Denis Kalinin

Practical Event Sourcing with Scala

Practical Event Sourcing with Scala

A comprehensive example based on Scala, Play and Akka Streams

Denis Kalinin

This book is for sale at http://leanpub.com/practical-event-sourcing-with-scala

This version was published on 2021-01-25

Contents

Preface

Event sourcing is an architectural pattern that implies recording the state of the system as a series of events. This approach is somewhat different from a *traditional* approach, which only maintains the current state of the system.

The event sourcing pattern is known to have many advantages. Since the event store already contains all events ever happened, we get a reliable auditing mechanism for free. Moreover, these events can also be used for analytics to get more insight into how the system behaved at some particular moment in the past.

However, implementing the system based on event sourcing is considered to be quite a challenging task. Part of the reason is that event-based approach is not very common, and best practices are not well-known. The developers implementing such a system are usually on their own as seemingly ubiquitous Internet resources are usually limited to only covering basics.

Another interesting thing about event sourcing is that most people seem to associate it with the Enterprise world. Things like domain-driven design (DDD), command-query responsibility separation (CQRS) often go hand in hand. At the same time, typical Web startups usually start with a more traditional CRUD approach (Create-Read-Update-Delete) but often switch to event-based architecture later.

This book takes a rather unusual look at event sourcing. Instead of emphasizing domain modeling, we will concentrate on technological aspects of building an event-based Web application. Our question-and-answer service, although relatively minimalistic, will nevertheless contain many attributes of a typical social-networking site like *tagging* and *upvoting*.

We will be using Scala as the main backend language throughout the book. I believe that this a very good choice, because it makes a good compromise between heavy-weight languages like C# or Java and overly concise languages like Ruby or Python. You don't need to be a Scala guru, but if you want to benefit most from reading the book, you need to at least be familiar with the syntax. If you're coming from another language, I suggest you read through my 40-page Scala tutorial released as a free sample for "Modern Web Development with Scala"[1].

Another prerequisite is familiarity with Web technologies. We're not going to spend time discussing HTTP methods and basic syntax of JavaScript, so readers should already know what they are for. However, will discuss in some details different JavaScript libraries as we start introducing them into our project. Since the project contains quite a lot of code, this book is accompanied with a code repository. All repository code is split into chapters so that the reader will be able to easily follow what we achieved at some particular point.

[1] https://leanpub.com/modern-web-development-with-scala

Chapter 1. The initial project

In this chapter, we will prepare the environment and initialize the project which will serve as a starting point of our application.

Preparing the environment

Since Scala is a JVM language, you need to have a Java Development Kit (JDK) installed on your machine. Some of the libraries that we will be using throughout the book require Java 8, so make sure to install JDK 8.

There are many ways to obtain JDK, and all of them are relatively straightforward. If you're running a Linux distribution, then there's a good chance that it has a prebuilt OpenJDK package in its repository. For example, on Ubuntu you can get the latest update for OpenJDK 8 using the following command:

```
$ sudo apt-get install openjdk-8-jdk
```

You can also download a prebuilt OpenJDK package from AdoptOpenJDK and then install it manually:

- Go to https://adoptopenjdk.net/releases.html[2] and download the latest JDK 8 (look for "OpenJDK 8 with HotSpot" and make sure that the full name looks something like "jdk8u172-b11", which means "version 8, update 172"). For Linux you'll probably end up getting a `tar.gz` file. Extract the contents of the archive anywhere (for example, to `~/DevTools/java8`);
- On Windows, just follow the instructions of the installer;
- Create an environment variable `JAVA_HOME` and point it to the JDK directory;
- Add the `bin` directory of JDK to your `PATH` so that the `java` command works from any directory on your machine;

On Linux, you can accomplish both goals if you add to your `~/.bashrc` the following:

```
JAVA_HOME=/home/user/DevTools/jdk8u172-b11
PATH=$JAVA_HOME/bin:$PATH
```

[2] https://adoptopenjdk.net/releases.html

Another option is to add the above lines to the `~/.profile` file. The use of these files usually differs in various distributions, but in Ubuntu `~/.profile` is a good place to put environment variables if you want to make them available to GUI programs.

For our application, we will also build a frontend workflow based on NodeJS. This means that you'll need to have two NodeJS tools installed - `node` (the NodeJS executable) and `npm` (the NodeJS package Manager). Since NodeJS is actively developed, I usually recommend installing `nvm` - the NodeJS version manager. This tool will allow you to switch between different versions of NodeJS without reinstalling it system-wide. In order to install it, just go to their website[3] and follow the instructions.

After `nvm` is installed, use it to obtain NodeJS itself:

```
$ nvm install v14.15.1
```

Add the following line to the very end of your `.bashrc` to make sure that NodeJS v14.15.1 is used by default:

```
nvm use v14.15.1 > /dev/null
```

Open a new terminal and check that `npm` is also available:

```
$ npm --version
6.14.9
```

Finally, we will be running all supporting services such as databases or message brokers as Docker[4] images. This simplifies everything tremendously, but you'll need to install both Docker and Docker Compose. The installation of Docker differs significantly from one operating system to another, but on Ubuntu it usually boils down to adding the official Docker repository and then getting the binaries from there:

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
$ sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
$ sudo apt-get update
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
```

On Windows or Mac, Docker comes with a sophisticated installer that takes care of the entire process.

[3] https://github.com/nvm-sh/nvm
[4] https://www.docker.com/

Initializing the project

Right now, `sbt` is the most popular Scala build tool. It is extremely powerful, and we will be using it throughout the book.

There are several ways to obtain `sbt`. For example, you can download the `zip` or `tar.gz` archive from the official Web-site[5]. After extracting the contents somewhere (for example, to `~/DevTools/sbt`), add the `sbt` executable to your `PATH`.

As always, you can add the `SBT_HOME` environment variable by editing the `.bashrc` file:

```
1 SBT_HOME=/home/user/DevTools/sbt
2 PATH=$JAVA_HOME/bin:$SBT_HOME/bin:$PATH
```

Starting with version 0.13.13, `sbt` is capable of initializing projects from Giter8[6] templates, which are little more than GitHub repositories with some metadata. I prepared such a template specifically for this book so that you will be able to easily create a new project without writing any boilerplate. In order to initialize a new project in a new directory called `event-sourcing-app`, type the following command:

```
$ sbt new denisftw/play-event-sourcing-starter.g8
```

`sbt` will prompt you to type in the project name. Type `event-sourcing-app` and `sbt` will create a directory with this name and initialize a new project there.

If you navigate to a newly created directory, you will see that there are a lot of files. We will discuss the project structure in details in the next section, but now let's check that everything actually works.

First, download the frontend dependencies by invoking the following command:

```
$ npm i
```

The `i` option here stands for `install`, so `npm` will look at the dependencies sections in the `package.json` file and install all necessary packages into the `node_modules` subdirectory. It may take some time, but after it's done, you will be able to compile frontend assets by typing the following:

```
$ npm run watch
```

In another (new) terminal window, run `bash run-sbt.sh` from the project directory:

[5] http://www.scala-sbt.org/download.html
[6] https://github.com/foundweekends/giter8

```
$ bash run-sbt.sh
Listening for transport dt_socket at address: 9999
[play-event-sourcing-starter] $
```

This will start `sbt` in debug mode so that it will be possible to connect your IDE to the running application and check what's going on.

In order to start the Web server, simply type `run` and press Enter:

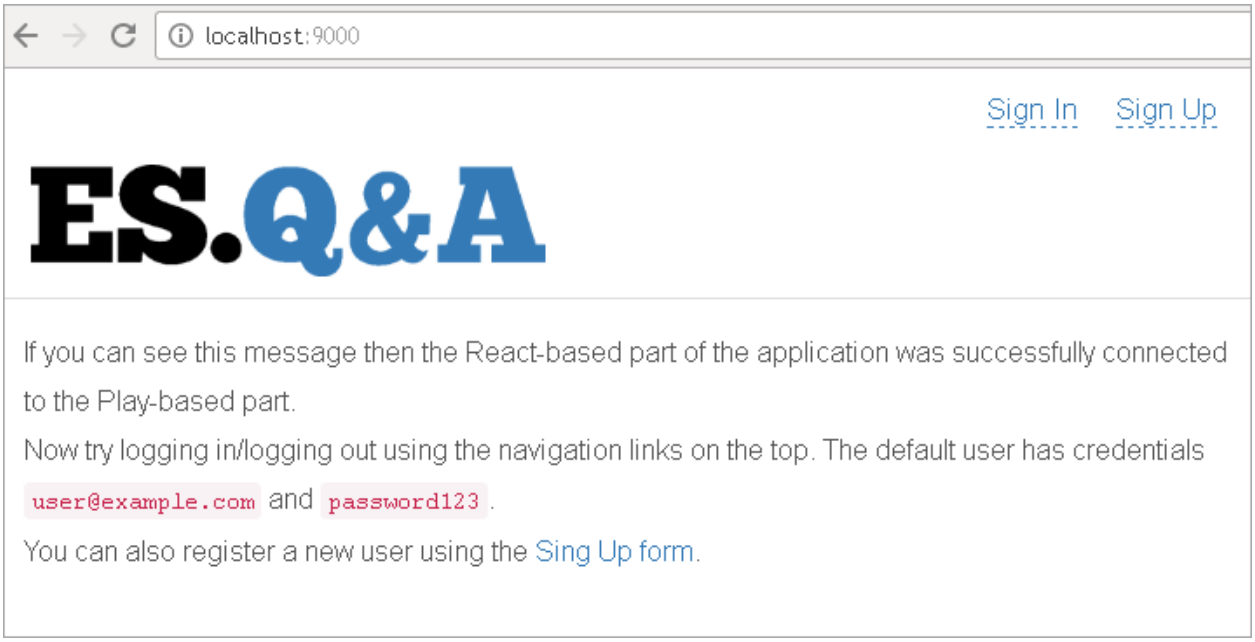
```
[play-event-sourcing-starter] $ run

--- (Running the application, auto-reloading is enabled) ---

[info] p.c.s.AkkaHttpServer - Listening for HTTP on /0:0:0:0:0:0:0:0:9000

(Server started, use Ctrl+D to stop and go back to the console...)
```

Finally, open yet another terminal tab, go to the `stack` directory and start the accompanying crew of services by typing the following:

```
$ cd stack/
$ docker-compose up
```

You will see a lot of logs in different colors, which should serve as an indication that all services are up and running. Since this is not a book about Docker, all necessary services are included from the very beginning even though some won't be used until final chapters.

Finally, now you can open a new browser tab and navigate to `localhost:9000`. You should see the following page:



The Welcome page

If you see it, congratulations! If not, try to analyze the error message. If it says something about connection pool, it's very likely that the PostgreSQL server is not started. There could be a problem with Docker, and it's usually easy to understand what went wrong by analyzing the logs.

When you resolve the problem, try running the server again. First, press Ctrl+D to stop the current task and return to the interactive shell. Second, type `run` and press Enter one more time.

The backend part

Now that we have a working application, let's stop for a moment and take a look at its internals. The backend part of our app is based on the Play framework. This framework is quite similar to other full-stack frameworks such as Django and Rails, so if you're familiar with them, you will probably notice a lot of similarities.

Here are the main parts:

part	description
`build.sbt`	contains typical `sbt` stuff like library dependencies, project name, versions etc
`public`	contains static assets like images and compiled frontend assets like minified scripts and style sheets
`project/build.properties`	specifies the `sbt` version
`project/plugins.sbt`	specifies necessary `sbt` plugins, including Play itself
`conf/application.conf`	contains configuration for the app, for example, database connection properties
`conf/logback.xml`	contains logging configuration

part	description
`routes`	specifies exposed routes as HTTP endpoints and maps them to Scala controller methods
`app`	contains Scala files

The Scala files are organized into packages, so it makes sense to discuss them here as well:

package	description
`controllers`	contains controllers
`dao`	contains low-level services that talk directly to the database
`model`	contains data types (mainly case classes) that are used by all other parts of the app
`security`	contains actions for user authentication
`services`	contains higher-level services
`util`	contains utility classes
`views`	contains Play templates

Controllers have methods directly tied to HTTP endpoints. If you look at the `conf/routes` file, you will see that every HTTP endpoint is always mapped to a specific controller method. For example:

```
1 GET     /login    controllers.AuthController.login()
2 POST    /login    controllers.AuthController.doLogin()
```

When Play receives a `GET` request on `/login`, it will serve this request using the `login` method of the `AuthController`:

```
1 // class AuthController
2 def login = Action { request =>
3   Ok(views.html.security.login(None))
4 }
```

Roughly speaking, a Play action is a function that accepts a `Request` and returns a `Result`. Wrapping the payload in `Ok` will send a status code of `200`, but there are other constants available as well. The `views.html.security.login` part corresponds to the `login.scala.html` template that resides in the `views/security` directory. It doesn't make much sense to show an entire template, but here is one interesting part:

```
@(maybeMessage: Option[String])

@views.html.security.authMain(maybeMessage) {
    @maybeMessage.map { message =>
        <div class="login-page__message-panel">
            <div class="alert alert-danger" role="alert">@message</div>
        </div>
    }
}
```

The first line declares that this template takes one argument of type `Option[String]`. More generally, everything starting with the `@` character signifies a Scala expression. In particular, the `@views.html.security.authMain(maybeMessage)` expression references the `authMain` template. The `maybeMessage` has a type of `Option[String]`, and there is no reason why calling the `map` shouldn't work here, so it does. The same goes for `@message`, which simply replaces this expression with the text content.

The DAO services in our application are implemented using ScalikeJDBC[7]. This library allows users to write type-safe SQL queries easily. Here is the typical usage from the `UserDao`:

```
// class UserDao
import scalikejdbc._
import util.ThreadPools.IO
def getUsers: Future[Seq[User]] = Future {
  NamedDB(Symbol("auth")).readOnly { implicit session =>
    sql"select * from users".map(User.fromRS).list().apply()
  }
}
```

The `NamedDB` specifies the name of the database we want to use. The `readOnly` method opens a connection in the read-only mode. Inside the block, we can write arbitrary queries and turn them into `SQL` objects by using the `sql` interpolation. The `map` method can be used to convert generic result sets to domain objects. Finally, the `list` method instructs ScalikeJDBC to retrieve a list of objects, and `apply` actually executes the query. Note that the `apply` method needs a `DBSession` defined in the implicit scope, so if you forget to mark `session` as `implicit`, code will not compile.

One important thing to note here is that our `getUsers` method returns a `Future`. Every time we use `Future.apply`, we need to specify an execution context. That execution context will be used to execute the code within curly braces, which is this case means running a SQL query. Since executing the SQL query is a blocking IO operation, it's prudent to run it in a so-called blocking thread pool.

In fact, there are only two thread pools that we will be using throughout the book, and both are defined in the `ThreadPools` utility object:

[7] http://scalikejdbc.org

```
object ThreadPools {
  implicit val IO = ExecutionContext.
    fromExecutor(Executors.newCachedThreadPool())
  implicit val CPU = ExecutionContext.Implicits.global
}
```

A good rule of thumb is to use `IO` for everything that blocks or is not directly dependent on the computing power of the central processor. Therefore, we'll stick to `IO` whenever we're writing code for DAO-classes that deal with the database. On the other hand, most higher level services rely primarily on the CPU, so for them it's better to stick to the default execution context.

Getting back to the database example, the `User` class from the previous code snippet is actually quite simple:

```
case class User(userId: UUID, userCode: String,
  fullName: String, password: String, isAdmin: Boolean)
```

There is also a companion object containing the `fromRS` method for mapping result set rows to domain entities:

```
object User {
  def fromRS(rs: WrappedResultSet): User = {
    User(UUID.fromString(rs.string("user_id")),
      rs.string("user_code"), rs.string("full_name"),
      rs.string("password"), rs.boolean("is_admin"))
  }
}
```

As it was already stated previously, the `services` package contains higher-level services that don't typically interact with the database directly. Instead, they delegate low-level work to lower-level DAO classes:

```
class UserService(userDao: UserDao) {
  import util.ThreadPools.CPU
  def getUserFullName(userId: UUID): Future[Map[UUID, String]] = {
    userDao.getUsers.map { users =>
      users.map { user =>
        user.userId -> user.fullName
      }.toMap
    }
  }
}
```

Here, the `UserService` accepts a `UserDao` as a constructor parameter and then uses it within its methods. As we will see later, this forms the basis of our dependency injection mechanism. Note also that as a higher-level service `UserService` uses a CPU-bound thread pool for its operations.

The `security` package contains two action builders - `UserAuthAction` and `UserAwareAction`. These action builders can be used in controllers instead of regular `Actions` we've seen earlier. The detailed explanation of the Play API is definitely beyond the scope of this book, but here is the basic idea.

We pass an instance of the `UserAuthAction` to a controller as a constructor parameter. Then, using this instance, we can create a new `Action` as usual:

```
1  // simplified
2  class AuthController(userAuthAction: UserAuthAction)
3    extends AbstractController {
4    def restricted = userAuthAction { request =>
5      val user = request.user
6      // ...
7    }
8  }
```

The `UserAuthAction` makes sure that only authenticated users can proceed; everyone else is sent to the login page. Within the curly braces we have access to a request wrapper that has a type of `UserAuthRequest` defined as follows:

```
1  case class UserAuthRequest[A](user: User,
2      request: Request[A]) extends  WrappedRequest[A](request)
```

The most important part of this class is the `user` field. Since the `UserAuthAction` restricts certain endpoints to only authenticated users, we can always get a `User` object back if we need. The `UserAwareRequest`, which works together with the `UserAwareAction`, is similar, but it doesn't redirect unauthorized users to the Login page:

```
1  case class UserAwareRequest[A](user: Option[User],
2      request: Request[A]) extends WrappedRequest[A](request)
```

In this case, user information may be not available, therefore the type of the `user` field is `Option[User]`.

Controllers and services take other services as their constructor parameters. But how does it work in practice? The `conf/application.conf` file defines a number of configuration settings, in particular the following one:

```
play.application.loader = "AppLoader"
```

When the application starts, Play instantiates a class called `AppLoader` (it resides in the root package) and calls its `load` method. This, in turn, triggers the loading of the `AppComponents` class. `AppComponents` has the following structure:

```
class AppComponents(context: Context) extends
  BuiltInComponentsFromContext(context) with EvolutionsComponents
  with DBComponents with HikariCPComponents with AssetsComponents {
  // ...
  lazy val sessionDao = wire[SessionDao]
  lazy val userDao = wire[UserDao]
  lazy val userService = wire[UserService]
}
```

This class glues together pretty much every service that our Play application uses. Essentially, this process consists of two parts. First, we mix in built-in traits and by doing so gain access to standard Play services such as `Configuration` (gives access to the application settings in runtime) or `ConnectionPool` (which is based on HikariCP[8]). Then, inside the class we instantiate necessary services by invoking their constructors and passing necessary parameters to them. The reason we don't call the constructors directly is the presence of the `wire` macro that does it for us. At compile time, the above snippet will be transformed into something like this:

```
// trait AppComponents
lazy val sessionDao = new SessionDao
lazy val userDao = new UserDao
lazy val userService = new UserService(userDao)
```

The `wire` macro does this substitution based on parameter types, whereas the order of initialization is resolved by the Scala compiler itself.

Talking of the `wire` macro, it comes from a library called MacWire[9]. This library is not bundled with Play, so it's defined as a dependency in the `build.sbt` file. Let's take a closer look at this file. The first part is extremely straightforward:

```
name := """play-event-sourcing-starter"""
organization := "com.appliedscala.streaming"
version := "1.0-SNAPSHOT"
scalaVersion := "2.13.4"
```

It simply defines the name of the project (`artifactId` in Maven terms), the organization name (`groupId`) and the version. It also specifies the version of Scala.

The next part simply enables Play itself, because from SBT perspective, Play is just a plugin:

[8] https://github.com/brettwooldridge/HikariCP
[9] https://github.com/adamw/macwire

```
lazy val root = (project in file(".")).
  enablePlugins(PlayScala)
```

The next line is actually Play-specific:

```
pipelineStages := Seq(digest)
```

This setting essentially enables fingerprinting of static resources (works only in production mode), which makes resource caching easier.

Finally, the last part lists all application dependencies:

```
libraryDependencies ++= Seq(
  jdbc,
  evolutions,
  "com.softwaremill.macwire" %% "macros" % "2.3.3" % "provided",
  "org.postgresql" % "postgresql" % "42.2.18",
  "org.scalikejdbc" %% "scalikejdbc" % "3.5.0",
  "org.scalikejdbc" %% "scalikejdbc-config"  % "3.5.0",
  "ch.qos.logback"  %  "logback-classic" % "1.2.3",
  "de.svenkubiak" % "jBCrypt" % "0.4.1"
)
```

`jdbc` and `evolutions` are built-in Play modules. Everything else is a third-party dependency. All of them are summarized below:

dependency	description
MacWire	enables compile-time dependency injection via the `wire` macro
PostgreSQL	enables low-level (JDBC) access to PostgreSQL database
ScalikeJDBC	simplifies access to the database
Logback	a logging framework used by ScalikeJDBC and others
jBCrypt	cryptographic hash library (used for generating password hashes)

The application settings are set in the `application.conf` file. Let's take a look at this file starting with the first part:

```
play {
  http.secret.key = "changeme"
  i18n.langs = [ "en" ]
  application.loader = "AppLoader"
  evolutions.autoApply = true
}
```

The `application.conf` file uses a JSON-like format called HOCON[10] (Human-Optimized Config Object Notation), which is promoted by Lightbend. This format is quite popular in the Scala community, and it's often used in non-Play projects as well.

The first part of the configuration file specifies Play-related settings:

setting	description
`http.secret.key`	application-wide secret key (used internally by Play for things like encryption and fingerprinting)
`i18n.langs`	the list of supported languages (used by Play I18N mechanism)
`application.loader`	specifies a custom application loader
`evolutions.autoApply`	instructs Play to apply evolutions automatically when needed

Let's look at the second part of `application.conf`:

```
1  db {
2    auth {
3      driver=org.postgresql.Driver
4      url="jdbc:postgresql://localhost:5432/authdb"
5      username=scalauser
6      password=scalapass
7      poolInitialSize=1
8      poolMaxSize=5
9      ConnectionTimeoutMillis=1000
10   }
11 }
```

The database connection settings are mostly self-explanatory. Note that `auth` specifies the name of the database, so everywhere in the application, Play will refer to this database as "auth". We've already seen it in the `getUsers` method, but we also use this name when we define database evolutions.

Since our authentication database is called `auth`, we must put evolution scripts into the `conf/evolutions/auth` folder. The evolution scripts must be named `1.sql`, `2.sql`, `3.sql` etc. In our case, we only use evolutions for convenience purposes so that we don't have to execute SQL scripts manually. Therefore, we have only one script that creates two tables - `users` and `sessions` and inserts a default user named `Joe Average` with login `user@example.com` and password `password123`. We instructed Play to apply evolutions automatically, so when your browser loaded the home page for the first time, necessary tables had already been created and users inserted.

[10] https://github.com/lightbend/config

The frontend part

The frontend is mostly defined by two files - `package.json` and `webpack.config.js`. The `package.json` file contains lots of settings, but we're only interested in three specific sections. These sections are:

- `dependencies`
- `devDependencies`
- `scripts`

The `dependencies` section lists all frontend dependencies that the application has. For example, if your application uses React[11], you'll need to add React to this section. Developers rarely edit this section manually, but instead, they instruct `npm` to install the required package and save it as an application dependency at the same time. To do that, they simply type something like this:

```
$ npm i react -S
```

The `-S` option instructs `npm` to add the package as an *application* dependency. The initial template comes with the following application dependencies:

package	description
`axios`	simplifies sending HTTP requests
`react`	helps to develop UI using VirtualDOM
`react-dom`	integrates React with the DOM model
`bootstrap-scss`	makes the application look decent

In contrast, the `devDependencies` section lists all dependencies that your project uses during development. For example, if you want to use Webpack[12] to build your app bundle, you will need to add `webpack` to this section via the following command:

```
$ npm i webpack -D
```

Since the Giter8 template we used earlier comes with already defined frontend dependencies, you don't need to do any of that.

Finally, there is the `scripts` section that defines two scripts:

[11] https://reactjs.org/
[12] https://webpack.github.io

```
"scripts": {
  "watch": "webpack --mode development --watch",
  "prod": "webpack --mode production"
}
```

So, when you type `npm run watch`, it will start Webpack in the monitor mode. If you type `npm run prod`, it will compile the frontend assets and minimize them for production.

The initial `package.json` defines a number of packages as development dependencies, including Babel, Webpack and several Webpack loaders. Babel[13] is an EcmaScript6-to-JavaScript5 transpiler. It allows developers to use the new language features[14] which are not yet available in all browsers, such as arrow functions, the `class` syntax, new keywords etc. Here is the summary of main Babel packages included in the project:

package	description
`@babel/core`	the core part of the library
`babel-loader`	provides integration with Webpack
`@babel/preset-env`	manages syntax transformations
`@babel/preset-react`	adds support for React templates

The project also comes with ESLint pre-installed. ESLint is a JavaScript linter and it helps identify potential problems in code on the fly. Provided that you use an editor that supports ESLint (both VS Code and IntelliJ IDEA Ultimate both do) you will be able to reap the benefits of having your code constantly checked against the set of best practices.

Webpack is both a build tool and module bundler for JavaScript. In our case, it performs the following tasks:

- allows us to work with separate JavaScript files as independent modules
- transforms (using the Babel loader) JSX and modern JavaScript into regular JavaScript code
- combines all of these files into a single script and provides source maps for in-browser debugging

Webpack also performs similar tasks for style sheets. For this project, we use a CSS preprocessor called Sass[15], which enhances ordinary CSS syntax with additional goodies like variables and imports.

In this book, we're not paying much attention to CSS and styling in general. The sample repository includes all necessary code that makes our application look decent, but we're not going to discuss how it is achieved at all.

Now that we know what we need Webpack for, let's take a look at the most important part of its main configuration file - `webpack.config.js`:

[13] https://babeljs.io
[14] http://es6-features.org
[15] http://sass-lang.com

```
module.exports = {
  entry: './ui/entry.js',
  output: {
    path: path.resolve(__dirname, 'public/compiled'),
    filename: 'bundle.js'},
  // ...
}
```

The `entry` setting specifies the starting point of the bundle, and in our case, this is the `entry.js` file. This is a very simple file that only contains two references:

```
import './scripts/app/main.js';
import './styles/style.scss';
```

Both `main.js` and `style.scss` also reference other files. Webpack starts its processing from the `entry.js` file, analyzes it, processes all of its references, analyzes them, processes their references and so on. Eventually, everything that our project uses will be linked together.

The `output` setting of the `webpack.config.js` specifies the output directory. In the previous section, I pointed out that the `public` directory contains static assets like images and fonts as well as compiled scripts and style sheets. Play uses this directory only as a source of static files. If you look at this folder, you'll see that it contains several subdirectories:

subdirectory	description
`compiled`	contains JavaScript and CSS bundles created by Webpack as well as source maps
`images`	contains the logo and favicon
`js`	contains auxiliary JavaScript libraries not included in the bundle

The `compiled` subdirectory is not under version control and can be seen as a bridge between the frontend and backend parts of the app.

The next part of `webpack.config.js` deals with actual source files:

```
// module.exports
module: {
  rules: [
    {
      test: /\.jsx?$/,
      include: /ui/,
      use: {
        loader: 'babel-loader'
      }
    },
    // ...
```

Out of the box, Webpack only works with regular JavaScript files. For everything else, it requires *loaders*. This part of the configuration file actually defines which loader to use for each type of file. First, the `babel-loader` takes `*.js` and `*.jsx` files written in EcmaScript 6 and transforms them into a single JavaScript 5-compliant bundle. Which language features to use is specified in the `.babelrc` file:

```
{
  "plugins": [
    "@babel/plugin-proposal-class-properties"
  ],
  "presets": [
    [
      "@babel/preset-env", {
        "targets": {
          "browsers": ["last 2 versions"]
        }
      }
    ],
    "@babel/preset-react"
  ]
}
```

If you're wondering about class properties, this is the kind of code they allow us to write:

```
class AppComponent {
  init = () => {
    this.initLoginRedirecting();
    this.renderComponent();
  };
}
```

It may not be necessary to write the `init` method as a class property in this particular example, but their presence makes it much easier to write event listeners for React components, and we're going to write many of those in next chapters.

The next part of `webpack.config.js` deals with styles:

```
rules: [
  // ...
  {
    test: /\.scss$/,
    use: [
      MiniCssExtractPlugin.loader,
      'css-loader',
      'sass-loader'
    ]
  }
]
```

This part may look really cryptic, but essentially, it instructs Webpack to take all `*.scss` files and combine them in a single CSS file. And the final rule instructs Webpack to treat stuff like images and fonts as binaries and do not attempt to parse them:

```
rules: [
  // ...
  {
    test: /\.(eot|woff|woff2|ttf|svg|png|jpg)$/,
    use: {
      loader: 'url-loader?limit=1'
    }
  }
]
```

After discussing Webpack, let's take a look at the actual app. The `ui` folder contains all frontend assets:

```
$ tree ui -L 1
ui
├── entry.js
├── scripts
└── styles
```

Not surprisingly, JavaScript (ES6) source files reside in the `scripts` directory, whereas style (SCSS) files are kept in the `styles` directory.

The JavaScript app starts with the `main.js` file, which merely imports the `AppComponent`:

```
import AppComponent from './AppComponent.jsx';

const appComponent = new AppComponent();
appComponent.init();
```

The `AppComponent` itself is written as an ES6 class:

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

class AppComponent {
  // ...
}

export default AppComponent;
```

Note that the `class` keyword is merely syntactic sugar over the JavaScript inheritance model. This, however, makes programming somewhat more pleasant and straightforward. Using this syntax, we can define methods with arrows and access class members via `this`. Everything we want to use outside this file needs to be exported, thus we're exporting the component.

```
// class AppComponent
init = () => {
  this.initLoginRedirecting();
  this.renderComponent();
};
```

The `init` method simply calls two other methods - `initLoginRedirecting` and `renderComponent`. The `initLoginRedirecting` method adds a response interceptor that analyzes errors:

```
// class AppComponent
initLoginRedirecting = () => {
  axios.interceptors.response.use((response) => {
    return response;
  }, (error) => {
    if (error.response.status === 401) {
      window.location = '/login';
    }
    return error.response;
  });
};
```

When the response status is "401 Unauthorized", the user will be redirected to the login page.

Finally, the `renderComponent` method makes React render a small HTML fragment on the page:

```
renderComponent = () => {
  const reactDiv = document.getElementById('reactDiv');
  if (reactDiv !== null) {
    ReactDOM.render(
      <div className="view-home-composite__react-panel__welcome-text">
        If you can see this message then ...
      </div>, reactDiv);
  }
}
```

This method first checks whether the page contains an element with the specific `id`. This check makes sense, because some pages (for example, the login page) are not supposed to include any React by design. If such an element exists, the React part of the app will be rendered inside the given HTML element. It may be unusual to see HTML markup inside of a JavaScript file, but as we will discover later, this approach gives developers a lot of flexibility. Also note that React templates must use `className` instead of `class` for specifying style classes of HTML elements.

We haven't written any code so far, but we will start doing that in the next chapter. If you want to follow along, remember to initialize the app using the `denisftw/play-event-sourcing-starter.g8` template as shown above. The final project is available in the book repository. Feel free to refer to Appendix A for more details.