MERN stack application for student lab management

Master Thesis

Vincent Glauser

April 2025

Thesis supervisors:

Prof. Dr. Jacques Pasquier-Rocha
and
Quentin Nater
Software Engineering Group

Software Engineering Group
Department of Informatics
University of Fribourg (Switzerland)

Acknowledgements

I would also like to thank all the people who helped me with this project. Thanks to Prof. Dr. Jacques Pasquier-Rocha and Quentin Nater, I was able to explore a completely new kind of work. I would also like to thank my colleagues and my family for their support throughout this time.

Abstract

The modern approach to structuring web services is to use a REST architecture. We are using a web development MERN stack to implement a management system for a student lab. Adding the component to share documents via this platform simplifies the correction of lab work and saves time for organisers and participants.

Keywords: Web Service, REST, MERN

Table of Contents

List of Figures

1 Introduction

1.1. Motivation and Goals

Web development stacks are used for the client-server model. The motivation is to add some functionality to a web service. We will use a RESTful service with a MERN stack. Before we start with the content, we give an overview for the organisation of the work, notations and conventions.

1.2. Organization

Chapter 2: Theory

This chapter briefly introduces definitions and terminology. These key concepts have been used throughout this project. First we define URI and HTTP. Then we introduce the terms API and REST API. Next, we explain the client-server model with the DBMS and web development stacks. Finally, we describe use cases.

Chapter 3: Project

The subject is to define the inspiration and idea of this project. The use cases and the entity relation model are presented and the tasks and functionalities of the different parts are defined. The choice of the stack is explained and the parts of the stack with the software used for the project are given.

Chapter 4: Developer details

In this chapter we explain the folder structure, the libraries used in the backend and the frontend. We take a closer look at the database, server and client and present small snippets of code to understand the implementation.

Chapter 5: Results

This chapter presents the project from a user's point of view. We show pictures of the website and explain the workflow. We have also included images of the database collections for the developer.

Chapter 6: Discussion

In this section, we present, analyse and group the problems we encountered during the project and their solutions.

Chapter 7: Conclusion

The last chapter concludes the project with a review of the work done. We give an outlook and final statements for the project.

Appendix

Contains extracts of artefacts or service messages, abbreviations and references used throughout this work.

1.3. Notations and Conventions

- Formatting conventions:
  - Abbreviations and acronyms as follows Application Programming Interface (API) for the first usage and API for any further usage;
  - `http://localhost:3000` is used for web addresses;
  - Code is formatted as follows:

```
public double division(int _x, int _y) {
    double result;
    result = _x / _y;
    return result;
}
```

- The work is divided into seven chapters that are formatted in sections and subsections. Every section or subsection is organized into paragraphs, signalling logical breaks.
- Figure s, Table s and Listings s are numbered inside a chapter. For example, a reference to Figure j of Chapter i will be noted *Figure i.j.*
- As far as gender is concerned, I systematically select the masculine form due to simplicity. Both genders are meant equally.

2
Theory

2.1. URI: Uniform Resource Identifier

The Uniform Resource Identifier (URI) assigns to each web document a syntax with a unique address. An example of an URI is *http://www.theory.com.* All the different web-based concepts are resources and the URI is the unique identifier to get the address of these resources. [10]

Uniform Resource Locator (URL) is a specific type of URI that tells us how to get that resource. An example of an URL is *http://www.theory.com:8080/chapter1/file1?title=School& subtitle=Work# section1.* The general form of an URL is *Scheme://Host:Port/Path/ QueryParameters/FragmentIdentifier* but only the *Scheme://Host* is required. [2]

2.2. HTTP: Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) is a language used by different computers to communicate over the Internet. [10]

HTTP takes a document and puts an envelope around it. This packet is sent to the server, which will receives it and sends back a response document. The envelope structure is strict but the document content is not. [16]

The HTTP method indicates the information about what type of request the packet is. The five most common methods are GET, POST, PUT, PATCH and DELETE. The GET asks the server for a particular representation of a resource. POST sends data to create a new resource. The PUT method updates an existing resource. PATCH also updates the resource but only requires specific changes, whereas PUT updates the resource completely. The DELETE method deletes the resource from a specific URL. [1]

2.3. API: Application Programming Interface

An application programming interface (API) is an interface for a better communication between user and a computer program. The API presents data and functions in a simpler way so that information can be easily exchanged. [10]

2.4. REST: Representational State Transfer

The Representational State Transfer (REST) is an architectural style for designing APIs for web services. An API that follows the rules of REST is called RESTful. We give some of the rules for the REST API URIs:

- Use '/' for hierarchical relationships
- The last character should not be a '/'
- Use '-' instead of '_' to join names
- Use lower case characters
- Do not add file extensions
- Document names are singular
- Collection or store names are plural

There are other rules, such as HTTP methods (GET, POST, etc.), response status codes (200-204, 301-307, etc.), HTTP headers (type, length, etc.) and many more. [10]

2.5. Client-Server Model

The client-server model is an architecture that provides the communication between clients, such as a web browser like Firefox, and a server. The client sends requests and the server processes and responds to the client. We call the client part *frontend* and the server part *backend*.

The advantage of this separation is that we can handle multiple clients simultaneously and centralise the data management. This architecture also allows us to scale the system, so we can add clients or servers to handle the workload. Another advantage is data security, as we can protect sensitive information from different users. The server controls what data is accessible to incoming requests.

The disadvantages are that the server is now a single point of failure. If the communication between the client and the server is poor, we will have performance problems. Costs and resources are increase when a server is used for multiple clients. [7]

2.6. DBMS: Database Management Systems

To store huge amounts of data in a structured way, we need Database Management Systems (DBMS). The database is stored at somewhere on the computer and the DBMS is a software that surrounds it and applies specific rules to the data and describes the relationships between them. A DBMS can manage more than one database.

There are different types of DBMS, such as relational, hierarchical, network, object-oriented or NoSQL database systems. Some examples of DBMS are Oracle, MySQL, SQL Server, SQLite, DB2 or MongoDB. [4]

2.7. Web Development Stacks

A web development stack is a set of different tools for building web applications. Examples of stacks are MERN, MEAN and MEVN. These acronyms represent the different technologies: MongoDB (M), Express (E), React (R), Node (N), Angluar (A) and Vue (V). Another example is LAMP, which stands for Linux (L), Apache (A), MySQL (M) and PHP (P). These technology stacks work well together and may have different programming languages. Each stack has it's advantages and disadvantages and the different parts can be swapped. For example, we swapped React with Angular for the MERN stack and got the MEAN stack. [6]

2.8. Use Cases

The methodology for describing the function of a system is a use case. The requirements help to identify the interactions of a user with the system. The three important elements are that there is an actor, a system and a goal. [3]

2.9. ERM: Entity Relation Model

The Entity Relation Model (ERM) is a model used for the logical structure of a database. We represent the entities by rectangles and ellipses for attributes of the entities. The diamond describes the relationship and the line connects the entities to the relations. [8]

3
Project

3.1. Project inspiration

At the University of Fribourg, all physics majors and minors have to complete the laboratory experiments *APLabs* in their first year. There are 19 labs in total and the students have to complete 18 of them. Groups of two students are formed. The students have to read a document before the corresponding lab at home, work in the lab for less than four hours. At home they write a report and send it back to the assistant. This report will be corrected and the students will have to correct the report. When there are no more errors in the report, the lab is considered completed. After the 18 labs, the APLabs is finished and the ECTS credits are transferred to the student.

The website *aplabs-physics.unifr.ch* manages all students and labs. It is possible to register and log in to a personal account. First you have to create a group with another student. Then the possible labs are displayed and the group can register for a specific lab. It is possible to download the documents for the labs, but it is not possible to upload files. All reports and corrections must be printed and returned to the Physics Department building. The website also contains rules, information and an analysis tool to create a scatter plot.

In the year 2020 & 2021 there were restrictions to be able to attend the labs because of the COVID-19 pandemic. Online labs were created, but reports and corrections were still printed.

3.2. Project idea

This project is about creating a client-server model. The idea was to extend the functionality of the website to include the ability to upload and download student reports and assistant corrections in the form of PDF files. The project was to start with nothing and build more and more functionality, but the focus was on file management for the users. First, we had to create a schema to identify several components before writing the code. Use cases and ERMs were created to define the project.

3.3. Use Cases

The student, the assistant and the administrator are the users. The system manages the profile (registration, login and password), the lab (create, register, withdraw, view, modify, delete, credits), the report (add, download and delete) and the mark (add, download, delete, accept, view). The use cases 3.1 show how users interact with the system. The aim is to separate the tasks and access of different users.

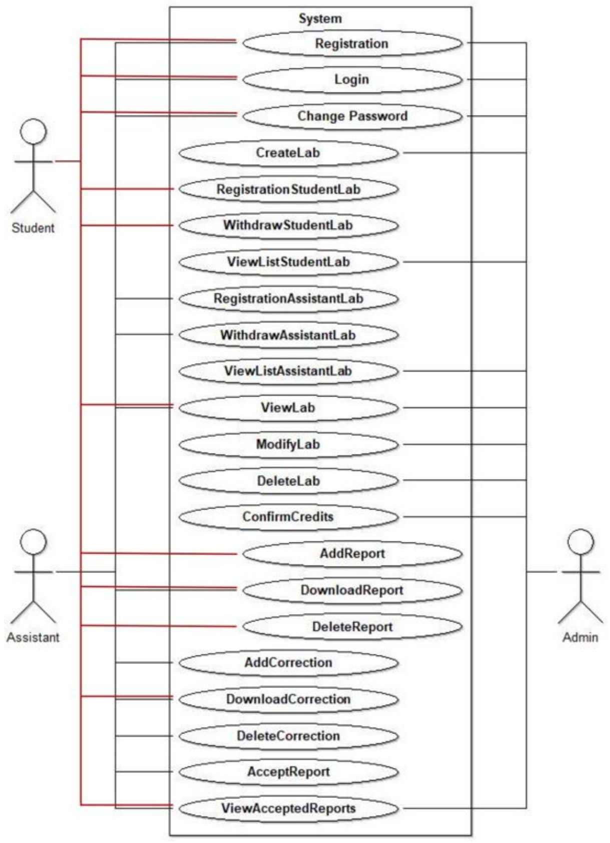


Fig. 3.1.: Use cases of student, assistant and admin with system

3.4. ERM

The entities are User (Student, Assistant, Admin), Document (Report, Correction) and Laboratory. The relations between the entities are IS-A, HandsIn, CorrectTo, Submit, WorksAt, Manages, Organises and HasA. Each line contains a 1 or mc for the MC-Notation that describes how many of each entity are related. The meaning of 1 is that only one entity is related, m means multiple and c means 0 or 1 entities are related. So, for example, one reads Laboratory-(1)-HasA-(mc)-Document the as follows: A laboratory has 0 or 1 or multiple documents. But the document has only one related lab and is not shared by multiple labs.

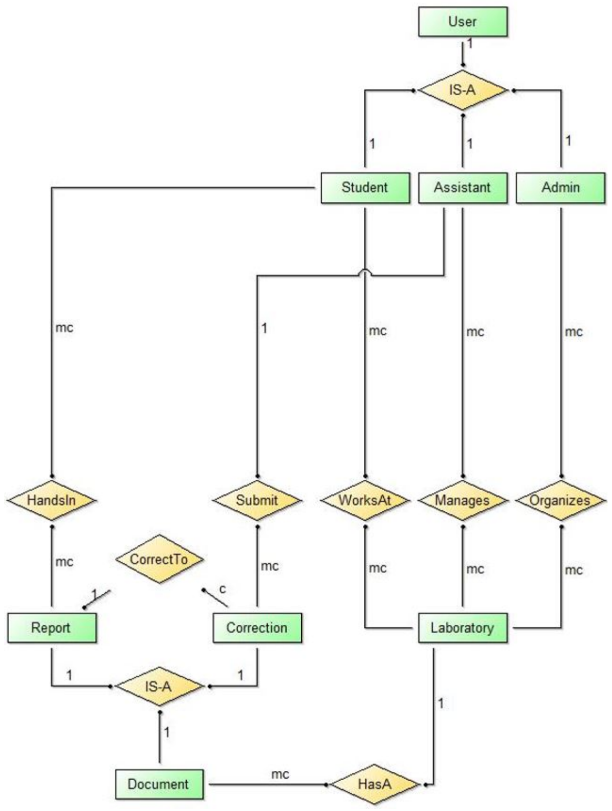


Fig. 3.2.: ERM in MC-Notation with no attributes

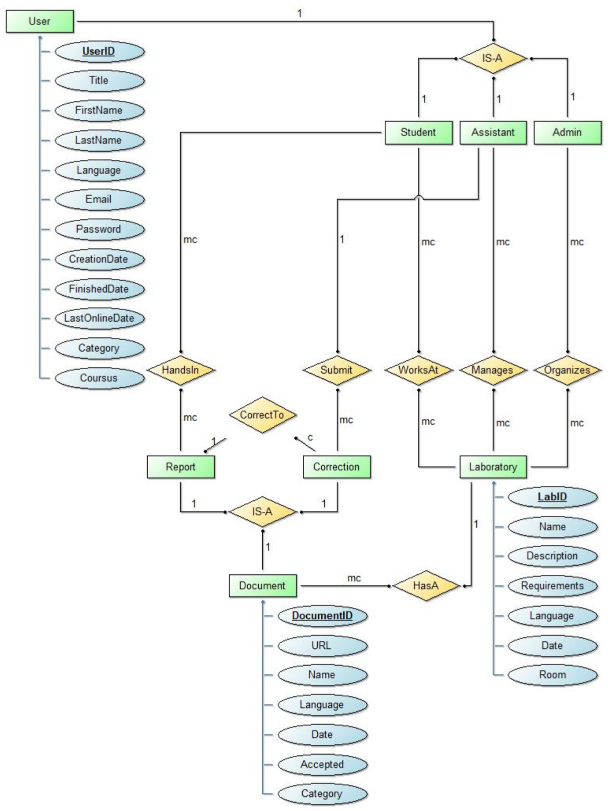


Fig. 3.3.: ERM in MC-Notation and attributes with primary key underlined

3.5. Stack choice

Now that the use cases are defined, the next step is to choose the stack for the project. First of all, it's important to note that all stacks could theoretically be chosen for any project. Each stack has different advantages and disadvantages. Here are four of the most popular full stacks, along with their pros and cons.

1. MEAN
   + One programming language (JavaScript)
   + Operating system independent
   + Scalable and flexible
   - Fast pace of updates results in incompatibility
   - MongoDB can lose data if overloaded
2. MERN
   + One programming language (JavaScript)
   + React gives a excellent experience for the user
   + Big support from community
   - React is a library and not a full framework
   - No direct backend server calls
   - Not good for large-scale applications

3. MEVN
   + One programming language (JavaScript)
   + Rapid development and effectiveness
   + Platform independent
   - Not large support and few active programmers
   - Lack of plugins
   - MongoDB is not suited for Multi-Object Transactions
4. LAMP
   + Classic, old, reliable, flexible
   + Stability, simplicity
   + Robust for large-scale applications
   - Apache is not highest performing webserver
   - MySQL gets less popular than NoSQL databases
   - Gets less attention as JavaScript stacks

I had never used a full stack before, so the simplicity of having one programming language was a reason not to choose the LAMP stack and other stacks that used multiple languages. The differences between MEAN, MERN and MEVN did not seem very great. Many websites mentioned the stacks in following order: MERN, MEAN, MEVN. The MERN stack was chosen, because of the large community support and the number of citations on several sites. There were more sources and examples for the MERN stack. For example a search on the Google engine with *MERN Stack*, *MEAN Stack* and *MEVN Stack* returns 4'120'000, 1'250'000 and 193'000 results respectively. [11]

The MERN Stack Crash Course Tutorial from Net Ninja on Youtube can be used to start as a beginner, code at the same time and build more elements later. [12]

3.6. MERN stack

The MERN stack consists of the following parts:

MongoDB

A NoSQL database that uses Javascript Object Notation (JSON) to manage the content. All the information about the users or documents is stored here.

Express

This is the web application framework for the server part. It uses the middleware and controller design of modern web applications.

React

React is the frontend library and manages the user interface. Hooks are used to view and change the state of components.

Node

Node is a runtime environment with has an event-driven architecture.

3.7. Software

To handle all the different tools we used the following software:

Bee-up 1.6 Modelling Tool

A modelling tool for different modelling languages, such as use cases and ERM.

Git

Software for storing and sharing programming files and version control.

MongoDBCompass

A Graphical User Interface (GUI) for MongoDB. MongoDBCompass is the database manager.

Mozilla Firefox

A free web browser. Firefox is the project's client.

Postman

A tool for testing and improving the API. Postman checks the endpoints of the application.

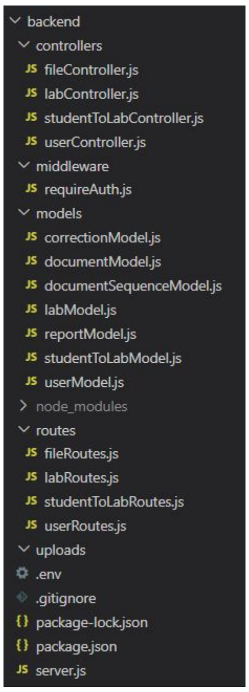
Visual Studio Code

An Integrated Development Environment (IDE) for working with the scripts.

4 Developer details

4.1. Backend

The following image shows the structure of the backend folder. The different names and subfolders help to understand the logic of the developer details.



```
backend
  controllers
    fileController.js
    labController.js
    studentToLabController.js
    userController.js
  middleware
    requireAuth.js
  models
    correctionModel.js
    documentModel.js
    documentSequenceModel.js
    labModel.js
    reportModel.js
    studentToLabModel.js
    userModel.js
  node_modules
  routes
    fileRoutes.js
    labRoutes.js
    studentToLabRoutes.js
    userRoutes.js
  uploads
  .env
  .gitignore
  package-lock.json
  package.json
  server.js
```

Fig. 4.1.: Structure of backend folder

The backend uses Node Package Manager (npm) where additional libraries are introduced:

bcrypt

A library to hash passwords. We can generate a random and unique string called *salt*. We append the salt of length ten to the password and hash it to gain security.

```
const salt=await bcrypt.genSalt(10)
const hash=await bcrypt.hash(password,salt)
```

dotenv

A module that loads the *.env* file in the current folder and stores the contents in *process.env*. We can get the *PORT* value with *process.env.PORT*.

express

The Node.js module framework. The application is started with *express()* and features added with *use()* for the location of uploaded documents, multiple router requests and router redirections:

```
const app=express();
app.use('/uploads', express.static('uploads'));
app.use((req,res,next)=>{next()})
app.use('/api/labs',labRoutes)
```

jsonwebtoken

A JSON Web Tokens (JWT) library. This token is exchanged between two parties to identify the user. We can create with *sign()* and compare with *verify()* the tokens:

```
jwt.sign({_id: _id}, process.env.SECRET, {expiresIn: '3d'})
const {_id}=jwt.verify(token, process.env.SECRET)
```

mongoose

Mongoose is a modelling tool for MongoDB that validates and casts types. It is also possible to add a timestamp to an object to keep track of the creation and last update time. The database is connected with the function:

```
mongoose.connect(process.env.MONGO_URI)
```

multer

A middleware to handle the uploading of files. We define the storage location to the folder *uploads* and the upload to the storage:

```
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, 'uploads/');
  },
  filename: (req, file, cb) => {
    cb(null, `${Date.now()}-${file.originalname}`);
  }
});

const upload = multer({ storage: storage }).single('file');
```

validator

A module for checking strings of a certain structure. The functions *isEmail()* and *isStrongPassword()* are used: [13]

```
if(!validator.isEmail(email)){
  throw new Error('Email is not valid')
}
if(!validator.isStrongPassword(password)){
  throw new Error('Password is not strong enough')
}
```

4.1.1. Database

MongoDB is a NoSQL database. This means that the data is not stored in tables but is stored in JSON format for an object. The attributes of objects can change over time. Multiple objects are grouped together in a collection. Different collections are stored together in a database. In MongoDBCompass, different databases can be managed when connecting to the *mongodb://localhost:27017*. To access a specific database called *MERN* we use the *mongodb://localhost:27017/MERN*.

The folder *models* contains all the mongoose schemas. For example in the file *userModel.js* we define a user. All users have a unique email and the students need additional attributes *major*, *minor* and *courseName*. The *finishedDate* is the only attribute that is not needed, because it is not clear when a student will finish all the labs. The *isActive* field determines whether the students or assistants are still working on labs or not.

```
const userSchema=new mongoose.Schema({
  firstName:{
    type: String,
    required: true
  },
  lastName:{
    type: String,
    required: true
  },
  email:{
    type: String,
    required: true,
    unique: true
  },
  role:{
    type: String,
    required: true,
    enum: ['student','assistant','admin']
  },
  password:{
    type: String,
    required: true
  },
  isActive:{
    type: Boolean,
    required: true
  },
  major: {
    type: String,
    required: function() { return this.role === 'student'; }
  },
  minor: {
    type: String,
    required: function() { return this.role === 'student'; }
  },
  courseName: {
    type: String,
    required: function() { return this.role === 'student'; }
  },
  finishedDate: {
    type: Date,
    required: false
  }
})
```

4.1.2. Server

The terminal command *npm start* in the backend folder will run *node server.js*. The file *server.js* initiates the application, defines the upload folder, determines multiple router requests, defines the routers, connects to the port 27017 for the *MERN* database and listens to the port 4000 for the server.

When a request is intercepted by the server, it is first sent to the routes, then to the middleware and then handled by the controllers. If there is an error it will throw the error back, otherwise it will send back the answer of the controller functions.

Let's say the server is listening to a POST request with the URL */api/file/upload*. Because of the code in the server *app.use('/api/file',fileRoutes)* it will redirect the request to the file *fileRoutes.js* in the folder *routes*. The token is verified by the middleware with *const _id=jwt.verify(token, process.env.SECRET)* . Assuming the token was valid and we continue the handling of the request. The line *router.post('/upload', handleUpload);* sends it on to the file *fileController.js* in the *controllers* folder. The *handleUpload* function is checking the user and file. It creates a report or correction for the student or assistant respectively. The document is created and stored in the document sequence if there is already one, else it will create the sequence. It then sends back the result back to the server.

For all other URLs we have a similar procedure. For example, for registration and login we do not need to authenticate the user with a token, because we create a new token. We present the implemented structure of the server:

```
// load .env file
require('dotenv').config();

// libraries and routes
const express=require('express');
const mongoose=require('mongoose');
const labRoutes=require('./routes/labRoutes');
const userRoutes=require('./routes/userRoutes');
const studentToLabRoutes=require('./routes/studentToLabRoutes');
const fileRoutes=require('./routes/fileRoutes');

// application
const app=express();

// look if there is a body in json
app.use(express.json())

```

```
// serve static files from the uploads directory
app.use('/uploads', express.static('uploads'));

// manage different routes
app.use((req,res,next)=>{next()})

//routes
app.use('/api/labs',labRoutes)
app.use('/api/user',userRoutes)
app.use('/api/studentToLab',studentToLabRoutes)
app.use('/api/file',fileRoutes)

// connect to db
mongoose.connect(process.env.MONGO_URI)
    .then(()=>{app.listen(process.env.PORT,()=>{
        console.log(`Connected to DB and listening on port ${process.env.PORT}`)
    })})
    .catch((error)=> console.log(error))
```

4.2. Frontend

The frontend also uses npm with some libraries:

cra-template

The official template for the react application. It is not used explicitly.

date-fns

A package for time manipulation.

```
import formatDistanceToNow from 'date-fns/formatDistanceToNow';

<p><strong>Creation Date: </strong>{formatDistanceToNow(
    new Date(studentLab.createdAt), { addSuffix: true })}</p>
```

react

A JavaScript library for user interfaces.

```
import {createContext, useReducer} from 'react';

export const StudentLabsContext = createContext();
const [state, dispatch]=useReducer(studentLabsReducer, {studentLabs:null})
```

react-dom

This module handles the Document Object Model (DOM) and works together with react.

```
import ReactDOM from 'react-dom/client';

const root = ReactDOM.createRoot(document.getElementById('root'));
```

react-router-dom

React-router handles the react routers for requests.

```
import {Link} from 'react-router-dom';

<Link to="/">
    <h1>Lab Overview</h1>
```

```
</Link>
<Link to="/reports">
    <h1>Reports</h1>
</Link>
```

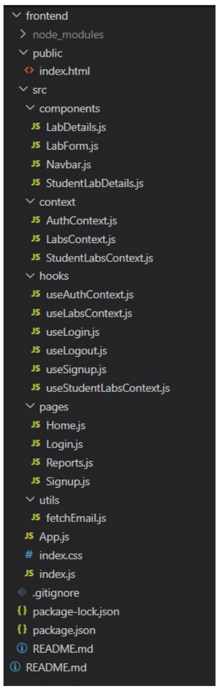
react-scripts

React-scripts is a package that provides scripts and configurations for starting, building, testing and ejecting the application. [14]

```
"scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
}
```

4.2.1. Client

We first present the structure of the folder for an overview and the different file names.



```
frontend
  node_modules
  public
    index.html
  src
    components
      LabDetails.js
      LabForm.js
      Navbar.js
      StudentLabDetails.js
    context
      AuthContext.js
      LabsContext.js
      StudentLabsContext.js
    hooks
      useAuthContext.js
      useLabsContext.js
      useLogin.js
      useLogout.js
      useSignup.js
      useStudentLabsContext.js
    pages
      Home.js
      Login.js
      Reports.js
      Signup.js
    utils
      fetchEmail.js
    App.js
    index.css
    index.js
  .gitignore
  package-lock.json
  package.json
  README.md
README.md
```

Fig. 4.2.: Structure of frontend folder

In the frontend folder, the command *npm start* will launch *index.html*. The *root* object from the react-dom package is shown in a division (<*div*>) in the body part (<*body*>):

```
<!DOCTYPE html>
<html lang="en">
  <head>
    // existing code
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
  </body>
</html>
```

The standard browser opens automatically and all the objects in root are shown. The port 3000 is used for the client.

First, we need to introduce the React hook features *useState*, *useEffect*, *createContext*, *useContext* and *useReducer*.

useState

The useState hook takes two arguments, the variable that defines the current state and the function that changes the state of the variable. It is used to display objects when the state is correct.

```
const [name, setName] = useState('Lab 1');
setName('');
```

useEffect

The UseEffect hook is a function that handles the states in the background. Usually we fetch data from an API endpoint. It is not used to display objects directly, only to update the information. [15]

```
useEffect(() => {
   fetchEmail(studentLab.fk_adminId, setAdminEmail, user.token);
   fetchEmail(studentLab.fk_assistantId, setAssistantEmail, user.token);
}, [studentLab.fk_adminId, studentLab.fk_assistantId, user.token]);
```

createContext

A function that creates the context that holds the state.

```
export const StudentLabsContext = createContext();
<StudentLabsContext.Provider value={{...state, dispatch}}>
   {children}
</StudentLabsContext.Provider>
```

useContext

The useContext function is used to automatically send data down the component tree, making the information available to all elements.

```
const context=useContext(StudentLabsContext);
```

useReducer

This function manages the state logic of react. It takes two parameters, a reducer function and the initial state. The reducer function takes the state and a action and returns a new state.

```
export const studentLabsReducer = (state, action) =>{
   switch(action.type){
      case 'SET_STUDENT_LABS': return{studentLabs: action.payload}
      // existing code
   }
}
const [state, dispatch]=useReducer(studentLabsReducer, {studentLabs:null})
```

When a user interacts with the browser, the web page sends a request to the server and refreshes the page with the new data.

Let's say that the user clicks on the *Upload* button of the *StudentLabDetails*. The button will call the *handleUpload* function with the corresponding ID of the *studentLab*. The user is prompted to select a file from their computer. We append the file and send a POST request with the URL */api/file/upload*. The body contains the file and the header contains the user's token. From the previous chapter we know the steps on the server that will place the file in the appropriate folder.

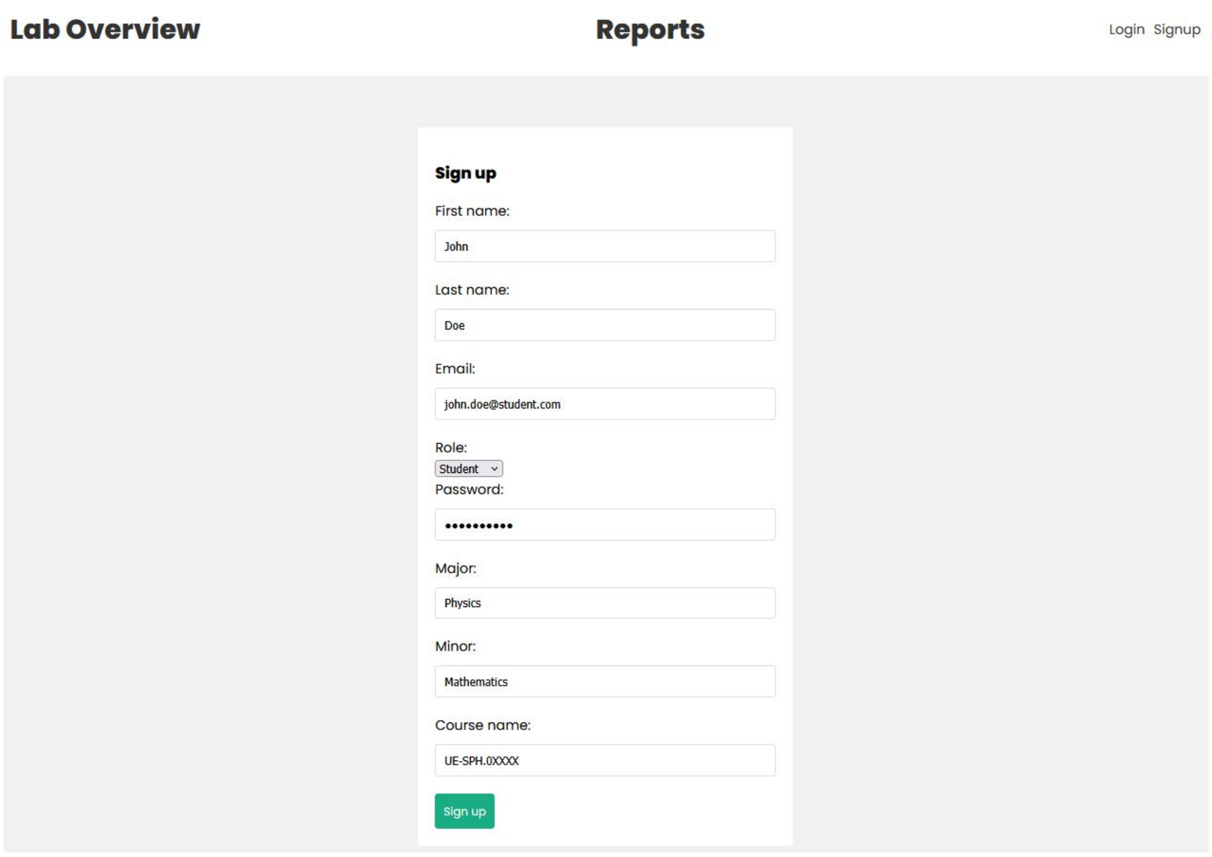
The other buttons and text fields are treated in a similar way. We will now present the code for the home page, which will show the labs to the user, once they have successfully logged in:

```
import {useEffect} from 'react';
import {useAuthContext} from '../hooks/useAuthContext';
import {useLabsContext} from '../hooks/useLabsContext';

import LabDetails from '../components/LabDetails';
import LabForm from '../components/LabForm';

const Home=()=>{
  const {user}=useAuthContext();
  const {labs, dispatch: labDispatch}=useLabsContext();

  useEffect(() => {
    const fetchLabs=async()=>{
      let url ="";
      if (user.role === 'student')  {url = '/api/labs/student/active';}
      if (user.role === 'assistant') {url = '/api/labs/assistant/active';}
      if (user.role === 'admin')  {url = '/api/labs';}

      const response = await fetch(url, {
        headers: {'Authorization': `Bearer ${user.token}`}
      });
      const json=await response.json();

      if (response.ok){labDispatch({type: 'SET_LABS', payload: json})}
    }
    if (user) {fetchLabs();}
  }, [labDispatch, user]);

  return(
    <div className="home">
      <div>
      {user.role === 'admin' &&
        <div>
          <p>Lab Form</p>
          <LabForm/>
        </div>
      }
        <div className="labs">
          {labs && labs.map((lab) => (<LabDetails key={lab._id} lab={lab}/>))}
        </div>
      </div>
    </div>
  )
}

export default Home;
```

5 Results

5.1. Profile

All users, i.e. students, assistants and administrators, must register and log in to be able to modify any labs and documents. We give an example for the students who want to register on the lab management page. If the role of the user is student, then the fields major, minor and course name must also be filled in. An admin or assistant doesn't need these field and only fills in the first name, last name, email, role and their password. For the convenience of the developer we fill out all fields with the following attributes: *John, Doe, john.doe@student.com, student, abcABC123!, Physics, Mathematics, UE-SPH.0XXXX*. If we want to generate many different users, we simply append a number to the email, e.g. *john.doe@student1.com, john.doe@student2.com,* etc. Another option is to use the role as a part of the email address. For example *john.doe@admin.com, john.doe@assistant1.com,* etc. This helps the developer to avoid filling out the form every time a small change is made.



Sign up

First name: John

Last name: Doe

Email: john.doe@student.com

Role: Student

Password: ••••••••••

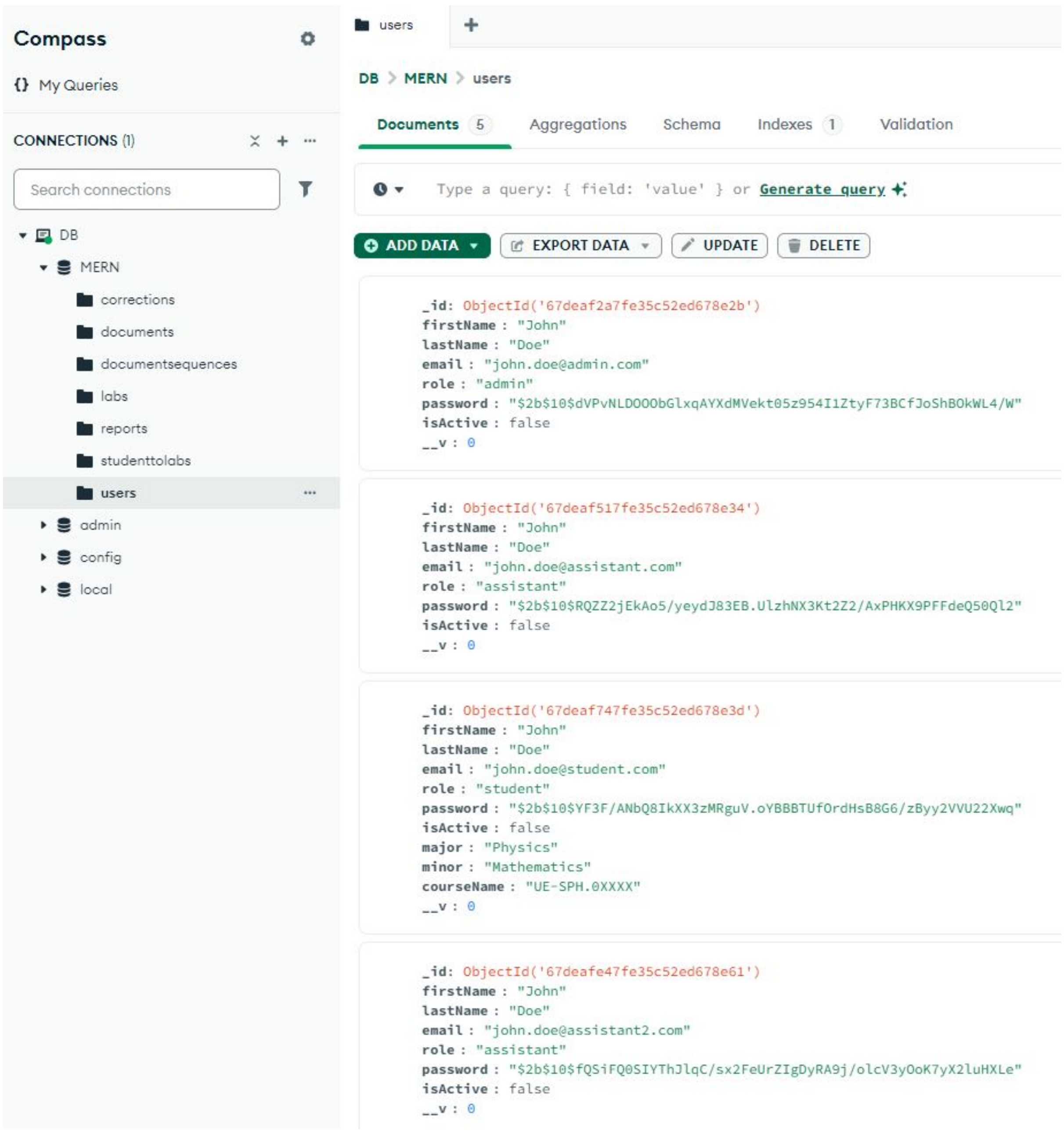
Major: Physics

Minor: Mathematics

Course name: UE-SPH.0XXXX

Fig. 5.1.: Register page of user

The MongoDBCompass software allows the developer to check that the users have been created correctly. The MERN database contains the collection users with all the different users. You can see that the student has more fields than the other two roles. MongoDB automatically assigns the fields *_id* for the unique identity of an object and *__v* for the version, which starts with the value 0 and is incremented by 1 each time the object is updated. Another detail is that all the hashed passwords are different, even if the user's password was always *abcABC123!*. This is due to the salt we add after the user password to increase security.



```
_id: ObjectId('67deaf2a7fe35c52ed678e2b')
firstName : "John"
lastName : "Doe"
email : "john.doe@admin.com"
role : "admin"
password : "$2b$10$dVPvNLDOOObGlxqAYXdMVekt05z954I1ZtyF73BCfJoShBOkWL4/W"
isActive : false
__v : 0

_id: ObjectId('67deaf517fe35c52ed678e34')
firstName : "John"
lastName : "Doe"
email : "john.doe@assistant.com"
role : "assistant"
password : "$2b$10$RQZZ2jEkAo5/yeydJ83EB.UlzhNX3Kt2Z2/AxPHKX9PFFdeQ50Ql2"
isActive : false
__v : 0

_id: ObjectId('67deaf747fe35c52ed678e3d')
firstName : "John"
lastName : "Doe"
email : "john.doe@student.com"
role : "student"
password : "$2b$10$YF3F/ANbQ8IkXX3zMRguV.oYBBBTUfOrdHsB8G6/zByy2VVU22Xwq"
isActive : false
major : "Physics"
minor : "Mathematics"
courseName : "UE-SPH.0XXXX"
__v : 0

_id: ObjectId('67deafe47fe35c52ed678e61')
firstName : "John"
lastName : "Doe"
email : "john.doe@assistant2.com"
role : "assistant"
password : "$2b$10$fQSiFQ0SIYThJlqC/sx2FeUrZIgDyRA9j/olcV3yOoK7yX2luHXLe"
isActive : false
__v : 0
```

Fig. 5.2.: User in MongoDB database

After registration, the admin must activate the students and assistants. Users can now log in to their account. They will need their first name, last name, email, role and password.

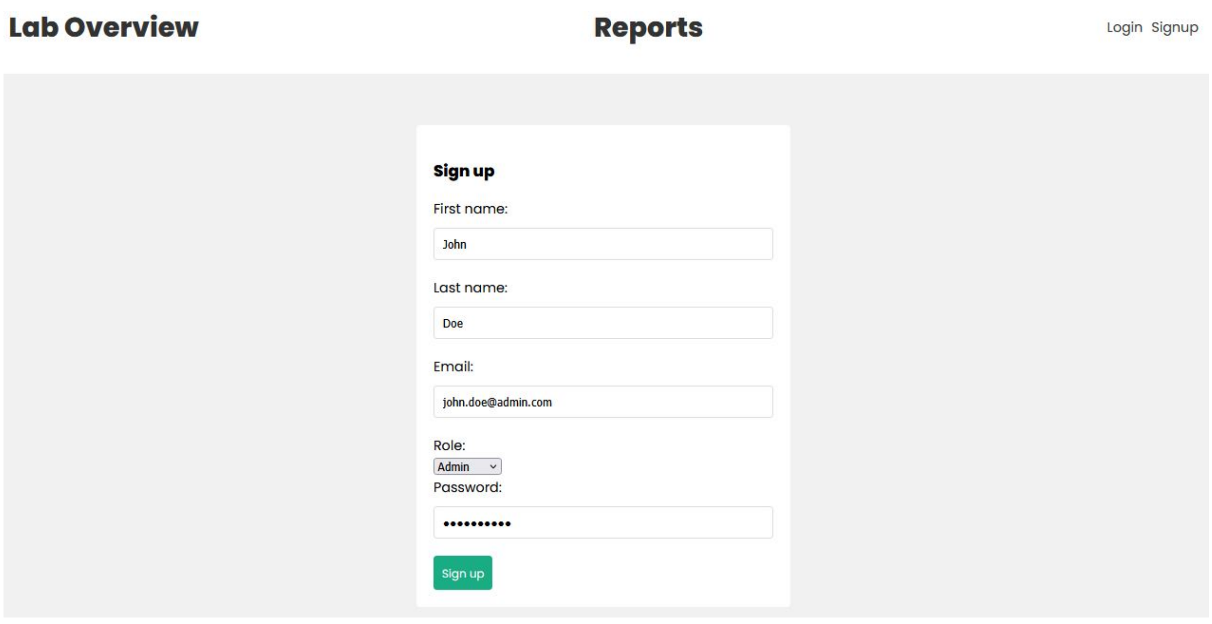
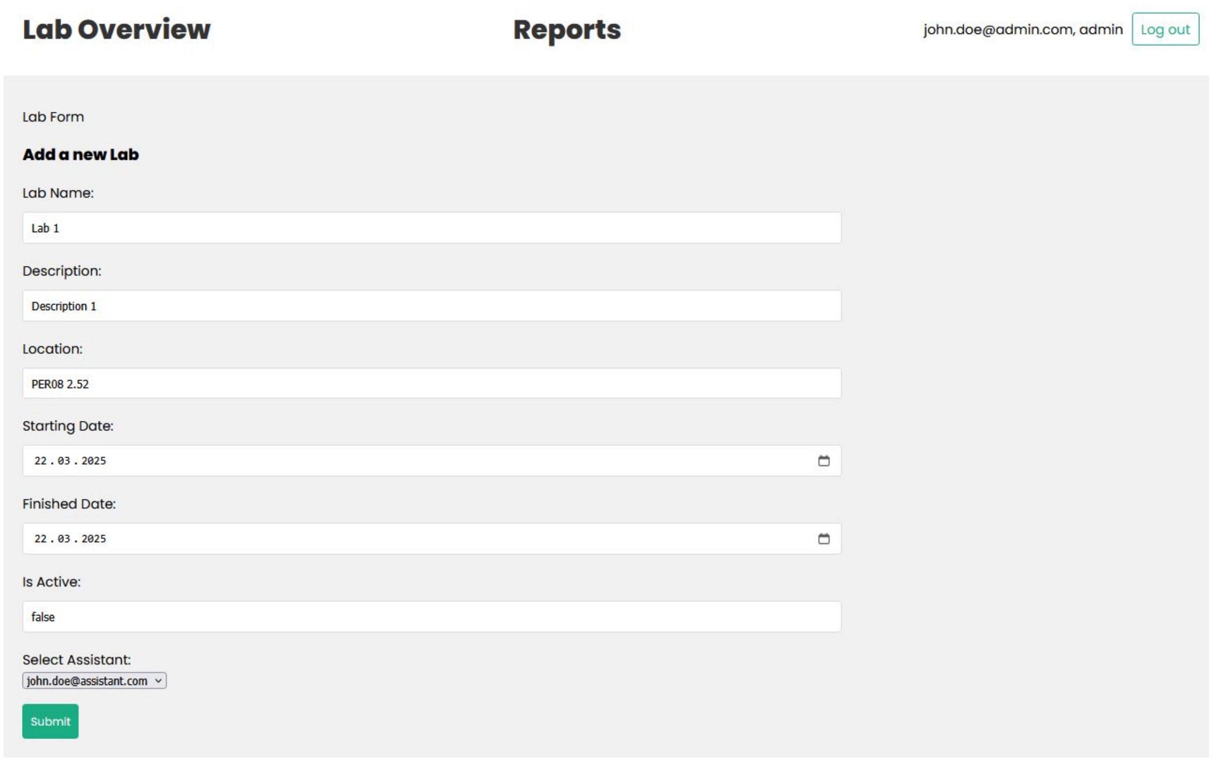

Lab Overview
Reports
Login Signup
Sign up
First name:
John
Last name:
Doe
Email:
john.doe@admin.com
Role:
Admin
Password:
••••••••••
Sign up

Fig. 5.3.: Login page of user

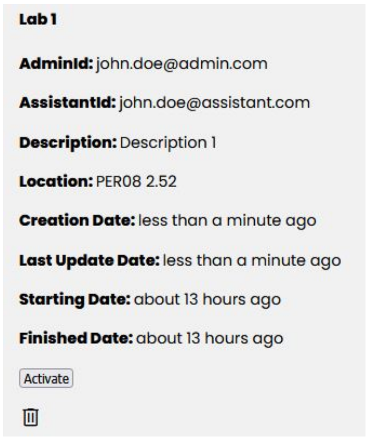
5.2. Lab

The admin can create laboratories with the fields lab name, description, location, starting date, finished date, is active and the selected assistant. The fields containing a date must be in the specific format DD.MM.YYYY or can be selected by clicking on the calendar icon to the right of the field. The assistant can not be entered manually but needs to be an activated assistant. The drop down menu shows all possible assistants. If the lab is not activated, then it will not be visible for students or assistant. The admin can activate the lab later or deactivate it if they see an error. The admin can also delete labs by clicking on the trash can icon.


Lab Overview
Reports
john.doe@admin.com, admin
Log out

Lab Form
Add a new Lab
Lab Name: Lab 1
Description: Description 1
Location: PER08 2.52
Starting Date: 22 . 03 . 2025
Finished Date: 22 . 03 . 2025
Is Active: false
Select Assistant: john.doe@assistant.com
Submit

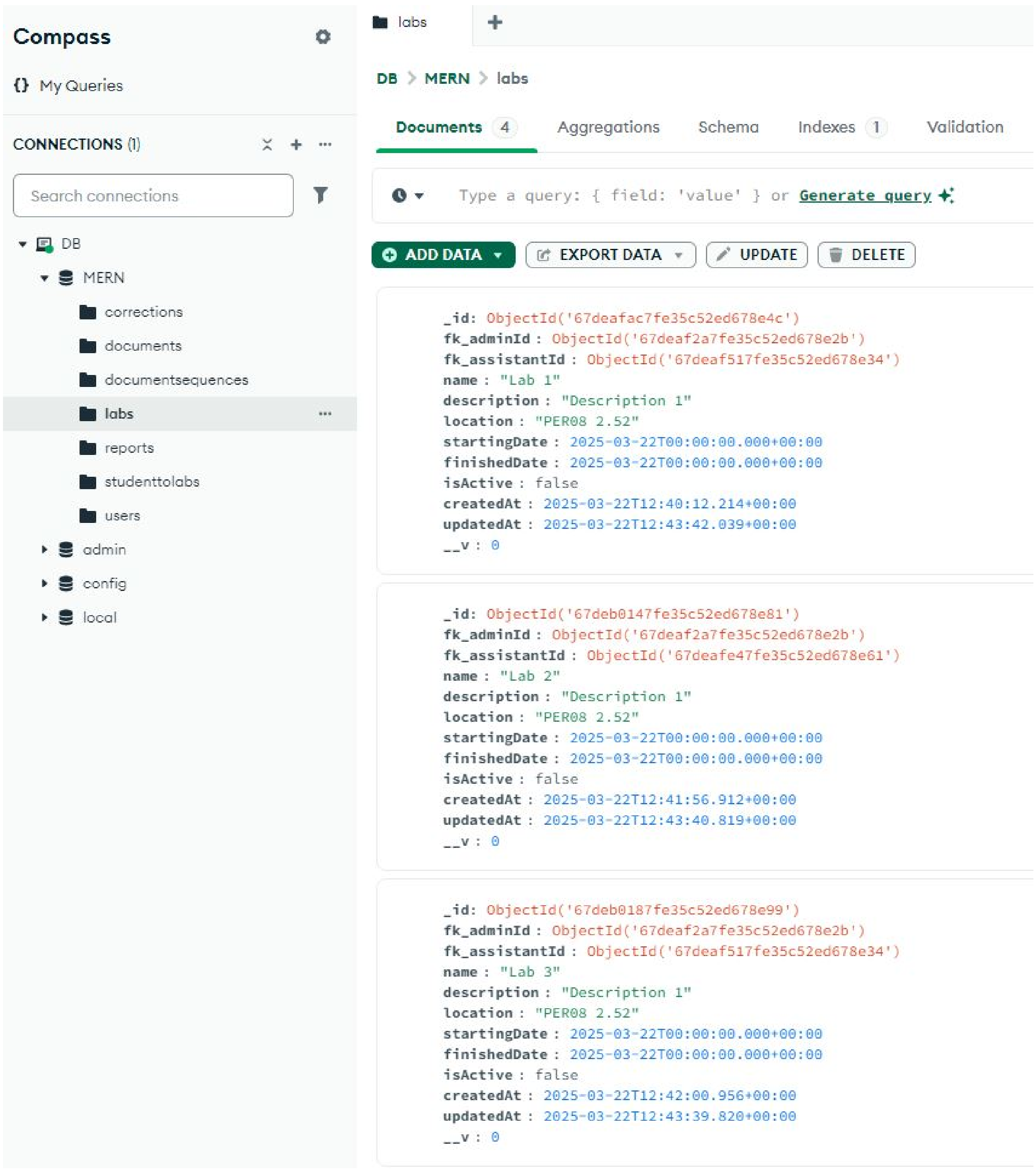
Fig. 5.4.: Admin creates a new lab with an assistant


Lab 1

AdminId: john.doe@admin.com
AssistantId: john.doe@assistant.com
Description: Description 1
Location: PER08 2.52
Creation Date: less than a minute ago
Last Update Date: less than a minute ago
Starting Date: about 13 hours ago
Finished Date: about 13 hours ago
Activate

Fig. 5.5.: Admin modify and delete an existing lab

The labs are stored in the database with the identity of the admin and assistant as foreign key *fk_ adminId* and *fk_ assistantId*. The lab has only one admin and one assistant. The timestamp attribute has been used for the labs so that the *createdAt* and *updatedAt* fields are automatically added by the database. This is useful for sorting the labs by creation date or last update date.



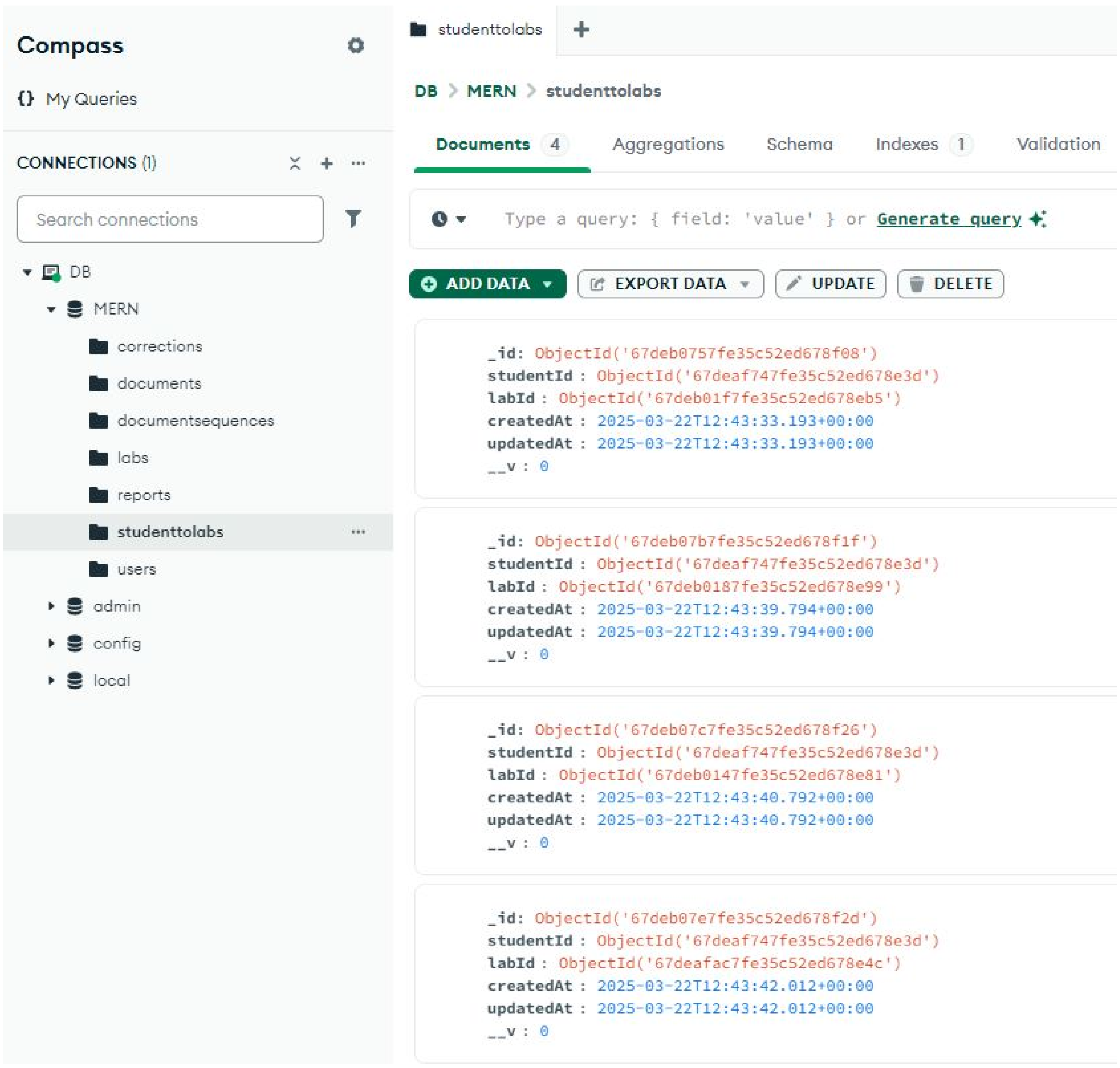
```
_id: ObjectId('67deafac7fe35c52ed678e4c')
fk_adminId : ObjectId('67deaf2a7fe35c52ed678e2b')
fk_assistantId : ObjectId('67deaf517fe35c52ed678e34')
name : "Lab 1"
description : "Description 1"
location : "PER08 2.52"
startingDate : 2025-03-22T00:00:00.000+00:00
finishedDate : 2025-03-22T00:00:00.000+00:00
isActive : false
createdAt : 2025-03-22T12:40:12.214+00:00
updatedAt : 2025-03-22T12:43:42.039+00:00
__v : 0
```

```
_id: ObjectId('67deb0147fe35c52ed678e81')
fk_adminId : ObjectId('67deaf2a7fe35c52ed678e2b')
fk_assistantId : ObjectId('67deafe47fe35c52ed678e61')
name : "Lab 2"
description : "Description 1"
location : "PER08 2.52"
startingDate : 2025-03-22T00:00:00.000+00:00
finishedDate : 2025-03-22T00:00:00.000+00:00
isActive : false
createdAt : 2025-03-22T12:41:56.912+00:00
updatedAt : 2025-03-22T12:43:40.819+00:00
__v : 0
```

```
_id: ObjectId('67deb0187fe35c52ed678e99')
fk_adminId : ObjectId('67deaf2a7fe35c52ed678e2b')
fk_assistantId : ObjectId('67deaf517fe35c52ed678e34')
name : "Lab 3"
description : "Description 1"
location : "PER08 2.52"
startingDate : 2025-03-22T00:00:00.000+00:00
finishedDate : 2025-03-22T00:00:00.000+00:00
isActive : false
createdAt : 2025-03-22T12:42:00.956+00:00
updatedAt : 2025-03-22T12:43:39.820+00:00
__v : 0
```

Fig. 5.6.: Database table labs

Multiple students can have multiple labs and we need to create a new table to assign each student to each lab. We call this table *StudentToLabs* to associate the students with labs. To do this we need the identity of the lab and the student. This table only links the two objects for the mc-mc relation that was defined for the ERM.

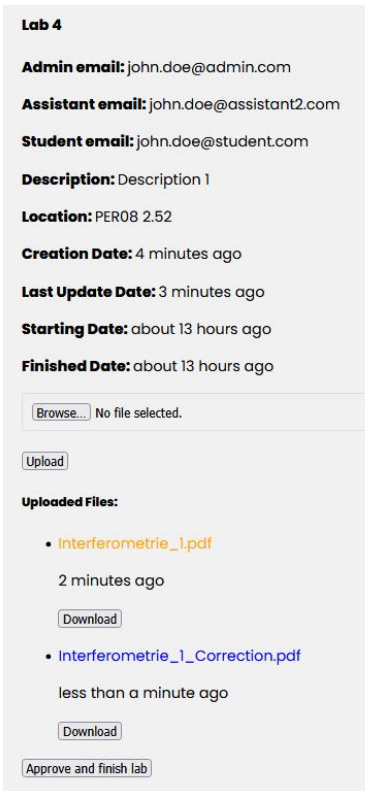


```
_id: ObjectId('67deb0757fe35c52ed678f08')
studentId : ObjectId('67deaf747fe35c52ed678e3d')
labId : ObjectId('67deb01f7fe35c52ed678eb5')
createdAt : 2025-03-22T12:43:33.193+00:00
updatedAt : 2025-03-22T12:43:33.193+00:00
__v : 0

_id: ObjectId('67deb07b7fe35c52ed678f1f')
studentId : ObjectId('67deaf747fe35c52ed678e3d')
labId : ObjectId('67deb0187fe35c52ed678e99')
createdAt : 2025-03-22T12:43:39.794+00:00
updatedAt : 2025-03-22T12:43:39.794+00:00
__v : 0

_id: ObjectId('67deb07c7fe35c52ed678f26')
studentId : ObjectId('67deaf747fe35c52ed678e3d')
labId : ObjectId('67deb0147fe35c52ed678e81')
createdAt : 2025-03-22T12:43:40.792+00:00
updatedAt : 2025-03-22T12:43:40.792+00:00
__v : 0

_id: ObjectId('67deb07e7fe35c52ed678f2d')
studentId : ObjectId('67deaf747fe35c52ed678e3d')
labId : ObjectId('67deafac7fe35c52ed678e4c')
createdAt : 2025-03-22T12:43:42.012+00:00
updatedAt : 2025-03-22T12:43:42.012+00:00
__v : 0
```

Fig. 5.7.: Database table of students to labs

5.3. Document

All documents are stored in a sequence of documents. They can be a report from a student or a correction of the assistant. The document sequence is ordered by the creation date of the file. Reports are displayed with an orange color and the corrections in blue. The assistant and student can see all the documents and can download them or upload new files. The file type is not restricted, but will be in most cases a PDF-file. With the date-fns package, dates are not displayed in the format of year, month, day, hour, minute, second, but as a relative time distance. For example *2 minutes ago* or *less than a minute ago*.


Lab 4

Admin email: john.doe@admin.com

Assistant email: john.doe@assistant2.com

Student email: john.doe@student.com

Description: Description 1

Location: PER08 2.52

Creation Date: 4 minutes ago

Last Update Date: 3 minutes ago

Starting Date: about 13 hours ago

Finished Date: about 13 hours ago

Uploaded Files:

- Interferometrie_1.pdf

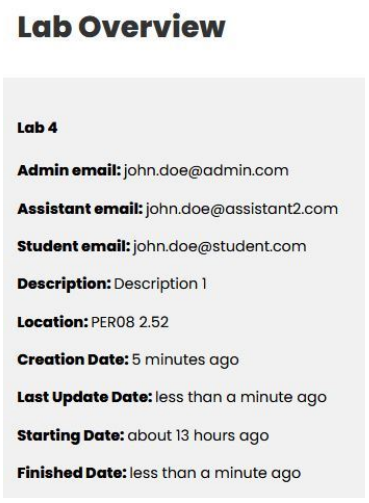
  2 minutes ago

- Interferometrie_1_Correction.pdf

  less than a minute ago

Fig. 5.8.: Example of report and correction that have been uploaded

The assistant is able to approve and finish the lab once the report has no more errors. When this button is clicked, then the student and assistant will no longer see the lab and the admin sees that this lab has been finished. When all the labs have been completed the admin can give the credits for the course.


Lab Overview

Lab 4

Admin email: john.doe@admin.com

Assistant email: john.doe@assistant2.com

Student email: john.doe@student.com

Description: Description 1

Location: PER08 2.52

Creation Date: 5 minutes ago

Last Update Date: less than a minute ago

Starting Date: about 13 hours ago

Finished Date: less than a minute ago

Fig. 5.9.: Admin sees the accepted labs of the students

6 Discussion

6.1. Use cases and ERM

The first step of this project was to create the use cases and the ERM. In this part I saw that most of the hierarchies were clear, such as *Does the student and the assistant see all documents or not?* and *Can an assistant have several labs?*. On the other hand, questions like *Is there only one admin or several?* and *Can an assistant also be an admin?* were not clear at the beginning.

I limited the options as much as possible and only changed them if they caused problems for the future of the project. This is why I rejected having multiple admins in the beginning, because that is how APLabs is structured at the moment. But having different admins gives the possibility to create and manage multiple labs in parallel. So we generalised to multiple admins. For the second example it was clear that the assistant and the admin should be separated. The tasks of assistants and admins are different and should not be mixed. This is the reason, why we split the users into the three categories.

Even with these problems, they could be resolved quickly and were corrected by the assistant and professor to be able to move on the next steps.

6.2. Web Development Stack

Once the structure was in place, the next step was to choose a web development stack. The recommendations were not clear because I didn't know the different stacks and their advantages. The difficulty was to choose the stack to avoid future incompatibilities, but also to know that any stack could theoretically be used for any project.

The solution was to try different stacks such as MEAN, MEVN and MERN. I followed tutorials that worked up to a certain point. Mismatching versions of packages made progress difficult, for example the MEVN stack with Vue2.0 and Vue3.0. The frontend error messages were hard to spot and took a lot of time to understand. Small syntax errors led to error messages. Other tutorials didn't structure the files into appropriate folders, which slowed down further work with this code.

After finding the Net Ninja tutorial, I used the MERN stack to continue the project. The folder structure and the existing code helped to expand it to the current implementations.

6.3. Node

For all the three stacks we used Node.js. The similar names like Node, Node.js, nvm, npm, nodemon, node_modules etc. were confusing when trying to start the frontend and backend with the terminal commands. After a few examples the confusion became less and less with time.

6.4. Folder structure

The folder structure of the tutorials was not the same, which made it complex to combine two ideas from two tutorials. The content and structure were so different that I was unable to merge the tools. Some tutorials only focused on one topic, such as the registering and logging of users, saving documents or showing the labs. The URLs or variable names were hard-coded and not generalised to different files. Another question about structure was whether it was better to use one or more *.gitignore*, *package.json* and *package-lock.json*.

In this project I mainly used the folder structure of Net Ninja. The files were split into different groups and this made it easier to write new code. I decided to create a *.gitignore*, *package.json* and *package-lock.json* for each of the frontend and backend folders.

6.5. Documents storage

After implementing user registration and login, the question of where to store documents arose. Up to this point, all user and lab information was stored in MongoDB. It is possible to store large documents (>16 MB) in MongoDB using GridFS. The large file is split into smaller pieces. [9]

I implemented this part, but after a discussion with the assistant I changed the way the documents are stored. I store the files in the uploads folder of the backend folder. The database stores the URL to access the files on the server. This reduces the load on the database and the server can respond directly to the client.

6.6. Microsoft Copilot

The Visual Studio Code can be connected with the GitHub account. The Microsoft Copilot can be used to chat and correct the code. This tool helped at the beginning of the project. For completely empty files or empty folders it can help to kick start the first lines of code. After several files and folders this tool became difficult to use. Some parts of the code were just copied without changing the content. Some variable names were not used from other files. Small examples worked very well. When the project had multiple files, I had to specifically write all the steps that needed to be implemented. Much of the code produced contained small bugs and hindered the progress.

The solution was to tell Copilot to scan all the files and write only the code that had changed, without rewriting the whole file. Sometimes it helped to create a new discussion to clear the memory. All the old variable names and functions didn't reappear in the code. It also helped to ask Copilot which file I needed to change some lines in. I also tried online AI tools, but the results were similar to Copilot.

The benefit for starting a project is clear, but for generalisation and efficient coding practices it is not a great tool. This could be an artefact of the fact that most of the code online is smaller projects and examples that don't contain a lot of complexity, and the AI was trained on this kind of data.

7 Conclusion

7.1. Review

This work started with use cases and ERM. We implemented the feature to upload new files to a MERN stack. Users can register and login to the RESTful web service. The students and assistants share their documents, so they don't have to print all the files. This facilitates proofreading and saves time for both parties.

7.2. Outlook

The project could be extended to include additional features. For example, you could implement a button for changing the password and confirming the credits. You could also create a page with all the documents for the labs that need to be read. There is also a plotting page on the current site which could be modified to allow not only linear but also logarithmic scatter plots. Another feature would be to swap the x and y axes and add other functionality.

7.3. Final statements

This master thesis shows how to build a RESTful web Service using a stack. Iteratively the service grows larger and contains more parts, so that the initial idea resembles to the final product.

A
Common Acronyms

API	Application Programming Interface
DB	Database
DBMS	Database Management System
DOM	Document Object Model
ERM	Entity Relationship Model
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
JWT	JSON Web Tokens
LAMP	Linux, Apache, MySQL, PHP
MEAN	MongoDB, Express, Angular, Node
MERN	MongoDB, Express, React, Node
MEVN	MongoDB, Express, Vue, Node
MC	Modified Chen
npm	Node Package Manager
PDF	Portable Document Format
PHP	Hypertext Preprocessor
REST	Representational State Transfer
SQL	Structured Query Language
URI	Unified Resource Identifier
URL	Uniform Resource Locator

B
License of the Documentation

C
Website of the Project

Clone the Github repository and follow the instructions in the README file:
`https://github.com/VincentGlauser/Info30`

References

[1] 9cv9 HR and Career Blog | Top Rated by Readers. What are get, post, put, patch, delete? a walkthrough with javascript's fetch api. `https://medium.com/@9cv9official/what-are-get-post-put-patch-delete-a-walkthrough-with-javascripts-fetch-api-17be3`, 2019. Accessed: 14.03.2025. 5

[2] Abhirup Acharya. Uri vs urn vs url : Key distinctions explained. `https://medium.com/@abhirup.acharya009/uri-vs-urn-vs-url-key-distinctions-explained-dec8e02ebd18`, 2023. Accessed: 14.03.2025. 4

[3] Kate Brush. What is a use case? `https://www.techtarget.com/searchsoftwarequality/definition/use-case`, 2022. Accessed: 21.03.2025. 6

[4] Omar Elgabry. Database — introduction (part 1). `https://medium.com/omarelgabrys-blog/database-introduction-part-1-4844fada1fb0`, 2016. Accessed: 14.03.2025. 6

[5] Inc. Free Software Foundation. Free documentation licence (gnu fdl). `http://www.gnu.org/licenses/fdl.txt`, 2008. Accessed: 14.04.2025. 34

[6] Christi Gorbett. Best web development stacks to use in 2025. `https://www.nobledesktop.com/classes-near-me/blog/best-web-development-stacks`, 2025. Accessed: 21.03.2025. 6

[7] Harsh Gupta. Client-server architecture explained with examples, diagrams, and real-world applications. `https://medium.com/nerd-for-tech/client-server-architecture-explained-with-examples-diagrams-and-real-world-applic`, 2021. Accessed: 14.03.2025. 5

[8] Kartik. Introduction of er model. `https://www.geeksforgeeks.org/introduction-of-er-model/`, 2025. Accessed: 21.03.2025. 6

[9] Kushagra Kesav. Storing data (images / audio / video) 16 mb in mongodb or gridfs? `https://www.mongodb.com/community/forums/t/storing-data-images-audio-video-16-mb-in-mongodb-or-gridfs/215074`, 2023. Accessed: 28.03.2025. 30

[10] M. Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O'Reilly Media, 2012. 4, 5

[11] Rahul Mathur. Mean vs mern vs mevn vs lamp stack for development. `https://www.arkasoftwares.com/blog/`

mean-vs-mern-vs-mevn-vs-lamp-stack-for-development/, 2024. Accessed: 21.03.2025. 11

[12] Net Ninja. Mern stack tutorial 1 - what is the mern stack? https://www.youtube.com/watch?v=98BzS5Oz5E4&list=PL4cUxeGkcC9iJ_KkrkBZWZRHVwnzLIoUE, 2022. Accessed: 21.03.2025. 11

[13] Inc. npm. Bcrypt, dotenv, express, jsonwebtoken, mongoose, multer, validator. https://www.npmjs.com/, 2025. Accessed: 15.04.2025. 14

[14] Inc. npm. cra-template, date-fns, react, react-dom, react-router-dom, react-scripts. https://www.npmjs.com/, 2025. Accessed: 15.04.2025. 18

[15] Okoro Emmanuel Nzube. React hooks – how to use the usestate & useeffect hooks in your project. https://www.freecodecamp.org/news/how-to-use-the-usestate-and-useeffect-hooks-in-your-project/, 2024. Accessed: 15.04.2025. 19

[16] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly Media, 2007. 4