MESOC Client-side Web Application Architecture in Node.js, Express.js, and React

Kuharić, Valentin

Undergraduate thesis / Završni rad

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Rijeka / Sveučilište u Rijeci**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:195:746996

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-09-13**

Repository / Repozitorij:

Repository of the University of Rijeka, Faculty of Informatics and Digital Technologies - INFORI Repository

University of Rijeka - Department of Informatics

Jednopredmetna informatika

Valentin Kuharić

MESOC Client-side Web Application Architecture in Node.js, Express.js, and React

Bachelor's thesis

Mentor: prof. dr. sc. Sanda Martinčić - Ipšić

Rijeka, June 2021.

ODJEL ZA INFORMATIKU
SVEUČILIŠTE U RIJECI

Rijeka, 8.5.2021.

Zadatak za završni rad

Pristupnik: Valentin Kuharić

Naziv završnog rada: **Arhitektura klijenta web aplikacije u Node.js, Express.js i React-u - primjer MESOC alat**

Naziv završnog rada na eng. jeziku: **MESOC Client-side Web Application Architecture in Node.js, Express.js and React**

Sadržaj zadatka:

Cilj završnog rada je osmišljavanje arhitekture klijentske strane web aplikacije za MESOC Toolkit, te njen razvoj i objavljivanje. Aplikacija će biti realizirana pomoću Node.js javascript runtime okruženja, Express.js okvira te React biblioteke za korisničko sučelje (frontend). Pisani dio završnog rada će uključivati kratko obrazloženje odabira ovih tehnologija te arhitekture i opis procesa razvoja početne stranice MESOC Toolkita u Node.js + Express.js okruženju, te razvijanje korisničkog sučelja (frontend-a) React aplikacije. Kod React aplikacije opisat će se postavljanje općeg prikaza aplikacije te upravljanje korisničkim podacima (registracija, prijava, stanje prijave, itd.).

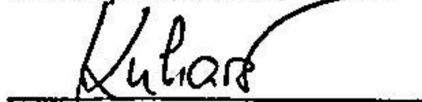
Mentor

Prof. dr. sc. Sanda Martinčić-Ipšić

Voditelj za završne radove

doc. dr. sc. Miran Pobar

Zadatak preuzet: datum



(potpis pristupnika)

// Abstract

This thesis describes the requirements and the thought process of designing the architecture for the frontend part of the web-based application MESOC Toolkit. It also includes the process required to set up and deploy the application. The user management process will also be described in the react application.

Application is written in Javascript using the Node.js runtime environment, enabling us to run javascript code outside of the browser. The application consists of two parts: The Nodejs + Expressjs based server and the React-based SPA (single page application). This design pattern enables us to have the development workflow and power of SPA while having the SEO (search engine optimization) advantages for the homepage.

Keywords: spa, seo, react, nodejs, expressjs, mesoc toolkit, frontend, architecture, javascript, web

Arhitektura klijenta web aplikacije u Node.js, Express.js i React-u - primjer MESOC alat

Sažetak

U ovom završnom radu opisan je proces osmišljavanja arhitekture klijentske strane web aplikacije za MESOC Toolkit, te njen razvoj i objavljivanje. Aplikacija će biti realizirana pomoću Node.js javascript runtime okruženja, Express.js okvira te React biblioteke za korisničko sučelje (frontend).

Pisani dio završnog rada će uključivati kratko obrazloženje odabira ovih tehnologija i arhitekture te opis procesa razvoja početne stranice MESOC Toolkita u Node.js + Express.js okruženju, te razvijanje korisničkog sučelja (frontend-a) React aplikacije. Kod React aplikacije opisat će se postavljanje općeg prikaza aplikacije te upravljanje korisničkim podacima (registracija, prijava, stanje prijave, itd.).

Ključne riječi: spa, seo, react, nodejs, expressjs, mesoc toolkit, frontend, arhitektura, javascript, sučelje, aplikacija, web

Table of Contents

Abstract 1

Sažetak 3

Table of Contents 4

1. Introduction 5

2. Background terminology and basics of web application architectures 6

- 2.1. Basics of HTTP servers and web applications 6
- 2.2. Multi-tier architecture 6
- 2.3. Advantages of multi-tier architectures 7
- 2.4. Microservices architecture 8
- 2.5. Single Page Application 8
- 2.6. Technologies used 8
- 2.7. Search Engine Optimization 9

3. MESOC Toolkit architecture 10

- 3.1. The requirements and challenges 10
- 3.2. The frontend architecture 10
- 3.3. Advantages of MESOC design 11

4. Homepage application 12

- 4.1. How handlebars templating works 12
- 4.2. Comparison of Node.js and ExpressJS 13
- 4.3. Folder structure 14
- 4.4. Index.js 15

5. Single page application 16

- 5.1. Basics of React 16
- 5.2. Folder structure 17
- 5.3. App.js 17
- 5.4. User management 18

6. Deployment process 19

- 6.1. Deployment 19
- 6.2. Configuration and environment variables 19

7. Conclusion 21

Appendix 22

References 23

Table of Figures 24

1. Introduction

The client-server model is a distributed application structure that partitions workloads of a service called servers and service requesters, called clients. Often clients and servers communicate over a computer network on separate hardware (L. Shklar, R. Rosen, 2003). Clients, therefore, initiate communication sessions with servers, which await incoming requests. Examples of computer applications that use the client–server model are email, network printing, and the World Wide Web, etc...

In the client–server model, the server is often designed to operate as a centralized system that serves many clients (Raymond Greenlaw, Ellen M. Hepp, 2003). The computing resources must be scaled appropriately to the expected workload. Many systems like load balancing are employed to enable scaling to more than one physical machine.

The client-server model is a type of tier-two software application architecture, allowing us to separate the presentation layer from the rest of the application and enable us to distribute it over the internet. This approach makes modifications to the application and the data easier, without needing to update the client's application.

The MESOC web application architecture employs the client-server model, expanded as a multi-tier architecture design which makes the developers job easier since different teams depend less on the progress and changes of other parts of the system.

In this thesis, the frontend architecture in Node.js, Express.js, and React is proposed and elaborated. The thesis is structured as follows: Chapter 2 covers the basics of web application architecture and different ways to structure a system, Chapter 3 covers the actual MESOC architecture, the advantages and disadvantages of the chosen approach, Chapter 4 explains the Express homepage application, the folder structure, packages and practices used, Chapter 5 covers the React single page application, how React works, the folder structure, routing, components and user management, Chapter 6 shows the deployment process of both apps and how to configure them, and Chapter 7 sums up the benefits and the tradeoffs made with this approach, together with the source code and future possible improvements.

2. Background terminology and basics of web application architectures

This chapter aims to explain the similarities and differences between terms used in web application design patterns. Also, some of the most common designs will be mentioned.

2.1. Basics of HTTP servers and web applications

The term web server can refer to hardware or software, or both of them working together.

1. Hardware side - a web server is a computer that stores web server software and all website's components. A web server is connected to the Internet and supports physical data interchange with other devices connected to the web.
2. Software side - a web server is a computer program (one or more parts) that controls how web users access hosted resources. At a minimum, this is an HTTP server. An HTTP server is software that understands URLs (Unified Resource Locator) and the HTTP protocol (Hyper-Text Transfer Protocol) (MDN Web Docs, 2021). An HTTP server takes incoming requests and responds with the requested resources and it can be accessed via the domain name it corresponds to.

A static web server consists of a computer machine connected to the internet (hardware) with an HTTP server (software). We call it "static" because the server sends its hosted files without modifications to the client. This is the most simple example of a two-tier software application design for the web as represented in Figure 1.

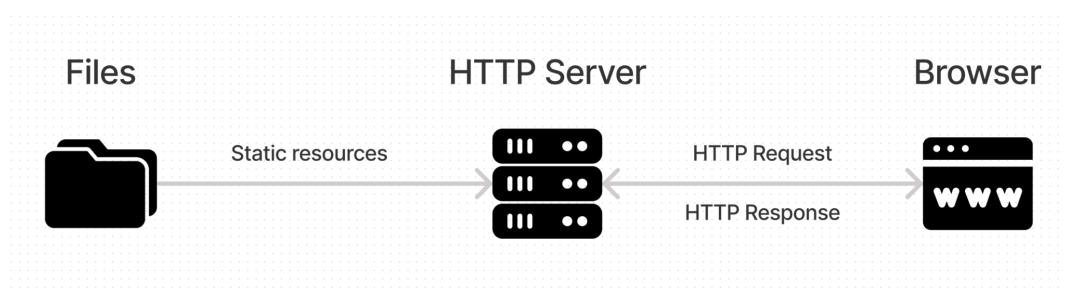


Figure 1: Diagram showing a basic request-response exchange between a client and a server (HTTP server)

A dynamic web server consists of a static web server paired with some other software, most commonly an application server and a database. We call it "dynamic" because the application server updates the hosted files before sending content to your browser via the HTTP server as shown in Figure 2.

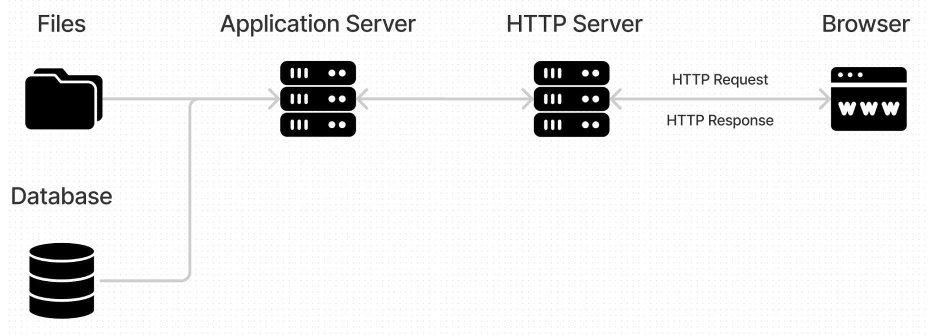


Figure 2: Diagram showing a dynamic web server paired with a database

2.2. Multi-tier architecture

The approach of splitting an application into tiers can be taken further into multi-tier architecture as shown in Figure 3. In multi-tier architecture, both parts of the client-server model can be further subdivided to improve the application if it is appropriate (The Open University, 2020).

1. The client - the client-side can be subdivided into more parts, for example; to process some data received from the server and to present the information
2. The server - the server software might include a database, paired with the web server program. Usually, when adding the database (located in the data tier) a middleware program needs to be added (located in the middle tier). Middleware programs are applications that handle the business logic, and they are located between the server tier and the database tier. An application that uses middleware to handle data requests between a user and a database is said to employ a multi-tier architecture. Note that a database can be further subdivided into two or more, enabling easier maintenance and upgradability.

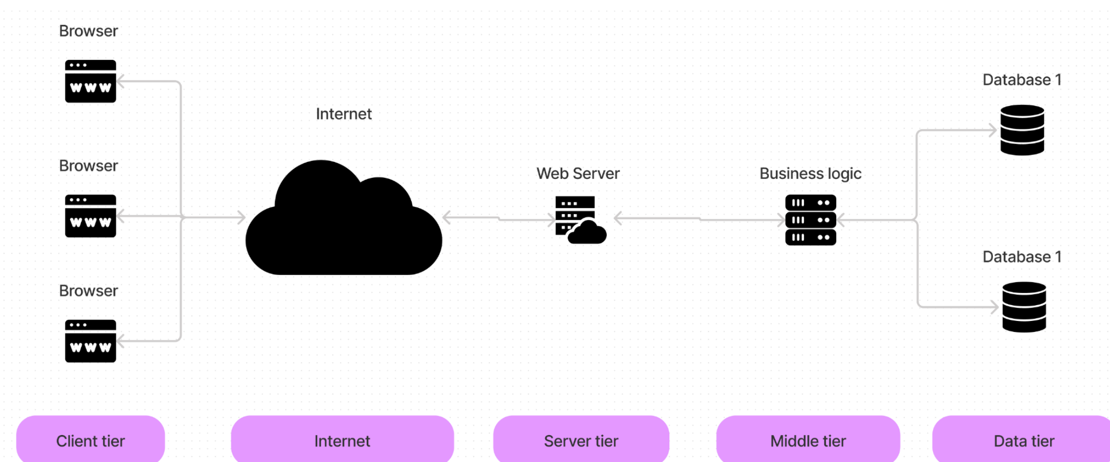


Figure 3: Diagram of tiers in a multi-tier architecture

2.3. Advantages of multi-tier architectures

As in the two-tier approach, there are advantages to breaking down the application into multiple tiers (The Open University, 2020). Each tier can be changed more easily since it depends less on other parts of the system. This again depends on how careful we are to adopt an approach that ensures undependability from all perspectives. Simply breaking the application into chunks doesn't

guarantee this; we also need to adopt suitable standards and specify precise and limited interactions between the tiers. N-tier architecture is for many medium to large systems de-facto a standard practice. Many more clients besides web browsers are now available that can realistically be interchanged without prohibitive effort, including databases and web servers.

2.4. Microservices architecture

Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of services that are (Chris Richardson, 2020):

- Highly maintainable and testable,
- Loosely coupled,
- Independently deployable,
- Organized around business capabilities.

The microservice architecture enables the rapid, frequent and reliable delivery of large, complex applications. It also enables an organization to evolve its technology stack easily over time, since all parts can be completely replaced by a different codebase, as long as the communication protocol stays the same.

A microservice is not a layer within a monolithic application. Rather, it is a self-contained piece of business functionality with clear interfaces and use cases.

2.5. Single Page Application

A SPA (Single-page application) is a web application implementation that loads only a single web document, and then updates the content of that single document when different content is to be shown (MDN Web Docs, 2021).

This, therefore, allows users to use websites without loading whole new pages when content is to change, which can result in performance gains and a more dynamic user experience, with some tradeoffs such as search engine optimization and complicated navigation (compared to a more traditional approach). Some of the most popular SPA frameworks are React, Angular and Vue.JS.

Since content gets rendered dynamically on the client-side, search engine crawlers have a hard time seeing content on a single page, so they can’t index the site properly.

2.6. Technologies used

Node.js is an open-source, cross-platform, back-end JavaScript runtime environment that runs on the V8 engine and executes JavaScript code outside a web browser (Node.js, 2021). Node.js enables writing command-line tools and for server-side-scripting - running scripts server-side to produce dynamic webpage content before the page is sent to the client’s web browser (an example of a dynamic web server).

When developing web applications in Node.js is de-facto standard framework developers use is ExpressJS. ExpressJS is a Node.js framework that simplifies writing code for serving routes. ExpressJS provides a thin layer of fundamental web application features (ExpressJS, 2020).

2.7. Search Engine Optimization

Search engine optimization is a process of improving the ranking of a website on search engines such as Google, Bing, Yahoo etc.

Search engine crawlers, or more commonly named “bots”, collect information from websites and store them in an index. Algorithms analyze that data and they rank websites according to some predetermined factors (MOZ, 2021). Some of the most important factors in search engine rankings are keywords and HTML semantic elements.

To control the indexing of a website, there are rules one can write in a *robots.txt* file which crawlers follow, that specifies which parts of the website they can index, and which they are forbidden to index. It’s important to note that this doesn’t affect the visibility of these pages.

3. MESOC Toolkit architecture

3.1. The requirements and challenges

MESOC Toolkit has a few requirements:

- Be deployed on the web,
- Have a seamless user experience,
- Have great SEO ranking,
- Work with the MESOC API and the MESOC Repository.

Therefore, the frontend had to be structured and technologies had to be used in such a way to achieve these requirements. Since the MESOC API and MESOC repository already existed, a monolithic approach was not an option.

Also, the complexity of the application and its rich feature set meant that a multi-tier architecture was the right approach.

3.2. The frontend architecture

The frontend architecture of the MESOC Toolkit application consists of two programs: the ExpressJS server and the single page application written in React as shown in Figure 4.

The ExpressJS server is used for serving HTML files with little complexity, such as the homepage, the terms and services, and the privacy page. These pages need to have a good SEO ranking, so serving them via a simple ExpressJS server as static files enables us to do so. All other pages are handled by the React single page application.

The two applications work in tandem, providing a seamless user experience connected via URL redirects.

The React SPA interfaces with the MapBox API (Mapbox, 2021) on the “/browse” page. The MapBox API provides the necessary map tiles to display the map.

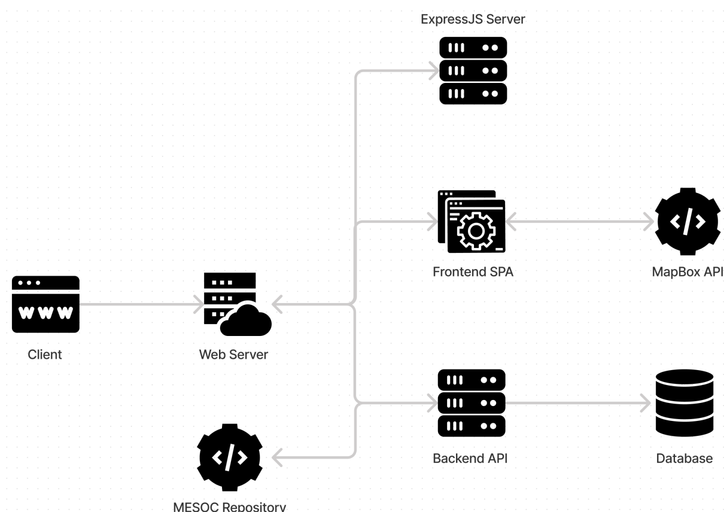


Figure 4: Diagram showing the MESOC Toolkit web architecture

3.3. Advantages of MESOC design

The benefit of having two applications is enjoying advantages from both approaches. ExpressJS serves static files that can be easily indexed on search engines while writing all business logic in a SPA package enables an easier development process than a more traditional approach, and the client (user) benefits from a more seamless experience powered by a single page load combined with better overall performance.

4. Homepage application

The homepage server is written using the Node.js runtime environment, giving us the ability to run Javascript code outside of the browser. Using it we run an HTTP server. To simplify the development process and to shorten the development time, the ExpressJS framework is used. Another important package is Handlebars (express-handlebars, 2020), which is a templating engine enabling us to reuse HTML templates and import specific data into them based on the client's request.

4.1. How handlebars templating works

Written in JavaScript, Handlebars.js is a compiler that takes any HTML and Handlebars expression and compiles it down to a JavaScript function. This derived JavaScript function takes one parameter, an object (some data) and returns a string with the HTML and the object properties' values inserted into the HTML. The result is a string that has the values from the object properties inserted in the relevant places that gets inserted as HTML onto the page.

The three main parts of Handlebars templating:

1. Handlebars.js expressions - a simple Handlebars expression is written like this, where content can be a variable or a helper function (with or without parameters):

   ```
   {{ content }}
   ```

2. Data object - The second piece of code in Handlebars templating is the data we want to display on the page. We pass the data as an object (a regular JavaScript object) to the Handlebars function. The data object is called the context. This object can be composed of arrays, strings, numbers, other objects, or a combination of these.

   If the data object has an array of objects, we can use Handlebars *each* helper function to iterate the array, and the current context is set to each item in the array.

   ```
   //The customers object has an array of objects that we will pass to Handlebars:

   var theData = {customers:[{firstName:"Michael", lastName:"Alexander", age:20}, {firstName:"John", lastName:"Allen", age:29}]};
   ```

3. The compile function - The handlebars *compile()* function takes the template as a parameter and it returns a JavaScript function. We then use this compiled function to execute the data object and return a string with HTML and the interpolated object values. Then we can insert the string into the HTML page.

```
var theTemplate = Handlebars.compile (theTemplateScript);

// Returns this: $(document.body).append (theTemplate (theData));

Figure 5
```

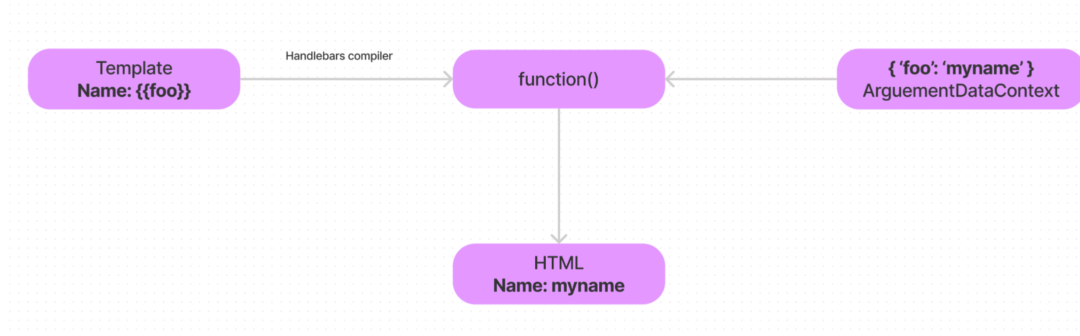


Figure 5: Diagram showing the flow of data and data transformation of Handlebars (simple)

Templating endpoints gives us higher customization and makes potential changes in the future easier.

4.2. Comparison of Node.js and ExpressJS

ExpressJS simplifies the coding process by providing controller, middleware and routing functionalities. To compare the two workflows, the same functionality of starting an HTTP server and creating an endpoint that will return an HTML file with the text "Hello from server" is written below, in Node and Express (GeeksForGeeks, 2020):

1. Nodejs

```
const http = require('http');

const server = http.createServer((req, res) => {

    res.setHeader('Content-Type', 'text/html');

    res.write('<html>');

    res.write('<head><title>GeeksforGeeks</title><head>');

    res.write('<body><h2>Hello from Node.js server!!</h2></body>');

    res.write('</html>');

    res.end();

});

server.listen(3000, ()=> {

    console.log("Server listening on port 3000")

});
```

2. ExpressJS

```
const express = require('express');

const app = express();

app.get('/', (req, res) => {

    res.send('<h2>Hello from Express.js server!!</h2>');

});

app.listen(8080, () => {

    console.log('server listening on port 8080');

});
```

4.3. Folder structure

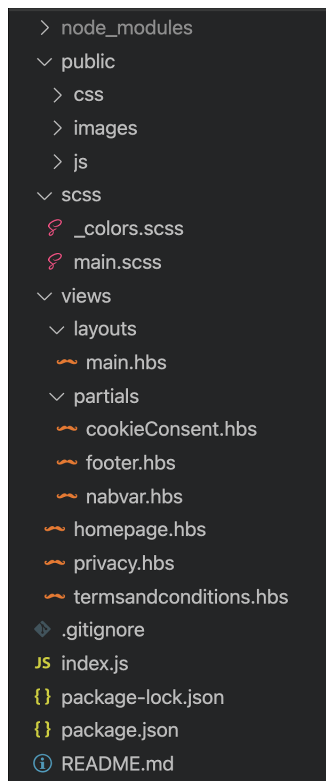


Figure 6: Express application folder and file structure

To maintain code readability and an organized structure, code has been separated into smaller files and all files have been placed in their respective folders as shown in Figure 6.

Since this project is based on Node.js, the *package.json* file that is generated by npm (node package manager) stores the project's metadata, including name, author, version, dependencies, scripts and more. The *package-lock.json* file is automatically generated for any operations where npm modifies the *node_modules* folder (where dependency packages get installed) or the *package.json* file.

The *public* folder stores all files that can be accessed directly via the URL. It contains folders *css*, *images* and *js*. For example, if we wanted to access the file *file1.js* inside the *js* folder, the URL will look like this: *schema:domainname:port/public/js/file1.js.*

The *scss* folder stores *.scss* files containing styling instructions for the HTML content. SCSS later gets compiled down to CSS, which can be found in the *public/css* folder.

The *views* folder stores all the *.hbs* files that handlebars uses to generate the final HTML file that gets sent to the client. The *layouts* folder stores handlebars data that is the base for the final content. *Partials* folder stores small parts of HTML that can be injected into the final file (navigation bars, footers etc.) - we can simply include the partial into the final file using one line of code (like this `{{> nabvar}}`), as opposed to writing the whole code snippet every time. In the *views* folder, we store the *.hbs* files that are the actual page content (*homepage.hbs,* t*ermsandconditions.hbs* and *privacy.hbs*).

README.md is a markdown file that Github uses to display a project's documentation.

The file *.gitignore* stores instructions that tell the Git versioning control system MESOC uses which files to ignore when committing changes.

4.4. Index.js

Index.js is the root file of the project and it starts the application.

At the top of the file we include the necessary packages.

Then we load the configuration settings from the configuration file (will be explained in the deployment section).

In case the port isn't specified in the configuration file, we make a constant and set it to 4001.

On line 11 we put the new express application into the app variable.

On line 13 we set the *public* folder as static, making its content available via the URL.

On line 15 we use the *cookie-parse*r package for parsing the Cookie header.

On line 17 through 20 we set the application to use the handlebars engine and provide the necessary configuration.

On line 22 we set the view engine setting the value of *.hbs*.

On line 24 we find a middleware function called *getCookieConsent*. This function runs before every endpoint, checks if the *mesoc_cca* cookie exists and passes it down to the next function.

On lines 34, 45, 53 we find the endpoints of the express application. These functions called route methods or route functions are attached to the app. All three endpoints use the GET method, followed by the route path, then the *getCookieConsent* middleware and then the actual function. All three functions serve the same purpose - render the HTML content and respond to the request with

the content. Function *res.render()* takes as parameters the name of the *.hbs* file that stores the content and an object in which we provide the values of the variables mentioned in the *.hbs* files.

Lastly, on line 62 we start the HTTP server with *app.listen()* function, which takes in as parameters a port value and an optional callback function, which we use to tell the administrator the server is successfully running by printing to the console "Server is running on port 4001".

5. Single page application

The single-page application is written in React (React, 2021) and it handles all the business logic of the frontend part of the MESOC Toolkit. React is a JavaScript library for building user interfaces and frontend applications. React itself is connected with state management and rendering that state to the DOM (Document Object Model), so additional packages and libraries were used to expand the application's feature-set.

5.1. Basics of React

The standard application programming interface for HTML documents is the Document Object Model (DOM), which defines the logical structure of documents and the way a document is accessed and manipulated. When working on client-side applications the DOM starts getting complicated and slow, so React implements a virtual DOM that is basically a DOM tree representation in JavaScript. Unlike manipulating the DOM directly, React elements are computationally cheap to create, so React manipulates its virtual DOM, and then tries to find the most efficient way to update the DOM based on the virtual model (GeeksForGeeks, 2021).

React has a few different kinds of components, but for simplicity the *React.Component* subclass will be explained.

```
class ShoppingList extends React.Component {
  render() {
    return (
      <div className="shopping-list">
        <h1>Shopping List for {this.props.name}</h1>
        <ul>
          <li>Instagram</li>
          <li>WhatsApp</li>
          <li>Oculus</li>
        </ul>
      </div>
    );
  }
}

// Example usage: <ShoppingList name="Reactname" />
```

React updates the components (and the DOM) every time the data (application state) changes. Here, *ShoppingList* is a React component class, which takes in parameters called *props* (short for "properties") and returns a hierarchy of views to display via the *render()* method.

The render method returns a description of what should be displayed on the screen. React takes that description, which is a React element - a lightweight description of what to render - and manipulates the DOM to match the description. To make the process of describing easier, React uses the JSX syntax which makes these structures easier to write. JSX combines HTML and JavaScript into a format easy to write and read.

In the example above we can see a way to use and render a prop to the screen with *{this.props.name}* which we provided in the commented example usage with *name="Reactname"*.

5.2. Folder structure

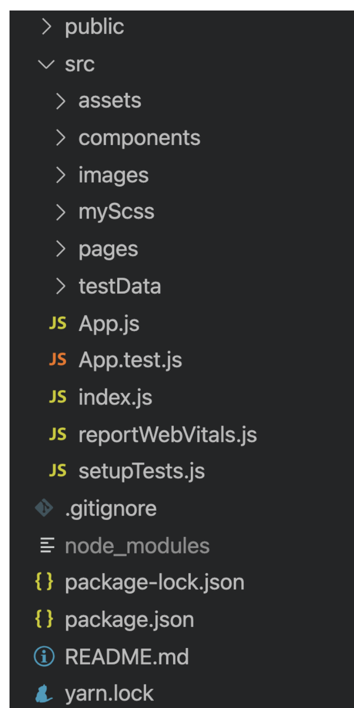


Figure 7: React SPA folder and file structure

The *package.json* and *package-lock.json* files are generated by the node package manager (npm) and serve the same purpose as the files in the express app project with the same name. The file *yarn.lock* serves the same purpose as the *package-lock.json* file but gets automatically created by the *create-react-app* toolchain. Yarn is a substitute for npm, and they work almost identically. In the MESOC SPA npm was used.

README.md is a markdown file that Github uses to display a project's documentation.

The *public* folder hosts the main *index.html* file and related files like a favicon, *robots.txt* etc.

The *src* folder contains the React *App.js* file, which is the main root container for all other react components. In *index.js* main react component *App* from *App.js* gets rendered inside an HTML DOM element called with id of *root*.

Other folders are self-explanatory as shown in Figure 7. Folder *assets* holds images, fonts and stylings (from a library), *components* hold react components, *images* holds main svgs and pngs, *pages* holds react components that represent different views e.g. pages, *myScss* holds more stylings and *testData* holds dummy data for testing in JSON format.

5.3. App.js

As previously mentioned, in *index.js* the React gets rendered into the *index.html* file inside a div element with an id of *root*. But the core of the React application itself is the *App.js* file - the main component that mounts all other components.

In MESOC the file *App.js*, other than being the main component, serves two purposes: holding main app state data, and having routing functionality for the whole app.

In the beginning of the file we import all libraries and components we need. Then, in the *App()* function, we use the hooks API to create variables to store global application data, such as the user token, is the user verified and more. Using the react-cookie package we can easily read and store cookie data containing user information and read it into application state and then use it.

```
const [userToken, setUserToken] = useState(null);
```

After managing the application state we use the use effect hook to run a piece of code when the component mounts and at every update. In our case, we use it to read the browser cookie and update the application accordingly.

In the *return()* statement return multiple elements inside the *React.Fragment*. Inside, we check if the application is ready with the *appReady* variable and if it is, we return a *div* inside which we specify all the routes the application will have. Firstly we specify the root route which always loads the navbar, and then we list all other routes inside the *Switch* tags that render only one route that matches the URL at a time. For each route, we specify the URL suffix for that route and the component it will load. We also provide specific props to components that need them. To finish, we render the loading animation until the components load completely.

5.4. User management

The MESOC toolkit application differentiates three user types: unregistered, logged-in but unverified and logged-in and verified.

A user can interact with the MESOC toolkit even if unregistered. Such a user has access to the MESOC map and he can analyze the data on a city level.

The user's account lifecycle starts with creating an account. By providing an email and a password, a user's account will be created and he will be able to log in. At that point, his privileges and power don't change until he verifies his account. Upon registration, an email will be sent to the user's email address containing a verification link with a confirmation button. After verifying his account,

he gains access to his personal document repository where he can upload and analyze his own documents with respect to the MESOC matrix and other MESOC documents.

6. Deployment process

6.1. Deployment

To deploy the frontend part of the MESOC toolkit we need to deploy the two applications mentioned above - the express server and the react SPA. MESOC toolkit uses the Nginx web server (Nginx, 2020), although alternatives can be used. Both codebases are stored on github in their respective repositories.

1. The express server - first step is to clone the git repository to the machine that will host the application stack. Navigating inside the folder, we run the command in the terminal *npm install* to install the necessary dependencies the application needs to work. Then, we need to run the node (express) application locally on a specific port and keep it running. All that remains is to route all incoming requests that match a specific subdomain to the app in a reverse proxy configuration using an Nginx config file.

```
location / {
          root           /var/www/mesoc-hp;
          proxy_pass http://localhost:4001;
          proxy_http_version 1.1;
          proxy_set_header Upgrade $http_upgrade;
          proxy_set_header Connection 'upgrade';
          proxy_set_header Host $host;
          proxy_cache_bypass $http_upgrade;
  }
```

2. React SPA - first step is to clone the git repository to the machine that will host the application stack. Navigating inside the folder, we run the command in the terminal *npm install* to install the necessary dependencies the application needs to work. Then, we need to build the application with the command *npm run build*. This command will generate a *build* folder containing a production build of the application. All that remains is to route all incoming requests for a specific subdomain to the application. React generates static files so no process needs to run in the background.

```
location / {

    #root /var/www/mesoc-frontend/build;

    root /var/www/mesoc-frontend/build;

    try_files $uri $uri/ /index.html =404;

  }
```

6.2. Configuration and environment variables

Both applications require minor configuration to work. The express app needs a *PORT* and a *WEBAPP* variable, which points to the subdomain the SPA is located, and the SPA needs *REACT_APP_MAPBOX_STYLE* and *REACT_APP_MAPBOX_ACCESS_TOKEN,* which point to the mapbox map link and the user token. Applications use the *dotenv* package (dotenv, 2020) for easier configuration.

Using dotenv, we create a new file with extension *.env* that isn't a part of the codebase. This file will contain application secrets and configuration data, such as database logins, API tokens etc. These secrets are written in a key-value format. An example is shown below.

```
SECRET_VARAIBLE=secretValue
```

To use dotenv in the express app we use it by including by importing the package and using the *dotenv.config()* function that takes as a parameter the path to the *.env* file, while in the SPA dotenv gets configured automatically by the *create-react-app* toolchain.

When it comes to updating the applications when changing the *.env* files the express app needs to be restarted and the SPA needs to be rebuilt since it builds static files.

As previously mentioned the react SPA uses mapbox API to receive map tiles to display on the MESOC map. To get the necessary mapbox user token and map URL an administrator needs to register on the Mapbox web application, duplicate the MESOC map from the public link and fill the *.env* file with his user token and map link.

7. Conclusion

In this thesis we have laid out the basics of web applications and JavaScript web technologies, we have discussed the MESOC toolkit frontend architecture, its shortcomings and advantages, and how the applications work. We have shown how to configure the application stack and deploy it.

Much more can be said about the technologies and design patterns used. Considering the requirements of the MESOC toolkit, exploring different approaches and principles would be worthwhile. Also, there are numerous situations where the code can be written in a simpler form, making maintenance and further development easier. One functionality this project currently lacks is unit testing. Unit testing wasn't introduced to the MESOC toolkit due to development deadlines, but this is something every project needs, even web applications of this scale. In that case, one would use packages such as Mocha (Mocha, 2021) to write unit tests and create an easy npm command to run them inside the *package.json* file.

At this point, the application stack is production-ready and can be deployed and used commercially, even though it lacks polish in some areas. Future work includes the polishing mentioned before and improving the user experience in the form of feedback forms. The document pipeline and data presentation is still in their early stage, so further testing is required in the form of different presentations of data.

Appendix

Appendix includes a CD-ROM containing a copy of the web application featured in the thesis and the accompanying deployment and configuration instructions.

References

L. Shklar, R. Rosen (2003). Web Application Architecture: Principles, protocols and practices. Retrieved from
http://bedford-computing.co.uk/learning/wp-content/uploads/2016/07/Web-Application-Architecture-Principles-Protocols-and-Practices.pdf

Raymond Greenlaw, Ellen M. Hepp (2003). Encyclopedia of Information Systems. Retrieved from https://www.sciencedirect.com/topics/computer-science/client-server-model

The Open University, (2020). An introduction to web applications architecture. Retrieved from

https://www.open.edu/openlearn/science-maths-technology/introduction-web-applications-architecture/content-section-1.2

IBM Cloud Learn hub. Three-Tier Architecture. Retrieved from

https://www.ibm.com/cloud/learn/three-tier-architecture#:~:text=Three%2Dtier%20architecture%20is%20a,associated%20with%20the%20application%20is

MDN Web Docs, (2021). What is a web server? Retrieved from

https://developer.mozilla.org/en-US/docs/Learn/Common_questions/What_is_a_web_server

MOZ, (2021). How search engines work: crawling, indexing and ranking. Retrieved from

https://moz.com/beginners-guide-to-seo/how-search-engines-operate

Chris Richardson, (2020). Microservice architecture. Retrieved from

https://microservices.io/

MDN Web Docs, (2021). SPA (Single-page application). Retrieved from

https://developer.mozilla.org/en-US/docs/Glossary/SPA

GeeksForGeeks, (2021). React.js (Introduction and Working). Retrieved from

https://www.geeksforgeeks.org/react-js-introduction-working/

React, (2021). Tutorial: Intro to React. Retrieved from

https://reactjs.org/tutorial/tutorial.html

CreateReactApp toolkit documentation. Retrieved from

https://create-react-app.dev/

Mocha JavaScript testing framework, (2021). Retrieved from

https://mochajs.org/

GeeksForGeeks, (2020). Node.js vs Express.js. Retrieved from

https://www.geeksforgeeks.org/node-js-vs-express-js/

Node.js (Version 14.17.1) (2021). Retrieved from https://nodejs.dev/

ExpressJS (Version 4.17.1) (2020). Retrieved from https://www.npmjs.com/package/express

React (Version 17.0.1) (2020). Retrieved from https://reactjs.org/

express-handlebars (Version 5.2.0) (2020). Retrieved from https://www.npmjs.com/package/express-handlebars

dotenv (Version 8.2.0) (2020) Retrieved from https://www.npmjs.com/package/dotenv

Nginx (Version 1.18.0) (2020). Retrieved from https://www.nginx.com/

Mapbox, (2021). Retrieved from https://docs.mapbox.com/

Table of Figures

Figure 1: Diagram showing a basic request-response exchange between a client and a server (HTTP server) 7

Figure 2: Diagram showing a dynamic web server paired with a database 7

Figure 3: Diagram of tiers in a multi-tier architecture 8

Figure 4: Diagram showing the MESOC Toolkit web architecture 11

Figure 5: Diagram showing the flow of data and data transformation of Handlebars

(simple) 14

Figure 6: Express application folder and file structure 15

Figure 7: React SPA folder and file structure 18