or by other such manual sharing techniques. Fortunately, this isn't necessary, because Vert.x has a higher-level concept for reuse: **Modules**.

A Vert.x module is a collection of verticles and other code that is packaged as a unit, and then referenced from other Vert.x modules or applications.



The structure of a module is really quite simple. It is a directory that contains a module descriptor file, the module code (usually in the form of one or more verticles), and other resources.

The module descriptor is a simple JSON file, always named `mod.json`, which defines general information about the module, such as its dependencies, licensing, and authors.

Every Vert.x module has a name and a version. When you include a module in a Vert.x application, you specify both the name of the module and the version you want to use. This is similar to how package managers work in many programming languages. For example, if you're familiar with Maven dependencies in Java, or Gem dependencies in Ruby, Vert.x module management should seem familiar.

We have already used a couple of modules in our application: the web server module and the Mongo Persistor module, which we have specified in our `app.js` deployment verticle. In this chapter, we are going to write one module of our own and include it in our application. This is the recommended approach with nontrivial Vert.x applications. Divide them into a collection of modules, where each module has a specific purpose.

Distributing modules

Although we won't be using modules across projects in this book, it is something most Vert.x developers want to do sooner or later. In Vert.x, module distribution can be achieved through module repositories.

As we discussed in *Chapter 3*, *Integrating with a Database*, Vert.x leverages the Maven dependency infrastructure for module distribution. You can upload a Vert.x module to a private or public Maven repository for others to download. To make this even easier, Vert.x provides project templates for the Maven and Gradle build tools. See the Maven and Gradle manuals at `http://vertx.io` for more information.

If you want to publish a module as an open source project, you can also register it in the **Vert.x Module Registry** at `http://modulereg.vertx.io`, where other Vert.x users can discover it.

Creating a module

Our image generator module won't be reused in other projects at this point, so we'll use the easiest method for creating one, and that is within the directory structure of our existing application.

If you take a look at the `mods` subdirectory of our project, you'll see two subdirectories: `io.vertx~mod-mongo-persistor~2.0.0-final` and `io.vertx~mod-web-server~v2.0.0-final`. These are the modules Vert.x has downloaded for us from the public module repositories.

Vert.x module directory names follow the naming convention `[package]~[name]~[version]`. It is recommended to use a Java style reverse domain name (such as `io.vertx`) as the package name to prevent naming conflicts when modules are distributed.

Let's add our new module in the `mods` directory.

The module directory

Create a directory for the module using the following command:

```
mkdir -p mods/com.vertxbook~mod-svg2png~1.0
```

Here, `com.vertxbook` is the package name for the module, `mod-svg2png` is the name of the module, and `1.0` is the version of this module.

Module descriptor

Our module will be a runnable module, meaning that it can be deployed with the `deployModule` function (similar to the Web Server and the Mongo Persistor). Runnable modules must define a main verticle, which is the verticle that will be started when the module itself is started.

Our module will also be a worker module, because generating PNG images can take some time to run. See the following tip for a discussion on worker modules.

Worker verticles

As we discussed in *Chapter 2*, *Developing a Vert.x Web Application*, Vert.x has a specific and well-defined concurrency model for controlling where and when the verticle code is running. An important part of the concurrency model is the concept of a reactor. Each verticle is run in a specific reactor event loop, and there is a limited number of those event loops in any Vert.x instance (based on the number of CPU cores on the server).

One of the implications of this model is that a verticle running in a reactor event loop should never reserve the loop for a long period of time. If a single verticle spends too much time on the event loop, other verticles cannot proceed and the reactor becomes exhausted. This means that a normal verticle should never run code that has to wait for IO operations or uses a lot of CPU time. For this kind of code, Vert.x provides the concept of worker verticles, which are verticles that run outside of the Vert.x event loops, in a separate background thread pool.

Worker verticles can communicate via the event bus just like other verticles, but they are not assigned to specific threads like normal verticles. Because they don't strictly follow the Vert.x concurrency model, they also don't reap the scalability benefits of that model. For this reason, it is recommended to use them sparingly, only when necessary.

Create the module descriptor in a file named `mod.json` in the module directory, and set its contents to:

```
{
  "main": "PNGExporter.java",
  "worker": true
}
```

The main verticle set here is `PNGExporter.java`. We will create it in a moment.

This is everything we need to configure in the module descriptor, though there are many additional supported configuration entries. See the Vert.x modules manual at `http://vertx.io` for more information on module configuration.

Libraries

Our SVG to PNG conversion code will make use of the Apache Batik library (`http://xmlgraphics.apache.org/batik/`), which is a toolkit for working with SVG documents in Java. Batik is a large library with many features, but we are going to use just one of them; that is, the ability to turn SVG documents into PNG images.

We need to obtain the Batik library and include it in our application. In practice, that means putting some Batik JAR files in the classpath of our module. Vert.x knows that any JARs that are in a subdirectory named `lib` should be added to the module's classpath, so that is where we will put them.

Create the `lib` subdirectory for our module using this command:

```
mkdir -p mods/com.vertxbook~mod-svg2png~1.0/lib
```

Open `http://xmlgraphics.apache.org/batik/download.html` in your web browser, find the latest Batik binary download archive, download it, and unpack it somewhere on your machine.

In the extracted directory structure, there is a `lib` subdirectory with several JAR files in it. Copy the JAR files over to the newly created `lib` directory in our module:

```
cp [batik_dir]/lib/*.jar mods/com.vertxbook~mod-svg2png~1.0/lib
```

When you need to use the same Java library in many Vert.x modules, it can quickly become unwieldy to have to keep copying its JAR files around. Although you cannot directly include plain Java libraries from Maven repositories, there's a useful trick to get around this limitation. That is to package the JARs up as a nonrunnable Vert.x module (one without any verticles). This module can then be defined as a dependency of any module that needs the library. It can also be uploaded to Maven repositories for distribution. See the Vert.x modules manual for more information on including modules in other modules.

Implementing the module

Now, we are ready to implement the actual code for our module. We could do this in JavaScript similar to what we've been doing so far, but because we're integrating a Java library, the most suitable language is Java.

A Java verticle is a Java class that extends the class `Verticle` defined by the Vert.x framework. All Java verticles must implement the method `public void start()`, which is called when the verticle starts.

The contents and APIs used to implement the Java verticles are pretty much the same as what we have seen in JavaScript. We'll just need a bit more structure and bookkeeping because of the static nature of the Java language.

Java is a compiled language, but there is no need to compile Java verticles before running them. Vert.x can pick up the Java source files and compile them at runtime.

Create the file named `PNGExporter.java` in the `mods/com.vertxbook~mod-svg2png~1.0` module directory (matching the main verticle name we configured in `mod.json`), and add the basic Java verticle structure:

```
import org.vertx.java.platform.Verticle;
public class PNGExporter extends Verticle {
  public void start() {
  }
}
```

Let's define the event bus message handler that takes some SVG data and some CSS styles as arguments and responds with a PNG image. Whereas in JavaScript an event handler is a plain JavaScript function, in Java it is an implementation of the `Handler` interface. Let's register such an implementation on the event bus:

```
import org.vertx.java.core.Handler;
import org.vertx.java.core.eventbus.Message;
import org.vertx.java.core.json.JsonObject;
import org.vertx.java.platform.Verticle;
public class PNGExporter extends Verticle {
  public void start() {
    Handler<Message<JsonObject>>exportHandler =
      new Handler<Message<JsonObject>>() {
      public void handle(Message<JsonObject> message) {
        String svg = message.body().getString("svg");
        String css = message.body().getString("css");
        message.reply(new JsonObject().putString("data",
          getPng(svg, css)));
      }
    };
    vertx.eventBus().registerHandler(
      "com.vertxbook.svg2png", exportHandler);
  }
}
```

Here, we defined an inline implementation of the `Handler` interface named `exportHandler`. `Handler` is a generic interface, typed by the kind of object it handles. In this case, it is a `Message` object, which is the type of every event that's passed over the event bus. `Message` itself is also a generic type, defined by the type of the event payload. We will pass the JSON data with the event, just as we have done before, so that we can set the type as `JsonObject`.

In the handler implementation we reach into the JSON object passed with the event and get the `svg` and `css` attributes from it. We expect those to hold the SVG and CSS data. We then call the `getPng()` method with that data. Then we respond to the event with a new JSON object that has a single key `data` whose value is the resulting PNG data.

Finally, we register this handler on the Vert.x event bus in the address `com.vertxbook.svg2png`.

Hopefully, you can see the similarities here to what we have done before in JavaScript.

Let's finalize the verticle by adding a method which, given some SVG and CSS data, produces a PNG image. Since the data transferred over the event bus is a JSON object and JSON does not support binary data, we need to encode the PNG image into a base64 string before returning it. For that purpose, we also define a couple of helper methods:

```
import java.io.*;
import org.apache.batik.transcoder.*;
import org.apache.batik.transcoder.image.*;
import org.apache.batik.util.*;
import org.vertx.java.core.Handler;
import org.vertx.java.core.eventbus.Message;
import org.vertx.java.core.json.JsonObject;
import org.vertx.java.deploy.Verticle;
public class PNGExporter extends Verticle {
  public void start() {
    Handler<Message<JsonObject>> exportHandler =
      new Handler<Message<JsonObject>>() {
      public void handle(Message<JsonObject> message) {
        String svg = message.body().getString("svg");
        String css = message.body().getString("css");
        message.reply(new JsonObject().putString("data",
          getPng(svg, css)));
      }
    };
    vertx.eventBus().registerHandler(
        "com.vertxbook.svg2png", exportHandler);
  }
```

```
  private String getPng(String svg, String css) {
    String cssDataUrl =
      "data:text/css;base64,"+encodeBase64(css);
    try {
      PNGTranscoder transcoder = new PNGTranscoder();
      transcoder.addTranscodingHint(
        SVGAbstractTranscoder.KEY_USER_STYLESHEET_URI,
          cssDataUrl);
      TranscoderInput input = new TranscoderInput(
        new StringReader(svg));
      ByteArrayOutputStream output =
        new ByteArrayOutputStream();
      transcoder.transcode(input,
        new TranscoderOutput(output));
      return encodeBase64(output.toByteArray());
    } catch (TranscoderException te) {
      container.logger().error("Could not convert SVG", te);
      return null;
    }
  }
  private String encodeBase64(String input) {
    return encodeBase64(input.getBytes());
  }
  private String encodeBase64(byte[] input) {
    try {
         ByteArrayOutputStream out =
          new ByteArrayOutputStream();
      Base64EncoderStream encoder =
          new Base64EncoderStream(out);
      encoder.write(input);
      encoder.close();
      return new String(out.toByteArray());
    } catch (IOException ioe) {
      container.logger().error(
          "Could not encode Base64", ioe);
      return null;
    }
  }
}
```

This is the full implementation of our SVG to PNG verticle. Let's go over the new additions one by one:

- The `getPng()` method takes two string arguments: the SVG and the CSS data. It returns a string, which will be a base64-encoded PNG image.
- The first thing we do is to construct a Batik `PNGTranscoder`, which is an object that can transform an SVG document to a PNG image. To make the document look nice, we pass in our CSS stylesheet embedded in a data URL (`http://en.wikipedia.org/wiki/Data_URI_scheme`), and also encoded in base64 to the transcoder.
- We set up the input and output for the transcoder. The input is the SVG data wrapped in a `TranscoderInput` object. The output is a Java `OutputStream`, into which Batik will write the PNG. We use a `ByteArrayOutputStream` object so that we can access the resulting PNG bytes later.
- We then call the `transcode()` method of the transcoder. This will do the actual transcoding work.
- We encode the resulting PNG byte array to a base64-encoded string and return it.
- Any exceptions that may have happened during the transcoding are logged using the built-in Vert.x logger.
- The helper methods, named `encodeBase64`, implement the code for turning a byte array into a base64-encoded string. Batik includes an `OutputStream` implementation named `Base64EncoderStream` for this purpose. Because we're using arrays and strings instead of streams, we can use these wrapper methods for more convenient access to the encoder.

Here, we use the logging implementation provided by the Vert.x platform through `container.logger()`. The logger is similar to the one used in many languages and frameworks, and provides methods for logging messages of different severities: `trace`, `debug`, `info`, `warn`, `error`, and `fatal`. The output of the logger goes to a file named `vert.x log` in the system's `temp` directory by default:

```
tail -f $TMPDIR/vertx.log
```

The logfile location and other aspects of the Vert.x logger can be configured through the `conf/logging.properties` file in the Vert.x installation directory. See the Vert.x main manual for more information.

Deploying the module

Our export verticle is completed, but we haven't integrated it into the rest of our application yet. Vert.x doesn't automatically deploy modules from the `mods` directory. You always need to deploy them explicitly. Let's do that from the deployment verticle `app.js`:

```
var container = require("vertx/container");
container.deployModule("io.vertx~mod-web-server~2.0.0-final", {
  port: 8080,
  host: "localhost",
  bridge: true,
  inbound_permitted: [
    { address: 'mindMaps.list' },
    { address: 'mindMaps.create' },
    { address: 'mindMaps.delete' },
    { address_re: 'mindMaps\\.editor\\..+' },
    { address: 'com.vertxbook.svg2png' }
  ],
  outbound_permitted: [
    { address_re: 'mindMaps\\.events\\..+' }
  ]
});
container.deployModule("io.vertx~mod-mongo-persistor~2.0.0-final", {
  address: "mindMaps.persistor",
  db_name: "mind_maps"
});
container.deployVerticle('mindmaps.js');
container.deployVerticle('mindmap_editor.js');
container.deployModule('com.vertxbook~mod-svg2png~1.0', null, 3);
```

We called the `deployModule` function to deploy our new module, similar to what we have done for the Web Server and the Mongo Persistor modules earlier. In addition to the `package~name~version` string, we gave two additional arguments to the function:

- The module configuration object, which is `null` in this case, because our module doesn't need any configuration
- The number of module instances to launch. We launched three different instances of the module. This means that three export jobs can be running concurrently. We will discuss this and other performance tuning issues more in the next chapter

Because we will be calling the exporter from the web browser soon, we also need to expose the event handler on the event bus bridge. We've done this by adding its address to the `inbound_permitted` whitelist of the Vert.x web server configuration.

Although the PNG exporter code is organized as a module, there is nothing in the `PNGExporter.java` verticle that knows anything about modules. It is just a verticle, and it doesn't care whether it's in a module or not. In fact, you could run the PNG exporter as a standalone application by invoking `vertx run PNGExporter.java` in the module directory. However, in that case Vert.x would not recognize the module convention of putting libraries in the `lib` directory. Instead you would need to specify all the JARs used on the command line, using the `-cp` switch.

Integrating the client

At the user interface, we want to have a button visible on the screen when the user opens a mind map. By clicking on the button, the user will receive a PNG image of that mind map. This requires some additions to our HTML and client-side JavaScript code.

In `web/index.html`, just under the editor `<section>` element, and above the first `<script>` element, add a **Save as image** button:

```
<button class="save-as-png" style="display: none;">
  Save as image
</button>
```

The button will be hidden at first, because we only want to show it once a mind map has been opened.

Let's implement the button's behavior in `web/client.js`:

```
var eb = new vertx.EventBus(window.location.protocol + '//' +
  window.location.hostname + ':' +
    window.location.port + '/eventbus');
eb.onopen = function() {
  var renderListItem = function(mindMap) {
    var li = $('<li>');
    var openMindMap = function() {
      new MindMapEditor(mindMap, eb);
      $('.save-as-png').show();
      return false;
    };
    var deleteMindMap = function() {
      eb.send('mindMaps.delete', {id: mindMap._id}, function() {
        li.remove();
      });
      return false;
    };
```

```
    $('<a>').text(mindMap.name).attr('href', '#').on('click',
      openMindMap).appendTo(li);
    $('<button>').text('Delete').on('click',
      deleteMindMap).appendTo(li);
    li.appendTo('.mind-maps');
  };
  $('.create-form').submit(function() {
    var nameInput = $('[name=name]', this);
    eb.send('mindMaps.save', {name: nameInput.val()},
      function(result) {
      renderListItem(result);
      nameInput.val('');
    });
    return false;
  });
  $('.save-as-png').click(function() {
    var svg = $('.editor').html();
    var stylesheet = document.styleSheets[0];
    var css = '';
    for (var i = 0 ; i < stylesheet.cssRules.length ; i++) {
      css += stylesheet.cssRules[i].cssText;
      css += "\n";
    }
    eb.send('com.vertxbook.svg2png', {svg: svg, css: css},
        function(result) {
      if (result.data) {
        window.location.href =
            ''+result.data;
      }
    });
    return false;
  });
  eb.send('mindMaps.list', {}, function(res) {
    $.each(res.mindMaps, function() {
      renderListItem(this);
    })
  })
};
```

Here, when a mind map is opened, we find and show the **Save as image** button using jQuery. We attach a click handler to the button.

The first thing we do in the handler is grabbing the SVG markup of the current mind map by selecting the editor from the page, and reading its contents using jQuery's `html` function. The D3 code that we wrote in the previous chapter has populated this element with the mind map SVG.

We also grab the CSS rules needed to make the mind map layout the same in the PNG as it is in the browser. The API provided by browsers for this is slightly awkward, but it gets the job done.

Finally we send SVG and CSS data to the address `com.vertxbook.svg2png` on the event bus. When we get a result, we set the browser's address to a data URL containing the newly created PNG image. The user will then be able to see the image.

Now we have a fully functional **Save as image** feature in our application. You can try it by making a mind map, and then clicking on the **Save as image** button. Your browser should navigate to a PNG image of the current mind map. The PNG transcoding may take a moment on the first run, because the JVM is loading and initializing the Batik library. It will get significantly faster during the subsequent runs as the JVM warms up.

Polyglot programming in Vert.x

Vert.x is a truly polyglot programming platform, meaning that it is built from the ground up to support the applications and systems that mix and match many different programming languages. Now, we have seen the building blocks that make this possible.

Notice how we have both the JavaScript code and Java code in our application. We haven't had to write any foreign function interfaces or wrapper code to make them work together. This is largely because of the event bus. All cross-verticle communication happens as events and is handled asynchronously. Verticles communicate by passing data to specific addresses on the event bus, and never by calling each other's functions directly.

You don't even need to know in which language a specific verticle is written; it simply does not have any effect on the way you use it.

Summary

In this chapter, we have seen how we can extend the Vert.x applications by including a Java library. Because Vert.x runs on the JVM, we are able to use any Java library in our applications, and there are a lot of Java libraries in the world. In our case, SVG to PNG transformation is not a simple task; so we are fortunate to be able to leverage Apache Batik to do the heavy lifting.

In this chapter, we have covered:

- Defining our own Vert.x modules to enable sharing code between projects and organizations
- Creating worker verticles for CPU intensive tasks or tasks that use blocking IO operations
- Implementing a Java verticle
- Including Java libraries in the Vert.x modules and applications
- Using the Apache Batik library to turn a SVG document into a PNG image

This wraps up the implementation of our application. In the next chapter, we will deploy the application to a server. We will also discuss issues related to scaling Vert.x applications.

6
Deploying and Scaling Vert.x

In this chapter, you are going to learn how to deploy a Vert.x web application on a server, making it available on the Internet. You'll also learn how to set up deployment scripts that enable the continuous delivery of updated versions of your application.

Finally, we will discuss some of the basics of scaling the Vert.x platform to accommodate large amounts of traffic and data.

Deploying a Vert.x application

There are many ways to deploy Vert.x on different flavors of Unix and Windows, and it really doesn't require much more than installing Vert.x and routing HTTP and WebSocket traffic to it.

We will be walking through one deployment scenario, which has been shown to work well in the real world. It involves deploying our Vert.x application to an Ubuntu Linux server.

Setting up an Ubuntu box

We are going to set up an Ubuntu virtual machine using the Vagrant tool. This virtual machine will simulate a real production server. If you already have an Ubuntu box (or a similar Linux box) handy, you can skip this step and move on to setting up a user.

Vagrant (`http://www.vagrantup.com/`) is a tool for managing virtual machines. Many people use it to manage their development environments so that they can easily share them and test their software on different operating systems. For us, it is a perfect tool to practice Vert.x deployment into a Linux environment.

Install Vagrant by heading to the **Downloads** area at `http://vagrantup.com` and selecting the latest version. Select a package for your operating system and run the installer. Once it is done, you should have a `vagrant` command available on the command line as follows:

```
vagrant -v
```

Navigate to the root directory of our project and run the following command:

```
vagrant init precise64 http://files.vagrantup.com/precise64.box
```

This will generate a file named `Vagrantfile` in the project folder. It contains configuration for the virtual machine we're about to create. We initialized a `precise64` box, which is shorthand for the 64-bit version of Ubuntu 12.04 Precise Pangolin. Open the file in an editor and find the following line:

```
# config.vm.network :private_network, ip: "192.168.33.10"
```

Uncomment the line by removing the **#** character. This will enable private networking for the box. We will be able to conveniently access it with the IP address `192.168.33.10` locally.

Run the following command to download, install, and launch the virtual machine:

```
vagrant up
```

This command launches the virtual machine configured in the `Vagrantfile`. On first launch, it will also download it. Because of this, running the command may take a while.

Once the command is finished, you can check the status of the virtual machine by running `vagrant status`, suspend it by running `vagrant suspend`, bring it back up by running `vagrant up`, and remove it by running `vagrant destroy`.

Setting up a user

For any application deployment, it's a good idea to have an application-specific user configured. The sole purpose of the user is to run the application. This gives you a nice way to control permissions and make sure the application can only do what it's supposed to do.

Open a shell connection to our Linux box. If you followed the steps to set up a Vagrant box, you can do this by running the following command in the project root directory:

```
vagrant ssh
```

Add a new user named `mindmaps` using the following command:

```
sudo useradd -d /home/mindmaps -m mindmaps
```

Also specify a password for the new user using the following command (and make a note of the password you choose; you'll need it):

```
sudo passwd mindmaps
```

Installing Java on the server

Install Java for the Linux box, as described in *Chapter 1, Getting Started with Vert.x*. As a quick reminder, Java can be installed on Ubuntu with the following command:

```
sudo apt-get install openjdk-7-jdk
```

On fresh Ubuntu installations, it is a good idea to always make sure the package manager index is up-to-date before installing any packages. This is also the case for our Ubuntu virtual machine. Run the following command if the Java installation fails:
`sudo apt-get update`

Installing MongoDB on the server

We also need MongoDB to be installed on the server, for persisting the mind maps. Install MongoDB following the instructions in *Chapter 3, Integrating with a Database*.

Setting up privileged ports

Our application is configured to serve requests on port 8080. When we deploy to the Internet, we don't want users to have to know anything about ports, which means we should deploy our app to the default HTTP port 80 instead.

On Unix systems (such as Linux) port 80 can only be bound to by the root user. Because it is not a good idea to run applications as the root user, we should set up a special privilege for the `mindmaps` user to bind to port 80. We can do this with the `authbind` utility.

`authbind` is a Linux utility that can be used to bind processes to privileged ports without requiring root access.

Install `authbind` using the package manager with the following command:

```
sudo apt-get install authbind
```

Set up a privilege for the `mindmaps` user to bind to port 80, by creating a file into the `authbind` configuration directory with the following command:

```
cd /etc/authbind/byport/
sudo touch 80
sudo chown mindmaps:mindmaps 80
sudo chmod 700 80
```

When `authbind` is run, it checks from this directory, whether there is a file corresponding to the used port and whether the current user has access to it. Here, we have created such a file.

Many people prefer to have a web server such as Nginx or Apache as a frontend and not expose backend services to the Internet directly. This can also be done with Vert.x. In that case, you could just deploy Vert.x to port 8080 and skip the `authbind` configuration. Then, you would need to configure reverse proxying for the Vert.x application in your web server.

Note that we are using the event bus bridge in our application, and that uses HTTP WebSockets as the transport mechanism. This means the front-end web server must be able to also proxy WebSocket traffic. Nginx is able to do this starting from version 1.3 and Apache from version 2.4.5.

Installing Vert.x on the server

Switch to the `mindmaps` user in the shell on the virtual machine using the following command:

```
sudo su - mindmaps
```

Install Vert.x for this user, as described in *Chapter 1*, *Getting Started Vert.x*. As a quick reminder, it can be done by downloading and unpacking the latest distribution from `http://vertx.io`.

Making the application port configurable

Let's move back to our application code for a moment. During development we have been running the application in port 8080, but on the server we will want to run it in port 80. To support both of these scenarios we can make the port configurable through an environment variable.

Vert.x makes environment variables available to verticles through the container API. In JavaScript, the variables can be found in the `container.env` object. Let's use it to give our application a port at runtime. Find the following line from the deployment verticle `app.js`:

```
port: 8080,
```

Change it to the following line:

```
port: parseInt(container.env.get('MINDMAPS_PORT')) || 8080,
```

This gets the `MINDMAPS_PORT` environment variable, and parses it from a string to an integer using the standard JavaScript `parseInt` function. If no port has been given, the default value 8080 is used.

We also need to change the host configuration of the web server. So far, we have been binding to localhost, but now we also want the application to be accessible from outside the server. Find the following line in `app.js`:

```
host: "localhost",
```

Change it to the following line:

```
host: "0.0.0.0",
```

Using the host `0.0.0.0` will make the server bind to all IPv4 network interfaces the server has.

Setting up the application on the server

We are going to need some way of transferring the application code itself to the server, as well as delivering incremental updates as new versions of the application are developed. One of the simplest ways to accomplish this is to just transfer the application files over using the **rsync** tool, which is what we will do.

rsync is a widely used Unix tool for transferring files between machines. It has some useful features over plain file copying, such as only copying the deltas of what has changed, and two-way synchronization of files.

Create a directory for the application on the home directory of the `mindmaps` user using the following command:

```
mkdir ~/app
```

Go back to the application root directory and transfer the files from it to the new remote directory:

```
rsync -rtzv  . mindmaps@192.168.33.10:~/app
```

Testing the setup

At this point, the project working tree should already be in the application directory on the remote server, because we have transferred it over using rsync. You should also be able to run it on the virtual machine, provided that you have the JDK, Vert.x, and MongoDB installed, and that you have `authbind` installed and configured. You can run the app with the following commands:

```
cd ~/app
JAVA_OPTS="-Djava.net.preferIPv4Stack=true" MINDMAPS_PORT=80 authbind ~/
vert.x-2.0.1-final/bin/vertx run app.js
```

Let's go through the file bit by bit as follows:

- We pass a Java system parameter named `java.net.preferIPv4Stack` to Java via the `JAVA_OPTS` environment variable. This will have Java use IPv4 networking only. We need it because the `authbind` utility only supports IPv4.
- We also explicitly set the application to use port 80 using the `MINDMAPS_PORT` environment variable.
- We wrap the Vert.x command with the `authbind` command.
- Finally, there's the call to Vert.x. Substitute the path to the Vert.x executable with the path to which you installed Vert.x.

After starting the application, you should be able to see it by navigating to `http://192.168.33.10` in a browser.

Setting up an upstart service

We have our application fully operational, but it isn't very convenient or reliable to have to start it manually. What we'll do next is to set up an Ubuntu upstart job that will make sure the application is always running and survives events like server restarts.

Upstart is an Ubuntu utility that handles task supervision and automated starting and stopping of tasks when the machine starts up, shuts down, or when some other events occur. It is similar to the `/sbin/init` daemon, but is arguably easier to configure, which is the reason we'll be using it.

The first thing we need to do is set up an upstart configuration file. Open an editor with root access (using `sudo`) for a new file `/etc/init/mindmaps.conf`, and set its contents as follows:

```
start on runlevel [2345]
stop on runlevel [016]

setuid mindmaps
setgid mindmaps

env JAVA_OPTS="-Djava.net.preferIPv4Stack=true"
env MINDMAPS_PORT=80

chdir /home/mindmaps/app
exec authbind /home/mindmaps/vert.x-2.0.1-final/bin/vertx run app.js
```

Let's go through the file bit by bit, as follows:

- On the first two lines, we configure when this service will start and stop. This is defined using runlevels, which are numeric identifiers of different states of the operating system (`http://en.wikipedia.org/wiki/Runlevel`). 2, 3, 4, and 5 designate runlevels where the system is operational; 0, 1, and 6 designate runlevels where the system is stopping or restarting.
- We set the user and group the service will run as to the `mindmaps` user and its group.
- We set the two environment variables we also used earlier when testing the service: `JAVA_OPTS` for letting Java know it should only use the IPv4 stack, and `MINDMAPS_PORT` to let our application know that it should use port 80.
- We change the working directory of the service to where our application resides, using the `chdir` directive.
- Finally, we define the command that starts the service. It is the `vertx` command wrapped in the `authbind` command. Be sure to change the directory for the `vertx` binary to match the directory to which you installed Vert.x.

Let's give the `mindmaps` user the permission to manage this job so that we won't have to always run it as root. Open up the `/etc/sudoers` file into an editor with the following command:

```
sudo /usr/sbin/visudo
```

At the end of the file, add the following line:

```
mindmaps ALL = (root) NOPASSWD: /sbin/start mindmaps, /sbin/stop
mindmaps, /sbin/restart mindmaps, /sbin/status mindmaps
```

The `visudo` command is used to configure the privileges of different users to use the `sudo` command. With the line we added, we enabled the `mindmaps` user to run a few specific commands without having to supply a password.

At this point, you should be able to start and stop the application as the `mindmaps` user:

```
sudo start mindmaps
```

You also have the following additional commands available for managing the service:

```
sudo status mindmaps
sudo restart mindmaps
sudo stop mindmaps
```

If there is a problem with the commands, there might be a configuration error. The upstart service will log errors to the file: `/var/log/upstart/mindmaps.log`. You will need to open it using the `sudo` command.

Deploying new versions

Deploying a new version of the application consists of the following two steps:

1. Transferring new files over using rsync
2. Restarting the mind maps service

We can make this even easier by creating a shell script that executes both steps. Create a file named `deploy.sh` in the root directory of the project, and set its contents as:

```
#!/bin/sh
rsync -rtzv . mindmaps@192.168.33.10:~/app/
ssh mindmaps@192.168.33.10 sudo restart mindmaps
```

Make the script executable, using the following command:

```
chmod +x deploy.sh
```

After this, just run the following command whenever you want a new version on the server:

```
./deploy.sh
```

To make deployment even more streamlined, you can set up SSH public key authentication so that you won't need to supply the password of the `mindmaps` user as you deploy. See `https://help.ubuntu.com/community/SSH/OpenSSH/Keys` for more information.

Scaling a Vert.x application

Vert.x is a very efficient framework, and because of the asynchronous nature of Vert.x code, performance bottlenecks are much more likely to crop up in databases or other dependent systems. When you do need to scale the Vert.x application itself, the following techniques are available.

Verticle counts

In *Chapter 5*, *Polyglot Development and Modules*, when deploying the `svg2png` worker verticle, we deployed three instances of it. This is similar to setting up three concurrent worker threads for processing a task in a more traditional system. At most three instances of the same job can be running concurrently.

The same scaling technique is also available for non-worker verticles. For any verticle, we can specify the number of instances at deployment time. However, having too few instances of a non-worker verticle is usually not a bottleneck. Since almost all Vert.x code is asynchronous, a verticle typically spends most of its time doing nothing. Instead, it is usually waiting for at network request, a response from the database, incoming data from the file system, or some other IO operation. If a verticle is not spending most of its time doing one of those things, chances are that it should really be a worker verticle.

When you do need to scale up the number of verticles, you can do so when deploying. For example, we could deploy the following instances in our deployment verticle:

- Two web servers
- Two mongo persistors
- Three mind map verticles
- Three mind map editors
- Ten `svg2png` workers

The following listing shows what this would look like in `app.js`:

```
var container = require("vertx/container");

container.deployModule("io.vertx~mod-web-server~2.0.0-final", {
  port: parseInt(container.env.get('MINDMAPS_PORT')) || 8080,

  bridge: true,
  inbound_permitted: [
    { address: 'mindMaps.list' },
    { address: 'mindMaps.save' },
    { address: 'mindMaps.delete' },
    { address_re: 'mindMaps\\.editor\\..+' },
    { address: 'com.vertxbook.svg2png' }
  ],
  outbound_permitted: [
    { address_re: 'mindMaps\\.events\\..+' }
  ]
}, 2);
container.deployModule("io.vertx~mod-mongo-persistor~2.0.0-final", {
  address: "mindMaps.persistor",
  db_name: "mind_maps"
}, 2);

container.deployVerticle('mindmaps.js', null, 3);
container.deployVerticle('mindmap_editor.js', null, 3);
container.deployModule('com.vertxbook~mod-svg2png~1.0', null, 10);
```

You may wonder what happens when you have multiple instances of a web server or an event handler deployed. How are the requests and events distributed? In these cases, Vert.x uses an internal load balancer to distribute requests and events in a round-robin fashion. In the case of web servers (and other network servers), only one instance of the actual server is instantiated, and load balancing is done for your code, which is the handlers for the incoming requests and events.

Memory

If you have a large number of verticles instantiated, or you are processing large in-memory data structures within them, you may sooner or later run into memory limitations with the underlying Java Virtual Machine. You can tweak the JVM memory settings and any other JVM runtime configuration settings as you would with a plain Java application, by providing the JVM with command-line arguments. (See `http://docs.oracle.com/javase/7/docs/technotes/tools/windows/java.html` for more details.)

At launch, the `vertx` command takes whatever is currently in the `JAVA_OPTS` environment variable and applies that as command-line arguments to Java. For example, if we would like to expand the minimum heap size to 512 Megabytes, and the maximum heap size to 2 Gigabytes, we can launch Vert.x with the environment variable using the following command:

```
JAVA_OPTS="-Xms512M -Xmx2G" vertx run app.js
```

In the production deployment scenario we set up earlier in this chapter, you can provide these settings by adding them to the environment variable; we already have in the upstart configuration file `/etc/init/mindmaps.conf`, using the following commands:

```
start on runlevel [2345]
stop on runlevel [016]

setuid mindmaps
setgid mindmaps

env JAVA_OPTS="-Djava.net.preferIPv4Stack=true -Xms512M -Xmx2G"
env MINDMAPS_PORT=80

chdir /home/mindmaps/app
exec authbind /home/mindmaps/vert.x-2.0.1-final/bin/vertx run app.js
```

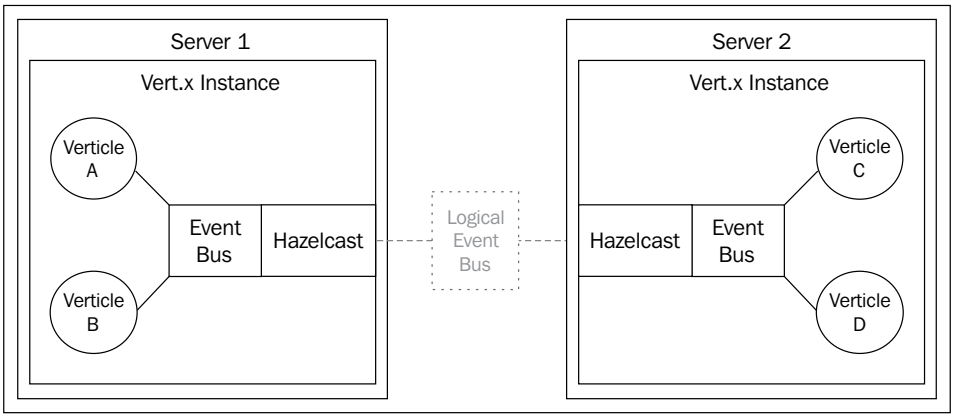
Clustering Vert.x

Vert.x is able to run in a clustered environment, allowing your systems to be distributed on multiple servers. This is achieved by connecting independent Vert.x instances together using a common logical event bus. For example, you can have verticles A and B running on **Server 1**, and verticles C and D running on **Server 2**, and have all the verticles collaborate over the event bus as if they were all in the same Vert.x instance. The cluster can grow and shrink dynamically. Verticles and Vert.x instances can join and leave the cluster during its lifetime.

The logical event bus is formed by connecting multiple Vert.x instances together using a **Hazelcast** data grid.

Hazelcast (`http://www.hazelcast.com/`) is an open source clustering tool for Java. It includes features such as distributed maps, sets, queues, dynamic discovery, and dynamic scaling. Vert.x ships with Hazelcast and there are no additional software installations needed to enable it.

When running in clustered mode, each Vert.x instance is also running a Hazelcast instance. A Vert.x cluster is formed when those Hazelcast instances are connected. Events over the Vert.x event bus are distributed over Hazelcast queues.



Creating a full-blown Vert.x cluster is outside the scope of this book, but let's go through a simple example that shows how it works. We are going to move the `svg2png` module out of our application to another Vert.x instance. This is a feasible scenario, since it is quite heavy compared to the rest of our code, and it could also be used by other applications in addition to our mind map application.

In this example, both Vert.x instances are going to be running on the same machine but there's nothing in Vert.x clustering or Hazelcast that requires it. That would defeat the whole point of clustering.

Find the file `conf/cluster.xml` in the Vert.x installation directory (on your local machine, not the virtual machine). Find the `<join>` element in the file and make the following changes to it:

```
<join>
  <multicast enabled="false">
    <multicast-group>224.2.2.3</multicast-group>
    <multicast-port>54327</multicast-port>
  </multicast>
  <tcp-ip enabled="true">
    <interface>127.0.0.1</interface>
  </tcp-ip>
  <aws enabled="false">
    <access-key>my-access-key</access-key>
    <secret-key>my-secret-key</secret-key>
    <region>us-east-1</region>
  </aws>
</join>
```

We have disabled the multicast discovery mode so that we don't need to deal with multicasting configuration for the purposes of this example. Instead, we have enabled the TCP-IP based discovery and changed the network interface it to `127.0.0.1` (localhost). This means Hazelcast peers will look for each other on the localhost network interface.

Find the file `conf/logging.properties` in the Vert.x installation. This file configures the Vert.x logger. Find the line in the file that sets the configuration level for `com.hazelcast`, and change it to the following:

```
com.hazelcast.level=INFO
```

This change is made so that we can see what Vert.x is doing when clustering mode is enabled. By default, it only logs about error situations.

Go to the application root directory, and start just the `svg2png` module with the cluster mode enabled:

```
vertx runmod com.vertxbook~mod-svg2png~1.0 -cluster
```

This will print out some clustering information, and then stay running quietly. The module is now running, but not yet communicating with the outside world because there's nothing else connected to the Vert.x instance's event bus.

Remove or comment out the line that deploys the `svg2png` module in `app.js`, as shown in the following line:

```
//container.deployModule('com.vertxbook~mod-svg2png~1.0', null,
  10);
```

The idea is to not run this module with our normal application. We want to connect to the module running in the other Vert.x instance instead. Now, start the application with cluster mode enabled using the following command:

```
vertx run app.js -cluster
```

This will also output some clustering information, including the fact that there are now two members in the cluster.

As you test the application, it should work just like before. The only difference is that under the hood, the SVG to PNG conversion is happening in another Vert.x instance and JVM, separated from the rest of the application. You can easily see how this model could be extended to distribute application components across many different servers.

Summary

Our application is finished, and it has been deployed for everyone to use! This concludes the basic real-time web application development workflow with Vert.x. In this chapter, you have learned the following things:

- How to set up a Linux server for Vert.x production deployment
- How to set up deployment for a Vert.x application using rsync
- How to start and supervise a Vert.x process using upstart
- How to scale the amount of concurrent processing in a Vert.x application by tweaking verticle counts
- How to provide additional memory to a Vert.x application and use other JVM command line arguments
- How to scale Vert.x up to multiple servers using clustering

Index

E

F

G

H

I

J

M

N

P

R

S

T

U

V

W

Thank you for buying

Real-time Web Application Development using Vert.x 2.0

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING

Ext JS 4 Web Application Development Cookbook

ISBN: 978-1-84951-686-0 Paperback: 488 pages

Over 110 easy-to-follow recipes backed up with real-life examples, walking you through basic Ext JS features to advanced application design using Sencha's Ext JS

1. Learn how to build Rich Internet Applications with the latest version of the Ext JS framework in a cookbook style
2. From creating forms to theming your interface, you will learn the building blocks for developing the perfect web application
3. Easy to follow recipes step through practical and detailed examples which are all fully backed up with code, illustrations, and tips

---

HTML5 Web Application Development By Example

ISBN: 978-1-84969-594-7 Paperback: 276 pages

Learn how to build rich, interactive web applications from the ground up using HTML5, CS3, and jQuery

1. Packed with example applications that show you how to create rich, interactive applications and games.
2. Shows you how to use the most popular and widely supported features of HTML5.
3. Full of tips and tricks for writing more efficient and robust code while avoiding some of the pitfalls inherent to JavaScript.

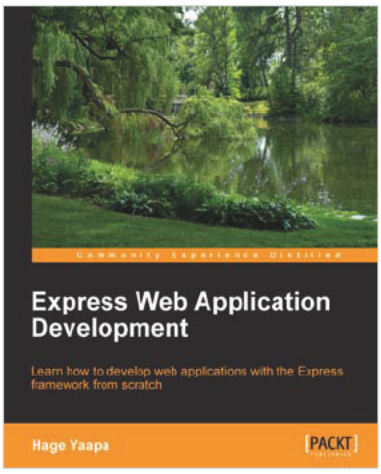
[PACKT] open source
community experience distilled
PUBLISHING

HTML5 iPhone Web Application Development

ISBN: 978-1-84969-102-4 Paperback: 338 pages

An introduction to web-application development for mobile within the iOS Safari browser

1. Simple and complex problems will be covered with examples and resources that backup the approach and technique.
2. Real world solutions that are broken down for multiple target audiences; from beginner developers to technical architects.
3. Learn to build true web applications using the latest industry standards for iOS Safari.



Express Web Application Development

ISBN: 978-1-84969-654-8 Paperback: 236 pages

Learn how to develope web applications with the Express framework from scratch

1. Exploring all aspects of web development using the Express framework
2. Starts with the essentials
3. Expert tips and advice covering all Express topics

Please check **www.PacktPub.com** for information on our titles