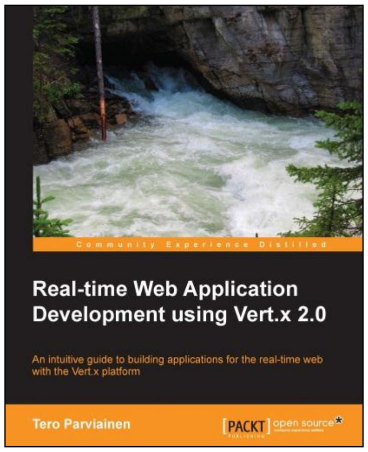
Real-time Web Application Development using Vert.x 2.0

Tero Parviainen



Chapter No. 2
"Developing a Vert.x Web Application"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.2 "Developing a Vert.x Web Application"

A synopsis of the book's content

Information on where to buy this book

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.

Real-time Web Application Development using Vert.x 2.0

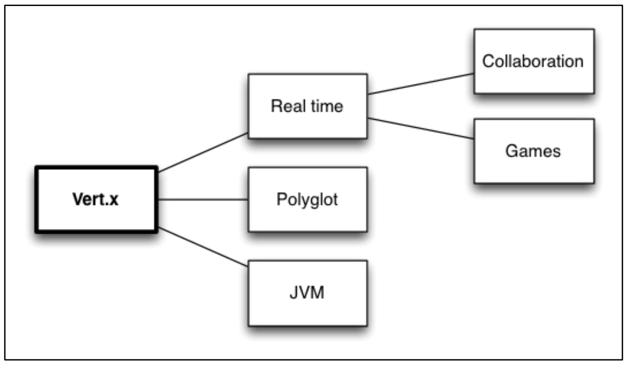
The real-time web is here. We've come a long way from the collection of linked, static pages that started it all; from the very first web applications in the 1990s, through the Ajax revolution about seven years ago, to the latest crop of highly dynamic web properties.

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 Vertex?

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.

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:

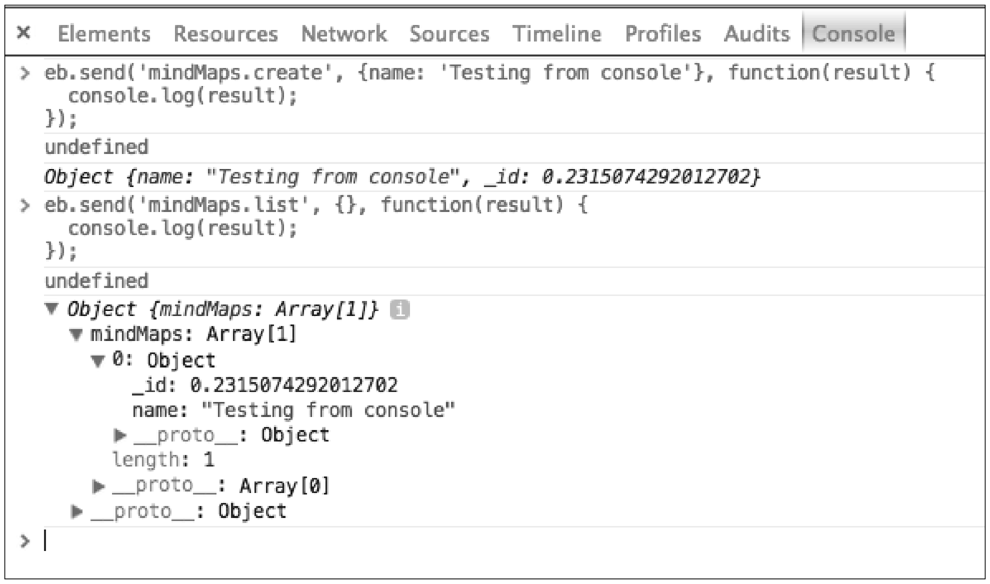
```
eb.send('mindMaps.save', {name: 'Testing from console'},
  function(result) {
  console.log(result);
});
```

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.



```
> eb.send('mindMaps.create', {name: 'Testing from console'}, function(result) {
    console.log(result);
  });
  undefined
  Object {name: "Testing from console", _id: 0.2315074292012702}
> eb.send('mindMaps.list', {}, function(result) {
    console.log(result);
  });
  undefined
  ▼ Object {mindMaps: Array[1]}
    ▼ mindMaps: Array[1]
      ▼ 0: Object
          _id: 0.2315074292012702
          name: "Testing from console"
        ▶ __proto__: Object
        length: 1
      ▶ __proto__: Array[0]
    ▶ __proto__: Object
> |
```

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.

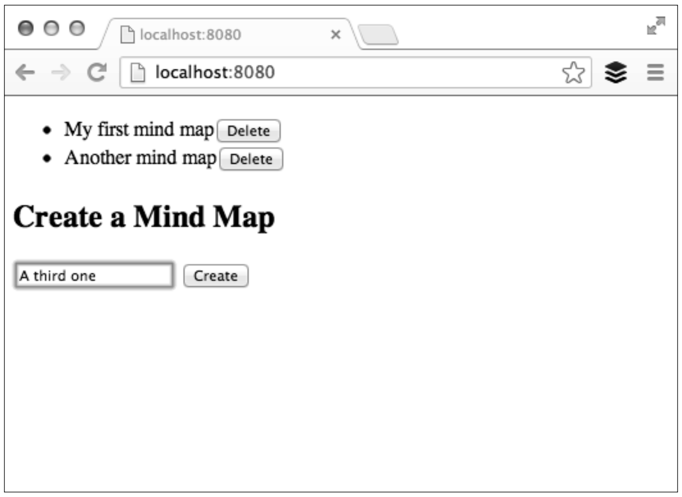
Add the highlighted code to `web/client.js` in order to handle this operation:

```
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:

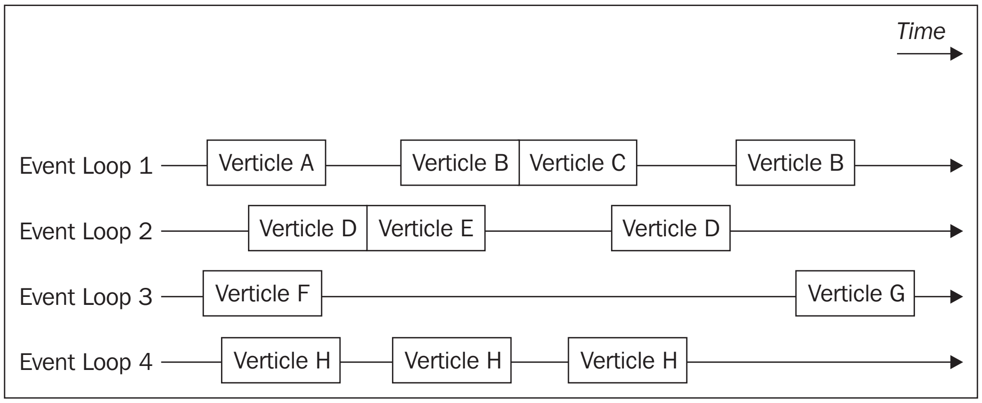
1. Each verticle is assigned its own Java classloader, which means it is impossible for verticles to access each other's state, even if it was defined in static Java variables or by other programming mechanisms that usually allow for global state to be shared.

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

Where to buy this book