CovidApp Documentation

1 Introduction

1.1 Overview

This application has been designed with Spring Data REST and its powerful backend functionality, combined with React's sophisticated features to build an easy-to-understand UI. The project is also managed with Maven, to build and launch the whole application with one command. The Spring Data part is designed to be accessed as an external server from the React application, indeed in this case the URL is hardcoded (http://localhost:8080/).

The whole system is launched by running *mvn spring-boot: run* on the console; this will install and run npm and all the plugins required for React automatically. Then spring is initialised, and the server starts running.

During the development of the application, there is no need to rebuild the whole project from the beginning, since there is the webpack plugin which allows to directly refresh the application on the browser, because it watches the bundle.js and recompiles it whenever a change to the React application is made. To run it, just type on the console the following command: *npm run-script watch.*

1.2 Versions, Dependencies, Plugins

With the Spring Data REST structure, the following dependencies were added:

- Rest Repositories
  Expose data repositories over REST via Spring Data REST.
- Thymeleaf
  A server-side Java template engine for web environments. Allows HTML to be correctly displayed in browsers and as static prototypes.
- Spring Data JPA
  Persist data in SQL stores with Java Persistence API using Spring Data.
- H2 Database
  Provides a fast in-memory database that supports JDBC API.

To build a nice interface with React, two plugins were used mainly:

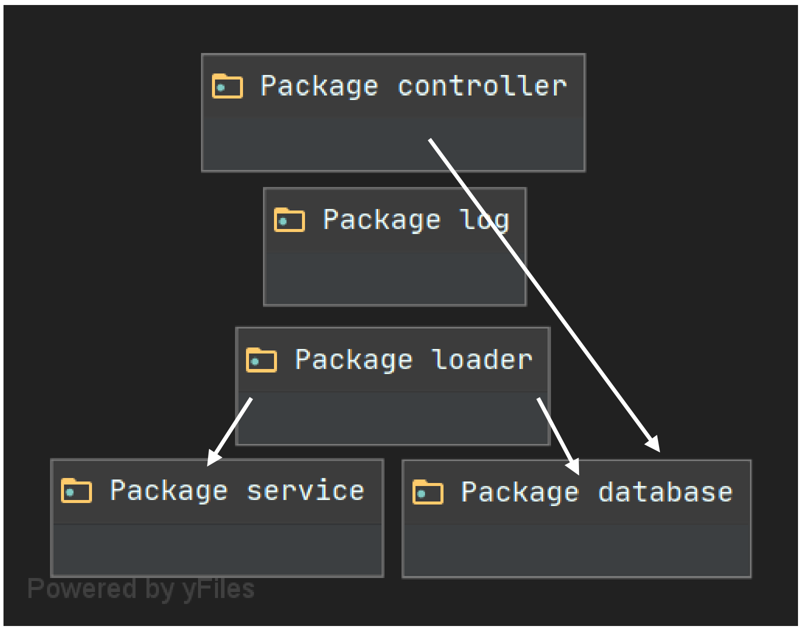
- Material UI
  React components focused on layout and usage (like Google)
- DevExtreme Reactive
  React components based on Material UI focused on data display

Versions:

- Java 13
- Spring Boot 2.3.0
- Node 12.16.3
- Npm 6.14.4
- React 16.5.2

2 Spring Data REST

2.1 Layered Structure



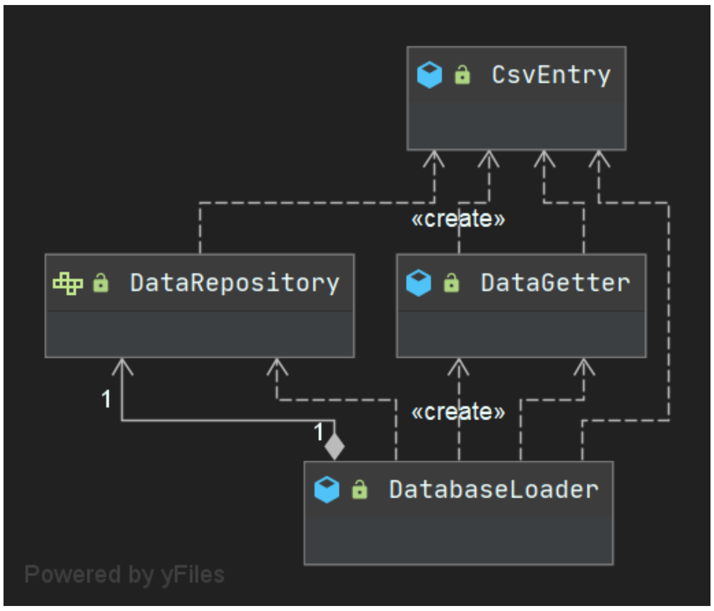
Layered structure of the Server-side application

2.1.1 Database, Service and Loader

In the upper image the structure of the Server-side module is shown. At the bottom we have a database (in this case a service package is also present in order to download the latest data from https://covid.ourworldindata.org/). The DataGetter.java in the service package downloads a CSV stream, parses it and returns a list of CSVEntry, which are the objects that will be store then in the CrudRepository that acts as database. CSVEntry objects are constructed with JPA annotations that are used to denote the whole class for storage in a relational table. The repository can be also changed to a PageAndSortingRepository, which can be helpful to make different and more complex requests since it enables automatic paging and sorting features for the data to be returned.

The loader layer takes the downloaded service data and loads it in the repository. In the repository there are also written some database queries in form of an API. This code is translated automatically by Spring in SQL queries and these APIs are accessed automatically by the controller, which is the only point communicating with the external application. Indeed, these APIs are reached by following the following URI:

../api/search/{*apiName*}?{*parameters*}



Dependencies relation between the repository, its loader and the data downloader

2.1.2 Logger

Since Spring MVC has built-in support for logging requests, it is easy to build a logging layer for the server application. Spring provides CommonsRequestLoggingFilter which can log request URL, body, and other related information. This Bean is a pre-defined filter, which can be customized with some method when the object is created. In order to display the logs in the console where Spring is launched, the log level of the filter was set to DEBUG in application properties.

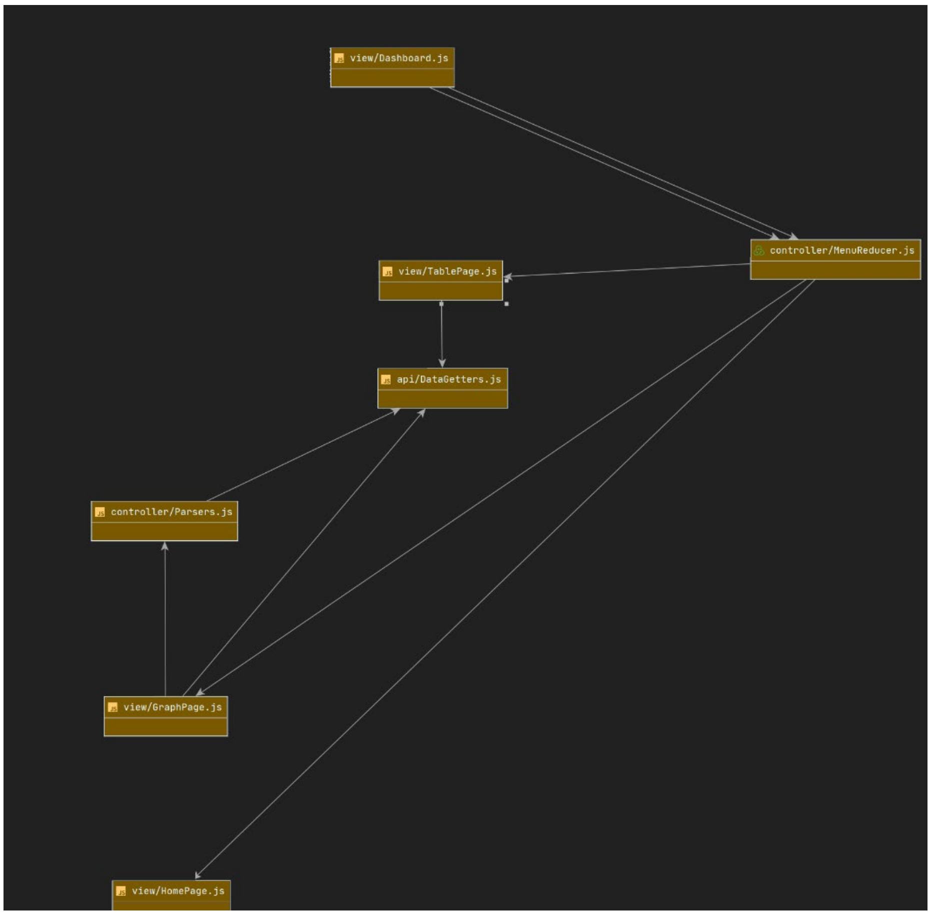
2.1.3 Controller

Spring Boot has an easy way to return the entry point of an application, and it is through an MVC controller. Here all the data requests are controlled based on the URI, and the controller returns something. In this case when http://localhost:8080 is requested, an index.html file is returned, which contains a reference to the bundle that is created by the react application part.

By default, Spring Data REST hosts a root collection of links at /. Because a web UI is hosted on that path, there was the need to change the root URI in the application.properties file with /api.

3 REACT-JS

3.1 LAYERED STRUCTURE



Dependencies of the Javascript modules

3.1.1 API

The API package contains the DataGetter.js file, which is just a collection of get-functions for each type of request. This component makes the requests with the help of the external axios framework, specified in the dependencies in package.json of the application. The functions are then used in TablePage.js and ChartPage.js to load the data inside their React states.

3.1.2 Controller

The main idea of the controller package is to contain the Reducers that manage the use and switch of components based, for example, on the user input, which is also what it has been done in this project. The Dashboard component returns the navigation bars (top and left bars) and based on the buttons that the user clicks, the custom page is loaded through the reducer.

The reducer state is hooked inside the Dashboard component with a default value. When a button is clicked, it dispatches the new value to the reducer which, based on the value given by the button, selects the appropriate component and returns it in the hooked state. Then React re-renders due the state change and the component selected is displayed to the user.

Dashboard.js

How the reducer's state is hooked

```
const [state, dispatch] = useReducer(MenuReducer, initialState);
```

The onClick lambda of the button that dispatches the selected button

```
<ListItem button key={text} autoFocus={index === 0} onClick={() => dispatch({type: text})}>
```

MenuReducer.js

The initial state of the reducer

```
export const initialState = {currentComponent: <HomePage/>}
```

The switch condition based on the button's name

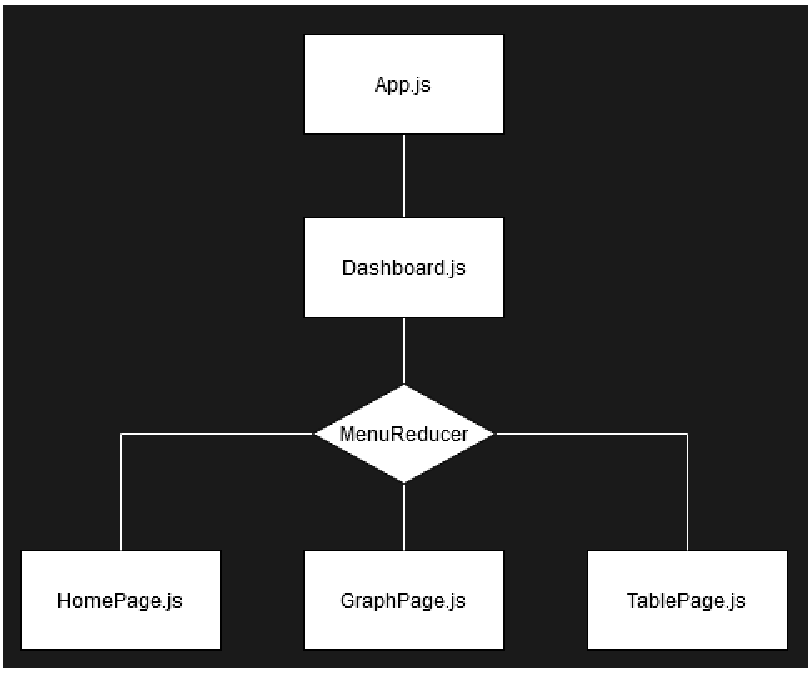
```
switch (action.type) {
  case 'Home':
    return {currentComponent: <HomePage/>};
  case 'Charts':
    return {currentComponent: <GraphPage/>};
  case 'Tables':
    return {currentComponent: <TablePage/>};
  default:
    return initialState.currentComponent;
}
```

3.1.3 View

The View package contains the main UI components of React.

React has a tree hierarchical structure. Therefore, the root component (App.js) is the entry point of the application which is thus characterized by a strong parent-child relationship. The App.js and all its children are compiled in a bundle.js file by React which is then referenced by the Index.html file returned by the server to the client.

To expand and create new functionalities, such as other types of graphs, the idea is then to create a graph component that can then be rendered as a child of GraphPage.js and pass the data, for example, as props or change it through hooks. If there is a component that should be switched with some input or state-change, then another implementation of a reducer would be an effective solution.



Tree-structure of the React application, with App as root node