Expertise
and insight
for the future

Anh Do

React server-side rendering with Scala.js and GraalVM

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

30 October 2019

Author Title Number of Pages Date	Anh Do React server-side rendering with Scala.js and GraalVM 32 pages 30 October 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Janne Salonen, Instructor

The topic of the thesis is to demonstrate the process of building a full stack application using Scala as the single programming language. It discusses the stack based on this language, and whether this architecture is practical for production use.

The motivation behind this study and Scala is the fact that although JavaScript remains the only language supported for building interactive applications on the browser, it comes with its own shortcomings. Scala offers a compelling alternative, rooted in its maturity with the Java ecosystem, and an ever-growing community around Scala.js, a compiler that produces JavaScript code from Scala source. This study introduces these tools and how they are used in building the application.

With related to isomorphic application, programs that run on both browser and server environments, the thesis also investigates polyglot features in GraalVM, a runtime for backend application. GraalVM can execute server-side rendering by executing JavaScript code built for the frontend. Its performance is measured against a pure client-side rendered versions.

The result from this study is a different way to build isomorphic application that does not necessitate using JavaScript, a language traditionally used for this goal.

Keywords	

Contents

List of Abbreviations

CPU	Central processing unit. The component in a computer that performs basics instructions such as arithmetic, control or input/output using an arithmetic logic unit, a register and multiple cache layers.
FCP	First contentful paint. The moment when the browser renders an element from the Document Object Model
FMP	First meaningful paint. The moment at which primary content appears on the webpage.
HTML	Hypertext Markup Language. The markup language used in the browser for displaying webpages.
HTTP	Hypertext Transfer Protocol. A protocol to transfer information on the Internet.
IDE	Integrated development environment. A set of tools assisting a developer with development work. It generally consists of a text editor, a code analysis system and a debugger.
JSON	JavaScript Object Notation. A representation format to serialize application native data into textual format.
JVM	Java virtual machine. A virtual machine that runs Java bytecode.
MIME	Multipurpose Internet Mail Extensions. An extension that allow communication protocols over the Internet to support many data formats such as Unicode, image, audio or video.
RPS	Request per second. The number of requests hitting an HTTP server every second.

SSR	Server-side rendering. A technique to return rich markup directly from the server, instead of an empty page to be populated from the browser.
URL	Universal Resource Locator. A reference to a resource with a protocol to retrieve it. To be used on the Internet.

1 Introduction

In software development, a solution stack is a group of closely related software components that operates towards a common goal. One approach is to build a central service with end-to-end responsibility (a monolith) with smaller auxiliary components around such as database and scheduled jobs. Another approach is to split the solution into smaller interconnected services (microservices) according to their functions. These functions include, but not limited to, payment, user interface, customer registry, etc. These two approaches are not mutually exclusive, but are two ends of a spectrum and applications can adopt a strategy in between for their development

Regardless of the approach, an application can most often be broken into at least two groups of components: the user interfaces and the backend services. This thesis showcases the development of a basic application with one user interface and one backend that also integrates with other external services. The backend service is written in Scala, while the user interface is written in Scala.js. The goal is to discusses the potential advantages of having these components sharing a very similar architecture, while still keeping them separated.

Lastly, by having the backend service runs on top of GraalVM and serve the user interface, the thesis also studies in depth a true polyglot environment and its capability. After that, the thesis benchmarks the application performance and the practicality of using polyglot environment in production.

These objectives shall be discussed in greater details in the next chapter.

2 Application fundamentals and thesis objectives

The software developed in this project, UserPage, is a simple web application with a user interface and one backend service. UserPage lets users browse the list of all available people on the site and calculate the geographical (great circle) distance between them. The user registry is not persisted in this project itself, but is provided by JSONPlaceholder[1], a third-party service which provide mocked data for testing purposes.

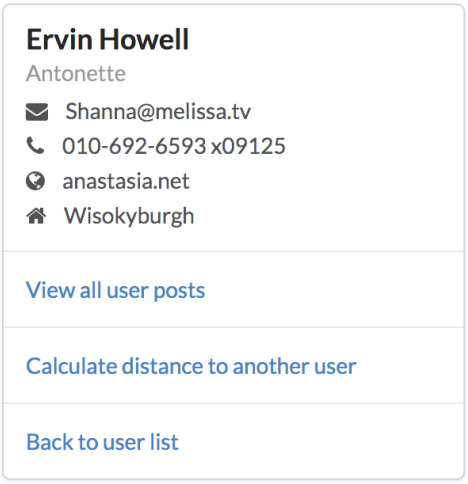


Figure 1. Final version of the user detail view of UserPage

Being simple feature-wise, UserPage reduces the scope of this project to focus only on the technical details of its architecture. It aims to study the topic listed below.

2.1 Develop and maintain a multi-component code base that is written in one single programming language

2.1.1 Scala

The backend service is written in Scala. Scala is a multi-paradigm programming language with an emphasis on type-safety. It incorporates features and patterns from both object-oriented and functional programming.

Scala is a good language of choice for UserPage backend service due to its maturity, excellent ecosystem and great interoperability with the JVM.

2.1.2 Scala.js

The user interface is written in Scala with Scala.js. Scala.js itself is not a different programming language, nor is it a dialect of Scala. It is a compiler that takes Scala code as an input and outputs JavaScript to be executed in the browser.

Choosing Scala for a web subproject deviates from the more popular approach of using a language belonging to the JavaScript family, such as JavaScript itself or TypeScript. The motivation behind this is that code compiled with Scala.js also inherits the advantages of the Scala programming language and benefits from existing Scala ecosystem. Many libraries written in Scala can also be cross compiled with Scala.js to work seamlessly on the web. The tradeoff in performance when writing source code in Scala is negligible, as Scala.js does a good job at optimizing the output JavaScript, as can be seen in its benchmark [2]:

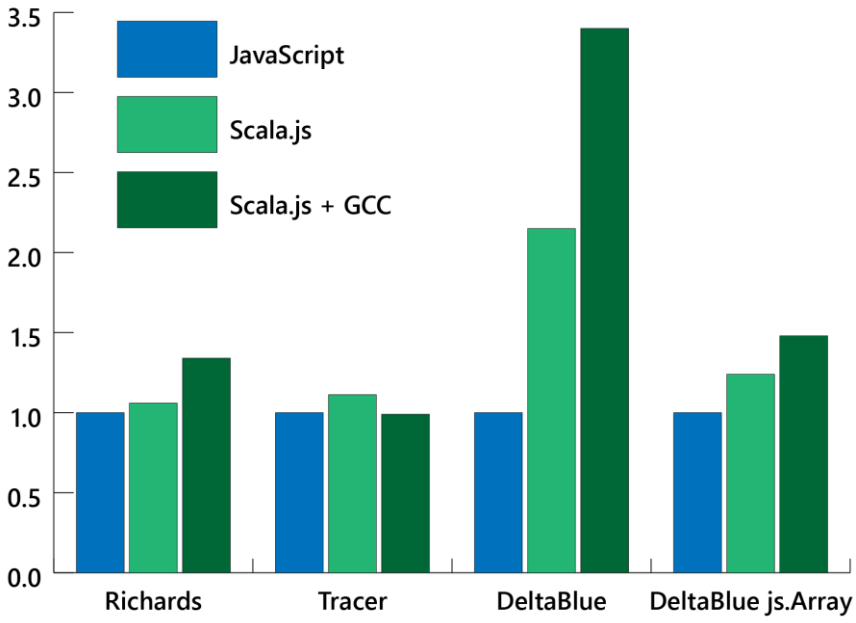


Figure 2. Various Scala.js-generated code benchmark against handwritten JavaScript

The benchmarks in Figure 2 show that Scala.js-generated code is quite performant, approach the speed of handwritten JavaScript in certain suites. In DeltaBlue suite, Scala.js

performs considerably worse than JavaScript (from 2 to 3.5 times worse). The performance degradation is improved drastically by using native JavaScript array (js.Array in Scala.js) instead of native Scala collections. This is a rule of thumb to keep in mind when optimizing Scala.js code whenever performance became critical in specific parts of the application. Native Scala implementations offer much better type-safety, better runtime checks and is more idiomatic to Scala, but are overall less performant. The last thing to note is that Google Closure Compiler optimizes the Scala.js bundle for size instead of speed, so in many scenarios it actually degrades the performance of the application.

Another potential disadvantage of using a non-JavaScript language is interoperability with JavaScript code, in order to access the vast ecosystem around JavaScript and the web. One way to mitigate this issue is to use existing Scala wrappers of popular JavaScript libraries, which can often be called Scala.js libraries. Another approach is to directly provide a Scala type definition, or a façade, to those libraries. Throughout this project, both approaches are exercised as UserPage employs wrapper libraries for the user interface framework React, while also implements type definitions for its component library, semantic-ui-react.

Behind the scene, Scala.js is a plugin on top of the Scala compiler. In general, the compilation consists of three phases [3]. The first phase is almost identical to the Scala for JVM compilation phase, which produces **.class** files. There are small differences, so it is not recommended to actually run these files on the JVM. They contain compilation information that can be used by external tools or IDEs. This phase also produces **.sjsir** files, short for Scala.js Internal Representation, which have types information necessary for the next phases of the compilation, and have many Scala featured replaced with their JavaScript equivalent. The second phase is a fast optimization phase. This is when the Scala.js compiler does two important optimizations: dead code elimination and inlining. Because Scala.js always analyze the program as a whole, it can calculate unused code branch from the libraries and remove those paths, reducing the final output in size. This operation is called dead code elimination, and is a common operation seen among JavaScript bundlers to reduce code size. The second optimization, inlining, investigates different language declarations, and using heuristics, replace them with simpler but equivalent code. Inlining normally includes replacing constant variables with actual literal constant, unrolling collection methods into imperative loops or substituting short function

calls with their implementations. This phase is generally quick and produces already executable JavaScript. The output JavaScript is also optimized to be more efficient than a naïve translation of the input **.sjsir**. During development, it is typical for a program to be repeatedly compiled to this stage for trial runs. Lastly, the final phase, full optimization, takes the output from fast optimization and runs it through the Google Closure Compiler. Because the input has been designed to be very compatible with the tool, Google Closure Compiler can be run at its Advanced Optimization mode, where it aggressively optimizes input code. The compiler also performs operations which boil down to inlining and dead code elimination, but this time with the most minimum output size as its goal. The final bundle is minified with its size typically in the range of 200KB, a 90% reduction from the original unoptimized code of 20MB [4].

2.1.3 Circe

Circe is a Scala library for serialization and deserialization of JSON. Circe is also cross compiled with Scala.js and is used in both the backend service and the web frontend of UserPage. Along with other dependencies that are later mentioned, circe shows very well the motivation behind sharing programming language between the two components of UserPage. Circe also helps with using JSON as a medium data format in server-side rendering that is discussed in the next objective.

2.2 Demonstrate server-side rendering on a JVM-based runtime

Server-side rendering is the way to display information on the web through HTML generated from computations on the server. Using languages such as PHP or Java, it has been the most common way to serve dynamic webpages on the Internet. Implement SSR in this sense is trivial and is not an objective of this thesis.

With the gaining popularity of JavaScript and AJAX over the years, and browsers becoming more powerful, it is possible to implement rich features on the browsers. Without having to reload the page, client-side scripting becomes the main method for websites to respond to user interactions. Web frameworks like Angular or React push this even further by managing the whole application state end-to-end. Client-side rendering refers to the method of serving webpages where an application running in the browser builds

and displays every piece of content from scratch (an empty document). This type of programs gives server-side rendering a new aspect: code isomorphism. An isomorphic application is one that can execute on both the server and the client.

Server-side rendering, in the context of code isomorphism, is the ability to pre-evaluate a client application, whether web or mobile, on the server and deliver as HTML a pre-rendered version of that application. This raw markup is then used in the process called hydration to enrich it with interactive behavior on the client side. The rendering and the hydration processes run through an almost identical set of render instructions, for example a React component tree, which is the isomorphic part of the code. In this thesis, only SSR in an isomorphic React application is discussed, although it inherits many characteristics from the general meaning of the term.

SSR offers many advantages and disadvantages comparing to client-side rendering. These aspects should be considered on a per-project basis in order to make the most suitable decision regarding whether to implement SSR for a user interface component. This study does not evaluate the benefits and tradeoffs of SSR for UserPage use case. It adopts SSR to demonstrate how such features could be implemented for this particular architecture of the application.

The second objective of this thesis is to develop an isomorphic program running on the browser and a server runtime that is based on JVM. This application should contain the SSR capability, and that the frontend it serves should go through the two-stage delivery of rendering and hydration. In order to provide SSR in Scala, UserPage uses Slinky and scalajs-react, two React wrappers for Scala.js.

2.3 Study the feasibility of running a polyglot runtime in production environment

UserPage backend service is run on GraalVM. It is a JVM which supports additional programming languages and interoperability between those languages in a shared runtime. [2] The GraalVM Java compiler can compile and execute Scala source code to run the main server. It also contains an ECMAScript 2019 compliant implementation, which allow GraalVM to execute JavaScript source code. This polyglot runtime is used

to execute the web frontend application to pre-render the markup for the user interface on the server, providing SSR capability.

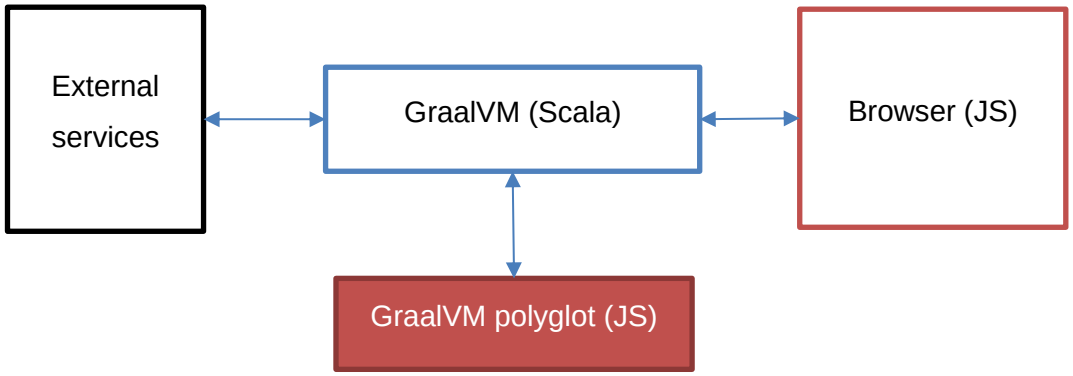


Figure 3. Server-side rendering on a GraalVM polyglot context.

Throughout the project, UserPage uses JSON as the medium to transport data between these two language runtimes. This approach requires serialization of data at the integration surface but produces reusable JSON that can be embedded into the HTML served to the browser. Benchmark and production optimizations are also done against this setup.

2.3.1 Lighthouse

Lighthouse is an automated tool that collect a website's performance metrics and gives analysis based on collected data [5]. Besides performance, Lighthouse can also analyze the page for possible improvements regarding search engine optimizations, accessibilities, Progressive Web App compliance and other best practices. It can produce limited-resources conditions such as CPU slowdown or network throttling to simulate lower-end mobile devices or slow network connections. Those conditions are produced out-of-the-box by Lighthouse with best-effort mechanism, and it gives instructions on how to even more realistically simulate these conditions.

In this project, Lighthouse is used solely for performance measurement. The main indexes are FCP, FMP and time-to-interactive. They have the highest impact to user experience in real world scenarios.

3 Scala implementation of UserPage

3.1 Set up the development environment

This section describes the steps for setting up a workspace suitable for this thesis. The following prerequisites are listed briefly:

- Use macOS operating system. Version 10.14+ is used in this project.
- Set up a basic Scala development environment [3] including sbt. Scala and sbt versions are configurable in the code base.
- Download and extract GraalVM Enterprise Edition for macOS for non-production license. The version used in this thesis is 19.2.0.1.

It is possible to build the application on a Linux operating system. In such case the GraalVM distribution for Linux should be used instead. It is not recommended to use Windows, however, as Windows support is experimental [4].

For brevity, this thesis does not opinionate on which IDE or text editor to be used. Whenever possible, operations are executed as command inputs to the terminal.

The extracted GraalVM folder should look similar to figure Figure 4:

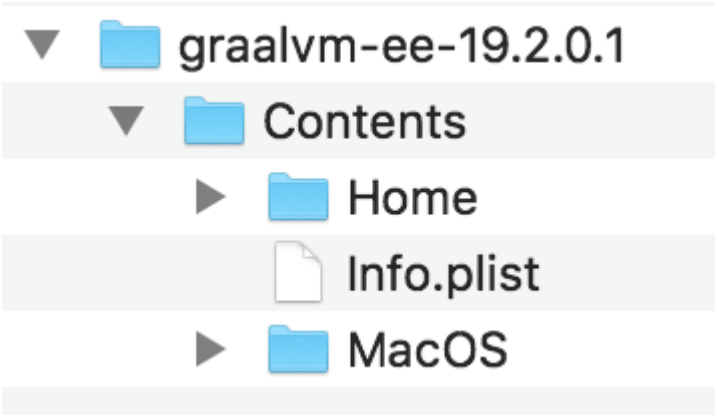


Figure 4. Extracted GraalVM distribution

The developer should take note of the path **./Contents/Home** as this is the JAVA_HOME path needed to run the application on GraalVM. It is recommended to set a symlink to this path from **~/bin/graalvm** as this path is used in the examples.

After these steps are done the developer is ready to start building UserPage.

3.2 Create the sbt boilerplate

The first thing to do is to configure the sbt project and plugins for UserPage. Set the sbt version to 1.2.8 in **build.properties** and add the following plugins to **plugins.sbt**:

```
addSbtPlugin("org.scala-js" % "sbt-scalajs" % "0.6.29")
addSbtPlugin("ch.epfl.scala" % "sbt-scalajs-bundler" % "0.14.0")
addSbtPlugin("org.portable-scala" % "sbt-scalajs-crossproject" % "0.6.0")
addSbtPlugin("pl.project13.scala" % "sbt-jmh" % "0.3.4")
```

Listing 1. Plugins added to plugins.sbt

In addition to Scala.js, the other three plugins are scalajs-bundler, scalajs-crossproject and jmh. The first plugin, scalajs-bundler, is used to build and bundle a Scala.js application using Webpack. One of the features it provides is a new configuration key to **build.sbt** called **npmDependencies**. This key is an alternative to the default **jsDependencies** from Scala.js which automatically creates a **package.json** file in the build folder to manage JavaScript dependencies through npm. It is the recommended approach by Scala.js for emitting JavaScript modules, which is how UserPage is built. Emitting modules is incompatible with **jsDependencies** [citation?].

3.3 Create the UserPage web client

UserPage is a single-page application that shows the posts of the users of the service and calculates the geographical distances between users. This section describes how to implement the frontend module of UserPage. It is important to first study the different modules of the frontend and how they interact:

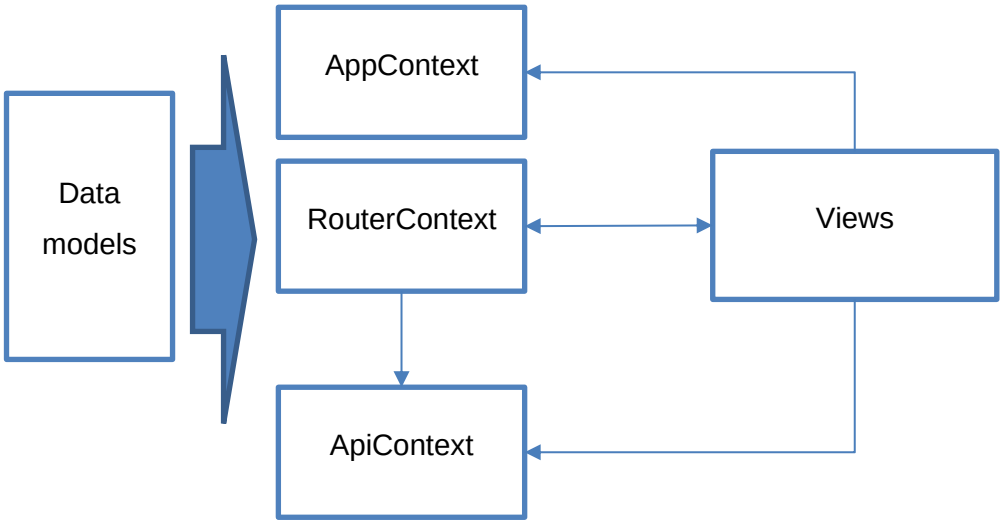
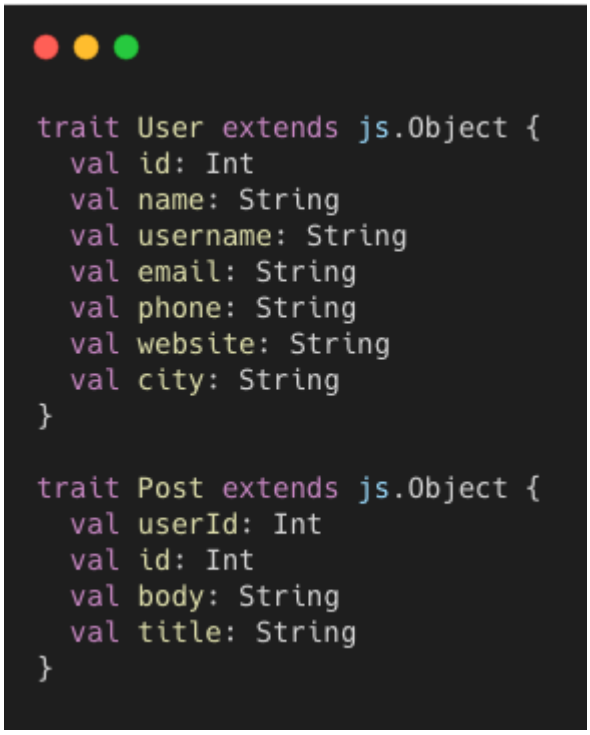


Figure 5. Main modules of UserPage frontend.

Figure 5 shows that the main modules are the views. They interact with each other and the outside world through other modules provided by React context mechanism. Each view represents a different logical page observable by the user. Navigation between the views is supported by RouterContext. The RouterContext is implements a client-side routing mechanism to move between the view without reloading the page. AppContext provides all application data available from initial page load. AppContext stores the list of users and the list of all posts grouped by each user. This data is needed in every view. Next, ApiContext exposes an interface to communicate with the backend server on demand to fetch more dynamic data. It is used in the view that compares two users. Lastly, the data models do not contain any behavior, but are used in every other module.

3.3.1 Data models

After studying the structure of the web client, the developer should start the implementation with defining the fundamental models for this project. The only two data models are User and Post. Firstly, they are defined as native JavaScript objects:



```
trait User extends js.Object {
  val id: Int
  val name: String
  val username: String
  val email: String
  val phone: String
  val website: String
  val city: String
}

trait Post extends js.Object {
  val userId: Int
  val id: Int
  val body: String
  val title: String
}
```

Figure 6. User and Post data models

Figure 6 shows how to define native JavaScript object types in Scala.js. They are required to extends from **js.Object** and any concrete member needs to be equal to **js.native**, signifying their lack of Scala implementation, and that the usage of their API depends solely on the existence of an equivalent JavaScript implementation during runtime. In UserPage case, since all models are just data objects without behavior, it is logical to declare them as Scala traits.

In further chapters, this thesis shows that the declaration above is only suitable for client-side code using Scala.js, and that the backend server requires an equivalent model declaration.

3.3.2 Client-side routing logic

In order to implement the client router, the developer needs to be aware of all possible views, and how they are represented as routes. In UserPage, there are a total of five different views, that are accessible through five routes and four React components:

Table 1. All views of UserPage

View	Component	Purpose
List of users	UserList	This view shows a list of all existing users.
User detail page	UserCard	This view shows the details of a single user.
User post list	PostList	This view shows a list of all posts created by one user.
List of users to compare with a selected user	UserList	This view allows selecting a second user to be compared side-by-side with another one from this list.
Users comparison page	CompareUsers	This view shows the details of two different users next to each other and calculate the geographical distance between them.

The router implementation that allows navigation between these views is shown below:

```
object ClientRouter {

  type RouteNav = RouterCtl[SpaPage]

  private val routerConfig = RouterConfigDsl[SpaPage].buildConfig { dsl =>
    import dsl._

    (emptyRule
      | staticRoute("users", UserListRoute)
      ~> render(ReactFragment(UserList(None).toScalaJSReact))
      | dynamicRouteCT("users" / int.caseClass[UserInfoRoute])
      ~> dynRender { page: UserInfoRoute =>
        ReactFragment(UserCard(page.id).toScalaJSReact)
      }
      | dynamicRouteCT(("users" / int / "compare" / int).caseClass[CompareUsersRoute])
      ~> dynRender { page: CompareUsersRoute =>
        ReactFragment(CompareUsers(page.id1, page.id2).toScalaJSReact)
      }
    )
      .notFound(redirectToPage(UserListRoute)(Redirect.Replace))
      .renderWith({ (_, resolution) =>
        <.div(^.padding := "30px")(
          resolution.render()
        )
      })
      .logToConsole
  }

  val (router, ctl) = Router.componentAndCtl(BaseUrl.fromWindowOrigin_/, routerConfig)

  val ctl: RouterCtl[SpaPage] = logic.ctl
}
```

Figure 7. Implementation of the client router

For brevity, only three routes are shown. It is very similar to CompareUsersRoute that the remaining two routes (for the list of posts and list of users to be compared) should be defined. From Figure 7, it is shown that the router should accept only routes of a specific types. **UserListRoute** and **UserInfoRoute** are case classes that extends the trait **SpaPage** that is extended by every route. A route consists of a path definition that parses and extracts parameters from the URL path and send them to the handler function that renders the matched route as React components. It is worth noting that all components have to use **toScalaJSReact** before being returned. It is because they are implemented in Slinky and the router only accepts scala-js-react components, hence an implicit method provided by Slinky is used to convert these components from a format usable by one library to the other. UserList is used as the fallback view in case the URL does not match any defined route. Finally, the router output is wrapped in some extra layout stylings before being rendered to the page.

With the data models and router completed it is now possible to implement the view components. The two other important modules, AppContext and ApiContext, are studied in more depth in further sections of the thesis where they are more relevant. Right now, it is safe to assume that they have the functionalities as are described in the beginning of section 3.3.

3.3.3 View components

The first view to create is the UserList. This view is used in two scenarios. The first one is as a landing page or overview, and the second scenario is when a user has to be selected in order to be compared to another pre-selected user. The code for this component should look like in Figure 8:

```
@react object UserList {

  case class Props(baseId: Option[Int])

  private val ListAll = FunctionalComponent[Seq[User]] { users =>
    val ctl = useContext(RouterContext)

    List(bulleted = true)(
      users.map({ user =>
        List.Item(js.undefined)(
          List.Content(ctl.link(UserInfoRoute(user.id))(user.name).toSlinky)
        ).withKey(user.id.toString)
      })
    )
  }

  val component = FunctionalComponent[Props] { props =>
    val items = useContext(AppContext).users.values.toSeq
    props.baseId match {
      case None => ListAll(items)
    }
  }
}
```

Figure 8. UserList component

This is the implementation of a basic React functional component written in Slinky. Slinky. The main advantage of Slinky is that it allows developers to build React component with an API very similar to that of native React. It achieves this with the help of the **@react** macro annotation placed on an object (UserList in this case). After a **case class Props** and a public constant **val component** are declared on the object, the macro generates an **apply** method on UserList and it then could be used as a normal functional component. From the figure above, it is shown that there is a private sub-component, ListAll, that is used when there is no **baseId** passed to UserList as props. ListAll only covers one of the two use cases of UserList, which is the overview page. For brevity, the implementation of another sub-component used when selecting the second user to compare is omitted from this report, but it is very similar to ListAll. This component, called ListCompare, is rendered when a base user ID is passed to UserList as React props and thus is rendered on the Some branch of the pattern match on **props.baseId**. ListCompare covers the remaining use case of UserList.

According to Figure 8, UserList is rendering different List leaf components. They are external components coming from a JavaScript library, semantic-ui-react. For the project to compile, a façade for these components needs to be defined. As an example, the type definitions of all List components are written like in Figure 9:

```
@react object List extends ExternalComponent {

  @js.native
  @JSImport("semantic-ui-react", "List")
  object Raw extends js.Object {
    val Content: js.Object = js.native
    val Item: js.Object = js.native
  }

  case class Props(bulleted: Boolean = false)
  val component = Raw

  object Content extends ExternalComponentNoProps {
    val component = Raw.Content
  }

  @react object Item extends ExternalComponent {
    case class Props(icon: js.UndefOr[String], content: js.UndefOr[String] = js.undefined)
    val component = Raw.Item
  }
}
```

Figure 9. A façade for semantic-ui-react List components

Figure 9 shows the Scala.js type definitions for List and its sub-component. At the beginning of the object a native JavaScript object is declared. It is also annotated with **@JSImport** to indicate that this object comes from a JavaScript module, semantic-ui-react, and is imported through the named import List. Slinky-specific **@react** macro can be seen annotating List and List.Item from the figure. In this case, the **val component** fields are assigned native objects from the imports directly instead of being assigned a concrete implementation (as in the case of UserList). This is because the native objects are already those components' implementations by the library, and the ExternalComponent abstract class is used to indicate the components are already implemented and ready to be used normally. It is worth pointing out that components extending ExternalComponentNoProps do not need a **case class Props** declared, and do not need to be annotated with **@react** either.

This is the final look of the list of user view:

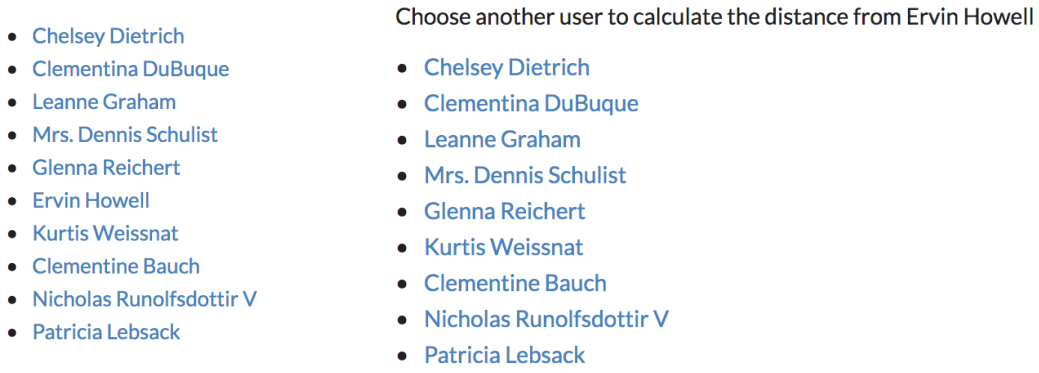


Figure 10. The final view of the user list overview (left) and when a user is preselected (right)

As is shown in Figure 10, user Ervin Howell who is already selected are excluded on the list to the right. One of the users in this list can be selected and the distance between that user and Ervin Howell is calculated in the user comparison view.

The implementations of the other view are not examined in depth, but they should follow the same approach as UserList. The UserPage web client is not ready by this point, but the rest of the modules are discussed together with the backend service in the next section.

3.4 Create the backend service and integrate with the web frontend

The second major component that needs to be implemented for UserPage to be functional is the backend service. Figure 11 illustrates the structure of this component:

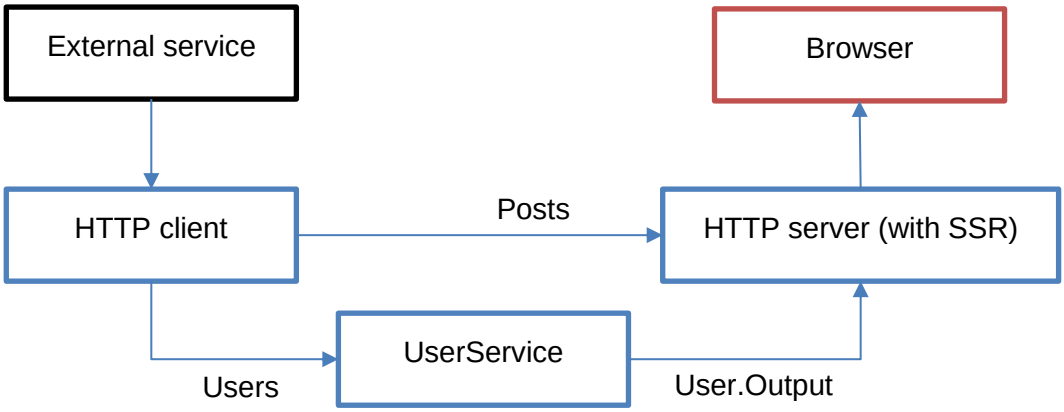
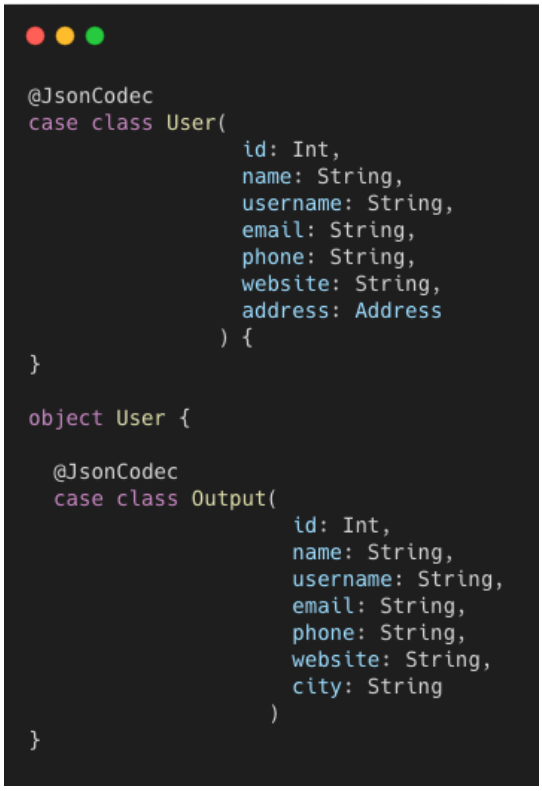


Figure 11. Packages of UserPage backend

The HTTP Server package is responsible for handling requests from the browser. It does not expose any data endpoints, but instead serve pre-rendered HTML embedded with data directly to the browser. It also serves the bundled JavaScript that is built by the web client component. The UserService package's functions consist of delivering all user data to be rendered and calculating the distance between users. Distance calculation is the only domain logic that UserPage application contains. ExternalService is a package representing data coming from the outside world. It returns the list of users in internal format to UserService and the list of all posts directly to the HTTP Server. ExternalService internally owns an HTTP Client that it uses to make requests to JSONTypicode for users and posts data.

As mentioned in section 3.3.1 regarding data models of UserPage, it is necessary to define a separate set of models to be used on the backend. The User class is defined like below:



```
@JsonCodec
case class User(
                  id: Int,
                  name: String,
                  username: String,
                  email: String,
                  phone: String,
                  website: String,
                  address: Address
                ) {
}

object User {

  @JsonCodec
  case class Output(
                     id: Int,
                     name: String,
                     username: String,
                     email: String,
                     phone: String,
                     website: String,
                     city: String
                   )
}
```

Figure 12. User data model for the backend service

It can be seen from Figure 11 and Figure 12 that there is a separate case class for User.Output. This is the type of User details returned from the backend to the web. The actual User class is only used internally in the server, and comes from JSONTypicode, the external user service consumed by UserPage. The **@JSONCodec** is a helper annotation by circe that automatically creates a pair of encoder and decoder to convert between JSON and these types.

The HTTP server is the module handling the client requests. It is written using Finch and is returning three types of content:

- HTML is the markup for UserPage frontend. The markup file is not stored in the filesystem as a static asset but is generated on every call to the backend by the server-side rendering function. It wraps the rendered component and preloaded data in a basic HTML skeleton.
- JSON data containing users' distance is the response to Ajax calls. The browser makes a request to fetch this data every time a user navigates to the comparison view. As the page is not reloaded in a SPA, only JSON data needs to be returned. The same calculation is also executed when a

user reloads this view completely, but instead of being fetched separately, the calculation result is delivered together with the markup.

- JavaScript files are loaded by <script> tags. These JavaScript files need to be generated from the frontend client subproject.

Like AppContext and ApiContext, the HTTP server contains logic related to server-side rendering that are discussed in more details in a later chapter.

3.5 Build and run the application

After both subprojects are implemented, the developer can start building and running UserPage. From the terminal, execute the following command:

```
sbt client/fastOptJS::webpack
```

This command tells Scala.js to build and optimize the JavaScript bundle from Scala code. **::webpack** is a task specific to scalajs-bundler which tells sbt to bundle JavaScript dependencies using Webpack. The output should look like in Figure 13:

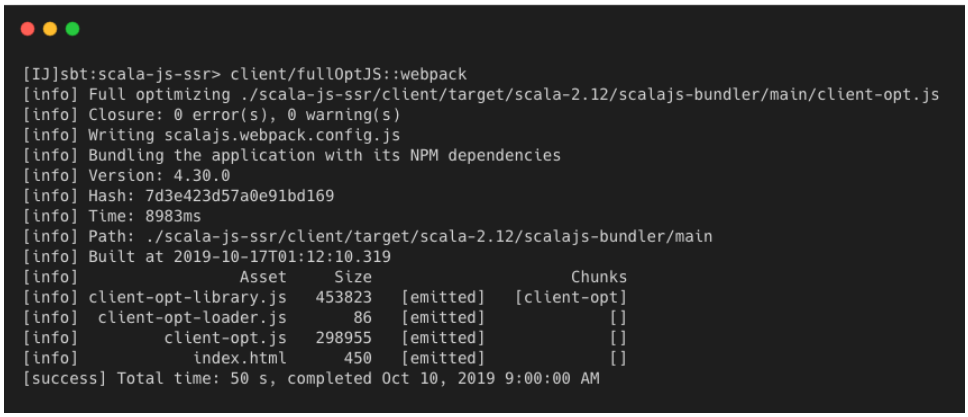


Figure 13. Building UserPage frontend module

Developers familiar with Webpack should recognize its output style. The next step is to run the backend server:

```
sbt -java-home ~/bin/graalvm server/run
```

After this, sbt starts compiling and running the code. The logs that signal server readiness should look like the below:

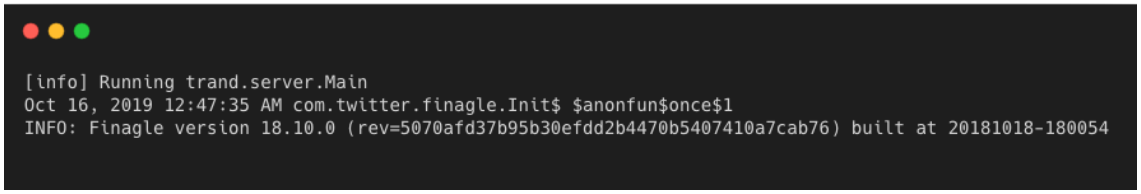


Figure 14. Logs from sbt when the backend server is ready to serve traffic

When the server is ready, depending on where it is run or deployed (in this case, locally), user can navigate to its host at path **/users** to see the UserList view. With this final step, the development of the first version of UserPage is now complete. As both frontend and backend module source code is written in Scala, this project finishes kickstarting a single-programming language codebase.

4 Server-side rendering in UserPage

4.1 Server-side rendering flow

In this section, the thesis discusses more deeply the different aspects related to server-side rendering on UserPage. For instances, how different modules on the backend and the frontend work together for this requirement. Figure 15 illustrates an overview of how the SSR flow on UserPage looks like:

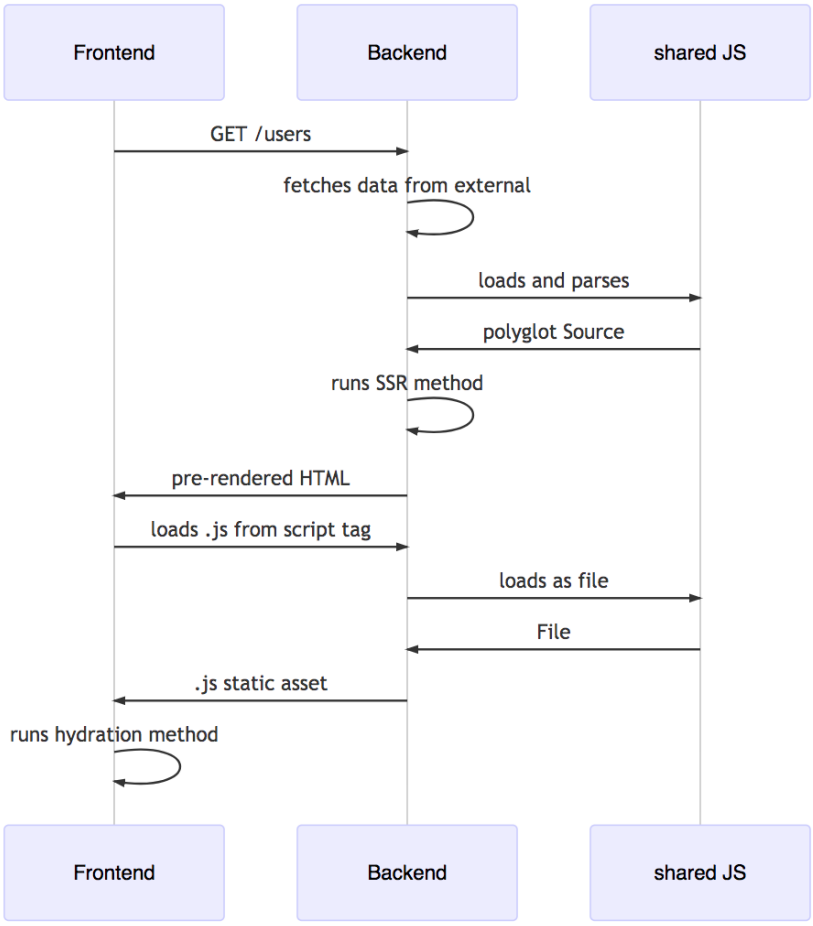


Figure 15. Server-side rendering sequence

According to Figure 15, the sequence starts with a user making a request to the backend from the web browser. The backend service fetches external data and does necessary calculation internally for the final dataset. It then loads the shared JavaScript source from the filesystem and parse it as a GraalVM polyglot source object. This shared JavaScript

is built exactly from the web client subproject. After loading the source successfully, it executes a method, **ssr**, responsible for rendering the application as HTML string. This HTML is then wrapped into an outer layer of HTML and returned to the browser. By this point the user has already seen a fully rendered page, though it is not responsive to interaction just yet. The pre-rendered HTML contains a script tag that instructs the browser to load the shared JavaScript from the backend service. In order to serve this JavaScript, the server loads the source again from the filesystem but this time as a static asset instead of code source. The static asset is sent back to the browser. Finally, the page parses the JavaScript source and executes another method, **hydrate**, to populate the client-side React app with states, reconciliate its virtual DOM with the real DOM and attach necessary event handlers to the real DOM.

4.1.1 Running JavaScript on the server

Figure 16 demonstrates the mechanism for loading and executing JavaScript on GraalVM:

```
object Main extends App {

  def wrapHtml(rendered: String, externalUsers: String, externalPosts: String, distance: Option[Double]) = {
    s"""
       |<html lang="en">
       |<head><title>UserPage</title></head>
       |<body>
       |<div id="root">$rendered</div>
       |<script>
       |  window.__USER_DATA__ = ${sanitizeJs(externalUsers)};
       |  window.__POST_DATA__ = ${sanitizeJs(externalPosts)};
       |  window.__DISTANCE__ = ${distance.map(_.toString).getOrElse("undefined")};
       |</script>
       |<script type="text/javascript" src="/js/shared.js"></script>
       |</body>
       |</html>
     """.stripMargin
  }

  val userList: Endpoint[String] =
    get(root :: "users").mapAsync { req =>
      externalData.map {
        case (usersStr, postsStr) => Loader.renderPath(req.path, usersStr, postsStr)
      }
    }

  val js: Endpoint[Buf] =
    get("js" :: path[String]) { fileName: String =>
      val script = new File(s"$srcDir/$fileName")
      val reader = Reader.fromFile(script)
      Reader.readAll(reader).map(Ok)
    }
}
```

Figure 16. Loader is responsible for loading and running JavaScript sources

The first method from the top, **makeSources**, is responsible for building a polyglot source from the File object, representing the JavaScript file loaded from the filesystem.

Next, the method **evalSources** receives such file and evaluates it using GraalVM JavaScript engine. This results in two methods defined in the polyglot binding, one of which, **ssr**, the server-side renderer, is returned. Lastly, the third and only public method, **renderPath**, executes the server-side renderer with all necessary parameters, wraps it in outer HTML and returns the result as markup string to the browser. An implementation of the main HTTP server calling the Loader module should look like below:

```
object Main extends App {

  def wrapHtml(rendered: String, externalUsers: String, externalPosts: String, distance: Option[Double]) = {
    s"""
       |<html lang="en">
       |<head><title>UserPage</title></head>
       |<body>
       |<div id="root">$rendered</div>
       |<script>
       |  window.__USER_DATA__ = ${sanitizeJs(externalUsers)};
       |  window.__POST_DATA__ = ${sanitizeJs(externalPosts)};
       |  window.__DISTANCE__ = ${distance.map(_.toString).getOrElse("undefined")};
       |</script>
       |<script type="text/javascript" src="/js/shared.js"></script>
       |</body>
       |</html>
     """.stripMargin
  }

  val userList: Endpoint[String] =
    get(root :: "users").mapAsync { req =>
      externalData.map {
        case (usersStr, postsStr) => Loader.renderPath(req.path, usersStr, postsStr)
      }
    }

  val js: Endpoint[Buf] =
    get("js" :: path[String]) { fileName: String =>
      val script = new File(s"$srcDir/$fileName")
      val reader = Reader.fromFile(script)
      Reader.readAll(reader).map(Ok)
    }
}
```

Figure 17. Main object which contains the HTTP server

It can be seen in the wrapper HTML that all necessary data is embedded into the markup. This data is needed for the hydration process on the client. Additionally, the returned HTML includes a script tag to load the same shared JavaScript. The HTTP server is configured to serve JavaScript asset for all requests coming to the path **/js**.

4.1.2 Running JavaScript on the client

By this point, the app has already appeared visually complete to the user. The next step is to hydrate the pre-rendered markup into a fully interactive web application using embedded data from the server.

```
object Main extends App {

  @JSExportTopLevel("ssr")
  def ssr(
    path: String, externalUsers: String, externalPosts: String,
    distance: js.UndefOr[Double]
  ): String = {
    ReactDOMServer.renderToString(RootComponent())
  }

  @JSExportTopLevel("hydrate")
  def hydrate(): Unit = {
    ReactDOM.hydrate(
      RootComponent(),
      document.getElementById("root")
    )
  }

  if (js.typeOf(document) != "undefined") {
    hydrate()
  }
}

@JSGlobalScope
@js.native
object GlobalScope extends js.Any {
  var __USER_DATA__ : js.Array[User] = js.native
  var __POST_DATA__ : js.Array[Post] = js.native
  val __DISTANCE__ : js.UndefOr[Double] = js.native
}
```

Figure 18. Hydration on the frontend

Though simplified, Figure 18 shows the main idea behind the hydration process. The embedded data is loaded into the GlobalScope object through the **@JSGlobalScope**

annotation. It is loaded into the global application context from there. There is a check behind this object that instructs the application to fallback to fetching external data when it is empty, which adds an extra layer of flexibility to this process. It can also be seen from the Main object that there are two different exported methods, namely **ssr** and **hydrate**, which matches the server-side rendering strategy discussed above. The hydration method is called automatically at the presence of a document object, which is used to identify the browser environment versus GraalVM.

After the flow is complete, it is easy to verify that (almost) identical executions take place on both the server and the browser:

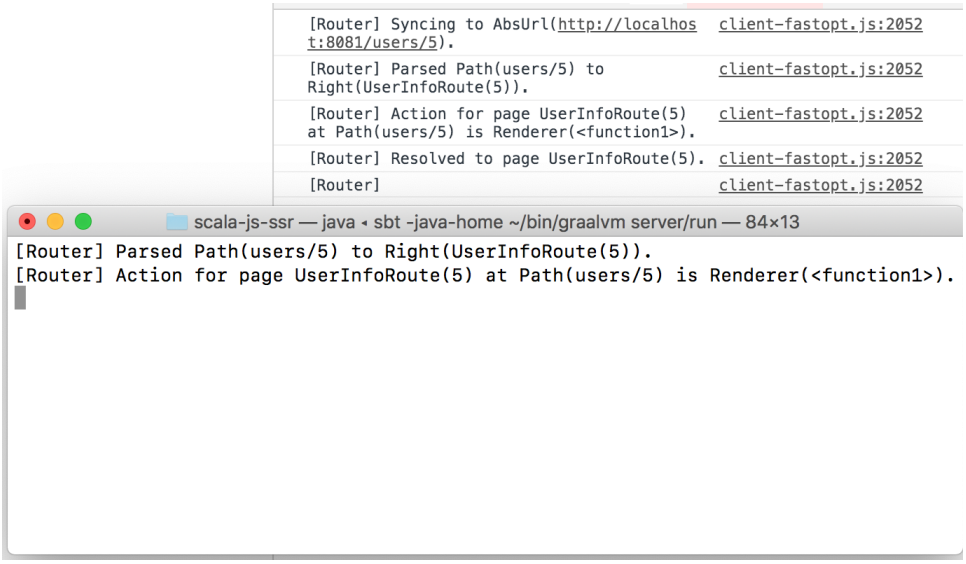


Figure 19. Router logging into the browser and server console

In the above figure, the user navigates to **/users/5** and the router evaluates twice, and both runs match UserInfoRoute and render the component for that route.

4.2 Server-side rendering performance

This section aims to evaluate the performance of server-side rendering in the specific scenario of UserPage. The project assumes that the solution stack remains written fully in Scala, thus the benchmark is only comparing the performance with and without SSR, and tries to draw observations from that test case.

4.2.1 Server response time

Firstly, it is important to point out that server response time, or latency, is not the suitable metrics to compare between SSR and client-side rendering. Because client-side rendering does no extra computation on the backend, its latency should be lower. The difference becomes significantly larger when the markup for this scenario can be served statically and appropriate caching strategy is configured on the backend. However, the measurement is still useful. The difference in latency can demonstrate the point that is discussed above very well. Additionally, server-side rendering latency metrics can still be used in later chapters to measure the impact of optimization strategies on the backend side.

Since currently UserPage has no capability to do client-side rendering, for SSR has been the design goal since the beginning, this feature needs to be added. This thesis does not go into details how it is implemented, but it is sufficient to say main difference is that the server serves in-memory static HTML string and on the browser, the normal React rendering process is used instead of the hydration process.

The tool used in this project to generate load and measure latency is **wrk**. The user comparison view is used for the benchmark due to its dynamic nature and data that is calculate internally by the backend. The performance test is run on a MacBook Pro 2015, 2,7 GHz Intel Core i5 CPU, 16 GB 1867 MHz DDR3 RAM. **wrk** is run on the same machine as the server, and there are other processes running on the same machine as the two programs, so deviations from intended results are unavoidable. However, this thesis assumes that longer-running test should provide adequately accurate benchmark, especially for just the purpose of comparing between different request flows.

Before measurement, any running server should be restarted. The first benchmark is for client-side rendering. In order for this benchmark to be effective, the server is configured to optimized for serving the static markup, by loading the HTML into memory as buffer and returned as-is from the endpoint. To run the test, execute the following command:

```
wrk –timeout=5s -d20 –latency –s bench.lua http://localhost:8081/
```

Listing 2. Benchmark client-side rendering

Listing 2 above generates requests for the same endpoint for 20 seconds with 5-second timeout configured for slow requests then prints out the results. The script, **bench.lua**, generates path for the request in the form of **/users/x/compare/y** where x and y are replaced with integer from 1 to 10 representing random user IDs. After 20 seconds the tool output the following result:

```
Running 20s test @ http://localhost:8081/
  2 threads and 10 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   431.72us  519.29us  23.10ms   96.36%
    Req/Sec    11.74k     1.20k   13.71k    81.75%
  Latency Distribution
     50%  345.00us
     75%  398.00us
     90%  553.00us
     99%    2.03ms
  467373 requests in 20.01s, 316.46MB read
Requests/sec:  23352.66
Transfer/sec:     15.81MB
```

Listing 3. Benchmark results for client-side rendering latency

Listing 3 shows that the average latency for this endpoint is 431.72µs. Other runs of this benchmark yield varying results in the range of 400-500µs. The exception to this is the first two runs, which yields an average latency of 15.82ms and 2.03ms, respectively. This is a characteristic of JVM-like runtime where warm up is needed to reach peak performance. GraalVM is not an exception to this. In order to properly warm up an application for the test, the simplest method is to just run several of the similar benchmark until a consistent latency is reached.

In real-world applications, with high enough traffic, latency impact to a small percentage of users could still result in big financial impact if the user base is large enough. For this reason, many real-world services often care more about 99-percentile latency (or p99 latency) or higher (p999 latency and so on). The benchmark in Listing 3 has a p99 latency of 2.03ms. In other tests, the same metric is in the range of 2 to 3ms. From this point on, the thesis will focus on reporting p99 latency instead of average.

The second test to be done is to benchmark server latency of the SSR application. The same command in Listing 2 can be used to run the test. The implementation is switched back to SSR on the server. The first few iterations of the benchmark result in high latency and a lot of timeout. The test should be rerun multiple times until consistent latency is reported:

```
Running 20s test @ http://localhost:8081/
  2 threads and 10 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   302.76ms  245.03ms   1.09s    76.51%
    Req/Sec    18.11      9.53    50.00     74.08%
  Latency Distribution
     50%  218.97ms
     75%  526.08ms
     90%  632.59ms
     99%  928.53ms
  699 requests in 20.03s, 19.93MB read
Requests/sec:     34.89
Transfer/sec:      0.99MB
```

Listing 4. Benchmark result for server-side rendering latency

As can be seen from Listing 4, the latency from SSR is much higher. The p99 latency in this test is 928.53ms, which is almost a second. In multiple runs of this benchmark, the p99 ranges from 800ms to 2s. However, this data is not enough to conclude about the user impact of the latency. In order to measure how the UserPage functions as a website, another tool that performs benchmark from the user's perspective is needed.

4.2.2 Page performance

The next step is to collect user-centric metrics from the browser. Lighthouse is used for this purpose. The test is first run using Simulated network throttling under the following settings [6]:

- latency: 150ms
- throughput: 1.6Mbps down / 750 Kbps up
- packet loss: none
- CPU slowdown: 4x.

After setting Lighthouse up, it is run against the client-side rendering feature:

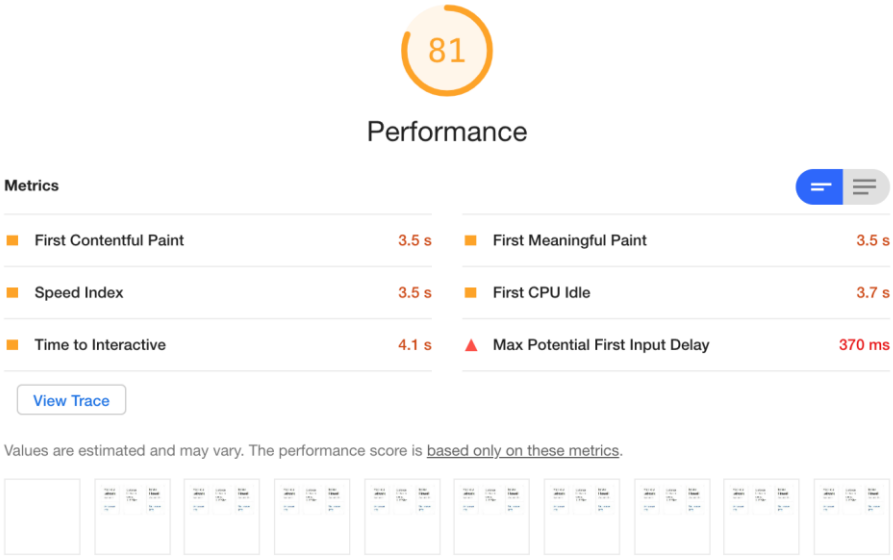


Figure 20. Client-side rendering Lighthouse report

Figure 20 shows that although server response time is very low for markup request. The FCP is still only after 3.5s. The reason for this is that the initial markup is empty, and more network requests must be made to the server to fetch users' data and calculations, which are dynamic in nature and require calls to other downstream services. This is illustrated well in the performance trace:

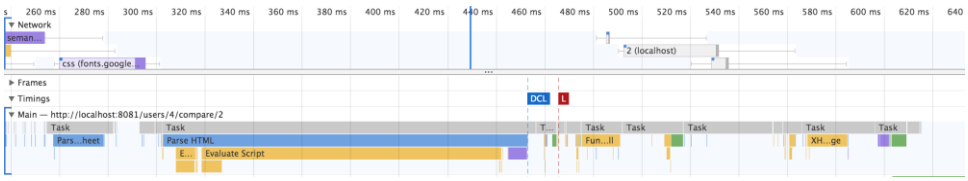


Figure 21. Performance breakdown of client-side rendering Lighthouse test.

As can be seen in Figure 21, more requests are made at 500ms mark on the timeline.

The next test is done against SSR React:

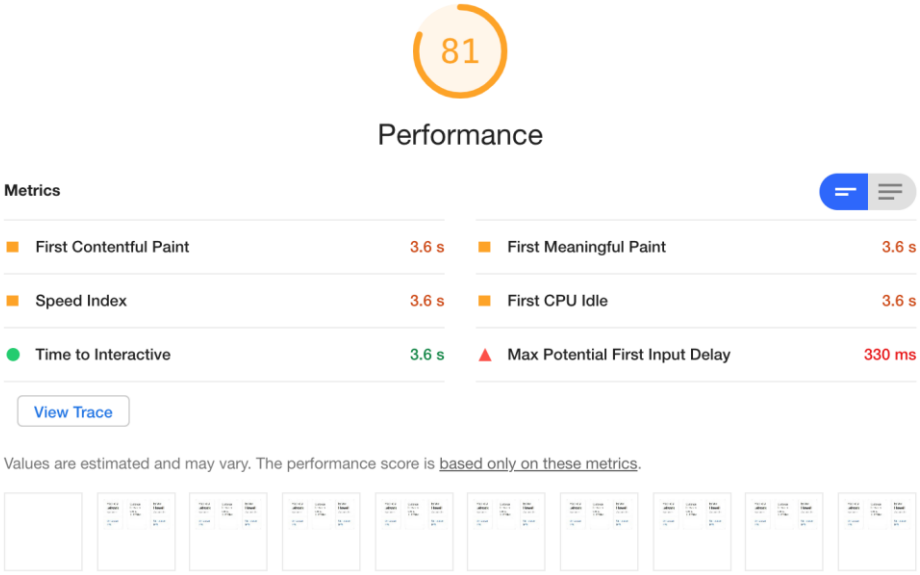


Figure 22. Server-side rendering Lighthouse report.

From Figure 22, it can be seen that the metrics are very similar to that of client-side rendering. FCP and FMP time of SSR is slightly lower but time to interactive are shorter (3.6s compared to 4.1s) and is considered satisfactory by the tool. While the overall performance scores are equal, this cannot be used to conclude that the performances of the two features are strictly equal. Vice versa, a difference by a small margin does not mean one performs better than other in all cases. The two approaches have very similar performance characteristics, even though server response time measurements are vastly different. This is illustrated by Figure 21 of the previous test and the network traces in this test in Figure 23 below:

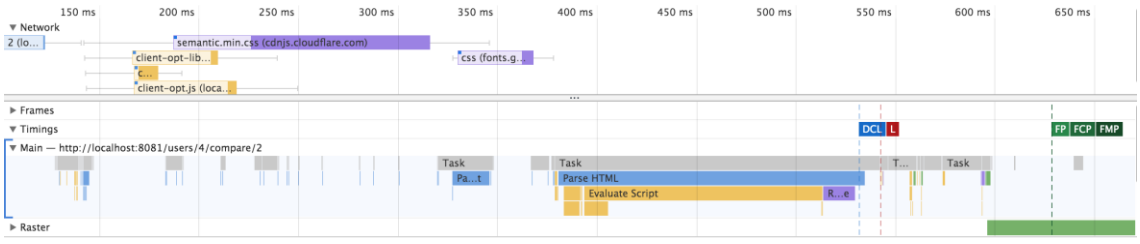


Figure 23. Performance breakdown of server-side rendering Lighthouse test.

As can be seen from Figure 23, there is no second roundtrip to the server for data. Users are able to see the proper view right after assets are loaded. This saves time before the

application can start hydration, which results in shorter time to interactive. The time saving is further amplified the worse the internet connection is.

5 Conclusions

The main objectives of this thesis focus on the technical aspects of an application. As a result, the mentioned application, UserPage, contains a very small set of features. Such features are powered by user events, network requests and client-side routing. Instead, stack architecture and project hierarchy are covered in depth. This study goes into details on the implementations of the most important modules, and how they interact with each other. There are not many conclusions to be drawn from this part, however, they demonstrate proper techniques on how to approach common problems in a React application, and how to solve them using idiomatic Scala solutions. The project also emphasizes on server-side rendering, a set of features providing pre-rendered HTML to the browser. The code which does the rendering on both browser and server is called isomorphic code. The term is popularized by isomorphic JavaScript, programs that runs natively on browsers and Node.js runtime. In this thesis, it focuses instead on Scala as the isomorphic source. This path arrived at an obstacle, for neither does Scala runs directly on the browser, nor JavaScript on a Java Virtual Machine. Thus, server-side rendering in Scala is a topic that necessitates the use of a runtime capable of executing JavaScript. GraalVM has been chosen for this purpose. Although GraalVM contains its own set of challenges, communicating through polyglot interface is straightforward and the program is demonstrated to run on both environments. Finally, the study analyzes performances of server-side rendering at different point in time, and from different perspectives. It finds out that although there is an increase in the response time, important metrics such as First Meaningful Paint time are preserved, and furthermore an improvement in Time to Interactive is observed. This comes to show that server-side rendering on GraalVM has the potential to operate for production use.

References

1 JSONPlaceholder - Fake online REST API for developers [Internet]. Jsonplaceholder.typicode.com. 2019 [cited 22 October 2019]. Available from: https://jsonplaceholder.typicode.com/

2 Performance [Internet]. Scala.js. 2019 [cited 22 October 2019]. Available from: https://www.scala-js.org/doc/internals/performance.html

3 Li H. Hands-on Scala.js [Internet]. Lihaoyi.com. 2014 [cited 22 October 2019]. Available from: http://www.lihaoyi.com/hands-on-scala-js/#HowCompilationWorks

4 Li H. Hands-on Scala.js [Internet]. Lihaoyi.com. 2014 [cited 22 October 2019]. Available from: http://www.lihaoyi.com/hands-on-scala-js/#WhyNoReflection?

5 Lighthouse | Tools for Web Developers | Google Developers [Internet]. Google Developers. 2019 [cited 22 October 2019]. Available from: https://developers.google.com/web/tools/lighthouse