

Phong Dao

SCALABLE REACT APPLICATION WITH REDUX AND TYPESCRIPT

Bachelor's thesis

Bachelor's thesis, Information Technology

T5616SN

2022



South-Eastern Finland
University of Applied Sciences

Author (authors)	Degree title	Time
Phong Dao	Bachelor of Engineering, Information Technology	March 2022
Thesis title		
Scalable React application with Redux and TypeScript		33 pages 0 pages of appendices
Commissioned by		
Supervisor		
Timo Hynninen		
Abstract		
<p>With the constant need for web applications nowadays, React has been a robust and reliable framework for web development. However, these websites can generate a considerable amount of data and require many features. Therefore, to have a website that can perform efficiently in the long term, developers must consider solutions to make their project scale-proof in the first place. Redux and TypeScript are of few tools that enable a React web application to have a robust state management process and a consistent codebase for future development.</p> <p>The main goal of this thesis was to elaborate on creating a full-stack React web application using MERN stack and how implementing Redux and TypeScript can help it be more scalable. Moreover, it can also be a brief guideline for people who want to build a React project using MERN stack.</p> <p>This thesis used a fully functioning full-stack project of a book store web as an example. In addition, the author showcased the development process of a MERN stack application and the effectiveness of using Redux and TypeScript through the documentation of building this project.</p> <p>In conclusion, by utilizing Redux and TypeScript with MERN stack, the author built a web application project that is scalable and easy to maintain and upgrade.</p>		
Keywords		
ReactJS, NodeJS, ExpressJS, MongoDB, Redux, TypeScript		

CONTENTS

1	INTRODUCTION	4
2	THEORETICAL IMPLEMENTATION	4
2.1	CONSTRUCTION OF MERN STACK APPLICATION	4
2.1.1	Node.js	5
2.1.2	React.js	6
2.1.3	MongoDB	9
2.1.4	ExpressJS	11
2.2	Redux	12
2.3	TypeScript	14
2.4	Practical project structure plan	16
3	PRACTICAL IMPLEMENTATION	17
3.1	Backend Express server	17
3.1.1	Backend structure	17
3.1.2	Features	21
3.2	Front end	23
4	CONCLUSION	32
	REFERENCE	33

1 INTRODUCTION

When technologies are constantly evolving, the Internet has become an essential part of our daily lives. According to The Digital 2021 Global Overview Report (January 2021), there were 4.66 billion active internet users worldwide, equivalent to 59.5 percent of the global population. We spent a considerable amount of our time surfing through websites on the Internet, ranging from e-commerce websites to social platforms. To catch up with current digital trends, businesses need to have their representation on the Internet to benefit from billions of active users there. A website is a great tool to reach and connect with the end-user.

Moreover, more features and trends will rise that we need to adapt to as time goes by. Therefore, when building a web application, it is essential to consider its scalability from the beginning. This route reduces the management effort and makes adding new features a more approachable task for the developer. This thesis will focus on how to build a scalable React web application using one of the most popular web stacks, MERN, by using Redux and TypeScript. These tech stacks enable us to implement a robust database and features like authentication/authorization and data storage.

2 THEORETICAL IMPLEMENTATION

In a MERN stack web application, we use NodeJs as our environment to run our JavaScript-based code. To develop our UI, we use ReactJs, a famous JavaScript library. In addition, this solution will work well with our implementation of Redux. Regarding handling the database of the web-app project, MongoDB is an excellent choice since it is lightweight and can scale very well.

2.1 CONSTRUCTION OF MERN STACK APPLICATION

A MERN stack application consists of multiple frameworks from backend to frontend and database. Therefore, to successfully build the MERN stack application, a developer needs to understand all of its elements.

2.1.1 Node.js

Node.js is one of the most popular open-source, cross-platform runtime environment for executing JavaScript code out of the web browser. Node.js is often used to build backend services such as API or Application Programming Interface. Backend services using Node.js are highly scalable and can handle real-time and data-intensive applications. Node.js popularity can be seen in the number of significant technology companies using it as a solution for their backend services. Since Node.js is built based on the JavaScript V8 Engine from Google Chrome, it is widely used by Netflix, NASA, Trello, PayPal, LinkedIn, Walmart, Uber, Twitter, Yahoo, eBay, GoDaddy, and so on. In addition, Node.js is used in the MERN stack because it utilizes JavaScript, which enables developers to apply their existing bits of knowledge in the frontend and transition to full-stack development.

Node.js is an asynchronous event-driven JavaScript runtime which means it uses a single-threaded event loop architecture. Requests from the client-side sent to the server will process each one of them and then return the response. Multiple threads are going to be used to handle multiple requests at the same time. Figure 1 shows the overall picture of Node.js architecture.

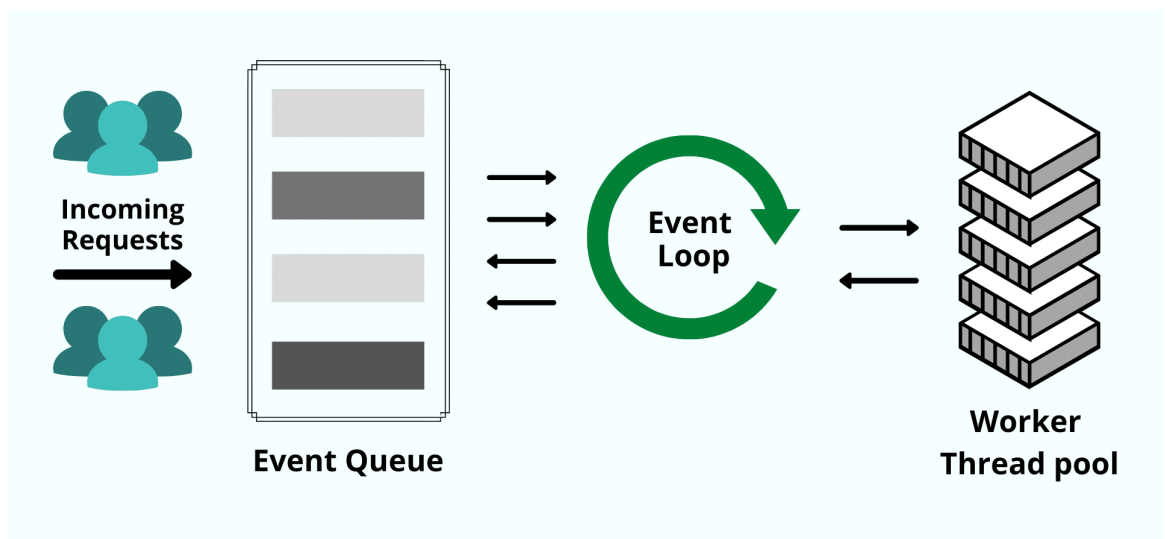


Figure 1. The architecture of Node.js server

Node.js server handles requests in a non-blocking fashion. Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem. Instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations the response comes back (Introduction to Node.js). In a real-world example, Node.js will not wait for the API to run entirely and return a response instead, it will move on to the following API request. This also applies to I/O operation in Node.js servers to save processing power, which boosts performance. The architecture Node.js follow enables it to handle thousands of simultaneous requests while maintaining a high-performance level.

```
38  const http = require('http');
39
40  const hostname = '127.0.0.1';
41  const port = 3000;
42
43  const server = http.createServer((req, res) => {
44    res.statusCode = 200;
45    res.setHeader('Content-Type', 'text/plain');
46    res.end('Hello World');
47  });
48
49  server.listen(port, hostname, () => {
50    console.log(`Server running at http://${hostname}:${port}/`);
51  });
```

Figure 2. Example of a simple Node.js server application

In Figure 2, we can see the setup of a Node.js server with HTTP requests.

2.1.2 React.js

React is a Javascript frontend framework that Facebook develops. Since the beginning of its release, it has consistently remained one of the most popular frontend frameworks. The main goal of React is to make it painless for developers to build an interactive user interface. React works based on states and components. This declarative view enables this framework to render just the right component when any state changes conveniently.

Component is the primary building block of React. To be able to achieve complex UIs and logic, each component has the ability to manage its own state. These components can be reusable, which is why React is one of the first choices for building a scalable web application. Reusing components can result in various development advantages, such as:

- Shorten development/refactoring times
- Reduce the number of bugs
- Improve debugging/maintaining process
- Achieve consistent throughout the codebase

React component works in a one-way flow. The component would take in Input known as Props and then render/output React element to the DOM. In React, components can be defined as classes or functions. When applying to the development process, it is advised to use functional components as much as possible.

A functional component is nothing more than a standard JavaScript function. As a function, it takes props as an argument, and returns React elements. The way a functional component return React element is different from a Class component which needs to have a separate render function. Furthermore, a functional component is considered a stateless component. This means the component will only receive data and then render it out in the form of a React element. We can also say that their primary responsibility is to render out the user interface.

However, since version 16.8 of React, where Hooks is introduced, functional component has become much more powerful. In addition, with the help of Hooks, a functional component can become stateful. Since then, it has been much preferred to use functional components since it has fewer codes to manage and is much easier to read. Furthermore, we can easily create and manage states without using a constructor. As a result, this approach to development will reduce the overall maintenance effort on the project.

```

7  const useBooks = (searchInput: string): [Book[]] => {
8    const books = useSelector((state: AppState) => state.book.books)
9    const [filteredBooks, setFilteredBooks] = useState<Book[]>([])
10   const dispatch = useDispatch()
11
12   const getBooks = useCallback(() => {
13     dispatch(fetchBooks())
14   }, [dispatch])
15
16   useEffect(() => {
17     getBooks()
18   }, [dispatch, getBooks])
19
20   useEffect(() => {
21     setFilteredBooks(
22       books.filter((book) => {
23         return book.title.toLowerCase().includes(searchInput.toLowerCase())
24       })
25     )
26   }, [searchInput, books])
27
28   return [filteredBooks]
29 }
30
31 export default useBooks
32

```

Figure 3. React functional component using Hooks

The functional component in Figure 3 uses the `useState` hooks to control the book's data within its scope. Sophisticated handling logic does not need to be passed down from the parent component.

A class component is much more complicated than a functional component, which means it has more code to maintain. To create a class component, you need to extend it from `React.Component`. The class component will inherit the render function with this extension to display the React element. A functional component accepts props as an argument, and then we can directly destructure them. On the other hand, in a class component, one has to refer to the props object with the keyword `'this'` to access any props passed down.


```

15 class ClassComponent extends React.Component {
16   constructor(props) {
17     super(props);
18     this.state = {
19       count: 0
20     };
21   }
22   render() {
23     const { title, description } = this.props;
24     return (
25       <div>
26         <h1>{title}</h1>
27         <p>{description}</p>
28         <p>count: {this.state.count} times</p>
29         <button onClick={() => this.setState({ count: this.state.count + 1 })}>
30           Add
31         </button>
32       </div>
33     );
34   }
35 }

```

Figure 4. A class component in React

Furthermore, the handling state process in a class component is also different from a functional component. Instead of calling a simple `useState()` hook, we have to initialize our state with the constructor as in Figure 4. The constructor will be called along with `super(props)` before any of our components is mounted.

As we can see, there are both pros and cons to functional or class components, but the modern React development, functional component has the upper hand in terms of benefits that it brings to the table.

2.1.3 MongoDB

MongoDB is a cross-platform document-oriented database program that is open source. MongoDB is NoSQL, its data is in the form of JSON-like document with optional schemas. The main attraction point of MongoDB is that it is very flexible and easy to scale.

MongoDB stores its data in documents and collections, unlike relational databases where tables and rows are used. A document is the basic unit of data in MongoDB. Their structure consists of multiple pairs of keys and the values,

along with a primary key to distinguish them from other documents in a collection. We can have various collections with a specific schema enforced with this approach. This will eliminate the need to filter out different document types.

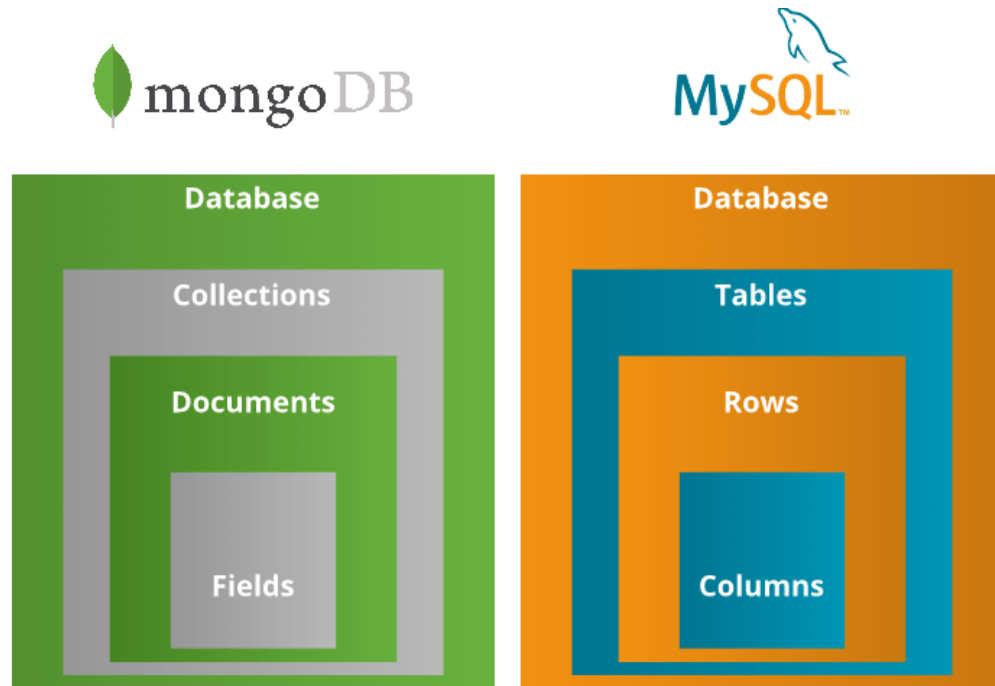


Figure 5. The database structure of MongoDB compare to MySQL

The difference between MongoDB data structure and conventional SQL database can be seen in Figure 5. MongoDB has a much more flexible structure instead of strict rows and columns.

```

1  _id: ObjectId('60536383c1b0810b7c2a6da6')      ObjectId
2  isbn: "9781449331818"                          String
3  title: "Learning JavaScript Design Patterns"    String
4  author: "Addy Osmani"                          String
5  publishedDate: "2012-07-01T00:00:00.000Z"      String
6  pages: 254                                     Int32
7  coverUrl: "http://covers.openlibrary.org/b/isbn/9781449331818.jpg" String
8  description: "With Learning JavaScript Design Patterns, you'll learn how to write be" String

```

CANCEL UPDATE

Figure 6. Example of a document element in MongoDB

A document in MongoDB always has an id Object to identify itself. For example, figure 6 is a book document of our web application database. The JSON-like structure makes it easy to modify and add data to the document.

MongoDB has five main technical features:

- **Support ad hoc queries:** In MongoDB, you can search the data you need by their fields. MongoDB supports field queries, range queries, and regular expression searches.
- **Indexing:** Indexing can significantly improve performance by reducing the time it takes to complete each query. MongoDB offers a broad range of indices and features with language-specific sort orders that support complex access patterns.
- **Replication:** Many potential points of failure can be avoided by using replication. These include both software like crashes or hardware failure. It can also help with load balancing when different clients access the same data.
- **Sharding:** This is the process of fragmenting an extensive database into smaller collections or so calls 'shards.'
- **Load balancing:** All of the features above contribute to the ability to handle large-scale databases. This also eliminates the need to use an external load balancer.

2.1.4 ExpressJS

Express a free and open-source backend web application framework for Node.js. Its primary purpose is to build web applications and APIs. Express has consistently been one of the most popular Node.js frameworks because of its flexibility and minimalism. For example, Express is perfect for building a web API that handles multiple GET, PUT, POST, and DELETE requests.

The main feature of Express:

- Fast development time
- Provide middleware
- Routing
- Templating
- Debugging

When receiving a request, an Express server will pass it through layers of middleware. The request will be passed through each middleware using the `next()` function.

2.2 Redux

Using React.js as a frontend framework, we can utilize its components and state to manage and display data. However, as the web application grows, there will be more and more components and a massive amount of data. Therefore, it will not be efficient to keep passing state from the parent component to its child in a deeply nested tree. Redux as a state management library for React.js, is the perfect solution for this problem.

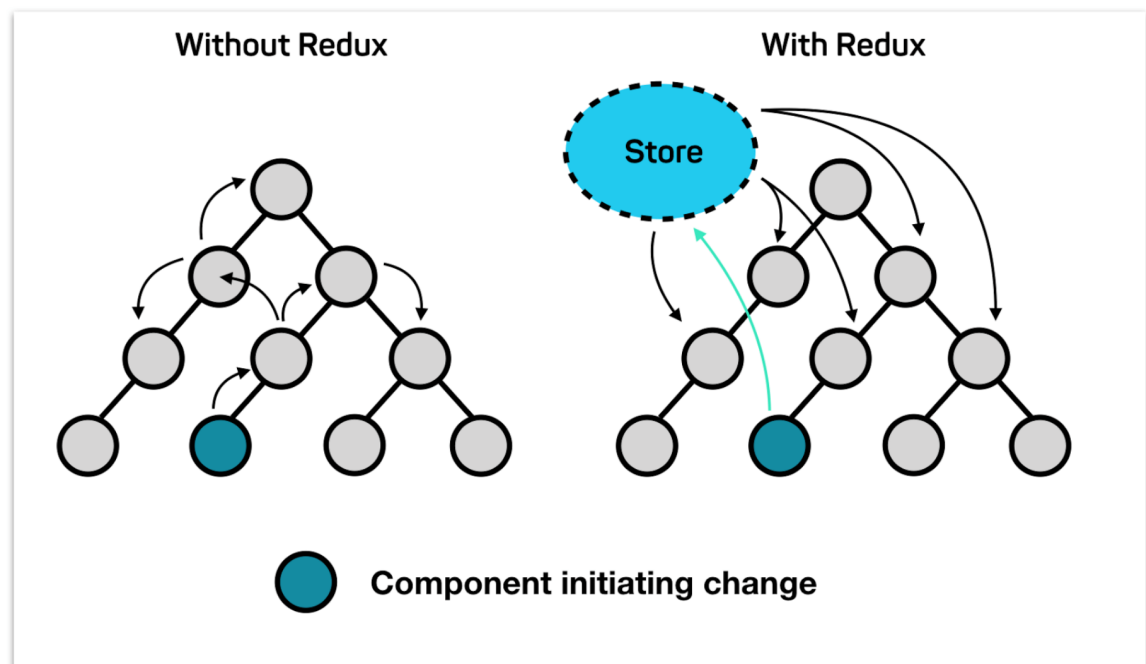


Figure 7. How states are handled with Redux

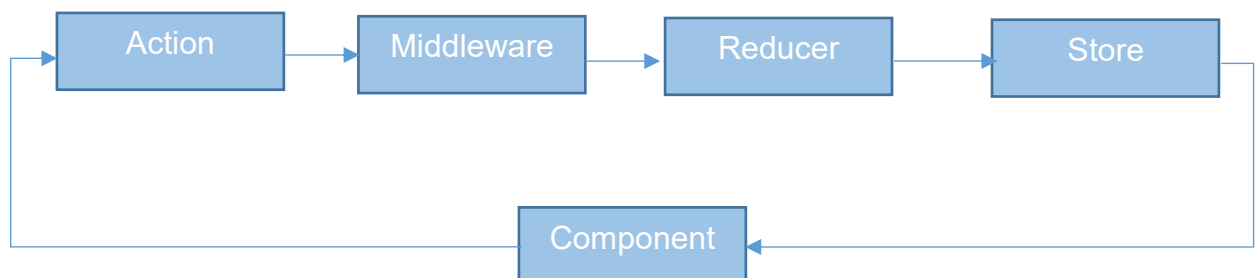


Figure 8. How components interact with Redux

The central concept of Redux includes Action, Store, Reducer, and Middleware. The flow of Redux action can be seen in Figure 8, where the data is handled outside of the component.

Redux takes the state outside of the component tree and centralizes it to a store. The store holds the reducer and the application's state and provides access to current actions/states. In Figure 7, we can see the benefit of having Redux, where there is only two clear data flow from the store to the component. The reducer is a function that takes in the state and the action to return a new state. The action object represents the type of modification we want to the state. We can see that the component only interacts with the action and receives the end state from the store. All the handling processes are separated from the component.

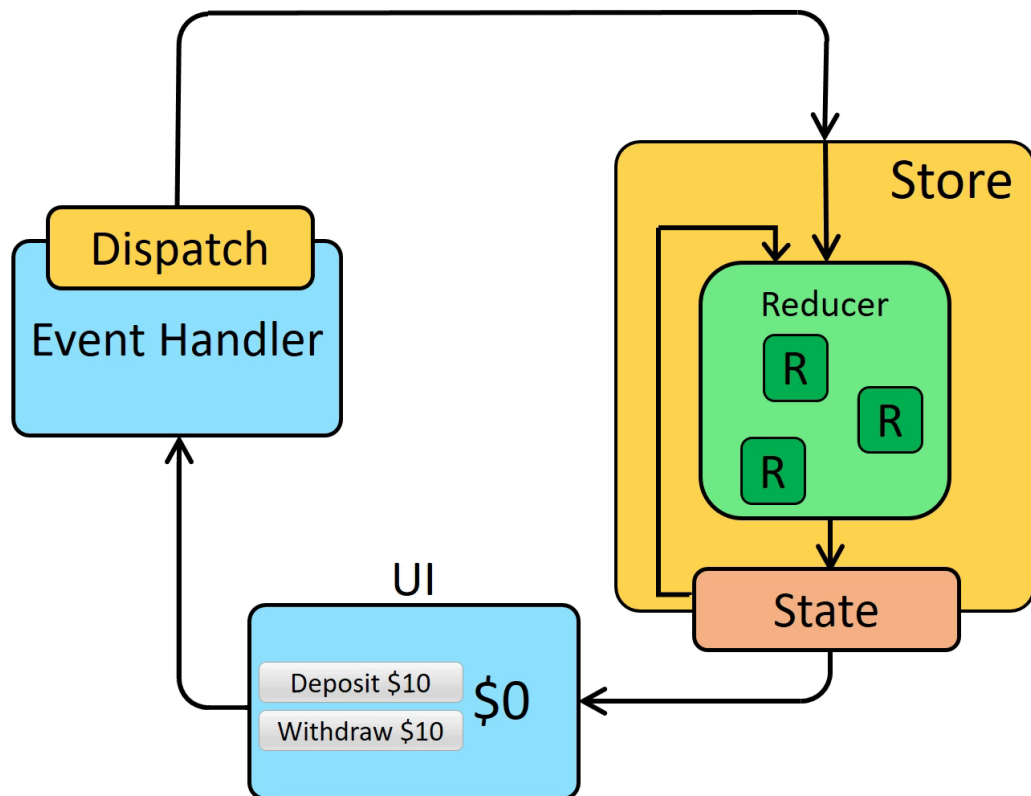


Figure 9. State mutation using action in Redux

Generally, a flow of a component using Redux would start with it dispatching an action. As in Figure 9, when the user triggers the Deposit or Withdraw action, the

correlated action will be dispatched. The Redux store will receive that action, pass it through necessary middlewares, and finally to the reducer. Next, the reducer will look into the action type and payload to update the state accordingly. Finally, the component will receive the new state and re-render when the state is updated.

2.3 TypeScript

TypeScript is a programming language that is built on top of JavaScript. It was developed and made public in October 2012 by Microsoft. Since its release, it quickly gained popularity as one of the most programming languages. In 2017, TypeScript became the official supported language of Google. To truly understand TypeScript, let us dive deeper into the definition and the intention behind TypeScript.

TypeScript can be defined as a strict superset of JavaScript. TypeScript solves one of the most prominent problems of JavaScript by adding a robust type system. Furthermore, TypeScript and JavaScript are interchangeable. Therefore, you can compile TypeScript code into vanilla JavaScript. This means that both TypeScript and JavaScript code can work together in a project. The typing in TypeScript can be optional. With the close nature of JavaScript, the learning curve of TypeScript is not too hard. Developers can quickly adapt to TypeScript with their knowledge of JavaScript. The main intention of TypeScript is to create scalability in a project. TypeScript excels in a large-scale project with many contributors. In these scenarios, you want your code to be as consistent and bug-free as possible, and TypeScript does precisely that.

Usually, in JavaScript, where type is not enforced, there will be errors that will only appear in runtime. These errors can be uncaught and surface in the production environment where users use the web application. Most of these errors can be caught immediately in the development process with typed code.

```
function capitalizeString(string) {
  return string.toUpperCase();
}

capitalizeString();
```

Figure 20. Example of JavaScript function with bug

Let us look at the example in Figure.10 above of a JavaScript function with a bug that can cause the web application to crash. However, the code above would still compile without any errors, which means we only know when a user experiences it in the web application. This is caused (by the fact that) the function above accepts a string as an argument, and when we call it below, we do not pass any string to it.

```
3  export const capitalizeString = (string: string) => {
4      return string.toUpperCase();
5  }
6
7  const capitalizeString: (string: string) => string
8
9  Expected 1 arguments, but got 0. ts(2554)
10 example.ts(3, 34): An argument for 'string' was not provided.
11 View Problem No quick fixes available
12 capitalizeString();
```

Figure 31. Example of a TypeScript function with a bug

On the other hand, the function in Fig.11 immediately throws an error when we do not pass a string to it. TypeScript helps negate these types of mistakes from reaching the production environment.

The primary type in TypeScript includes the following:

- Any
- Primitive: Number, Boolean, String, Void, Null/Undefined
- Array
- Enum
- Object
- Tuple
- Never

These types can be in the form of alias, literal, or mapped. TypeScript also enables us to create an interface that represents an object type.

2.4 Practical project structure plan

To have a scalable web application, we have to have a good project structure from the beginning to accommodate all tech-stack that we are using and also the growth in the future.

On the frontend side, when we want to have a file system where we can have easy access to our reusable components and our Redux actions and reducer.

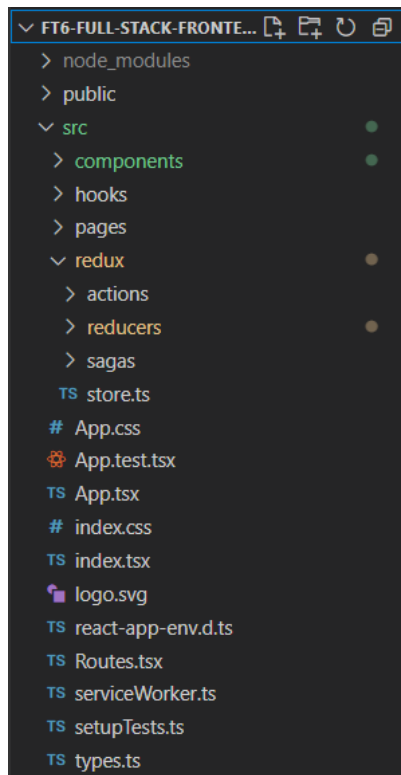


Figure 42. Frontend structure of the project

The pages folder will hold the main page of our application. These pages will act as a wrapper for our components, and it is also the primary place where we are handling our state. For Redux, we will separate folders for action, reducer, and saga.

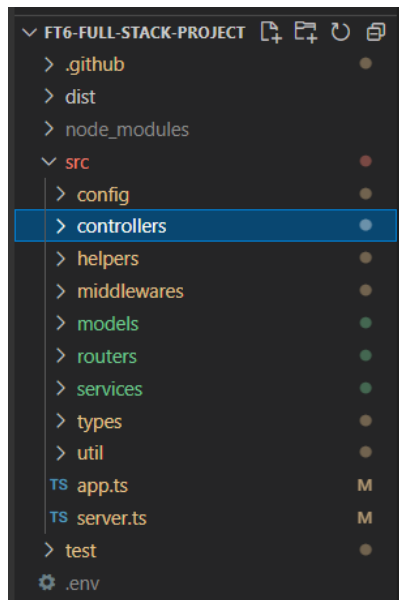


Figure 53. Backend structure of the project

3 PRACTICAL IMPLEMENTATION

In the implementation part, we will look at the book store web application to see the benefits of using TypeScript and Redux inside a MERN-stack web.

3.1 Backend Express server

3.1.1 Backend structure

One of the primary functions of our Express server is to manage and modify the data in our database. Therefore, one of the first things we need to set up in our backend structure is to create the data models and connect them with our MongoDB database. In our server.ts file, we will configure our Express server to connect with the designated MongoDB database URL.

```

7   const mongoUrl = MONGODB_URI
8   mongoose
9     .connect(mongoUrl, {
10      useNewUrlParser: true,
11      useCreateIndex: true,
12      useUnifiedTopology: true,
13    })
14    .then(() => {
15      // Start Express server
16      app.listen(app.get('port'), () => {
17        console.log(
18          ' App is running at http://localhost:%d in %s mode',
19          app.get('port'),
20          app.get('env')
21        )
22        console.log(' Press CTRL-C to stop\n')
23      })
24    })
25    .catch((err: Error) => {
26      console.log(
27        'MongoDB connection error. Please make sure MongoDB is running. ' + err
28      )
29      process.exit(1)
30    })

```

Figure 64. Database connecting structure on Express sever

To access and modify data on our server, we need to declare our data model. Then, with the help of TypeScript in our data schema, any information we get or modify will be consistent with typing that we designed for our database.

```

4   export type UserDocument = Document & {
5     firstName: string
6     lastName: string
7     email: string
8     joinDate: Date
9     borrowedBooks: String[]
10  }
11
12  const userSchema = new mongoose.Schema({
13    firstName: {
14      type: String,
15      index: true,
16    },
17    lastName: {
18      type: String,
19      required: true,
20    },
21    email: {
22      type: String,
23      required: true,
24    },
25    userType: {
26      type: String,
27      enum: ['USER', 'ADMIN'],
28      default: 'USER',
29    },
30    borrowedBooks: [{ type: String, ref: 'Books' }],
31  })
32
33  export default mongoose.model<UserDocument>('Users', userSchema)
34

```

Figure 75. User schema in Express server

According to Figure 15, all values of our User document are typed with TypeScript. Therefore, in sending requests to our server, any mismatch of type in the data will be detected immediately.

With the database connection set up, we can start to create the routing logic of our backend server. The routing process involves mapping our API endpoints to the specific server code that handles that request.

```
5  const router = express.Router()
6
7  // Every path we define here will get /api/v1/books prefix
8  router.get('/', findAll)
9  router.delete('/:bookId', deleteBook)
10 router.post('/', createBook)
11
12     Method Path Handle function
13 export default router
13
```

Figure 86. Router books endpoints in our Express server

In Figure 16, we can see the structure of our routing process. The 'router' represents the route handler for our book endpoints. In our BookStore application, we have a separate route handler for each database. This will ensure a consistent backend structure that can be scaled up when we have more data. The route handler will first look at the method of request. These methods can be get, post, delete, or put. Next, we specify the path name for that endpoint that contains the parameter. Lastly, we pass in the function that will handle that specific request. This function will receive three objects: the request, the response, and the next function. Let us look at an example of a handler function in our backend server.

```

83 // GET /books
84 export const findAll = async (
85   req: Request,
86   res: Response,
87   next: NextFunction
88 ) => {
89   try {
90     res.json(await BookService.findAll())
91   } catch (error: any) {
92     next(new NotFoundError('Books not found', error))
93   }
94 }
95

```

Figure 97. Handle function for book request

Figure 17 shows the `findAll` function for our request to get all books inside of our database. This function is responsible for calling the correct service and setting up the response. All of the functions to get or modify the web application data are defined in their own services file. By having separate service files for each of our data collection, it is much easier to add and manage the features of our web application.

```

3  ✓ function create(book: BookDocument): Promise<BookDocument> {
4    |   return book.save()
5    | }
6
7  ✓ function findAll(): Promise<BookDocument[]> {
8    |   return Books.find().sort({ name: 1, publishedDate: -1 }).exec()
9    | }
10
11 ✓ function deleteBook(bookId: string): Promise<BookDocument | null> {
12 |   return Books.findByIdAndDelete(bookId).exec()
13 | }

```

Figure 108. MongoDB query in Express server

In our Node.js environment, we can use the MongoDB query to access and modify our data. In Figure 18, the Express server can query and sort data with multiple parameters. The ability to handle complex queries is one of the strong points of the Express-MongoDB server. The Express server is the middle man between our React web application and our MongoDB database. The Express server handles the REST API calls from the frontend by connecting with the database.

The use of TypeScript can also be seen in Figure 18. The response of our function is typed to ensure the consistency of data on the front end and backend. This will ensure our front end will always get the correct data type to handle.

3.1.2 Features

The main features of our server are data access/modification, user log in and authentication. Our web application uses Google accounts and JWT authentication to handle the user's login process.

Our server uses a token-based approach when it comes to authentication and security. When the users successfully log in with their Google account, the server sends back a JWT token in the response. This token will be used to remember the user in future login and identify the valid client-side request to the server. This modern way of authentication ensures that the users will have an uninterrupted experience with our web application.

Passport.js is used to handle the Google and JWT logic on our Express backend server. Passport.js is a authentication middleware for Node.js server. It is widely used due to its flexibility and modularity. Passport.js includes a wide range of authentication strategies: username and password along with various famous third-party accounts like Google, Facebook, and Twitter.

```

8   const google = new googleStrategy(
9     {
10    clientID: process.env.GOOGLE_CLIENT_ID,
11    },
12    async (parsedToken: any, googleId: any, done: any) => {
13      const { given_name, family_name, email } = parsedToken.payload
14
15      const user = new Users({
16        firstName: given_name,
17        lastName: family_name,
18        email,
19      })
20      const loggedUser = await UserService.findOrCreate(user)
21
22      const token = JWT.sign(loggedUser.toJSON(), JWT_SECRET)
23      done(null, { token })
24    }
25  )
26
27  const jwt = new JWTStrategy(
28    {
29      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
30      secretOrKey: process.env.JWT_SECRET,
31    },
32    async (jwtPayload, done) => {
33      const { email } = jwtPayload
34      const user = await UserService.findByEmail(email)
35
36      if (!user) return done(null, false)
37
38      return done(null, user)
39    }
40  )
41
42  export default { google, jwt }
43

```

Figure 119. Google and JWT strategies for Passport.js

Figure 19 displays the strategies that we define for our google and jwt authentication. This middleware will be applied to our authentication endpoint. The Google strategies will handle the Google response triggered by the front end. After receiving the response, the middleware will check our database for an existing user. If not, it will create a new one. Finally, a token signed with the user information will be sent back to the client-side. That token will then be saved for future requests.

Our backend also includes a token verify middleware to check whether the requests our server received are coming from our client and not from other

malicious sources. Initially, the token we send back to the client is signed with our JWT secret key in the login process.

```
6  const tokenVerify = async (req: Request, res: Response, next: NextFunction) => {
7    passport.authenticate('jwt', (error, user) => {
8      if (error) return next(new InternalServerError())
9      if (!user)
10       return next(new UnauthorizedError('Invalid token, Please login again'))
11       req.user = user
12       return next()
13     })(req, res, next)
14  }
15
16  export default tokenVerify
17
```

Figure 20. Token verify middleware

In Figure 20, the middleware calls the passport's authenticate function targeting our jwt strategy. The strategy will extract the token from the request header and then check its authenticity using the secret key. Only when the token is valid will the middleware return a next() function that continues with the request's handle function. Therefore, our server will only send back data to valid and authenticated users.

3.2 Front end

The front end of this web application will handle displaying information to our users and communicate with the backend to access the database. The example project will display a book store where users can search and borrow books when logging in to the app.

The whole front end is broken down into small components for development. It is much easier to track, manage, and reuse your code with smaller components. Reusable components reduce the amount of code in the project. When refactoring the component, the developer only needs to modify one file, and changes will be applied anywhere that component is being used.

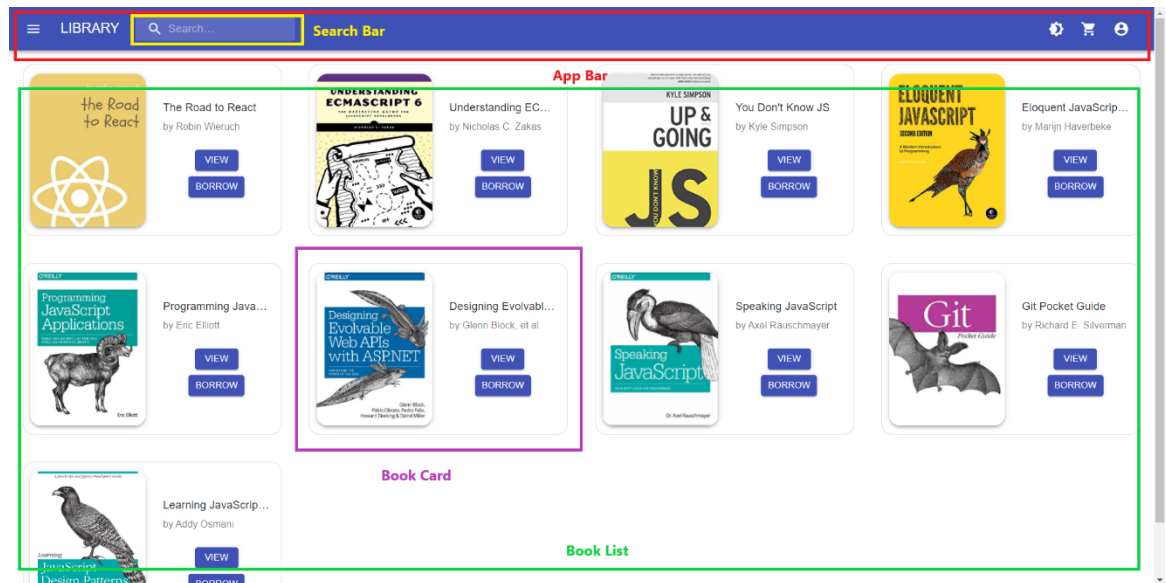


Figure 21. Front end component decomposition

Figure 21 shows how our front end applies component decomposition. Each component should serve only one purpose, like displaying content or holding inputs. Generally, states should be handled in components that stay in the upper tree. Smaller components below main task are to display content as it is.

```

9  const inCart = localStorage.getItem('inCart')
10 const darkMode = localStorage.getItem('darkMode')
11 const token = localStorage.getItem('token')
12 const loggedUser = localStorage.getItem('loggedUser')
13 const initState: AppState = {
14   book: {
15     books: [],
16   },
17   inCart: {
18     inCart: inCart ? JSON.parse(inCart) : [],
19   },
20   ui: {
21     darkMode: darkMode ? JSON.parse(darkMode) : false,
22   },
23   loginDetail: {
24     token: token ? token : '',
25     user: loggedUser
26       ? JSON.parse(loggedUser)
27       : {
28         email: '',
29         firstName: '',
30         lastName: '',
31         userType: '',
32         _id: '',
33         borrowedBooks: [],
34       },
35   },
36 }

```


Figure 22. The initial state of the web application

To display all the information fetching from the backend, the front end needs a place to store them, which is our Redux. The initial state is set in the store when the application first runs. In our web application, we use local storage to save many of our data, including book data, tokens, and user information. Therefore the first step is to check if there were any saved data inside our local storage. If not, the state is initialized with the default data structure that we defined (Figure 22).

The reducer and action also need to be set up to handle and modify all of our states. First, let us look at an example of how our book's state is handled.

```
6 export function fetchBooksSuccess(books: Book[]) {
7   return {
8     type: FETCH_BOOKS_SUCCESS,
9     payload: {
10      books,
11    },
12  }
13 }
14
15 export function fetchBooks() {
16   return ((dispatch: Dispatch) => {
17     return axios
18       .get('http://localhost:5000/api/v1/books')
19       .then((res) => {
20         dispatch(fetchBooksSuccess(res.data))
21       })
22       .catch((e) => {
23         console.log(e)
24       })
25   })
26 }
```

Figure 23. Fetching book data action

Figure 23 is the action for fetching our book data. Upon successfully fetching the data, it will trigger our action and send it to the reducer. The reducer will check for the exact type and then modify our state with the payload from the response.

```

3  export default function book(
4    state: BookState = {
5      books: [],
6    },
7    action: BookActions
8  ): BookState {
9    switch (action.type) {
10   case FETCH_BOOKS_SUCCESS: {
11     const { books } = action.payload
12
13     return { ...state, books }
14   }
15   default:
16     return state
17   }
18 }
19

```

Figure 24. Book reducer

In the book reducer, the initial state is spread out, and the new book data will be appended to that array (Figure 24). After the state is updated, any components using that specific state will receive the new state and re-render.

Throughout the process of updating the state reducer, the data stored in the local storage also need to be updated accordingly. Our web application uses Redux Saga to solve this problem. Redux Saga is a middleware library that allows Redux to interact with data outside its store asynchronously. These operations are also called as side effects. The function inside our sagas is set up in a way that it will listen to action being triggered in our web application. The function then can access the action's payload to initiate the necessary side effect with data.

```

5  function* toggleTheme(action: ToggleDarkMode): any {
6    const localData = yield localStorage.getItem('darkMode')
7    const darkMode = localData ? JSON.parse(localData) : false
8    localStorage.setItem('darkMode', JSON.stringify(!darkMode))
9  }
10
11 export default [takeLatest(TOGGLE_DARK_MODE, toggleTheme)]
12

```

Figure 25. Saga function for UI state

The function in Figure 25 is listening to the theme toggle action to make changes to the theme variable inside our local storage. Each time the action is called in our front end, the state will be updated in our Redux, resulting in visual change

for users. However, each time the web application page is reloaded, the state inside our Redux is reset. Therefore, the store will check our local storage for saved settings and apply them to the new state. This way, user preference will be remembered, which improves the overall user experience.

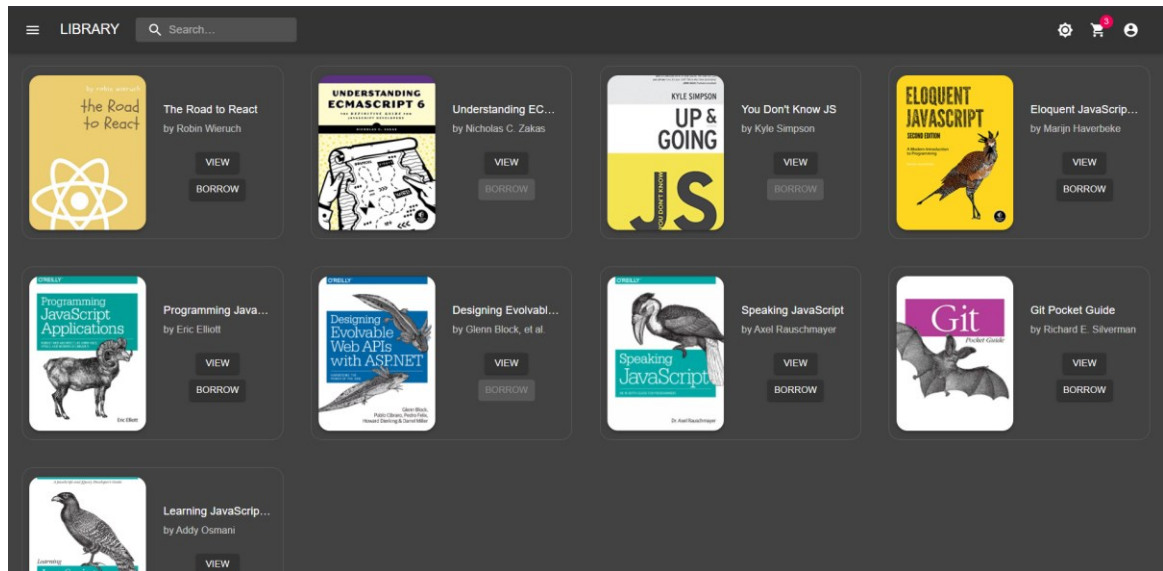


Figure 26. Web application in dark theme

The web application in dark theme (Figure 26) is one example of how using Redux as state management can enable you to quickly implement robust features on our application. Any component inside our component tree can access the UI theme state and render the content accordingly.

```

<IconButton
  color="inherit"
  onClick={() => {
    dispatch(toggleDarkMode())
  }}
>
  {darkMode ? <BrightnessHighIcon /> : <Brightness4Icon />}
</IconButton>

```

Figure 27. Icon component

The IconButton component can have different visuals based on the theme state set by the user (Figure 27). These states would have to be passed down from the parent component with traditional state management, resulting in complicated logic and a nested state tree. A nested state tree makes refactoring the code a

much more complicated task since the changes need to be applied to every place where the state is being passed.

One more example of utilizing Redux in our web application is the borrow and return feature. When the users perform the task of borrowing or returning a book, the application not only sends out an action to modify our Redux state but also sends out a POST request to our server. This POST request will update the user borrowed book array so that their data can be loaded in future login. In the action file, we create a function to make the POST request to our server and then return a dispatch action along with the response from our server.

```
27 export function addBorrowedBook(bookId: string) {
28   return {
29     type: ADD_BORROWED_BOOK,
30     payload: {
31       bookId,
32     },
33   }
34 }
35
36 export function postBorrowedBook(bookId: string) {
37   return ((dispatch: Dispatch, state: any) => {
38     return axios
39       .post(
40         `http://localhost:5000/api/v1/users/borrow/${
41           state().loginDetail.user._id
42         }`,
43         { bookId: bookId }
44       )
45       .then((res) => {
46         if (res.status === 200) dispatch(addBorrowedBook(bookId))
47       })
48       .catch((e) => {
49         console.log(e)
50       })
51   })
52 }
```

Figure 28. Add borrowed book function

The function will first make the POST request to our server using Axios and the data that need to be updated in the parameter (Figure 28). If only when the server returns a successful response with the status of 200, we would start to modify our state in Redux. With this approach, the data between our client side

and our database is always consistent. The risk of having mismatched data or updating information without saving it to the database is eliminated.

```
14  axios.interceptors.request.use(  
15    (config) => {  
16      config.headers.authorization = `Bearer ${token}`  
17      return config  
18    },  
19    (error) => {  
20      return Promise.reject(error)  
21    }  
22  )
```

Figure 29. Axios interceptors settings

Furthermore, to enforce the integrity of our data, the web application is configured with an Axios interceptor. This interceptor will append our request to the server with the Bearer token from the login process. The middleware in our server will then verify this token.

The borrowing and return feature also shows the flexibility of Redux in state management. The state of these books is accessed by multiple components in the application, including the cart icons from the app bar, the book display component, and the book cart component. First, the App bar will look at the book array to render the number of items in our cart to our icons. Next, the book card component will conditionally render the borrow button based on the state of whether the book has already been borrowed or not. Finally, the cart will render all the books inside the borrowed book array. If we have to pass all of these down from component to component, keeping track of our code would be tough. Any error and refactoring would affect the whole component tree. Thanks to Redux and its ability to access the state dynamically from within the component itself, this process has become more approachable.

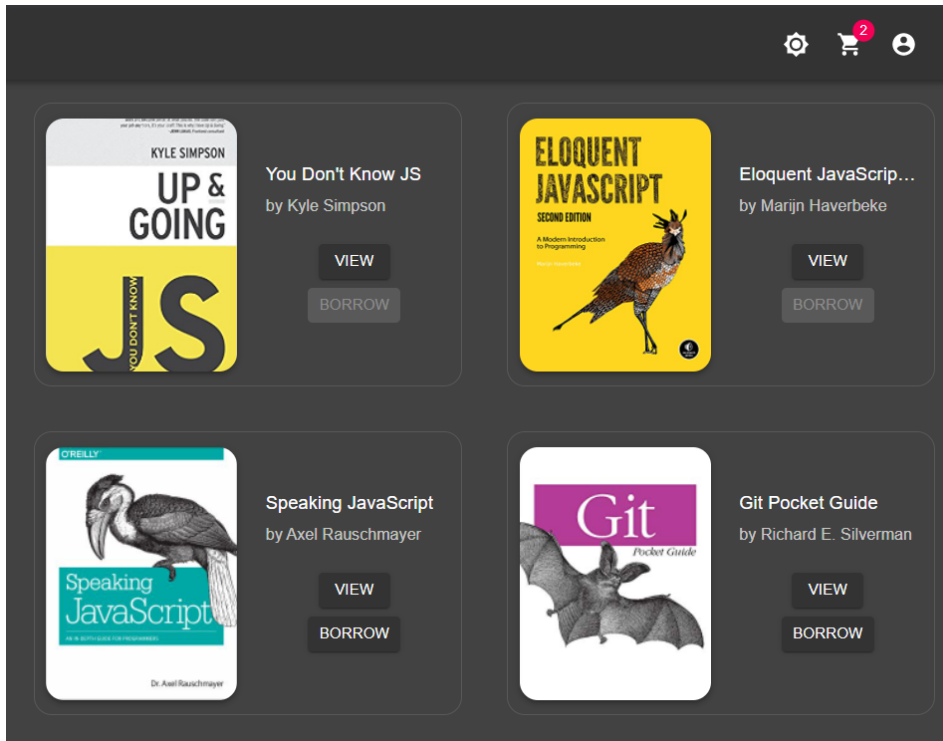


Figure 30. Cart icon and button component

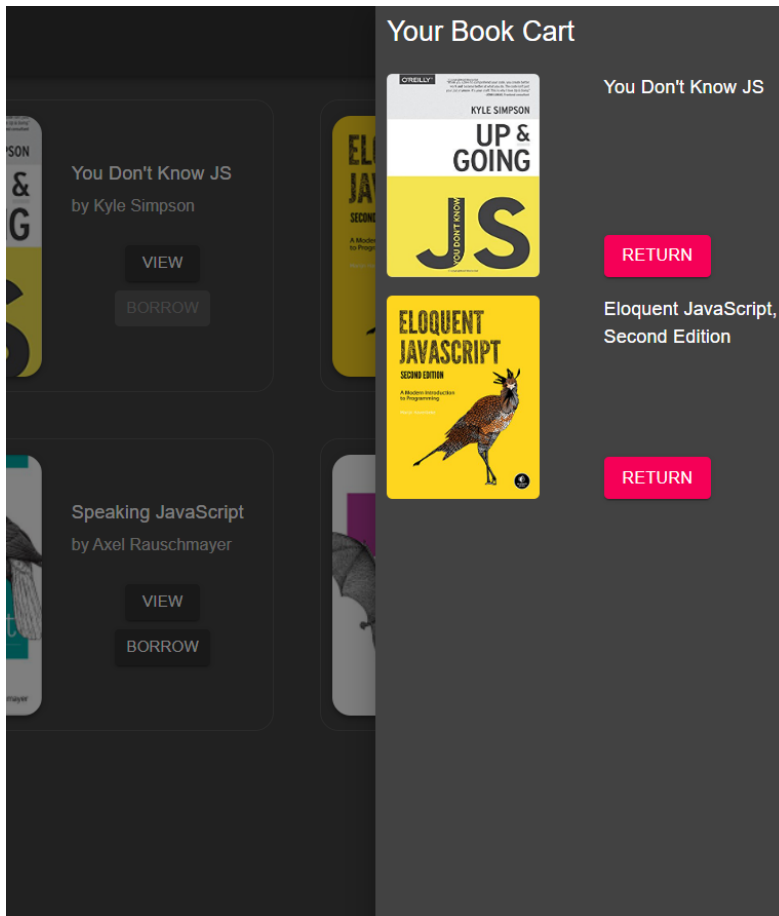
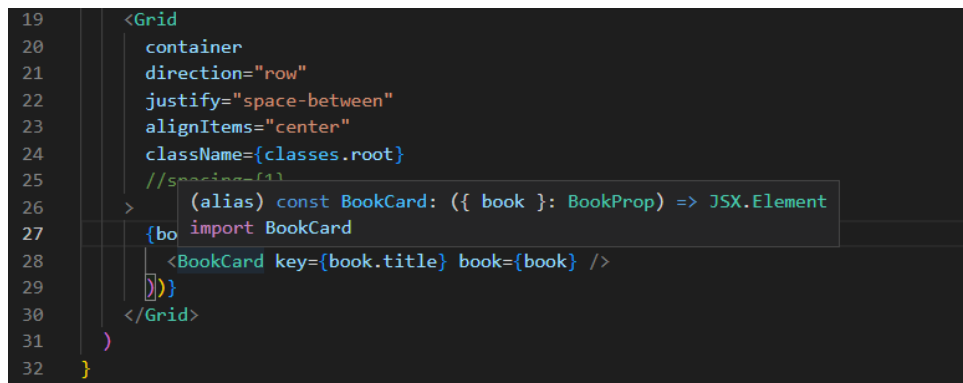


Figure 31. Cart component

In Figure 30, we can see the conditional visual effect of the application using the state in the Redux. This also applied to the cart component in Figure 31.

Besides Redux, the frontend development process also benefits a lot from using TypeScript. By enforcing TypeScript in each of our components, we can easily see what type of props it needs.

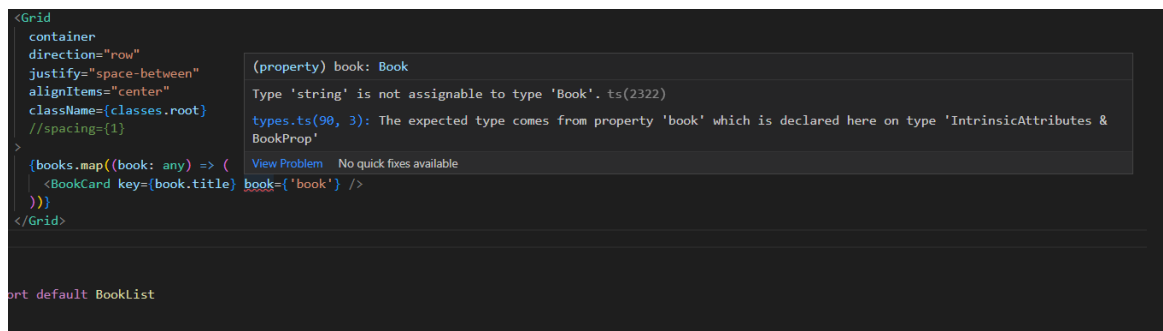


```

19     <Grid
20       container
21       direction="row"
22       justify="space-between"
23       alignItems="center"
24       className={classes.root}
25       //spacing={1}
26     >
27       (alias) const BookCard: ({ book }: BookProp) => JSX.Element
28       {books.map((book) => (
29         <BookCard key={book.title} book={book} />
30       ))}
31     </Grid>
32   )
  
```

Figure 32. BookCard component

In Figure 32, the BookCard component is being called inside its parent component, which is the BookList. When hovering over the component, the IDE shows that it requires a prop under the name 'book' with the type of BookProp. In this way, when multiple developers contribute to the project, they can easily interpret others people's code and reuse it.



```

<Grid
  container
  direction="row"
  justify="space-between"
  alignItems="center"
  className={classes.root}
  //spacing={1}
>
  {books.map((book: any) => (
    <BookCard key={book.title} book={'book'} />
  ))}
</Grid>
  
```

(property) book: Book
 Type 'string' is not assignable to type 'Book'. ts(2322)
 types.ts(90, 3): The expected type comes from property 'book' which is declared here on type 'IntrinsicAttributes & BookProp'

View Problem No quick fixes available

port default BookList

Figure 33. BookCard component

TypeScript also prevents us from passing props with the wrong datatype. This error can easily lead to undetected errors that may end up in the user's environment. However, when strict typing is enforced, the error can be seen

immediately in the development process. For example, in Figure 33, the BookCard is passed the book prop as a string which is not the correct type for that prop. In this case, the IDE immediately shows an error stating that the type string is not assignable to the Book type of our prop. Declaring TypeScript all across the code may take more development time, but its advantages outweigh the time cost. As a result, the project code base will be more consistent and scalable for the future.

4 CONCLUSION

This thesis aims to study how a MERN stack web application can benefit from integrating Redux and TypeScript. The main point that this thesis pointed out is that Redux and TypeScript can help a web application project become more scalable in the future. The general idea of a MERN stack web application is discussed throughout this thesis. With that information, this paper then introduced the concept of Redux and TypeScript and how it is integrated into the prototype.

The prototype used in this thesis is a web application of a book store where users can log in and perform actions such as borrowing or returning books. The author demonstrated the effect of using Redux and TypeScript by showcasing the development process on both the front and back end of the web application. The prototype is a fully functional full-stack project with features including login through a Google account, borrowing and returning books based on account, and interactive UI. With the presence of Redux and TypeScript, all those features work consistently and can also be expanded and scaled in the future. Even though the paper shows that the prototype has a solid code base for further growth, there was no example of further development in this project.

In the end, this thesis has successfully proved that using Redux and TypeScript brings significant advantages to the development process. This thesis can also be a guide when it comes to planning out a web application based on a MERN stack. The project needs to have a good structure from the beginning to be able to achieve scalability.

REFERENCE

Simon Kemp 2021. DIGITAL 2021: GLOBAL OVERVIEW REPORT. WWW document. Available at: <https://datareportal.com/reports/digital-2021-global-overview-report>
[Accessed March 2022].

Coding Jitsu 2021. Node JS Crash Course 2021. WWW document. Available at: <https://dev.to/w3tsa/node-js-crash-course-2021-2n96>
[Accessed 1 April 2022].

MongoDB, Inc. 2022. MongoDB Features. WWW document. Available at: <https://www.mongodb.com/what-is-mongodb/features>
[Accessed 3 April 2022].

Dan Abramov and the Redux documentation authors 2022. Redux Essentials, Part 1: Redux Overview and Concepts. WWW document. Available at: <https://redux.js.org/tutorials/essentials/part-1-overview-concepts>
[Accessed 3 May 2022].

Microsoft, Inc. 2022. TypeScript for the New Programmer. WWW document. Available at: <https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html>
[Accessed 3 May 2022].

Microsoft, Inc. 2022. TypeScript for JavaScript Programmers. WWW document. Available at: <https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html>
[Accessed 3 May 2022].