Single Page Application development with React and Vert.x

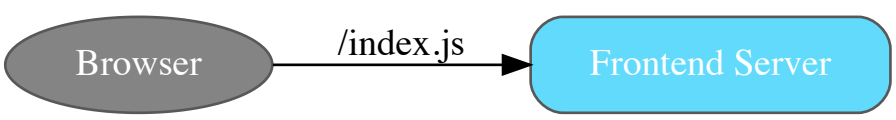
This document will show you how to develop a Single Page Application (SPA) with React and Vert.x.

What you will build

You will create a React frontend communicating over HTTP with a Vert.x backend.

Development workflow

When developing an SPA, it is very convenient to get instant feedback after updating a Javascript, HTML or CSS file. With React, this requires you start a development server with `npm` that handles requests for frontend resources:



But what about requests that must be processed by Vert.x? You will configure the project so that the frontend development server proxies API requests to the backend:

What you need

- A text editor or IDE
- Java 8 or higher
- Maven or Gradle
- Node
- `npm`
- `npx`, an NPM package runner

Create a project

The code of this project contains Maven and Gradle build files that are functionally equivalent.

Using Maven

Add the `vertx-web` dependency in you Maven POM file:

Maven `pom.xml`

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

Then add the `exec-maven-plugin`:

Maven `pom.xml`

```
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>exec-maven-plugin</artifactId>
      <version>1.5.0</version>
      <configuration>
        <mainClass>io.vertx.howtos.react.BackendVerticle</mainClass>
      </configuration>
    </plugin>
```

Using Gradle

Assuming you use Gradle with the Kotlin DSL, add the `vertx-web` dependency:

Gradle `build.gradle.kts`

```
dependencies {
  val vertxVersion = "3.7.0"
  implementation("io.vertx:vertx-web:${vertxVersion}")
}
```

Then configure the application main class:

Gradle `build.gradle.kts`

```
application {
  mainClassName = "io.vertx.howtos.react.BackendVerticle"
}
```

Expose a message service over HTTP

Let's start with the backend service. It shall handle requests on the `/api/message` path by retuning a greeting message:

Java `src/main/java/io/vertx/howtos/react/BackendVerticle.java`

```
Router router = Router.router(vertx);
Route messageRoute = router.get("/api/message"); // (1)
messageRoute.handler(rc -> {
  rc.response().end("Hello React from Vert.x!"); // (2)
});

router.get().handler(StaticHandler.create()); // (3)

vertx.createHttpServer()
  .requestHandler(router)
  .listen(8080);
```

1. A Vert.x Web `Route` is defined to match HTTP requests on the `/api/message` path
2. The `Route` handler replies to requests with a greeting message
3. The `StaticHandler` is required to handle requests for static resources

IMPORTANT / You may wonder why we need a `StaticHandler` if the frontend development server handles static resources? Keep in mind that when the whole application is built and put to production, the frontend will be bundled with and served by the backend HTTP server.

Before we can test the implementation, the `BackendVerticle` needs a `main` method:

Java `src/main/java/io/vertx/howtos/react/BackendVerticle.java`

```
public static void main(String[] args) {
  Vertx vertx = Vertx.vertx(); // (1)
  vertx.deployVerticle(new BackendVerticle()); // (2)
}
```

1. Create a `Vertx` context
2. Deploy `BackendVerticle`

You can run the application:

- straight from your IDE or,
- with Maven: `mvn compile exec:java`, or
- with Gradle: `./gradlew run` (Linux, macOS) or `gradlew run` (Windows).

> NOTE / The following example uses the HTTPie command line HTTP client. Please refer to the installation documentation if you don't have it installed on your system yet.

To receive a greeting, open your terminal and execute this:

```
http :8080/api/message
```

You should see:

```
HTTP/1.1 200 OK
content-length: 24

Hello React from Vert.x!
```

Displaying the message in the browser

We have a fully operational backend, we can create the frontend. To do so, run the `create-react-app` package with `npx`:

```
cd src/main
npx create-react-app frontend
```

This will:

- create a `package.json` file that defines dependencies as well as build and run scripts
- install the dependencies
- generate a skeleton application

> NOTE / In this how-to, the frontend code lives as part of the backend project, inside the `src` directory. This is easier to get started, in particular if your team includes more backend-oriented developers. However, as you project grows, you might prefer to split the frontend and the backend into separate modules.

The skeleton application is not of great interest here so let's remove it:

```
rm -rf frontend/src/*
```

Then open your favorite editor and implement the React frontend:

Javascript `src/main/frontend/src/index.js`

```
import React from 'react';
import ReactDOM from 'react-dom';

class Greeter extends React.Component { // (1)

  constructor(props) {
    super(props);
    this.state = { // (2)
      message: "Default message"
    }
  }

  componentDidMount() { // (5)
    fetch("/api/message")
      .then(response => response.text())
      .then(text => this.setState({message: text}));
  }

  render() { // (3)
    return (
      <div>
        <span>{this.state.message}</span>
      </div>
    );
  }
}

ReactDOM.render( // (4)
  <Greeter/>,
  document.getElementById('root')
);
```

1. Our frontend consists in a single React `Greeter` component
2. The `Greeter` component holds state, which is a message to display in the browser
3. The message is displayed within a simple HTML `span`
4. The `Greeter` component is rendered in the web page
5. After initial rendering, an HTTP request is sent to the backend; the result is used to udpate the component state

Last but not least, you must configure the frontend development server to proxy API requests to the backend. Open the `package.json` file in you editor and add:

Javascript `src/main/frontend/package.json`

```
"proxy": "http://localhost:8080"
```

That's it, open a terminal and start the frontend development server:

```
cd src/main/frontend
npm start
```

A browser tab should automatically be opened and pointing to http://localhost:3000.

You should see:

```
Hello React from Vert.x!
```

Putting it all together

In production of course you will not start a frontend development server. So the Maven POM (or Gradle build) file shall be configured to:

- run a frontend build
- copy the static files to the `src/main/resources/webroot` folder

> **NOTE** Do you remember the `StaticHandler` from the first section? It looks for static files in the `webroot` folder by default. This is why you must copy static files to `src/main/resources/webroot`.

Add these plugins to your Maven POM file:

Maven `pom.xml`

```
<plugin>
  <groupId>com.github.eirslett</groupId>
  <artifactId>frontend-maven-plugin</artifactId>
  <version>1.7.6</version>
  <configuration>
    <nodeVersion>v10.15.3</nodeVersion>
    <npmVersion>6.4.1</npmVersion>
    <workingDirectory>src/main/frontend</workingDirectory>
    <installDirectory>target</installDirectory>
  </configuration>
  <executions>
    <execution>
      <id>install-node-and-npm</id>
      <phase>generate-resources</phase>
      <goals>
        <goal>install-node-and-npm</goal> (1)
      </goals>
    </execution>
    <execution>
      <phase>generate-resources</phase>
      <id>npm-install</id>
      <goals>
        <goal>npm</goal>
      </goals>
      <configuration>
        <arguments>install</arguments> (2)
      </configuration>
    </execution>
    <execution>
      <id>npm-run-build</id>
      <phase>generate-resources</phase>
      <goals>
        <goal>npm</goal> (3)
      </goals>
      <configuration>
        <arguments>run build</arguments>
      </configuration>
    </execution>
  </executions>
</plugin>
<plugin>
  <artifactId>maven-resources-plugin</artifactId>
  <version>3.1.0</version>
  <executions>
    <execution>
      <id>copy-to-webroot</id>
      <phase>process-resources</phase>
      <goals>
        <goal>copy-resources</goal> (4)
      </goals>
      <configuration>
        <outputDirectory>${project.build.outputDirectory}/webroot</outputDirectory>
        <resources>
          <resource>
            <directory>${basedir}/src/main/frontend/build</directory>
            <filtering>false</filtering>
          </resource>
        </resources>
      </configuration>
    </execution>
  </executions>
</plugin>
```

1. Downloading Node and `npm` in the build directory allows to run the frontend build on CI where they might not be present
2. Download frontend dependencies with `npm install`
3. Create a production-ready build of the frontend

4. Copy static files to `src/main/resources/webroot`

If you use Gradle, first add the Gradle NPM plugin:

Gradle `build.gradle.kts`

```
import com.moowork.gradle.node.npm.NpmTask

plugins {
  java
  application
  id("com.moowork.node") version "1.3.1"
}
```

Then the configuration is similar to what we did with Maven:

Gradle `build.gradle.kts`

```
node {
  version = "10.15.3"
  npmVersion = "6.4.1"
  download = true
  nodeModulesDir = File("src/main/frontend")
}

val buildFrontend by tasks.creating(NpmTask::class) {
  setArgs(listOf("run", "build"))
  dependsOn("npmInstall")
}

val copyToWebRoot by tasks.creating(Copy::class) {
  from("src/main/frontend/build")
  destinationDir = File("${buildDir}/classes/java/main/webroot")
  dependsOn(buildFrontend)
}

val processResources by tasks.getting(ProcessResources::class) {
  dependsOn(copyToWebRoot)
}
```

Make sure all previous `npm`, `mvn` or `gradlew` executions are terminated and start the Vert.x server:

- with Maven: `mvn compile exec:java`, or
- with Gradle: `./gradlew run` (Linux, macOS) or `gradlew run` (Windows).

Browse to http://localhost:8080 and you should see:

```
Hello React from Vert.x!
```

Summary

This document covered:

1. the creation of a new React application with `create-react-app`
2. running a frontend development server (for live-reload) that delegates API requests to the Vert.x backend
3. bundling the frontend static files together with the Vert.x classes when going to production

See also

- Create React App
- The Vert.x Web documentation

Last published: 2019-08-13 15:06:50 +0000.