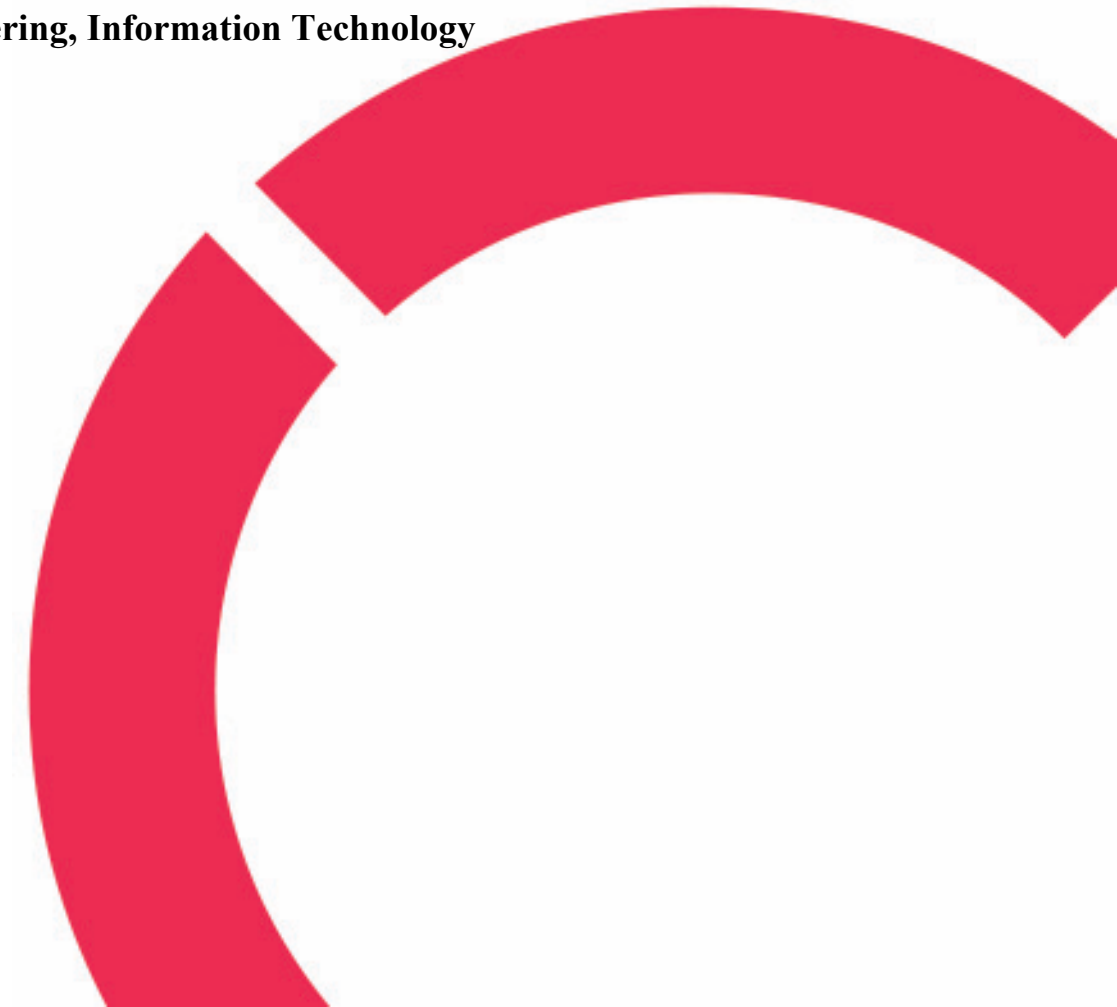
Duc Minh Nguyen

DESIGN AND IMPLEMENTATION OF A FULL STACK REACT AND NODE.JS APPLICATION

Simulating Driver's License Exams

Thesis
CENTRIA UNIVERSITY OF APPLIED SCIENCES
Bachelor of Engineering, Information Technology
May 2024



ABSTRACT

Centria University of Applied Sciences	**Date** May 2024	**Author** Duc Minh Nguyen
Degree programme Bachelor of Engineering, Information Technology		
Name of thesis DESIGN AND IMPLEMENTATION OF A FULL STACK REACT AND NODE.JS APPLICATION. Simulating Driver's License Exams		
Centria supervisor Heikki Ahonen		**Pages** 54 + 33
Instructor representing commissioning institution or company		

This thesis documents the development process of a web and mobile application that helps users in their daily practice in driver's license exams. The application is implemented by combining React for the front-end side and Node.js for the back-end. The thesis goes through the app's goals, design, and functionality, as well as React and Node.js theories with practical samples. The thesis is organized into six parts. The first part is a quick introduction and research questions. The second part is the setup of development tools in a development environment. The subsequent section covers front-end development with React, React Native, and React Native for Web, along with the design and implementation of React application in the next part. The fifth part focuses on the back end with MERN stack, and the final part focuses on implementation of Node.js app, database integration and APIs.

Key words
AWS, back-end, CSS, design, driver license, front-end, JSON, mobile, MongoDB, Node.js, React, React-Native, Redis, tests, web

CONCEPT DEFINITIONS

List of Abbreviations

AI - Artificial Intelligence
AWS - Amazon Web Services
CSS - Cascading Style Sheets
CLI - Command line interface
CI/CD - Continuous Integration and Continuous Delivery
DRY - Don't Repeat Yourself
DOM - Document Object Model
EC2 - Amazon Elastic Compute Cloud, is a web service provided by AWS
HTML - Hypertext Markup Language
JS - JavaScript
JSON - JavaScript Object Notation
JSX - JavaScript XML
JWT- Json Web Token
NPX - Node Package Execute – NPM package runner
NPM - Node Package Manager for JavaScript programming language
RN - React Native
RN Web - React Native for Web
SDLC - Software Development Life Cycle
SPA - Single page application
TS - TypeScript
UI/UX - User interface/User experiences
XML - Extensible Markup Language
YAML - Yet another markup language or YAML Ain’t markup language

ABSTRACT
CONCEPT DEFINITIONS
CONTENTS

PICTURES

TABLES

CODES

APPENDICES

1 INTRODUCTION

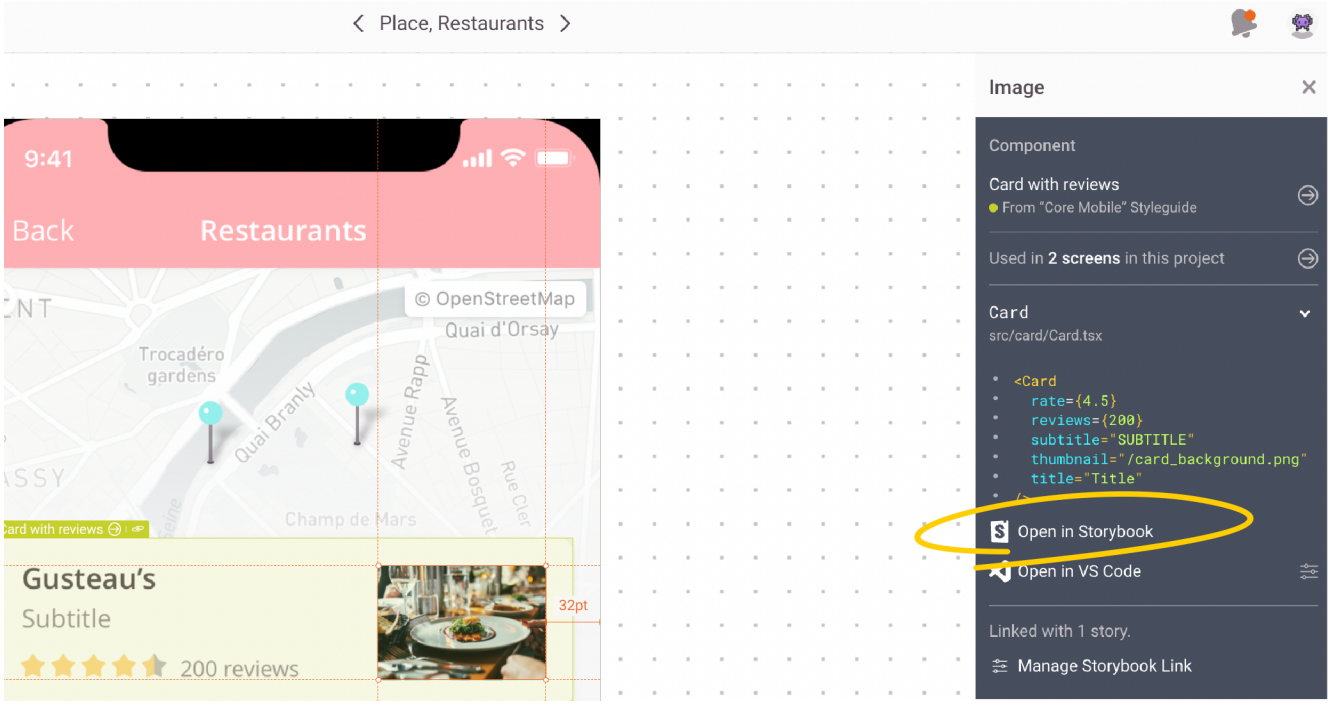
In everyday life, every single exam requires hard practice to pass. The thesis author lives in Denmark with his family while doing his thesis in Centria UAS. Based on his experiences with the driver's license theoretical exam, it could be said that the theoretical exam in Denmark differs from other countries. The fact is that the question system is not fixed and there is no common set of questions, instead, it undergoes annual changes. To pass the test, it is necessary to understand traffic laws. The exam has twenty-five questions, each with a maximum of four sub-questions. These questions are built on traffic laws and driving rules. If a participant fails a single sub-question, the question will be marked as failed as well. The participant needs a score of more than eighty percent to pass the exam. To pass the exam, it is necessary to use real experiments or learn all the questions by heart. Numerous applicants find it challenging. (Teoriklar 2023.)

The starting point of this thesis comes from a real-life problem above. The thesis project must be a cross-platform application that works well on iOS, Android, and web browser. It is a challenge, and it is motivation to apply programming skills in a real situation. The thesis answers three research questions. The first one is "Is there any technology that allows an individual programmer, or a limited resources team to efficiently develop a system like that in a short time at a low cost?". Node.js and React are the two most used frameworks/libraries among web developers worldwide as of 2023, 42.65% and 40.58% respectively (see Appendix 1). The second research question in the thesis is "Why React and Node.js are most used in web development?" Is there any special reason why developers prefer to use them for their projects? The last research question is "How React and Node.js can be applied to the thesis project in an efficient approach to accomplishing tasks?".

The trend of making SPAs with React is becoming popular among web developers. Handling routing on the client side allows users to have better experiences with smooth navigation because it avoids page refreshes (Dhiwise 2022). Especially with this thesis application, it could be used daily on a handset device, so making the best user experience is a must. Answering what is the best for all these questions is tough. It can cause an infinity of debates in developed communities. But, when considering an efficient approach for the thesis project, the combination of React and Node.js is a good choice at this moment. A brand-new application for practicing driver's license tests is a result of this thesis.

2 DEVELOPMENT TOOLS AND ENVIRONMENTS

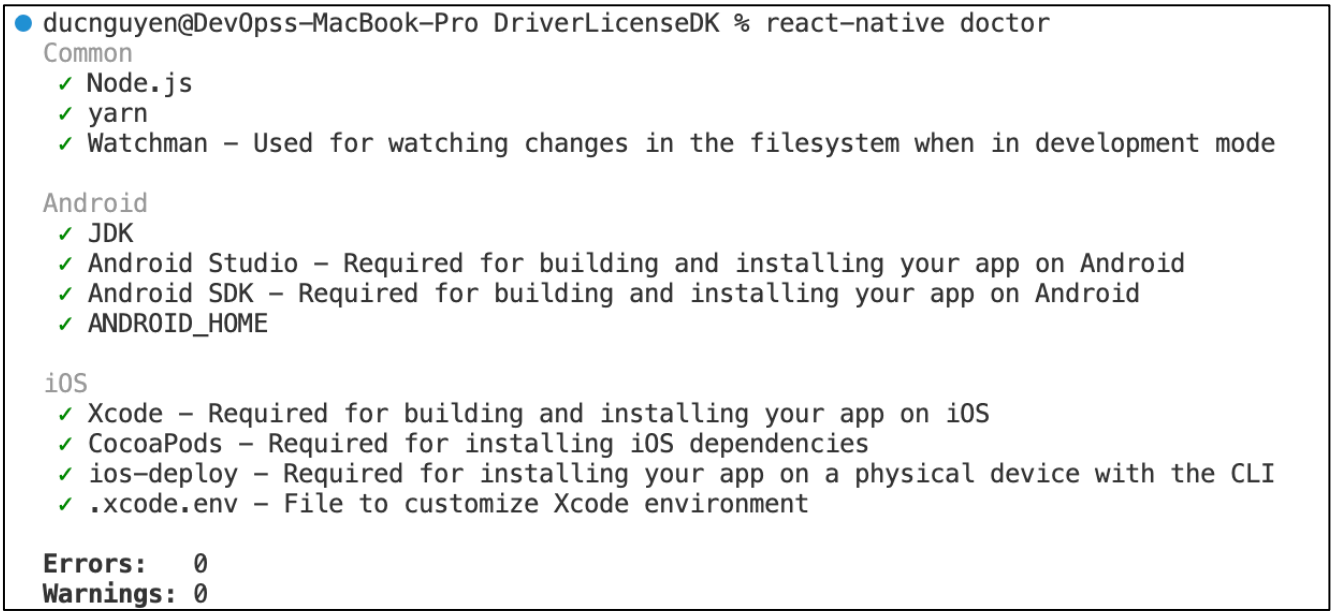
In SDLC, there are five stages such as requirement analysis, system design, implementation, testing, and maintenance (Pressman 2010, 40). In the first stage, a brainstorming meeting is created to find out ideas and user requirements. To visualize this idea, some design tools such as Sketch, Vision, Zeplin, or simple drawings on paper are used. It is important because it is the first document that transfers knowledge between a software development team. Ideally, Zeplin with an extension allows users to export design to code and connect the design with the codebase (Zeplin 2023). More details can be found in Zeplin's support document. PICTURE 1 below demonstrates the possibility of connecting React components to a design in Zeplin.



PICTURE 1. Connect Zeplin design to codebase (Zeplin 2023)

Regarding system design, there are five major activities, such as describing the environment, designing the application components, designing the UI, designing the database, and designing software classes and methods (Satzinger 2000). Choosing a programing language is a part of the system design environment. JS is the most popular programming language used in the world, and TypeScript is the fastest-growing programming language (JetBrains 2022). To write a program with JS, Visual Studio Code, IntelliJ, or even a command line text editor such as Vim is used. Git and GitHub are used to manage the codebase and CI/CD process. To work with mobile platforms, XCode and Android Studio are two IDEs

used to work with iOS and Android, respectively (React Native 2023). There are numerous IDEs, and tools must be installed. Therefore, to make sure all requirements are met, developers can use a command-line tool named React Native Doctor. It allows troubleshooting and automatically fixing errors in the evolution environment. Developers can type a shell script as 'react-native doctor' into the terminal. The tool will analyze the machine's environment and show the result set of what needs to be fixed (Bento 2019). PICTURE 2 demonstrates how the checking tool works.



PICTURE 2. Screenshot from React Native Doctor results

For debugging and testing, Chrome Developer Tools is used to inspect, edit, debug JS code, and profile the app's performance (React Native 2023). Cypress is used besides support testing, and it works well with Chrome (Cypress 2023). For the deployment stage, there are numerous options to choose, such as AWS, Azure, Firebase, and Heroku (Stepnov 2022). All these need time to dive deep to know which is suitable for the project. In this thesis project, a set of MongoDB for database management, Jest and Cypress for testing, and AWS for deployment were used.

3 FRONT-END DEVELOPMENT WITH REACT

Front-end frameworks and libraries have become an important part of the web development stack. In 2011, React was created by Facebook, now known as Meta. React is a free and open-source front-end JS library. It is used to build the application UI based on components. React was first announced by Facebook and then maintained by a wide community of developers and companies. React can be used not only to build both web and mobile applications but also macOS apps, Windows apps, tvOS, and React Native Skia. (React Native 2023.)

3.1 React core concepts

React includes four core concepts. They are component, JSX, prop, and state. First, the component is an isolated unit of UI. It is a JS function that follows React syntax to describe a part of the UI. It can be a view, a button, a label, or an input field, or an entire page. For instance, a simple component can be defined in CODE 1 below. A component named 'PlayAudioButton' is defined. It uses JS arrow function syntax and returns an 'Icon' component that is imported from another module. Numerous components can be combined to make a new one. That is the way components are reused. There are two ways to write a component in React, such as using a class component or using a functional component. There is no strict rule for writing it and it depends on developers' preferences. (React 2023.)

```
const PlayAudioButton = ({
  busy = false,
  onPress,
}: PlayAudioButtonProps) => (
  <Icon
    name="volume-high"
    size={24}
    color={busy ? palette.dark500 : palette.primary}
    disabled={busy}
    onPress={() => onPress && onPress()}
  />
);
```

CODE 1. Definition of a React component function (Thesis project)

Second, JSX is a syntax extension for writing HTML-like inside JS. JS is now in charge of the HTML, so rendering logic, markup, and CSS live together in a component. In JSX, there are some rules people

must follow: always return a single root element; in case developers need to return more than one element, they can wrap them up into a React Fragment by '<></>' or another '<View></View>'. When opening a tag, it must have to close the tag. JSX uses camel-case in syntax. It means that the component is 'PlayAudioButton', not like the 'Play_audio_bUtton' (React 2023). Third, 'props' stands for the property of a component. Because all elements created in React are JS objects. So, props are data of them. It can be any value or function (React 2023). In CODE 1 above, the component 'PlayAudioButton' has two properties named 'busy' and 'onPress' and they help to customize the component.

Lastly, React state is known as the component's memory. It is useful for handling user interaction, which makes data change. For example, in CODE 2 below, the loading state is used to store the state of the result when users press 'SignIn' button, which will call 'SignIn' API. Loading state will trigger 'SignInForm' to show a loading indicator. React uses 'React.useState()' to define a state. A pair of '[getter, setter]' will be available for controlling the state value change. In React, a function starting with the keyword 'use' is called a Hook. Hooks functions are special functions used by React to hook/inject into React components. 'useState' is a simple React Hook function. After calling the 'useState' hook, the React component registers to React where it wants to remember the state variable in the render tree. The value can be accessed by getter and setter functions. (React 2023.)

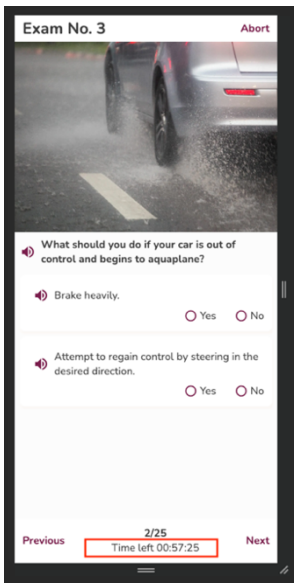
```
const SignIn = (props: RootParamsScreenProps) => {
  const [loading, setLoading] = useState(false);
  // other code goes here

  return (
    <SafeAreaView>
      <ScrollView>
        <View>
          <SignInForm
            onSubmit={handleSubmit}
            onForgotPassword={handleForgotPassword}
            onSignUp={handleSignUp}
            loading={loading}
          />
        </View>
      </ScrollView>
    </SafeAreaView>
  );
};
```

CODE 2. SignIn component with a loading state (Thesis project)

3.2 React rendering strategies

Rendering is a process of UI based on the app's state and props (React 2023). State is the heart of the component. From the initial stage, React will make a first render. After that, each state or prop change can trigger the re-rendering of a small part of the UI depending on how it wants the rendering behaviors to happen. A countdown hook is implemented using hook to calculate the time left (refer to CODE 24 of Appendix 6). The 'countDown' state is initiated with the first value of 'countDownTime'. Then another hook called 'useEffect' is used. 'useEffect' is a React Hook that allows the side effects code to change the component state. Inside this hook, an interval timer calculates days, hours, minutes, and seconds for each one-second pass. After all, these values are combined as a tuple structure. A 'Countdown' component uses that custom hook (see CODE 25 of Appendix 6). The hook is used inside the 'ExamControl' component. In every second, values of minutes and seconds will be updated. After that, the parent component will trigger the re-rendering of a child component named 'CoundownView'. A simple 'Text' component displays the time as 'HH:mm:ss' where 'HH' is hours, 'mm' is minutes and 'ss' is seconds in PICTURE 3 below.

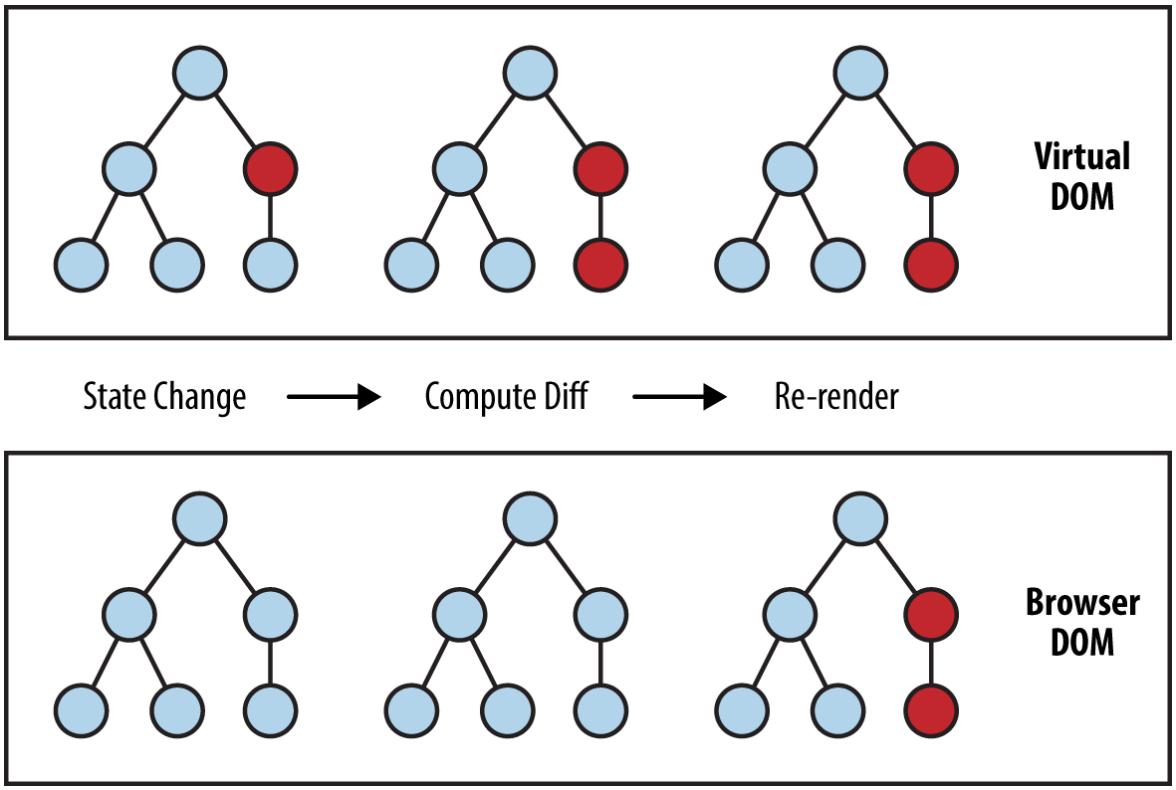


PICTURE 3. Countdown component re-rendering to update the time left for the exam (Thesis project)

To dig deeper into how the React engine re-renders behind the scenes, the virtual DOM will be mentioned right now. Virtual DOM, also known as VDOM, is a programming concept where an ideal, or virtual, representation of a UI is kept in memory and synced with the real DOM by a library such as 'ReactDOM'. This term is called reconciliation. In quick definition, reconciliation is the algorithm used to differ one tree to another to determine which parts need to be changed, inserted, or deleted. Updating

the DOM tree requires a complex algorithm to recalculate CSS and layouts, which impacts performance by causing delays. (React 2023.)

While the DOM represents the web document as nodes and objects, VDOM is a lightweight replica of the original DOM. DOM is created when a site is loaded while VDOM is initiated once from DOM. Whenever any state of the component is updated, these changes are updated in the VDOM first. The real DOM is not affected. It saves time and the browser's resources to re-render the DOM and display it. The React engine compares the change needed for VDOM and only the changed object is updated into the real DOM. This is a concept SPA provides: the web loads smoothly and faster than the original multiple pages. VDOM is off-screen rendered, it is not blocking the UI's render process as can be seen in PICTURE 4. (React 2023.)



PICTURE 4. Reconciliation in React (React 2023)

3.3 React principles

All the state components can be split into two main groups. They are the server state and client state. And managing data is half of React. It includes rendering data and binding data to the components,

layouts, routing, and styling components. It is not exaggerated to say that managing the state is the most important technique about React. As time passes, the more scale the application is, the more complex the management state must handle. It is a must-have suitable strategy to control the situation, and there are numerous principles to follow to make state management better. (Gamage 2023.)

3.3.1 SOLID Principles

The SOLID principles can be applied to manage the state in React effectively. SOLID stands for five design principles that make the codebase reusable, maintainable, scalable, and loosely coupled (Velkov 2023). The first principle is single responsibility, which states that each component should have one responsibility. If developers want to do more in a component, they should split it into smaller components, like classes. For example, in a component handling the theme, a custom hook retains the theme as a state, and provides a tuple of the theme and a function named 'themeToggler' to toggle the theme's value. The hook does nothing more than things related to the control theme in the app as shown in CODE 26 of Appendix 6.

The second principle is Open/Closed. The rule is based on the concept of being open for extension and closed for modification. All components in React follow this principle. Props can be passed down from the parent component to nested child components, but the direction from the child component to the parent is not allowed. To send the value from the child component to the parent, it uses another technique called 'useRef' or relies on a global state. The third principle is known as Liskov substitution. Components are replaceable with objects of their subtype, without disrupting their behavior of them from the beginning. For instance, if a component named 'CustomSwitch' extends React Switch, it should work the same as React Switch. It means that it can be pressed and toggled the value itself. (Velkov 2023.)

The next principle is interface segregation. This means that the larger interface should be split into smaller parts. For example, in Appendix 6, CODE 27 shows how the I-principle is applied. The big component 'UserProfileView' has a prop type for all interfaces 'IUserEmail', 'IUserPhone', and 'IUserSaveLogin'. This means that the parent component's props include all properties. In a child component named 'UserEmail', React only passes two properties named 'email' and 'submitEmail' from the parent into it, which is a normal behavior. But with the 'UserPhone' component, all properties are passed immediately. Therefore, the change in 'email' can trigger the re-rendering of the 'UserPhone' component. It is not the component's correct behavior, which means it is not good practice. (React 2023.)

The last principle is dependency inversion. It depends on abstractions, not concretions; high-level modules depend on low-level modules, and an abstraction interface between them is a must. For example, in this project, an interface is used to connect between a text-to-speech module and a component. The CODE 28, 29, and 30 of Appendix 6 prove the D-principle. The 'ITextToSpeech' interface is a bridge to connect the React component which wants to use the text-to-speech function to the 'react-native-tts' module on iOS and Android or 'window.speechSynthesis' with a web browser. In this way, modules can be connected as can be seen in CODE 28, 29, 30, 31 of Appendix 6.

3.3.2 DRY Principle

Developers should follow the DRY rule to avoid code duplication and improve the writing of reusable code (React 2023). In this rule, there are numerous rules they can usually do to make DRY: avoid copying and pasting code; use variables to store repeated values that can be used in many places of components; and separate common functions into utilities folder. For example, in this thesis project, from the beginning of the initial stage, the styled-components library is used to style all the components used in the app. That means the styling of the app, theme, fonts, colors, and spacing between views can be easily changed. (Styled-component 2023.)

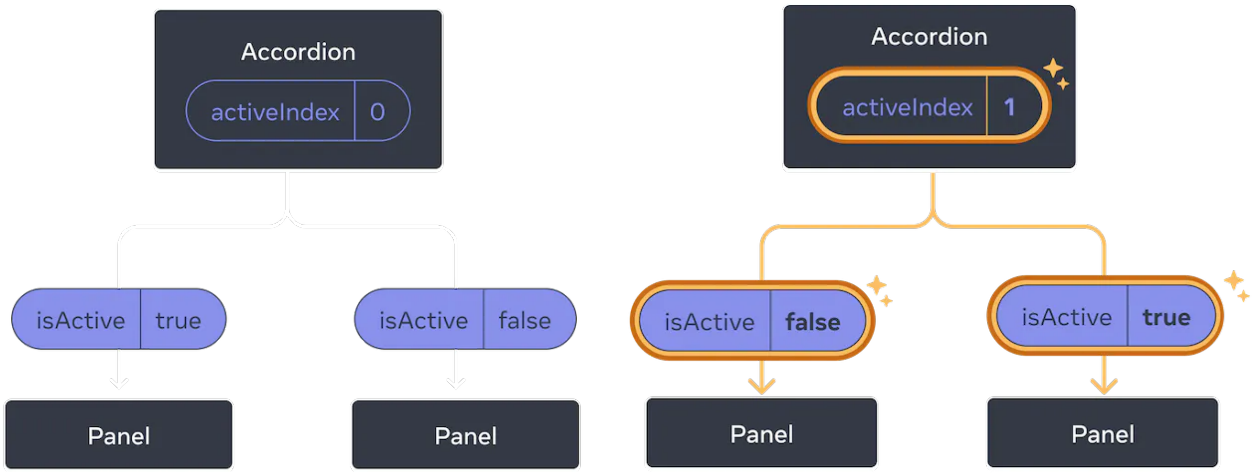
Furthermore, basic React components are abstracted into customized-based components. As shown in CODE 3 below, 'maxFont-SizeMultiplier' is a property of React Text, which allows the 'Text' component to be scaled up or down depending on the device accessibility setting. The maximum value can be changed in the 'TEXT_CONFIG' setup. Then another option of accessible properties like 'accessibilityLanguage', 'accessibilityElementsHidden', and 'importantForAccessibility' can enable or disable the screen reader to talk aloud the text to the user. As a result, developers can reduce code redundancy within 'Text' components. A minor change with 'BasedView' is also applied to all 'View' components. (Styled-component 2023.)

```
import styled, {css} from 'styled-components/native';
const BasedText = styled.Text.attrs(props => ({
  maxFontSizeMultiplier: TEXT_CONFIG.maxFontSizeMultiplier,
  accessible: props.accessible,
  accessibilityLanguage: solveTtsLanguage(),
  // iOS only
  accessibilityElementsHidden:
    props.accessible === undefined
      ? true
      : Boolean(props.accessible) === false
      ? true
      : false,
  // Android only
  importantForAccessibility:
    props.accessible === undefined
      ? 'no-hide-descendants'
      : Boolean(props.accessible) === false
      ? 'no-hide-descendants'
      : undefined,
}))``;
const BasedView = styled.View.attrs(props => ({
  accessible:
    props.accessible === undefined
      ? undefined
      : Platform.OS === 'android'
      ? props.accessible
      : undefined,
}))``;
```

CODE 3. ‘BasedText’ component in thesis project (Thesis project)

3.3.3 React data flow

There are two ways to share a state between components. One way is using ‘One-way data flow’, from high-level component to low-level component, parent to children. React team mentioned the ‘lifting state up technique’ in official documentation, starting with removing the state from all children, moving it to the parent, and finally passing props from the parent component to the children (React 2023). This technique has been visualized in PICTURE 5. The other way is using the Redux concept. Redux is a good choice library that allows the application to broadcast the variable changes to subscribed components. With Redux, the state is stored in a global store (Redux 2023). Various libraries have the same approach as Redux, such as MobX, RxJS and Zustand.



PICTURE 5. Lifting state up in React (React 2023)

3.3.4 React preserving and resetting state

The app's state is stored inside a render tree, not in the component. React keeps track of the state belonging to which component based on its place in the render tree. When a component is destroyed, the state that sticks with it is destroyed too. In the rendered tree, if the position of the component is changed, React will re-render it. It means that the state is also reset. The rules are the same component and at the same position in the render tree, the state is preserved. For different components at the same position, the state is reset, and its entire subtree is also reset. By default, React will keep the state unchanged when its place in the render tree is the same as before. But React allows developers to change this behavior by resetting the state manually. As shown in CODE 32 of Appendix 6, 'QuestionView' component is used in the 'Exam' component. Each time 'question.id' changes, by default, the 'QuestionView' without a key prop will not re-render because its position in the render tree is still the same. So, it must add the key prop with a value equal to 'question.id'. Each time a question changes, the 'id' will be updated, and therefore, the 'QuestionView' display is re-rendered to display a new question's image. (React 2023.)

3.3.5 React performance

Regarding application performance, mobile devices typically aim to display 60 frames per second. It means that each rendered frame has a maximum of 16.67 milliseconds to calculate and generate a static image to display on the screen (Android 2023). If the code needs more than 16.67 milliseconds to render a frame, that frame is dropped, lead to UI lag. Sometimes, the code may require a heavy calculation task

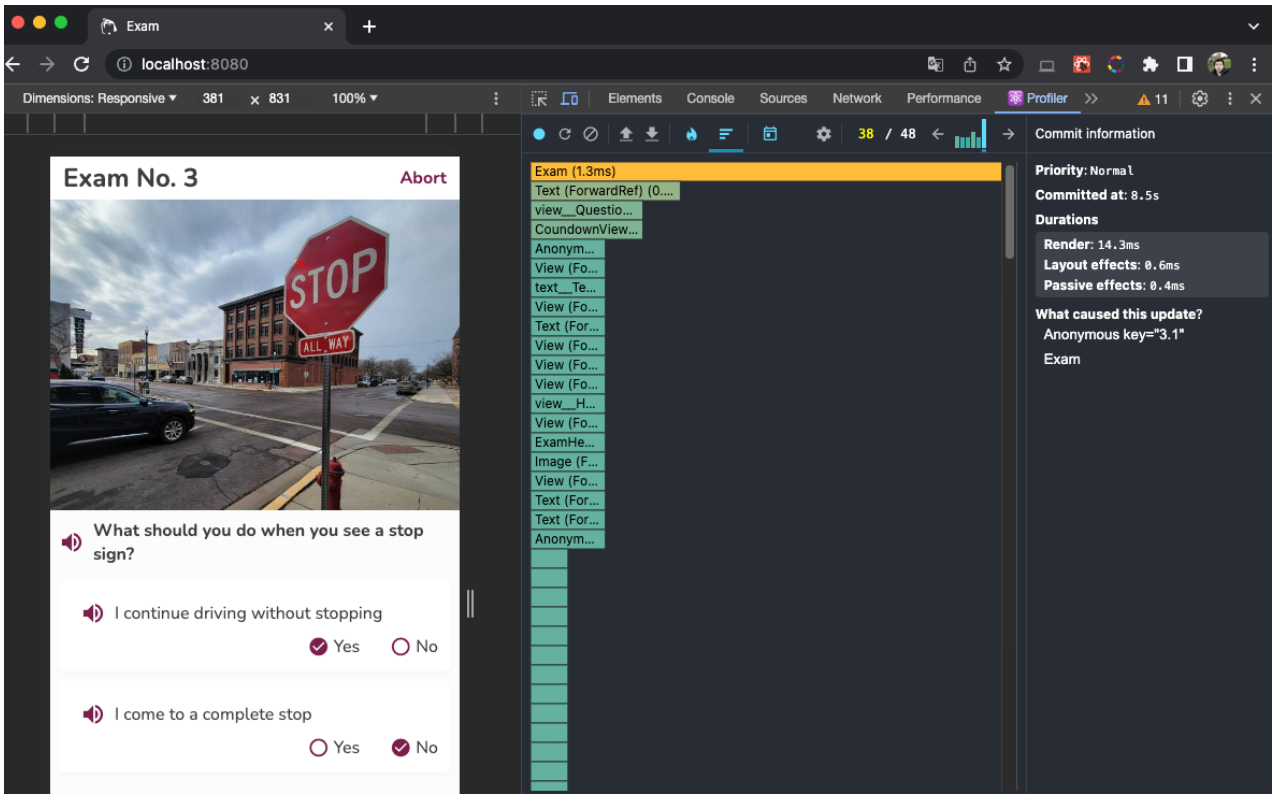
to generate the frame, and this leads to low-performance issues. Therefore, React provides some approaches to troubleshooting the issue. A React component '<Profiler/>' can be used to wrap around the React tree to measure rendering performance as can be seen in CODE 4 below. (React 2023.)

```
function onRender(
  id,
  phase,
  actualDuration,
  baseDuration,
  startTime,
  commitTime,
) {
  // Aggregate or log render timings...
}

<Profiler id="A heavy component need to be checked" onRender={onRender}>
  <CheckComponent />
</Profiler>
```

CODE 4. Manual measure component's render time with React Profiler component (React 2023)

These statistics can be tracked by enabling the Pert Monitor feature inside the React Native Dev tool or by using the React Developer Tool browser extension. It displays two tabs: Components and Profiler from the debugging panel. The Profiler tab is used to identify any part of the app that gives low performance. For example, in PICTURE 6 of the following page, a bar chart in the debugging panel demonstrates the render time of each component, and next to that is the reason for triggering its render. By means of this visualized chart, people know immediately where the problem occurs. The component named 'Exam' is rendered because the question number changed to the value '3.1' and the total render time is 14.3 milliseconds. This is a good render time and does not require further investigation in this component. But when the render time is more than 16.67 milliseconds, it is a must to refactor the 'Exam' component. This is just one example of how using a debugging tool can improve the quality of an app. (React 2023.)



PICTURE 6. Using Profiler tool in React to find the code made low performance (Thesis project)

3.3.6 React in Advance Level

According to the React Native DEV community, starting with the basics first and engaging in extensive practice is the key to leveling up in front-end development. Various tips and tricks from the community may be useful to other aspects outside of React from React TypeScript Cheatsheet in 2023. Firstly, developers must follow code conventions and lilting rules while working in a team. This makes code cleaner and more readable. With ESLint, developers can improve their code quality through its strict rules. Secondly, using a wrapper function for console logs can avoid the mistakes of logging data on production. Thirdly, using custom hooks allows developers to reuse the code logic. (React TypeScript Cheatsheet 2023.)

Developers can reduce code complexity by using a tool to measure the complexity score. For instance, ‘code-complexity’ is a good measure tool for JS and TypeScript. Developers can reduce the app bundle size by avoiding external libraries if they are not necessary and optimizing resources/assets or removing redundant code. Developers should cache images or requests as much as possible. It saves loading time, reduces workload on the server side, and sometimes it is useful for offline mode. They can avoid using

Expo also. Expo is an unnecessary layer for the application. It makes the app slower, has bigger app sizes, and has limitations in integrating native modules. Finally, accessibility is an important part of the application. Of course, there are no strict rules in many countries, but this improves the user experience and expands the user target (National Center of Deaf-Blindness 2023).

3.4 React Native

React Native combines the best things of native development of iOS and Android with React. React Native codebase can be written with JS or TypeScript and React Native does the hard part, rendering JS into native platform UI. From 2021, React Native can port to macOS and Windows as a Universal Windows Platform. It is a revolution and brings a lot of developers' attention to React Native compared with Flutter. To work with React and React Native, a strong understanding of JS and React is essential. Of course, it will be best if developers know both iOS and Android development processes. But with React Native, JS, and React knowledge is quite enough to start with. Developers write JS code once and run it everywhere. Their task is to write the React components. (React Native 2023.)

At runtime, React Native will create the corresponding native views from those. For example, a React component like View will be transformed into ‘UIView’ in iOS and ‘ViewGroup’ in Android perfectly, a Text component will be ported to ‘TextView’ in Android and ‘UITextView’ in iOS. React Native also provides tools for creating the React project template to work with mobile apps faster. It includes several core packages such as ‘react’, ‘react-native’, ‘babel’, ‘typescript’, ‘prettier’,' and ‘jest’. It is simple to set up the project from the guidelines of the React Native official site. React Native also provides a tool for debugging. It reduces the time for troubleshooting issues and turning React performance. (React Native 2023.)

3.5 React-native-web

There is a layer between React Native and React DOM while working with RN Web. It renders RN components into compatible JS code in a web browser. There are some packages which must be installed using RN Web, such as ‘react-dom’, ‘react-native-web’, ‘babel-plugin-react-native-web’. React Native developers have access to libraries that support both React Native and React Native Web. The React Native Directory serves as a valuable resource, allowing developers to efficiently locate suitable libraries

for their projects (React Native for Web 2023). Of course, there are several differences between RN and React DOM, including styling in the JS API, layout localization, and gesture systems such as pointer, mouse events, and keyboard (React Native for Web 2023). However, developers have several ways to handle the fragmentation between web platforms and mobile platforms. The most used way is using React 'Platform' to check the OS system and write code to handle each platform. Another method is to use file extensions to separate logic and behavior between the mobile and the web. Typically, file extensions '.ts' and '.js' are used for all platforms, but developers can use extensions '.web.ts' or '.web.js' for specific web platforms. To configure the 'web.ts' extension, CODE 5 below is used to update the configuration in the 'webpack'. In this code, 'entry', 'rules', and 'extensions' are required configurations.

```
const webpack = require('webpack');
// Other

module.exports = {
  entry: ['./index.web.js'],
  // Other config
  module: {
    rules: [
      {
        test: /\.(js|jsx|ts|tsx)$/,
        exclude: /node_modules/,
        include: path.resolve(__dirname, 'src'),
        use: ['babel-loader'],
      },
      {
        test: /\.(png|jpe?g|gif)$/i,
        use: [{loader: 'file-loader', },],
      },
      {
        test: /\.svg$/,
        exclude: /node_modules/,
        use: [{loader: '@svgr/webpack',},],
      },
    ],
  },
  resolve: {
    alias: { // alias between RN and RN for web goes here
    },
    extensions: [
      '.web.tsx', '.tsx', '.web.ts', '.ts', '.web.jsx', '.jsx', '.web.js',
      '.js', '.css', '.json',
    ],
    mainFields: ['browser', 'main'],
  },
};
```

CODE 5. Webpack config sample from thesis project (Thesis project)

4 DESIGN AND IMPLEMENTING OF REACT APPLICATION IN PRACTICE

The theoretical driver's license exam in Denmark consists of twenty-five main questions, each worth one point. These main questions may also have two, three, or four sub-questions, which require a simple yes or no answer. There is a limited time for completing all questions, applicants must quickly listen to the question, tick yes or no answer, and move on to the next question. After that, if the total point equals or exceeds twenty, it is marked as a pass and below twenty a failure. Moreover, the question bank is generated from traffic law and driver rules. (Teoriklar 2023.)

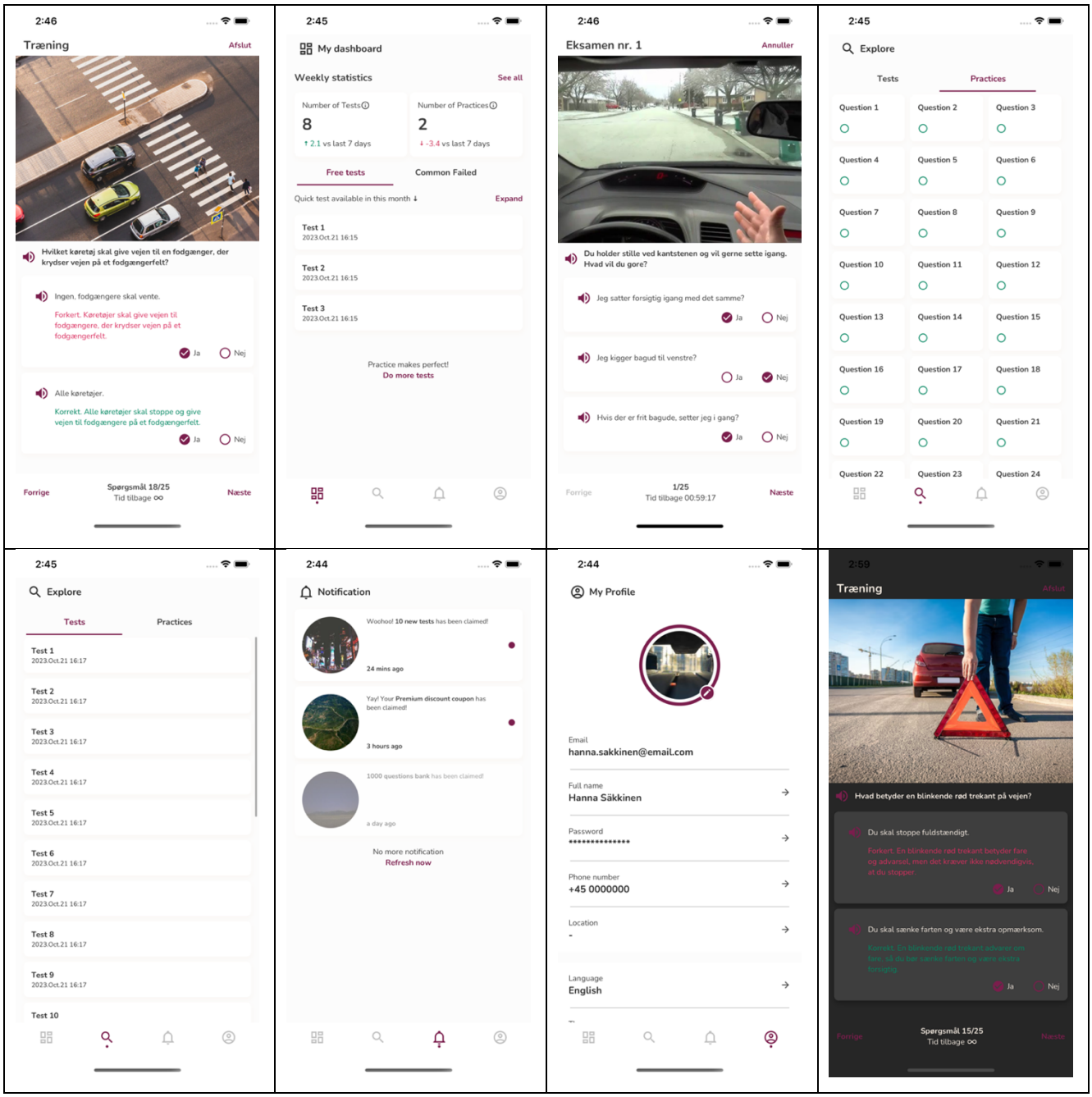
4.1 Application Design Overview

Regarding project requirements, firstly, two main modules need to be implemented. The practice view contains many questions, and the exam view includes real twenty-five-question tests. Other modules are added, such as a Dashboard for displaying the visualized learning process, a Notification for listing system notifications related to the learning process, and a Profile for showing user profiles and other preferences. PICTURE 7 below contains pictures of the application design. This design utilizes the material design pattern with many card views to split each functional area. To provide consistency in UI/UX, the web application looks and feels the same as the mobile application. Dark mode is also supported in the application. In the question view, the user can read or listen the question and tap the combo box yes or no to answer. After that, the answer sheet is stored, and the final score is calculated.

Secondly, this application offers support for accessibility functions while users do exams. It means that the user can utilize a screen reader to explore and navigate between elements on the screen, and double-tap to interact with combo boxes and buttons. There is a setting view that allows users to customize languages, text-to-speech rate, pitch, and male or female voice. This feature is a simulation of the real exam in Denmark while examinator will read the questions and applicants must answer quickly and move to the next question. Finally, the app supports offline mode. It means that after users see the question, they can see it again without a network. In a long-term project, there is a feature that allows users to download a batch of images and questions before opening them.

To clarify the application’s features, the system allows users to register a new account with an email address and a phone number. The account is an optional requirement to use the system. If users want to

use the full functionality of the app, they must sign in and buy a premium package. From the dashboard view, users can review their learning process on how many tests they did, and how many percent they passed. It also includes free tests for unauthenticated users. From the exam view, the user reads or listens to the question and answers all sub-questions below. After that, they can move to the next question. There is a limited time for doing an exam. But in practical mode, users can do these questions at their own pace and can see question hints with correct results right after they check the yes or no answer. The application provides basic functionalities for updating a user's profile, such as email, phone, and two-layer factor authentication. There is a setting view that allows users to change their preferences.



PICTURE 7. Application final design (Thesis project)

4.2 Setup Project

To initiate a new project, developers can use the React Native command line. Alternatively, they may choose to use Expo, which can expedite the project setup process (React Native 2023). Setting up without Expo may be complicated when working in a web environment. However, Expo, in the long run, has several limitations as mentioned earlier. For example, it lacks support for in-app purchases, has restricted RAM, and may result in a larger app size and lower performance compared to the original version (Pregasen 2023). From now on, the thesis project only uses RN CLI. The first step is creating a new project with RN CLI. The default template will be TypeScript because it is good practice for developers to force better code styling and structures (React 2023). By running from the terminal, CODE 6 below is executed to install a new project. Meanwhile, NPX is a Node package execute. Project uses the latest RN version '0.72.5', and the project name is 'DriverLicenseDK'.

Init new React Native project with RN CLI

```
npx react-native@latest init DriverLicenseDK
```

Install other packages to run web from a React Native project

```
yarn add react-native-web react-dom
yarn add -D webpack webpack-dev-server
```

CODE 6. Create new project with RN CLI (React Native 2023)

After that, the project requires a few missing packages to work with the web environment: 'react-native-web', 'react-dom', 'webpack'. Additional packages are installed with NPM or Yarn. Yarn is a package manager just like NPM and it is developed by Meta. React Native Dev refers to Yarn because it can install dependencies in parallel. It is also faster when downloading large files. The file 'package.json' requires the addition of scripts for web build process (see CODE 33 in Appendix 6 for an example script). Two essential files that need to be added to project's root folder, are 'index.html' and 'index.web.js'. The second file to be included is the main entry point for web environment as can be seen in CODE 34 of Appendix 6.

The next step is configuring the 'webpack' file. The file 'webpack.config.js' is added to the root folder. With this project, several packages such as 'babel', 'typescript', 'react-navigation', 'i18next', 'react-native-svg', 'react-native-linear-gradient', and 'lottie-react-native' are used. They have an alias library that works well with 'react-native-web'. The 'webpack.config.js' file requires an update to enable the use of these aliases (refer to CODE 35 in Appendix 6). The final step is linking libraries to run iOS and

Android. The script found in CODE 36 of Appendix 6 needs to be run within a terminal to complete this process. It will take some minutes to download and install pods that are needed for iOS building process. After that step, developers can run debug on iOS or Android with CODE 7 below.

```
yarn cache clean // For clean the build cache

yarn ios // For running iOS in simulator or devices

yarn android // For running with Android emulator or devices

yarn web // For running web with default browser
```

CODE 7. Running scripts (React Native 2023)

4.3 Code Convention in TypeScript

TypeScript is a strongly typed programming language that builds on JS. By using TypeScript in a React app, developers can decrease bugs in their code, and decrease the need for testing. The latest version of TS is 5.5 (TypeScript 2023). To enable linting with TS, the package '@tsconfig/react-native' is needed to be installed. After that, the 'tsconfig.json' file can be updated as can be seen in CODE 37 of Appendix 6. In the '@tsconfig/react-native/tsconfig.json' file, there are many rules that help fix the bad code. These rules can be overwritten in 'tsconfig' but keeping default rules is a good start. TS does not emit any JS output because it only complies React code for type-checking. The result after bundling is the same as a non-TS project. TS works well with 'react-native-web'. A package named 'babel-loader' is added to 'package.json' file and a new rule is appended in 'webpack.config.js' > 'module' > 'rules' as can be seen in CODE 8 below. (Phull 2020.)

```
{
   test: /\.(js|jsx|ts|tsx)$/,
   exclude: /node_modules/,
   include: path.resolve(__dirname, 'src'),
   use: ['babel-loader'],
},
```

CODE 8. Babel-loader config (of Thesis project)

Babel is a JS toolchain that is used to convert ECMAScript 2015+ code into a backward compatible version of JavaScript in current and older browsers or environments (Babel 2023). During the bundling

process, files are passed off to the Babel compiler by the Babel loader and the loader returns the processed code, which is used in the final bundle (Goswami 2020). After the configuration step, people can use TS in the React app. It is necessary to review how to use TS in this project. Props and state can be defined as type or interface as CODE 38 in Appendix 6. These properties are defined clearly to make sure that there is no misunderstanding when used in components. That is another reason why combining React with Node.js is great. It can easily transfer the definition types between the server side and the client side. There will be no conflict when parsing data from APIs in a client. The kind of interface that should be used depends on where to use it. For example, an interface should be used to bridge between two modules as SOLID principles. Meanwhile, type is preferred to use for props and state in a single component. This rule makes the code consistent. More code conventions in TS can be found in the React TypeScript Cheatsheet. (React TypeScript Cheat-sheet 2023.)

4.4 Styling UI Libraries & Layout

Developers can feel overwhelmed by the many UI libraries, but in this thesis project, "styled-components" is applied. For other UI libraries, there is a table that compares them in Appendix 2. There are some benefits when using styled-components lib. It allows the use of actual CSS in it. It supports styling in both RN and RN Web. Moreover, it supports server-side rendering (Styled-components 2023). To install 'styled-components', developers can execute the shell script from the terminal as CODE 39 in Appendix 6. Then, a color palette and a dimension list are defined. To describe a component style, spacing can be used as CODE 40 in Appendix 6. After that, 'darkTheme' and 'lightTheme' are defined as CODE 41 in Appendix 6. By doing this, all colors, spacing, sizes, and font sizes in each view, button and label are consistent and easy to write without remembering exactly what the value is. The next step is defining base types and interfaces as can be seen in CODE 9 on the following page. The type 'VariantSize' is the way to configure style in components without the need for the original React Stylesheet.

The library 'styled-components' allows users to re-style an existing component and create a new one. That styling chain can continue as a decorated design pattern. It can be used as CODE 42 in Appendix 6. Finally, to use these styled components in pages, developers can write a shorthand style as CODE 43 in Appendix 6. Developers should avoid breaking the DRY principle. Typically, developers have many options when styling the UI, but following all principles is a must. It makes our codebase clean, readable, and easy to maintain later. (Metsäpelto 2022.)

```
import {TextStyle, ViewStyle, FlexAlignType} from 'react-native';
export type VariantSize = 'xxs' | 'xs' | 's' | 'm' | 'l' | 'xl' | 'xxl';
export type JustifyContentType =
  | 'flex-start'
  | 'flex-end'
  | 'center'
  | 'space-between'
  | 'space-around'
  | 'space-evenly';

export interface IViewStyle extends ViewStyle {
  _gap?: VariantSize;

  // Other customize options …
  _fullWidth?: boolean;
  _alignItems?: FlexAlignType;
  _justifyContent?: JustifyContentType;
  _flex?: number;
  _backgroundColor?: string;
}

export interface ITextStyle extends TextStyle, IViewStyle {
  _color?: string;
  _fontSize?: VariantSize;
  _bold?: boolean;
  // Other customize options …
}

export const TEXT_CONFIG = {
  maxFontSizeMultiplier: 2.0,
};
```

CODE 9. Styled-components with base interface and type (Thesis project)

4.5 Supporting multiple languages

To support multiple languages in the app, two packages named ‘i18next’ and ‘react-i18next’ are chosen. The shell command is executed to install these packages as can be seen in CODE 44 of Appendix 6. After that, ‘i18n’ class can be configured as CODE 45 in Appendix 6. Then, create a new folder named ‘i18n/resources’ for holding files ‘vi.json’, ‘en.json’, ‘dk.json’. The localized file can look like CODE 10 below with many key-value pairs. Finally, create a new custom hook to control the value of the picked language in the whole application. The app can display all supported languages as Vietnamese, Danish, and English. To use translate with ‘react-i18next’, keys can be joined by ‘.’ such as the inline code ‘t('common.my_dashboard')’ with ‘t’ is the used hook provided by ‘react-i18next’.

```
// File en.json
{
  "common": {
    "feature_not_available_yet": "Feature not available yet",
    "my_dashboard": "My dashboard",
    …. other
  }
}

// File dk.json
{
  "common": {
    "feature_not_available_yet": "Funktionen er endnu ikke tilgængelig",
    "my_dashboard": "Min kontrolpanel",
    "explore": "Udforsk",
    "notification": "Meddelelse", …
```

CODE 10. Sample for localized string in json file (Thesis project)

4.6 Generate question bank and images

AI is useful in many aspects. In the scope of the thesis, the tool named ChatGPT-3 is used for generating a question list and suggesting a prompt which is used to generate images from it (OpenAI 2023). Another AI tool named MidJourney is used to generate images from text. To begin generating the question bank, a question interface is defined as CODE 46 in Appendix 6. After that, questions are generated by ChatGPT that conforms 'IQuestion' interface. The output should be in a JSON format. The result looks like CODE 47 in Appendix 6. There are many issues with generating questions with AI. First, the question words are not suitable or correct in meaning. Second, the complexity of the question may not be good enough for a real exam. Third, these questions can be duplicated. However, in the scope of the thesis, the generated result is kept as an example of effectively utilizing AI to support work. Finally, all the AI-generated questions are needed to review again.

4.7 Data flow and state management

The topic of data flow and state management in React applications is quite complex. A React client usually connects with the server side to request data. This data is called server state. Meanwhile, the client receives user's interactions and connects with external resources of the OS such as RAM statistics, battery level, OS preferences or memory, and SD storage. This offline data is called client state. At runtime, some exceptions occur, leading to crashes or side effects. They are called error states. To be

managed, the state can be divided into other categories with suitable strategies applied to each type. (Gamage 2023.)

4.7.1 Server state

Response data from the server side must be stored somewhere that can be accessible from other components later, not only on a component that requests the data. Its purpose is to reduce resources to request the same data again and again, reduce workload for the server side, and decrease response time, meaning smooth UX. It is called a caching technique. Many libraries support this feature, such as Apollo, SWR, RTK Query, and React Query. In this thesis, React Query is used. To install React Query, developers can execute the script as CODE 48 in Appendix 6. After that, its 'QueryClientProvider' is added to App component CODE 49 in Appendix 6. Next, create a new custom hook wrap-up 'useQuery' like CODE 50 in Appendix 6. After all, these hooks can be used inside components. For instance, in 'ExploreTests' component, it includes an infinity scrolling list. Each time users scroll the list to reach the bottom, 'fetchNextPage' will be triggered, and the new data will be updated, resulting in the addition of more items to the list from the scroll position as shown in CODE 11 below.

```
const infiniteResult = useAllTests();
  const { data, error, isLoading, fetchNextPage, isFetchingNextPage, hasNextPage } =
infiniteResult;
return (
    <List
      data={data ? data.pages.flat() : []}
      renderItem={({item}) => renderItem(item, handleNavigateTest)}
      onEndReachedThreshold={0.2}
      onEndReached={handleFetchMoreData}
    />
  );
```

CODE 11. Infinity scrolling list (Thesis project)

4.7.2 Error state – Error boundary

To handle exceptions, JS provides developers with a try-catch tool. But in the long run, developers need to create a general solution to concentrate all error handles in one place (React 2023). Another option is writing custom error boundary like CODE 51 in Appendix 6. From React 16, all errors are printed during rendering to the console in debug mode. Developers can see the error message and the JS stack and

component stack traces. It helps a lot while troubleshooting where exactly in the component tree the failure happened (React 2023). Developers can use a third-party library like ‘react-error-boundary’ to wrap around React components to catch errors and handle corner cases like showing error toasts or render a 404-error view. ‘react-error-boundary’ provides ‘useErrorBoundary’ hook for manual controlling show or dismissing error boundary as can be seen in CODE 12 below.

```
import {useErrorBoundary} from 'react-error-boundary';

function Example() {
  const {showBoundary} = useErrorBoundary();
  //...
  useEffect(() => {
    fetchScore(id).then(
      response => {
        // Set data in state and re-render
      },
      error => {
        // Show error boundary
        showBoundary(error);
      },
    );
  });
  //...
}
```

CODE 12. Sample of ‘useErrorBoundary’ (Thesis project)

4.7.3 Client state

Regarding client state, developers have many options to choose, such as Redux, Zustand, MobX, Akita, and Recoil. Each library has its advantages and disadvantages. Developers can choose one depending on their needs and preferences. Ideally, it is worth trying at least one time each to feel and analyze what is suitable in a React project. Zustand is used in this thesis. It is simpler than other libraries in many ways, such as no need to wrap up entire app with a provider or use hooks in any code place. A created store’s data can be any type of object or function, offering much greater flexibility in data management (Zustand 2023). By combining with the 'immer' library, it makes the next immutable state tree by simply modifying the current tree (Immer 2023). Two packages named ‘zustand’ and ‘immer’ are added to use Zustand and Immer in the React project (Davis 2023). The Zustand store type can be defined with all available local states as CODE 52 in Appendix 6.

Notifications, questions, and tests are not included in the client's state. This data needs to be fetched from the server side, so it belongs to the server state. Developers have many efficient ways to manage the server state (see section 4.7.1). In this thesis, preferences, answers, and study records will be used as offline data. To store them with Zustand, create a set of data and actions like CODE 53 in Appendix 6. Persist and 'immer' are two middleware that are useful with Zustand (Zustand 2023). Immer allows developers to update the immutable state in a short way inside the 'set' callback. Besides, persist creates local storage 'AsyncStorage', for the client state, persisting its data. After the previous step, the store can be implemented like CODE 54 in Appendix 6. 'CreateSelectors' creates shorthand use of Zustand store. Finally, 'useStore' can be hooked into any component or used directly outside React via the 'getState' function. CODE 13 below demonstrates how to use Zustand store in components.

```
i18n.use(initReactI18next).init({
  resources,
  lng: useStore.getState().setting.pickedLanguage, fallbackLng: 'en',
  compatibilityJSON: 'v3',
  interpolation: { escapeValue: false },
});
const ProfileDetails = (props: RootParamsScreenProps) => {
  const {t} = useTranslation();
  // Using Zustand store
  const pickedLanguage = useStore.use.setting().pickedLanguage;
  const setPreLanguage = useStore.use.setPreLanguage();
  const isDarkMode = useStore.use.setting().isDarkMode;
  const toggleDarkMode = useStore.use.toggleDarkMode();

  const handleChangeLanguage = async () => {
    let lang: SupportedLanguage =
      pickedLanguage === 'vi' ? 'dk' : pickedLanguage === 'dk' ? 'en' : 'vi';
    setPreLanguage(lang);
  };
  const handleChangeTheme = async () => {
    toggleDarkMode();
  };
}
```

CODE 13. How to use useStore in application (Thesis project)

In general, a good decision in system design allows developers to boost the development process. There is more than one solution for every aspect of React state management. The support of the React developers' community helps quickly choose a good library. Another factor is how developers applied them to the project. Consistent practice is the key to rapid skills for developers.

4.8 Mockup data with json-server

Regarding the interaction between the React application and the server side, it is a good approach when developers can develop all the front-end and back-end in parallel to save development time. It is not necessary to wait until APIs are ready to start with the front-end feature. From the previous session, an interface between client and server will be defined following the rules of the SOLID principles. An interface between how data types are defined and what is possible of variables/properties makes both client and server work perfectly, like gears in a watch. The interface can be shared between the front-end and back-end because of combining React and Node.js from the beginning, so people do not need more extra steps to map different data between client and server. (React 2023.)

Front-end developers have many approaches to simulating data from the server side via a test server. In this project, 'json-server' is used to create a testing server for this purpose. To install the tool, developers can use CODE 55 in Appendix 6 from the terminal. After that, create a test database in the file 'db.json' as CODE 56 in Appendix 6. To stream image resources with json-server, create a folder named as '/public/assets/images' in the root project, then copy the test files into (json-server 2023). Finally, to run the server from default port 3000, the CODE 57 can be executed with delay time parameter is optional (see Appendix 6). The result will be the same as CODE 14 below. Developers can access the test APIs from the React app or Postman without any issues. A variable environment can be pushed into the '.env' file to hide the API endpoint and help set up the CI/CD process to change the server URL in configs.

```
$ json-server db.json --static ./public --delay 1000
  Loading db.json
  Done
  Resources
  http://localhost:3000/notifications
  http://localhost:3000/questions
  http://localhost:3000/free-tests
  http://localhost:3000/tests
  http://localhost:3000/questions-statistic
```

CODE 14. Result after running json-server command in terminal (Thesis project)

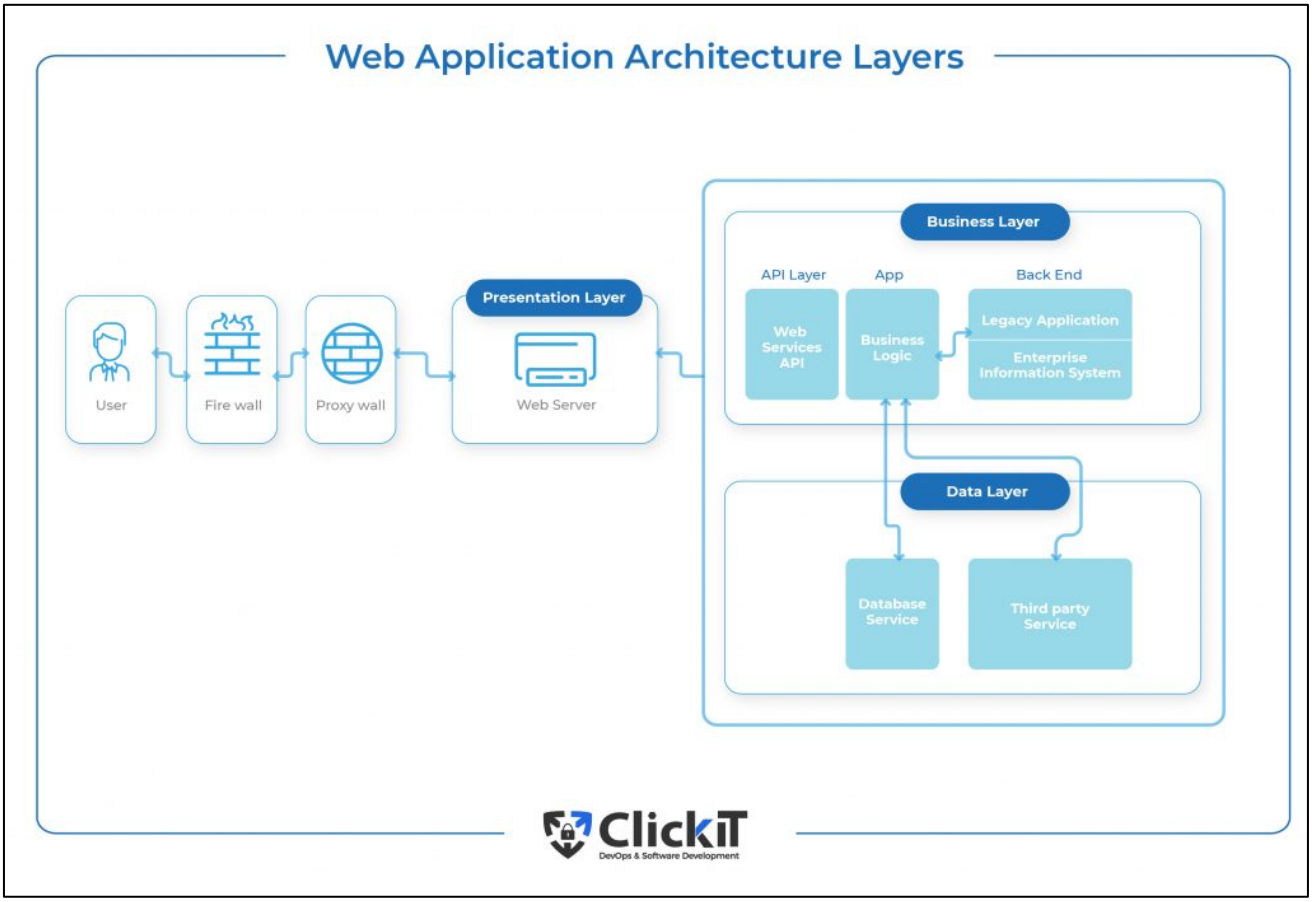
5 BACK-END DEVELOPMENT WITH NODE.JS

Node.js is an open-source server environment that runs on numerous platforms such as Windows, Linux, Unix, and macOS. Node.js utilizes JS to execute on the server, and it is utilized to construct the back-end of web applications. There are numerous packages and libraries available that can be used as Node.js modules, which makes application development faster and more efficient. Node.js has a unique advantage because millions of frontend developers that write JS for the browser are now able to write the server-side code in addition to the client-side code, without the need to learn a completely different language. For instance, with a few lines of code, developers can build simple web servers as can be seen in CODE 58 of Appendix 7. (Node.js 2023.)

5.1 MERN Web Architecture

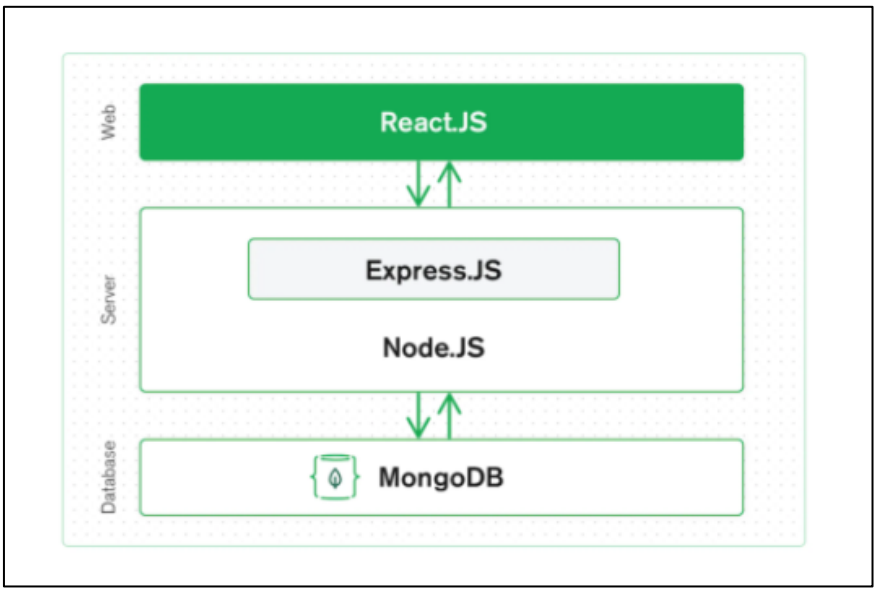
To start with back-end development, three-tier architecture in general software and web applications is analyzed. The three-tier architecture includes a presentation tier, an application tier, and a data tier. The web/mobile application works in the role of presentation tier because it interacts with users via web browsers or mobile applications. Meanwhile, the back-end application plays the role of application tier for processing data. It handles all business logic such as generating response data for the presenting tier, and adds, updates, or deletes data in the data tier. Database management servers such as MySQL, PostgreSQL, Oracle, Firebase, and MongoDB, are the data tier. This architecture is most used in web development, compared with two-tier and N-tier architecture. (IBM 2023.)

The three-tier architecture allows developers to speed up the development process because each tier can be worked on concurrently. Developers can use the latest and best languages and tools for each tier. Each tier can be scaled independently as needed, and security is improved because the presentation tier cannot connect directly with the data tier. Additionally, there are numerous factors such as firewalls, proxies, load balancers, and caching storage built in front of the application to protect it against security vulnerabilities. PICTURE 8 below represents web application architecture layers and its components. (IBM 2023.)



PICTURE 8. Web Application Architecture Layers (ClickITech 2022)

MERN is a three-tier web architecture that is used in the thesis project. MERN stands for MongoDB, Express, React, and Node. While MongoDB is an open-source database management program and a NoSQL database, Express is a web framework for Node.js (MongoDB 2023, MERN Stack Explained). Meanwhile, web or mobile applications are built with the React library and the Express framework. Express provides a mechanism to handle HTTP requests and generate responses through numerous middleware as a pipeline. Because of the combination of React and Node.js from the beginning, MERN is a good architecture to implement this system. PICTURE 9 below demonstrates how MERN is constructed. (MongoDB 2023.)



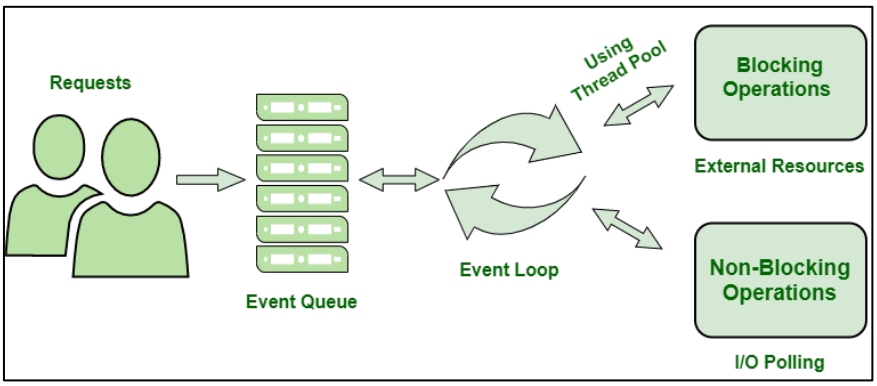
PICTURE 9. MERN Stack (MongoDB 2023)

5.2 Node.js Core Concepts & Modules

Node.js is built on the V8 JavaScript engine by Google Inc. Node.js is designed to build scalable network applications. Many connections can be handled in parallel based on threads. There are built-in modules and multiple libraries and frameworks built by the developer community. Consequently, the Node community helps developers boost the development process and maintenance tasks. Node.js architecture contains the concepts discussed in the following subchapters. (Node.js 2023.)

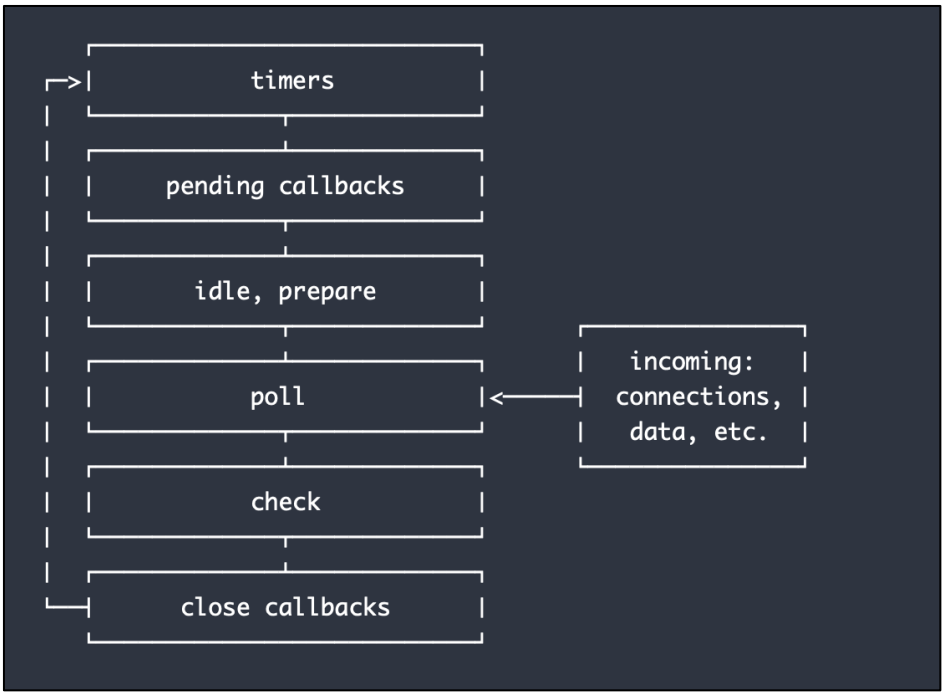
5.2.1 Event loop

Node.js is an event-driven technology, and it initializes variables and functions from the start. After that, it runs an event loop to listen to and handle events from connections. If there are no events to perform, Node application will be idle. This event loop is a fundamental concept of the Node.js design, called a “Single Threaded Event Loop”. JS is a single-thread language, which led to Node.js being designed without threads. But there are background threads, called worker threads, running in the background to perform blocking operations tasks (GeeksForGeeks 2023). PICTURE 10 below demonstrates how the Node.js Event Loop is constructed.



PICTURE 10. Workflow of Node.js Server (GeeksForGeeks 2023)

The Event Loop will check the nature of event requests and assign blocking requests to the process of completing the task using external resources. It can be confusing to say that Node.js is a single thread that runs tasks concurrently. As shown in PICTURE 11 below, an event loop is a loop that runs if the Node.js app is running. There are six phases in every loop: timers, pending callbacks, idle, poll, check, and close callbacks. There is one FIFO queue in every phase, which holds one or more callback functions that need to be executed. The event loop will try to operate callbacks in the phase queue until the queue has been exhausted or reached the maximum number of callbacks executed. After that, the event loop moves to the next phase and continues the flow. It executes one instruction at a time, but it gives the feeling that all tasks are running at the same time.



PICTURE 11. Event loop's order of operation (Node.js 2023)

5.2.2 NPM

In July 2023, there were three million packages that were listed in the NPM registry, making it the biggest single-language code repository in the world (Dobocan 2023). NPM is used to manage downloads of dependencies in a Node project and install them in the 'node_modules' folder from the project folder. NPM is also used to run tasks that are defined by key scripts in the 'package.json' file. The CODE 15 below is a sample of how to define tasks in a Node project and run a single task with 'npm run <task_name>'. Besides, specifying a version of a library with the command 'npm install <package_name>@<version>', helps to sync every developer with the same version of a package. (Node.js 2023.)

```
"scripts": {
    "ngrok": "ngrok http http://localhost:3001",
    "start:dev": "npx nodemon",
    "start": "npm run build && node build/index.js",
    "build": "rimraf ./build && tsc",
    "format:check": "prettier --check .",
    "lint:check": "eslint --ignore-path .gitignore --ext .js,.ts,.tsx .",
  },
```

CODE 15. Define tasks in Node.js project (of Thesis project)

5.2.3 Node Environment

In Node.js, the 'process' module provides the env property to read all the environment variables that were set when a Node application started (Node.js 2023). For example, to run an application in mode production, developers can use command as the CODE 59 in Appendix 7. Another way is using 'dotenv' package to load these variables during runtime. A '.env' file keeps all Node environment properties as CODE 60 in Appendix 7 and in a script file, using import to load 'dotenv', then read variables with 'process' module as CODE 16 in below.

```
import 'dotenv/config';

if (process.env.NODE_ENV !== 'production') {
  const accessLogStream = fs.createWriteStream(
    path.join(__dirname, 'access.log'),
    { flags: 'a' },
  );

  app.use(
    morgan('combined', { stream: accessLogStream,
      skip: function (req, res) { return res.statusCode < 400; },
    }),
  );
}
```

CODE 16. Using process module to read Node env to decide write logs onto file (Node.js 2023)

5.2.4 Non-Blocking I/O and Blocking I/O

In section Overview of Blocking vs Non-Blocking, Node team have referred to interaction with the system disk and network with support from a library named ‘libuv’. When a request comes to the server, Node.js triggers an off-passing request and does not wait for the response to avoid block operation. All the I/O tasks in Node.js can be handled asynchronously by ‘libuv’. Of course, there is also a synchronized version that blocks the I/O. For example, module ‘fs’ is used to read and write data logs into a file on disk and there are two versions of ‘fs’ as blocking and non-blocking I/O as CODE 17 below. It is necessary to consider using non-blocking code to allow for higher throughput. (Node.js 2023.)

```
import fs from 'fs';
// Blocking code
const data = fs.readFileSync('/file.md'); // Blocks here until file is read
// Non-blocking code
fs.readFile('/file.md', (err, data) => {
  if (err != null) throw err;
});
```

CODE 17. Blocking code and Non-blocking code (Node.js 2023)

5.2.5 Async and Await with Promise

According to MDN documents about Promise, a Promise object describes the eventual completion or failure of an asynchronous operation and its resulting value. A promise is resolved when its resolve

function is called or rejected with an exception uncaught within the async function. It allows asynchronous methods to return values at some point in the future, not immediately returning values like synchronous methods. In case developers need to wait for all values from two or more Promises, they can use the function 'Promise.all()' to get that. In Node.js, async and await are used to work with promise in asynchronous functions. It is a wrapper for restyling code and making code easier to read and use. For example, with and without 'await' and 'async', developers can write an asynchronous method such as CODE 18 below. In this sample, it uses 'Promise.prototype.then()' method which returns another Promise. In this way, promises can be chained. (MDN 2023.)

```
const samplePromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Success!'); // Return Success! after 250 ms
  }, 250);
});

// With await/async
const doSomething = async () => {
  const message = await samplePromise();
  console.log(`Yay! ${message}`);
};
doSomething();

// Without await/async
samplePromise.then((successMessage) => {
  console.log(`Yay! ${successMessage}`);
});

// Promise.all
const {message1, message2} = await Promise.all([
  samplePromise1(),
  samplePromise2(),
]);

// Chained Promises
samplePromise
  .then(handleFulfilled1)
  .then(handleFulfilled2)
  .then(handleFulfilled3)
  .catch(handleRejectedAny);
```

CODE 18. Sample of using Promise, async/await (MDN 2023)

5.2.6 Node.js Modules

In Node.js, modules are blocks of encapsulated code that do some functionalities and can be called by Node.js applications. They can be a file or multiple files or folders and can be loaded by CODE 61 in Appendix 7. There are three types of modules: core modules, local modules, and third-party modules (Node.js 2023). First, core modules are packaged inside Node.js. Some common core modules of Node.js can be found in Appendix 3. Second, local modules are created in Node.js applications locally by developers by providing methods, and attributes via ‘exports’ keyword. Then other files within the application can utilize the exported methods using the ‘import’ or ‘require()’ function. Last, third-party modules are modules that can be downloaded by NPM such as ‘express’, ‘mongoose’, ‘react’, 'cors’, and ‘nodemailer’.

5.2.7 Working with File System

Processing files is a critical task on the server side, present in virtually every system. For example, when users log in, updating their profile with a photo is a usual use case. A file has a set of metadata such as file owner, accessed time, updated time, and file size. They can be read by function ‘stat()’ via ‘fs’ module as CODE 62 in Appendix 7. In addition, ‘fs’ is a built-in module in Node.js to process with the file system. Module ‘fs’ provides methods to create, read, write, delete, rename, and track the changes in files and folders. For instance, in CODE 16 above, ‘fs’ provides the function ‘createWriteStream’ to write into file ‘access.log’. Function watch file in module ‘fs’ can be used for syncing the data content between clients. For example, multiple clients can access a file and update it, the function ‘watch()’ will listen for these changes and the application can use a web socket to send the changes of the file to other clients without delay same as a document service by Google Docs. The CODE 63 in Appendix 7 demonstrates how the ‘watch’ function works. (Node.js 2023.)

5.3 Database

There are numerous options for database management such as MySQL, PostgreSQL, SQLite, MongoDB, SQL Server, and Redis that Node.js can work with. MongoDB has become the most popular database for those learning to code behind MySQL, with 31.32% and 28.29% of professional developers using MongoDB in their project (Stack Overflow 2022). MongoDB allows users to build databases that are more future-proof with its scaling capabilities and flexible schema (MongoDB 2023). While most

developers refer to work with JSON because of its simplicity and flexibility, MongoDB created a Binary JSON format to support more data types than JSON, so it can be searched and indexed faster, which led to increasing performance (MongoDB 2023, Why use MongoDB and When to use it?). That is why MongoDB become an important part of this thesis project. To download and install the MongoDB driver, the package 'mongodb' is installed and used as CODE 19 below. The Mongo's scheme will be 'mongodb://<host>:<port>/database'.

```
const MongoClient = require('mongodb').MongoClient;
const url = "mongodb://localhost:27017/mydb";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  console.log("Database created!");
  db.close();
});
```

CODE 19. Connect MongoDB in local machine with 'mongdb' package (MongoDB 2023)

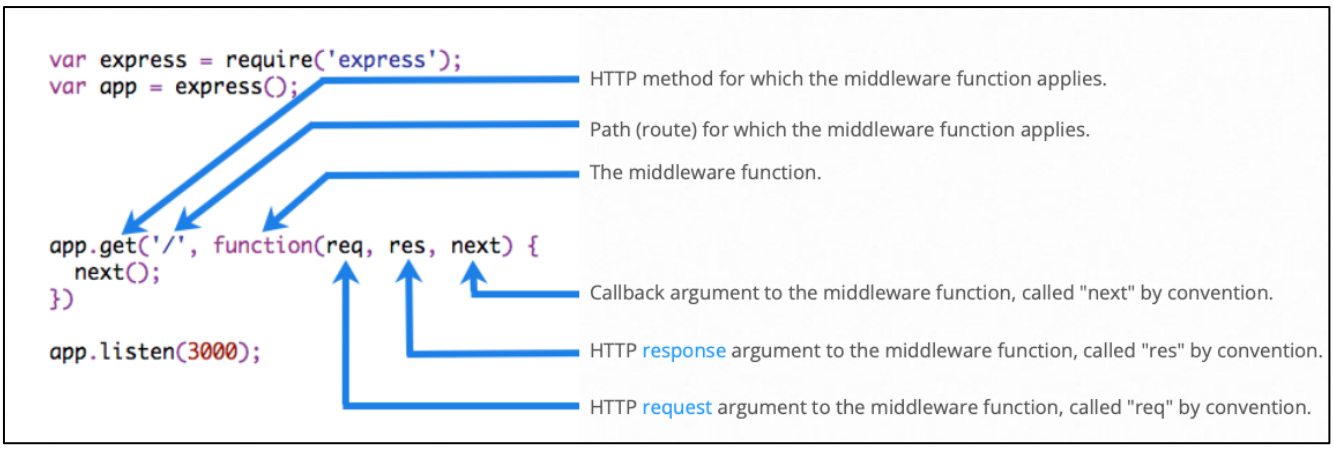
5.4 Express Server

Express.js is a web application framework built to create Node.js web-based applications. It provides an integrated environment to facilitate the rapid development of Node.js applications. Express includes several middleware modules that are used for additional requests and response activities (refer to Appendix 4). Express allows defining application routes using HTTP methods and URLs. To start coding with Express.js, in CODE 64 of Appendix 7, the 'express()' function is a top-level function exported by the 'express' module used to create a new Express application. Its function is like 'http' module inside Node.js. Additionally, in PICTURE 12 below, 'get()' is the HTTP method for which the middleware function applies. It can be 'get', 'post', 'head', 'update', and 'delete'. The 'get()' method receives two parameters, the first parameter is a string of a route path for which the middleware function applies. The second parameter is a middleware function. In this middleware function, there are three arguments which are HTTP request, HTTP response, and a callback to the middleware function. This format can be applied to other Express middleware. If the string path is not provided, then the middleware will be applied for all routes. (Express.js 2023.)

The route paths can be strings, string patterns, or regular expressions. Query strings are not part of the route path but route parameters. The router parameters are used to capture the values in the URL. These

values are populated in the 'req.params' object. For instance, if the router path is "/users/:userId/books/:bookId" and the request URL is 'http://localhost:3000/users/1/books/2', the route params will be '{"userId: 1, "bookId: 2}'. The hyphen and the dot are interpreted literally, and they can be used with router parameters for useful purposes. For example, if the route path is '/logs/:from-:to' and the request URL is 'http://localhost:3000/logs/10-20', the route params will be '{"from": 10, "to": 20}'. Or if the route path is '/float/:digit.:decimal' and the request URL is 'http://localhost:3000/floats/1.2', the router parameters will be '{"digit": 1, "decimal": 2}'. (Express.js 2023.)

Static is a built-in middleware function in Express. It serves static files which are stored in a directory path as the first parameter of 'static()' function. The CODE 65 in Appendix 7 represents how to use 'express.static middleware'. Another commonly used function of Express is 'Router()'. It creates a new router object that is an isolated instance of middleware and routes. Overall, a router acts like middleware itself and can be treated as a mini-application. It is suitable for separating logic code into other modules. A sample is provided in CODE 66 of Appendix 7.



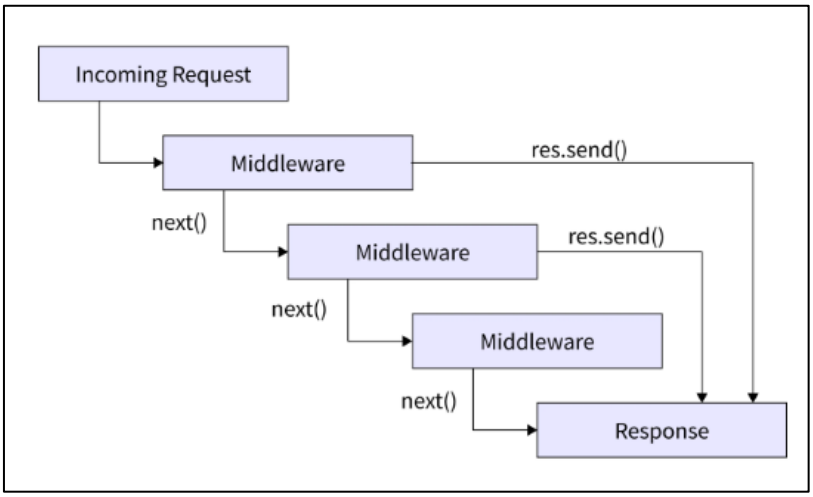
PICTURE 12. Elements of a middleware function call (Express.js 2023)

5.5 Express Middleware

Middleware is a function that accesses the request object, response object, and the next middleware function in the express application. Middleware functions can execute any code, make changes to the request and response, call the next middleware function in the chain, or end the request-response cycle. There are vast middleware functions that can be grouped into five types: application-level, router-level, error handling, built-in and third-party middleware. CODE 67 in Appendix 7 is a sample of using third-

party middleware to create a rate limitation in requests from the client side to limit requests per minute. (Express.js 2023.)

In a request-response cycle, there are numerous middleware are called before the final response is returned to the client. PICTURE 13 below demonstrates how the request-response cycle works. Leveraging middleware functions can include additional features and make a web application more secure and functional. With the capability to call multiple middleware functions in approach, the application becomes modular and flexible.

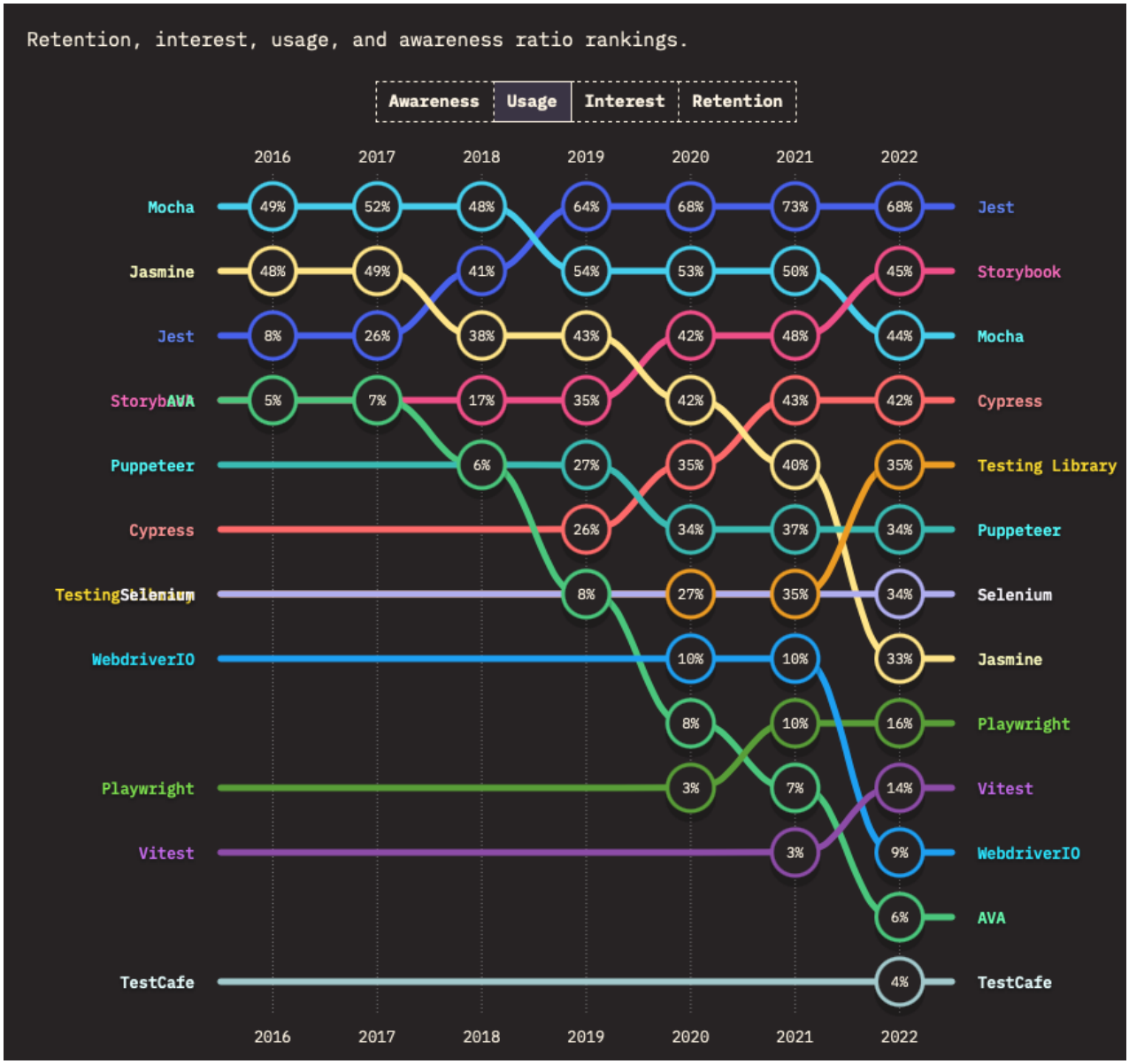


PICTURE 13. A request-response cycle in express application (Express.js 2023)

5.6 Testing

Testing is a crucial phase in SDLC, especially backend applications. There are numerous testing frameworks and libraries available for Node.js and support asynchronous testing. At the code level, unit testing is an important step to achieve of building a robust and reliable application. Jest, Storybook, Mocha, Cypress, and Testing Library are the top five testing frameworks that promise testing frameworks for Node.js (State of JS 2022). PICTURE 14 below represents the most used of testing frameworks and libraries in JS, and Jest is the most used in the list. Jest is a JS framework developed and maintained by Meta. Jest has several advantages compared with other competitors in this aspect (Jest 2023). Jest can reliably run tests in parallel by running them in their processes to maximize performance. It also generates code coverage by collecting information from the whole project, including not-tested modules. It has many built-in features for testing JS code, including mocking and snapshot testing. Basically, mocking allows to isolate a part of a module and test it with mockup data. This is useful for testing code that

depends on external resources, such as network requests or database requests. Meanwhile, snapshot testing is a technique that captures and compares snapshots of the expected output with the current returning result with the same parameters. (Jest 2023.)



PICTURE 14. Testing frameworks and libraries ranking (State of JS 2022)

To use Jest in a Node.js application, the Jest package is installed via NPM. Then a new test file with the extension '.test.js' or '.test.ts' can be created as CODE 20 below. In this code, jest, a test spec can be described as 'describe(name, fn)' and 'beforeAll(fn, timeout)' calls 'fn' function before any of tests in this file run. The 'expect()' function is used to test the response code that needs to be 200 as it passes the test. To run all test, developers can use 'jest' or 'jest –coverage' to get code coverage information in details.

```
import 'jest';
import type * as express from 'express';
import request from 'supertest';
import IntegrationHelpers from '../../../helpers/Integration-helpers';
describe('status integration tests', () => {
  let app: express.Application;
  beforeAll(async () => {
    app = await IntegrationHelpers.getApp();
  });
  it('can get server time', async () => {
    await request(app)
      .get('/api/status/time')
      .set('Accept', 'application/json')
      .expect((res: request.Response) => {
        console.log(res.text);
      })
      .expect(200);
  });
});
```

CODE 20. Testing with Jest (Jest 2023)

5.7 Server Security

Regarding security, Node.js's official website provides a list of security vulnerabilities such as denial of service of HTTP server, DNS rebinding, HTTP request smuggling, and malicious third-party modules. Node.js's popularity among backend developers makes it a target for online attacks. From section 5.1, MERN architecture with three-tier is used to protect user data in the data tier. Firewalls and proxies are built to proactively mitigate attacks directed from the frontend to the backend application. When installing new packages, NPM 'audit' and 'snyk' are tools for analyzing the project's dependencies and help developers to know vulnerabilities from third-party libraries. NPM audit can be used by typing the command into a terminal as can be seen in CODE 21 below. (Zanini 2023.)

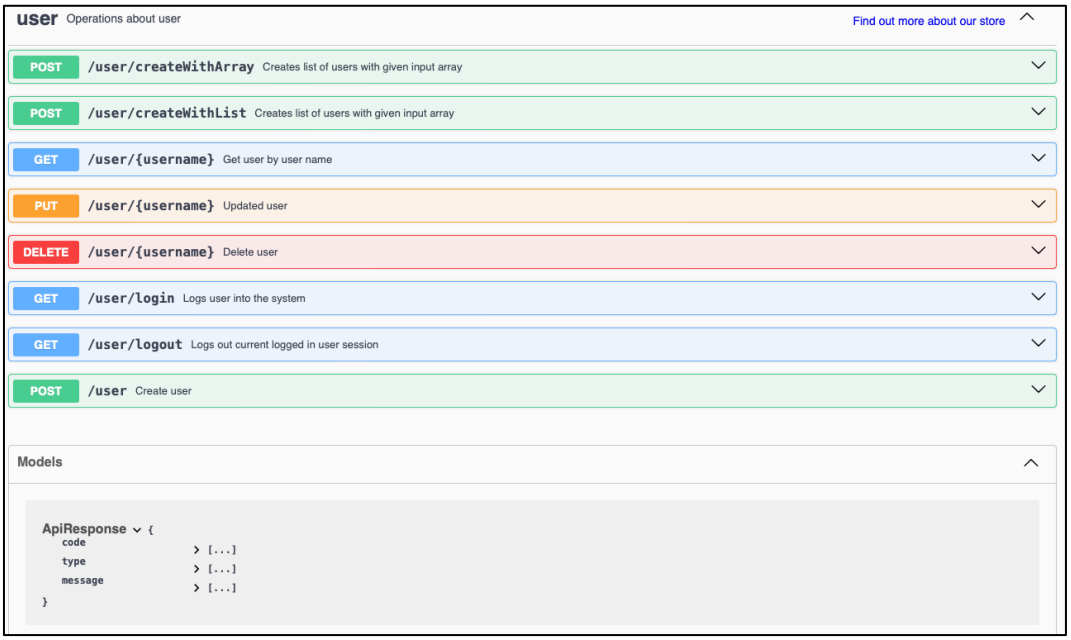
```
npm audit
...
found 4 vulnerabilities (2 low, 2 moderate)
run `npm audit fix` to fix them, or `npm audit` for details
```

CODE 21. Use npm audit to check security vulnerabilities from dependencies (Thesis project)

Snyk is a tool to check project dependencies against Snyk's Open-Source Vulnerability Database. In addition, 'express-session' middleware is used to project session cookies by renaming the cookies and encryption cookies with a secret key. The 'helmet' middleware is used to set the most important security headers such as 'X-XSS-Protection', 'X-Content-Type-Options', and 'Strict-Transport-Security'. Rate-limiting middleware from CODE 65 in Appendix 7 is a simple way to avoid DDoS and brute force attacks. Another way is using security linters such as 'eslint-plugin-security' which is ESLint rules to enforce security development in Node.js. Taking early action and focusing on security vulnerabilities within a Node.js application helps to reduce risks and creates a more reliable Node.js architecture. (Zanini 2023.)

5.8 REST API Documentation

API documentation is an important aspect of backend development. The API document is used by team members and other departments in the product team. It is a common way to communicate between frontend and backend teams because developers can understand how to structure requests and how to parse response data from the APIs by reading these documents. There are several ways to automatically generate these documents, and Swagger is an option used by this thesis project to implement the APIs documentation with a simple UI (Senger & Agrawal 2019). The Swagger team provides a library named 'swagger-ui-express' for integrating their tool into a Node.js application (Swagger UI 2023). PICTURE 15 below illustrates what Swagger UI looks like in a browser.



PICTURE 15. Sample of Swagger UI (Thesis project's swagger page)

6 NODE.JS IN PRACTICE

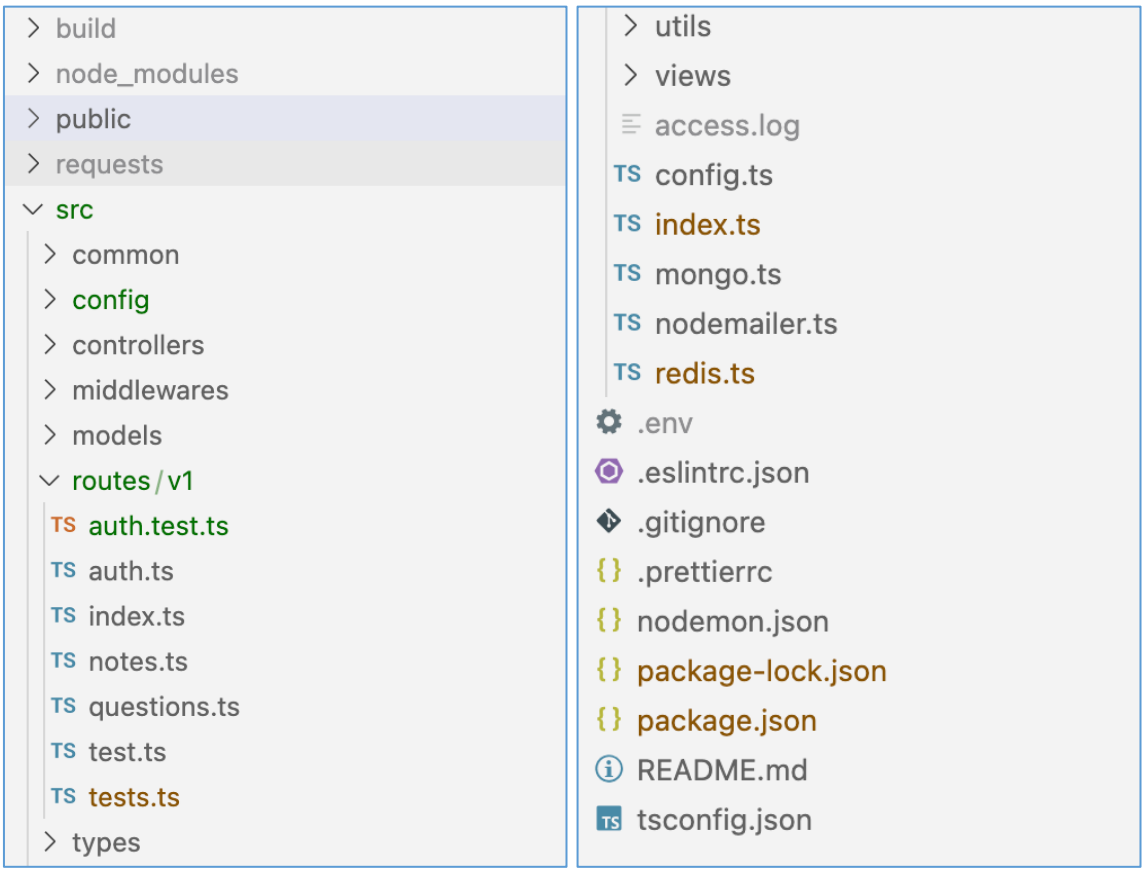
From section 3 and 4 about Front-end development with React, a React project was built with a simple UI and basic functionality to allow users to register accounts, login accounts, and fetch tests and questions. A set of interfaces connected between the front-end and back-end were built to speed up the development process. In this section, a new Node.js application is created to provide all the APIs that fulfills requirements from the front-end side. React application used the 'json-server' tool to create the mockup APIs and it works. Therefore, the minimum output of this Node.js application should work the same as the 'json-server' did.

6.1 Application Design

The Node.js application in the thesis follows the MERN architecture. It includes an express server inside a Node environment. The backend application connects to MongoDB server. Besides that, Redis is used to cache the user's token to add one more security layer into authentication feature. A JSON Web Token, known as JWT, is a standard that defines a compact way for securely sharing information between the client side and server side. The JWT data payload is signed using a secret with an encrypted algorithm as HMAC or a public-private key pair as RSA or ECDSA (IETF Data Tracker, RFC 7519). The most use of JWT is for securely sending authentication information, allowing users to access routes, services, and resources permitted with that token. JWT is good, but it is not perfect in all cases. For example, after users have a JWT token, they can send requests to the Node.js application without error until it expires, but what if a user wants to force log out of all sessions or just one session at a time? In that case, a simpler solution is to store the user's JWT token securely in the Redis database. JWT can be checked if it exists in Redis memory before the app allows the user's request to continue, otherwise it cannot be used anymore even if it has not expired. (JWT IO 2023.)

Node.js supports Typescript in coding. From the previous part of the React application, Typescript is used to enforce type safety in coding. It is applied to Node.js in the same way. Typescript can point out the mistake if developers call a function with the wrong parameters. In additional, ESLint is used to check syntax, find problems, and enforce coding style in the Node.js application. (Node.js 2023.)

Regarding code folder structure, code files are separated into several folders: config, controllers, middleware, models, routes, and views. Config folder contains configurations of JWT options such as secret, expired time, and limiter, MongoDB configs such as username, password, database host, Send-Grid config, and Redis server config. The 'dotenv' tool is used to keep sensitive information such as secret key, hash, username, and password for connecting databases into an environment file such as '.env.prod' or '.env.staging' or '.env'. Folder controllers keep all synchronous and asynchronous handler functions used in express routers. The middleware folder includes all the local middleware or third-party middleware wrappers such as verifying access tokens, identifying the user's role, validity of user's input, and limiter. Other built-in and third-party middleware are used in the main application or routers. Folder routes include all express routers such as auth, test, and question. The folder view contains HTML templates that are used to generate static content for response. Besides that, the public folder keeps all files that serve static resources. PICTURE 16 below demonstrates what the folder structure looks like.



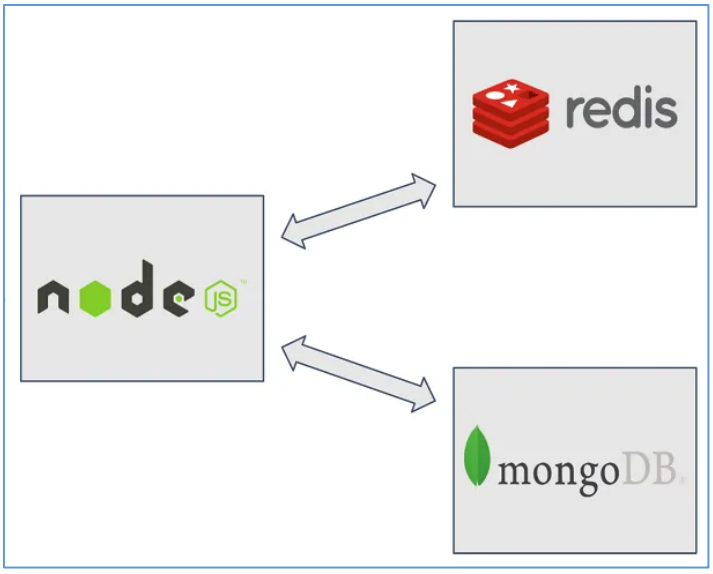
PICTURE 16. Node.js application's folder structure (Thesis project)

Unit testing requires to mock numerous of classes. The project has used the Jest framework for unit tests and test files have the same folder structures so that tests can be mapped with ease. The unit test is an

important phase in SDLC but in the scope of the thesis, unit tests only did the minimum tasks to describe how Jest is used, not covering all test cases.

6.2 Database Design

The Node.js application uses MongoDB as a primary database to store users, questions, tests, and user answers. Besides that, Redis is used to store JWT token. Redis is an in-memory key-value store which means it stores data on the host's RAM, not on disk. So Redis allows reading and writing data faster than performing disk operations. In this case, Redis plays the role of a session management system. PICTURE 17 below illustrates the relationship between Node.js, Redis and MongoDB. (Redis 2023.)



PICTURE 17. Node.js connect to MongoDB and Redis (Redis 2023)

When working with MongoDB, developers have several options for data interaction. These options include the native MongoDB driver named 'mongodb' and 'mongoose,' a Node.js-based Object Data Modeling library. Mongoose is a good option because it allows developers to enforce a specific schema at the Node application. It also provides an easier way for hooks, model validation, and other features to work with MongoDB. CODE 68 in Appendix 7 is a sample of how to implement data schemes in 'mongoose'. The 'validator' package is used to validate user's input such as email, phone number, and strong password. Then the scheme can be used as CODE 69 in Appendix 7. User's password is encrypted by simple 'bcrypt' tool before it is saved into database. CODE 22 below represents how Node.js app connects to MongoDB server. (Kukic 2021.)

```
db.mongoose
  .connect(
`mongodb+srv://${dbConfig.username}:${dbConfig.pwd}@${dbConfig.HOST}/${dbConfig.DB}?retryWrites=true&w=majority`,
    {},
  )
  .then(async () => {
    console.log('Successfully connect to MongoDB.');
    await initial();
    await addQuestions();
    await addTests();
  })
  .catch((err) => {
    console.error('Connection error', err);
    process.exit();
  });
```

CODE 22. Connect MongoDB via mongoose (MongoDB 2023)

To work with Redis, ‘redis’ package is installed via NPM. After that, Node.js application connects to Redis server as CODE 70 in Appendix 7. Then ‘redisClient’ can be used as CODE 71 in Appendix 7. By early checking the ‘accessToken’ that is gotten from the request headers, which exists in Redis, this technique can reduce resources of the Node server for processing the next step. Jwt package is used to verify and get ‘userId’ and modify requests by adding ‘userId’ into it. With this logic, the next middleware can check the role of the user easily.

6.3 APIs Design

For API versioning, routers are put inside version folders named ‘v1’, ‘v2’ in case the application supports several API versions at the same time. The final URL request will have the format ‘https://<server_host>/api/v1/<route_path>’. The implementation for API versioning can be found in CODE 72 in Appendix 7. In CODE 66 in Appendix 7, the express router contains numerous route paths with the format ‘router.<REST_method> ('/', [<middleware functions>], handler function)’. The middle parameter is a list of middleware that involves access, validation requests, and modification requests. For example, to access the Admin dashboard, the two middleware of ‘verifyToken’ and ‘checkIsAdmin’ can be used to detect the user’s role as admin or not. The last parameter is a handler function. This function is provided by controller modules. According to the Express.js API reference, express routers only accept the handler function that returns the type of void. Therefore, to make the asynchronous

function work in this case, the ‘express-async-handler package it is hot used as a wrapper for the asynchronous function as can be seen in CODE 23 below. There is a list of core APIs that can be seen in Appendix 5.

```
import asyncHandler from 'express-async-handler';

const signup = asyncHandler(
  async (req: Request, res: Response, next: NextFunction) => {
     // signup logic
  },
);
```

CODE 23. Sample use of express-async-handler (Thesis project)

6.4 Setup project

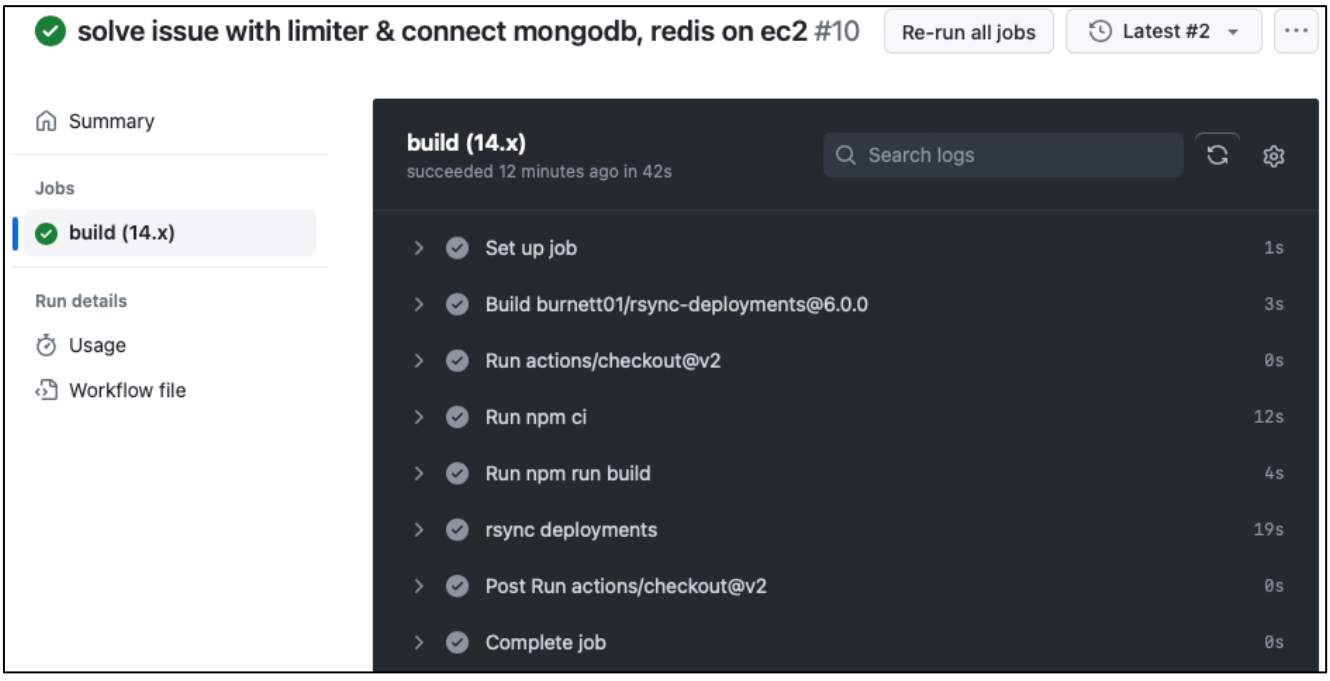
Before starting with the Node.js application, it is required to install Node.js into the development machine. Node.js can be installed on Windows, Linux, or macOS. NPM has come up with a Node.js setup package. By running ‘npm init’, it will create a ‘package.json’ file with basic contents such as name, version, description, and scripts. From the application design overview path, “express”, ‘mongoose’, ‘redis’, ‘dotenv’, ‘jsonwebtoken’, ‘nodemailer’, and ‘validator’ are installed as core packages. In additional, ‘nodemon’, ‘supertest’, and typescript are installed as dev dependencies. Nodemon is a tool that watches project directory and automatically restarts Node.js application if there are any changes. Nodemon can be configured by adding contents as can be seen in CODE 73 of Appendix 7. The React application is using the ‘dotenv’ tool to read variables environment from ‘.env’ file. It is a convenient way to change the host URL of server. After running the Node.js server, by running ‘ngork’ command, it creates a tuner from outside internet to server in local machine via a new URL. This URL can be used to connect the client and the server.

6.5 CI/CD & Containers Setup

Implementing a CI/CD pipeline is the final step of SDLC for continuously deploying Node.js applications to the cloud. The frequent changes of code will be automated built, tested, and deployed to production without errors made by humans. In addition, configuring the pipeline to enforce linting rules improves code quality, which means a robust and reliable codebase. There are numerous approaches

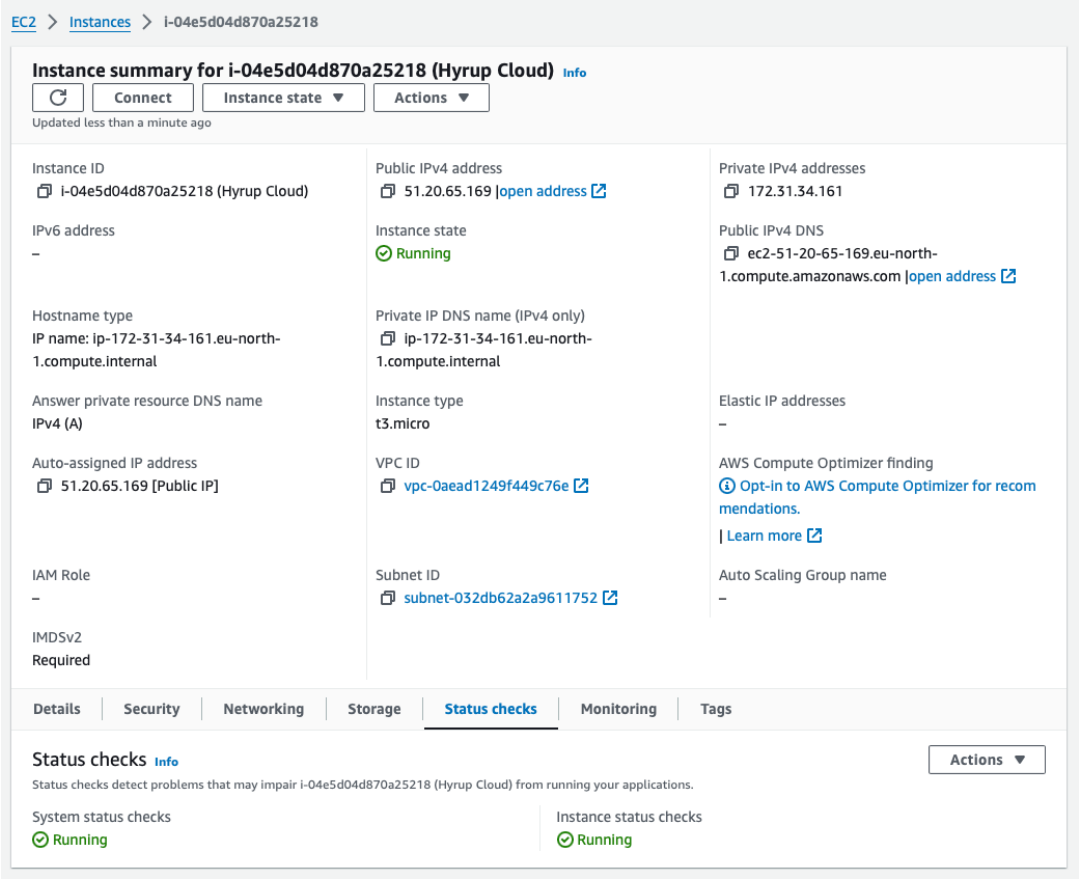
such as GitLab, Jenkins, CircleCI, and Travis CI for configuring CI/CD. In this thesis project, GitHub Action is used to set the CI/CD pipeline and AWS is used as a hosting service. A GitHub action is triggered to run every time there is a merge to the main branch that will always work in production. Another condition for running GitHub action is that it only allows merges when the branch is up-to-date with the main branch, therefore different developers cannot overwrite each other's changes. (GitHub 2023.)

In GitHub Action, the process flows, also known as Workflow, can run several jobs such as building, testing, linting, releasing, and deploying. In each job, developers can define specific step-by-step to run individual tasks. These jobs can be run in parallel and steps in each job will be executed sequentially. To configure the GitHub Action, a 'yaml' config file is created in the folder '.github/workflows' in the root repository folder. A basic workflow can be described in the YAML document by three elements such as 'name', 'on' known as the events that trigger the workflow to be executed, and 'jobs'. In the 'jobs' section, each job is listed as '<job_name>:', 'run-on' environment, 'steps' includes run scripts that will be executed. A simple workflow for building the application is shown as CODE 74 in Appendix 7. Each time a new commit is pushed onto main branch, GitHub Action trigger a CI workflow that configured by 'aws.yml' file. This workflow is visualized by GitHub UX as PICTURE 18 below. By the 'rsync deployment' action, a set of Node.js files are built in production mode and then is copied to an AWS EC2 instance. (GitHub 2023.)



PICTURE 18. CI/CD pipeline jobs in GitHub Action UX (Thesis project)

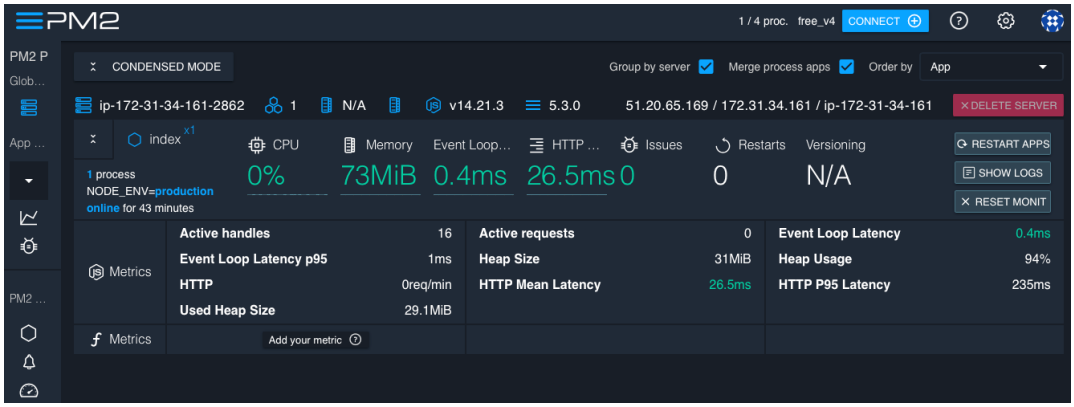
Regarding AWS EC2 service, Node.js applications can be hosted in a separate virtual server by EC2 instances. It provides scalable computing capacity in the cloud, allows developers to easily config security and networking rules, and manages storage, leading to a reduction in the cost of physical resources. With an EC2 instance, a Node.js application can be run in the same way as the local machine does. It means that developers can run shell scripts to install several packages such as 'nginx', 'pm2', 'git', 'nodejs', 'redis-cli', and config network proxy by manually. Developers can check the virtual machine status, configure, and monitor CPU utilization, and network in/out used via EC2 UX that looks like PICTURE 19 below. The IPv4 address of the instance is added to MongoDB whitelist IP addresses to allow connection between the Node.js app and the database hosted by MongoDB. This IP is also used to connect to the virtual machine via an SSH client so that developers can run scripts to set up the environment inside it. (AWS 2023.)



PICTURE 19. Preparing AWS EC2 instance to host Node.js application (Thesis project)

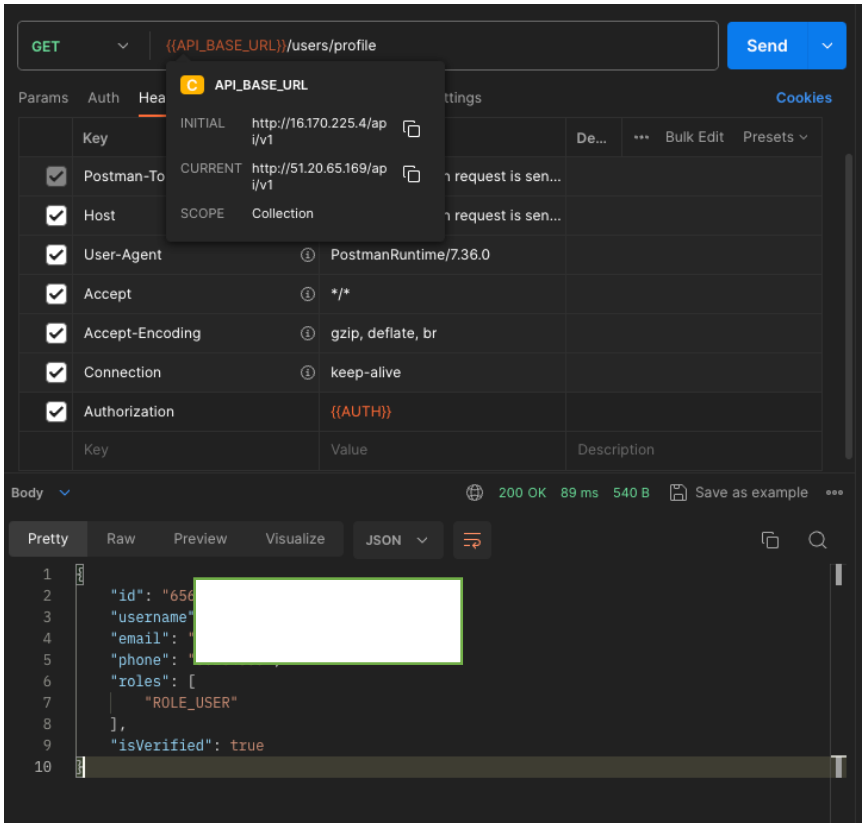
After the CI/CD step, a set of JS files that can run the Node.js app is copied into the '/var/www/app' folder in the EC2 virtual machine, To run the app and manage it, PM2 is used. PM2 creates a daemon process that helps to manage and keep the application online and restart the app if there are any changes

in files by command "'pm2 start '/var/www/app/index.js –watch'". Developers can check application status by PM2 UX like PICTURE 20 below.



PICTURE 20. Use PM2 tool to start and monitor Node.js application on AWS EC2 (Thesis project)

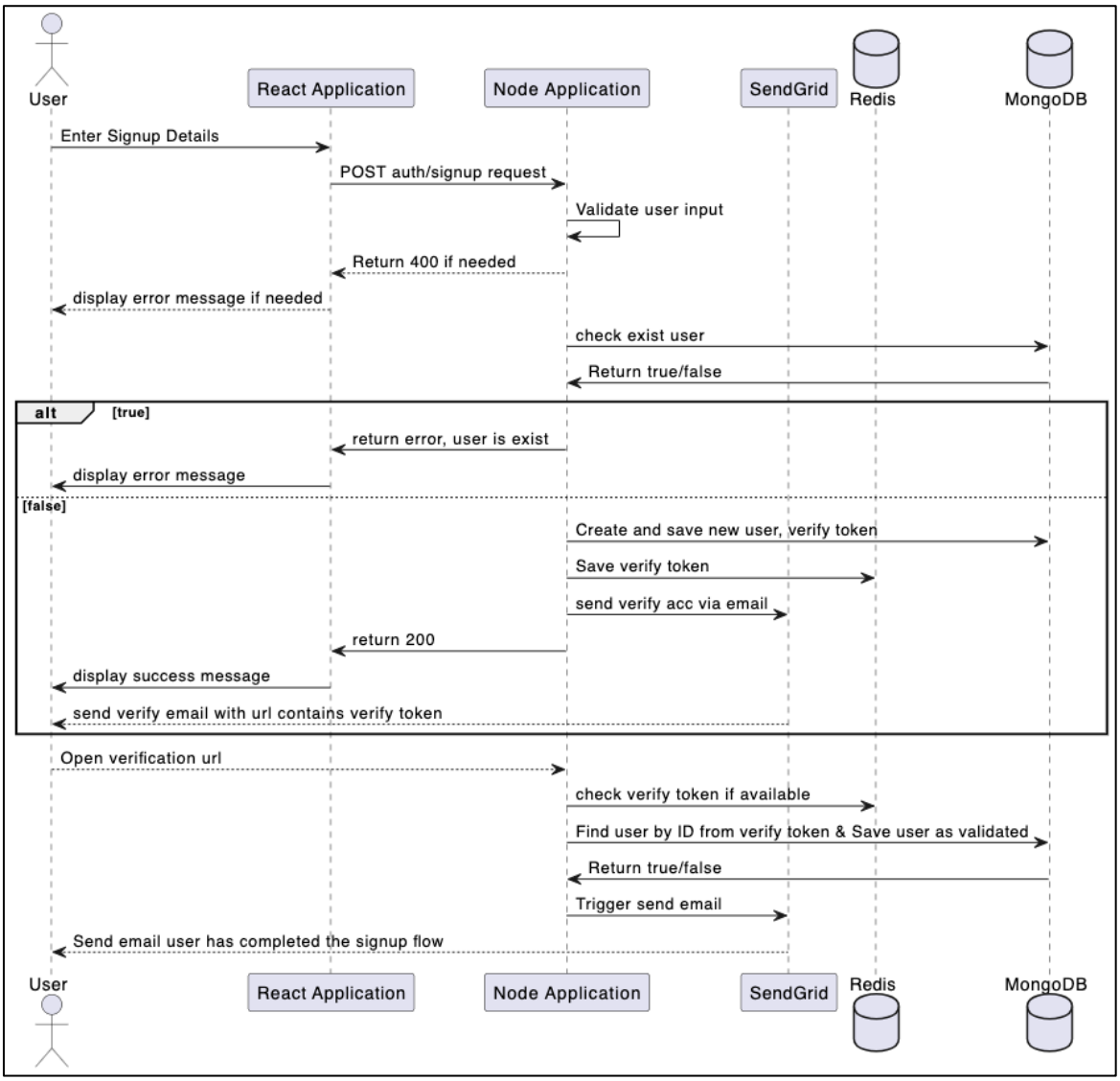
The UI provides real-time insights into the application's performance. It displays key metrics such as CPU usage, memory consumption, latency, and active requests. This allows developers to monitor the app's health and identify potential bottlenecks. Like the "server-json" component in the React portion, the public IP address of the EC2 instance serves as the API endpoint for this functionality. For reference, PICTURE 21 illustrates a sample request and response using the new URL.



PICTURE 21. Sample for getting user's profile with production endpoint (Thesis project)

6.6 Authentication Use Case

In this section, a typical use case named authentication is described as a sample of how a React application integrates with a Node.js back-end. Sign up workflow is shown in PICTURE 22 below. Node.js application plays an important role in handling signup requests, validating user input, and connecting MongoDB to create user records. The user's password is encrypted before being saved into the database. A verification token is created with the user's ID information to be used to verify the user's email address. After this step, the user can sign in with an email and password. SendGrid is also important in this use case because it will send directly to users the verification token in a link that can be clickable and open in a web browser. By opening the link, the browser sends a request 'auth/verifyAccount' to the Node app. This flow is a security technic reduces the risk of unauthorized users using the system or creating an account by bots. The sign-up code logic can be found in CODE 75 of Appendix 7.



PICTURE 22. Sign up flow diagram (Thesis project)

7 CONCLUSIONS

In summary, the three research questions can be answered. First, front-end development, particularly the use of React, demands a substantial knowledge base. Technic is updated more frequently. Developers must update their knowledge to adapt to its changes. React Native can cover almost all cases of front-end development, and it is a good choice to start with. Because of a wide range of dev communities and the JS language itself, it is the fastest to learn, allowing developers to become experts quickly and get the quickest support from the dev community. Of course, there are some alternative cross-platform frameworks like Flutter, Ionic, Cordova, and Xamarin, nowadays known as MAUI, but React Native is much more advantageous than Flutter and others.

Regarding back-end development, it focuses on creating server-side logic, infrastructure, and security. Especially in Node.js, developers have numerous options to solve their needs for handling databases, managing requests and responses, and ensuring the functionality and performance of the application. Node.js has become a popular choice for back-end development due to its capability and support from the developer community. Combining React and Node.js is a good solution to speed up the development process and reduce the cost of human resources. A development team or an individual developer with a basic JS skill set can fit in this situation.

There are React and Node.js problems that are not mentioned in the scope of this thesis, such as question bank, copyright protection of content and images, debugging, logging, data analysis, unit tests, security issues, device capabilities, defects, and issue solving. Because of limited time and resources, the project thesis provides only the core features of the application design. React and Node.js together have the potential to unlock exciting new features in future commercial products. An indie developer with JS knowledge and skills in React and Node.js can create a full stack web and mobile application in just a short time.

REFERENCES

Android. 2023. *Setting an optimal frame rate using Frame Rate API*. Available at https://developer.android.com/media/optimize/performance/frame-rate. Accessed 13.11.2023.

AWS. 2023. What is Amazon EC2. Available at https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html. Accessed 19.12.2023.

Babel. 2023. *What is Babel*. Available at https://babeljs.io/docs. Accessed 01.11.2023.

Bento, L. 2019. React Native Community, *Meet Doctor, a new React Native command*. Available at https://reactnative.dev/blog/2019/11/18/react-native-doctor. Accessed 27.10.2023.

ClickITech. 2022. *Web Application Architecture: The Latest Guide 2022*. Available at https://www.clickittech.com/devops/web-application-architecture. Accessed 20.11.2023.

Cypress. 2023. *Cypress Overview*. Available at https://docs.cypress.io/guides/overview/why-cypress. Accessed 01.11.2023.

Davis, A. 2023. *Creating a responsive, API-powered, PaaS-deployed, single-page application in React*. Available at https://www.theseus.fi/handle/10024/800734. Accessed 20.12.2023.

DhiWise. 2022. *Why to Choose React JS for Single Page Application Development*. Available at https://www.dhiwise.com/post/reactjs-for-single-page-application-development. Accessed 31.10.2023.

Dobocan, G. 2023. State Of Npm 2023: The Overview. Available at https://blog.sandworm.dev/state-of-npm-2023-the-overview#heading-total-packages-count-3342873. Accessed 20.11.2023.

Stepnov, E. 2022. *10+ Best Ways to Deploy React App*. Available at https://flatlogic.com/blog/best-ways-to-deploy-react-app. Accessed 31.10.2023.

Express.js. 2023. *Express middleware*. Available at https://expressjs.com/en/resources/middleware.html. Accessed 26.11.2023.

Express.js. 2023. *Req.params*. Available at https://expressjs.com/en/5x/api.html#req.params. Accessed 26.11.2023.

Express.js. 2023. Using middleware. Available at https://expressjs.com/en/guide/using-middleware.html. Accessed 26.11.2023.

Gamage, D. 2023. *How to Manage Server State With React Query*. Available at https://blog.bitsrc.io/how-to-manage-server-state-with-react-query-79557a605a22. Accessed 01.11.2023.

GeeksForGeeks. 2023. *Express.js*. Available at https://www.geeksforgeeks.org/express-js. Accessed 26.11.2023.

GeeksForGeeks. 2023. *Node.js Web Application Architecture*. Available at https://www.geeksforgeeks.org/node-js-web-application-architecture. Accessed 20.11.2023.

Goswami, P. Gupta, S. Li, Z. Meng, N. Yao, D. 2020. *Investigating The Reproducibility of NPM Packages*. IEEE International Conference on Software Maintenance and Evolution (ICSME), Adelaide, SA, Australia, 2020, pp. 677-681, doi: 10.1109/ICSME46990.2020.00071.

IBM, 2023. *What is three-tier architecture?* Available at https://www.ibm.com/topics/three-tier-architecture. Accessed 20.11.2023.

Immer, 2023. *Official docs*. Available at https://github.com/immerjs/immer. Accessed 27.10.2023.

Jest. 2023. Testing Asynchronous Code. Available at https://jestjs.io/docs/asynchronous. Accessed 26.11.2023.

Jest. 2023. *Snapshot Testing*. Available at https://jestjs.io/docs/snapshot-testing. Accessed 26.11.2023.

JetBrains, 2022. *The State of Developer Ecosystem 2022*. Available at https://www.jetbrains.com/lp/devecosystem-2022. Accessed 31.10.2023.

Json-server. 2023. *Getting started*. Available at https://github.com/typicode/json-server#getting-started. Accessed 01.11.2023.

Kukic, A. Vlaeva, S. 2023. *MongoDB & Mongoose: Compatibility and Comparison*. Available at https://www.mongodb.com/developer/languages/javascript/mongoose-versus-nodejs-driver/. Accessed 29.11.2023.

Metsäpelto, A. 2022. *Ways of Using CSS in React.js*. Available at https://www.theseus.fi/bitstream/handle/10024/746791/Metsapelto_Anna.pdf. Accessed 20.12.2023.

MongoDB. 2023. *MERN Stack Explained*. Available at https://www.mongodb.com/mern-stack. Accessed 20.11.2023.

MongoDB. 2023. *Why use MongoDB and When to use it?* Available at https://www.mongodb.com/why-use-mongodb#:~:text=Using%20MongoDB%20enables%20your%20team,a%20large%20number%20of%20teams. Accessed 20.11.2023.

National Center of Deaf-Blindness. 2023. *Why accessibility is important*. Available at https://www.nationaldb.org/for-state-deaf-blind-projects/accessibility-toolkit/why-accessibility-is-important. Accessed 18.11.2023.

Node.js. 2023. *About Node.js*. Available at https://nodejs.org/en/about. Accessed 20.11.2023.

Node.js. 2023. *What is the Event Loops?* Available at https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick. Accessed 22.11.2023.

OpenAI. 2023. *Introducing ChatGPT*. Available at https://openai.com/blog/chatgpt. Accessed 01.11.2023.

Phull, R. 2020. *Getting Started with Typescript with React: 4 Easy Steps*. Available at https://www.codementor.io/@rajjeet/getting-started-with-typescript-4-easy-steps-1do2bnxl6i. Accessed 01.11.2023.

Pregasen, M. 2023. *Putting the Expo vs React Native debate to rest*. Available at https://retool.com/blog/expo-cli-vs-react-native-cli. Accessed 01.11.2023.

Pressman, R. 2010. *Software Engineering, A Practitioner's Approach, 7th edition*. ISBN 978–0–07–337597–7 — ISBN 0–07–337597–7.

React. 2023. *Preserving and Resetting State*. Available at https://react.dev/learn/preserving-and-resetting-state. Accessed 01.11.2023.

React, 2023. *Quick Start*. Available at https://react.dev/learn. Accessed 27.10.2023.

React, 2023. *Render and Commit*. Available at https://react.dev/learn/render-and-commit. Accessed 31.10.2023.

React-i18next, 2023 .*Introduction*. Available at https://react.i18next.com. Accessed 27.10.2023.

React Native, 2023. *React Native – Learn once, write anywhere*. Available at https://reactnative.dev. Accessed 27.10.2023.

React Native, 2023. *Setting up the development environment*. Available at https://reactnative.dev/docs/environment-setup. Accessed 31.10.2023.

React Native, 2023. *React Native Directory*. Available at https://reactnative.directory. Accessed 27.10.2023.

React Native Dev. 2023. *Out-of-Tree Platforms*. Available at https://reactnative.dev/docs/out-of-tree-platforms. Accessed 31.10.2023.

React Native for Web. 2023. *Multi-platform setup*. Available at https://necolas.github.io/react-native-web/docs/multi-platform. Accessed 01.11.2023.

React Query, 2023. *Official Docs*. Available at https://tanstack.com/query/v3/docs/react/overview. Accessed 27.10.2023.

React TypeScript Cheatsheet. 2023. Available at https://react-typescript-cheatsheet.netlify.app/docs/basic/recommended/codebases. Accessed 27.10.2023.

Redux, 2023. *Getting started with Redux*. Available at https://redux.js.org/introduction/getting-started. Accessed 27.10.2023.

Satzinger, J. Jackson, R. Burd, S. 2000. *Systems Analysis and Design in a Changing World*. Available at https://www.cerritos.edu/dwhitney/SitePages/CIS201/Lectures/IM-7ed-Chapter06.pdf. Accessed 31.10.2023.

Senger, A. Agrawal, S. 2019. *API Modeling and Description Languages*. Available at https://agrawalshyam.com/reports/Seminar_Web_Engineering_API_Modelling.pdf. Accessed 20.12.2023.

Shahab, H. 2023. 9 React components libraries for efficient development in 2024. Available at https://ably.com/blog/best-react-component-libraries. Accessed 20.12.2023.

Stack Overflow. 2022. Result Developer Survey 2022. Available at https://survey.stackoverflow.co/2022/#most-popular-technologies-database-prof. Accessed 26.11.2023.

State of JS. 2022. *Ranking Testing frameworks and libraries*. Available at https://2022.stateofjs.com/en-US/libraries/testing. Accessed 26.11.2023.

Statista, 2023. *Software development - statistics & facts*. Available at https://www.statista.com/topics/1694/app-developers. Accessed 27.10.2023.

Styled-components, 2023. *Official docs*. Available at https://styled-components.com/docs. Accessed 27.10.2023.

Swagger. 2023. *Swagger UI*. Available at https://swagger.io/tools/swagger-ui. Accessed 26.11.2023.

Teoriklar, 2023. *Tips for the theory test*. Available at https://www.teoriklar.eu/doc/menu/6/gode-raad-foer-teoriproeven. Accessed at 31.10.2023.

Velkov, K. 2023. *Mastering React JS SOLID Principles*. Available at https://blog.stackademic.com/react-js-mastering-react-js-solid-principles-dfb48d03e565. Accessed 01.11.2023.

W3Schools. 2023. Node.js Build-in Modules. Available at https://www.w3schools.com/nodejs/ref_modules.asp. Accessed 25.11.2023.

Zanini, A. 2023. *Best Practices for Securing Node.js Applications in Production*. Available at https://semaphoreci.com/blog/securing-nodejs. Accessed 26.11.2023.

Zeplin, 2023. *Getting started with Zeplin for developer*. Available at https://support.zeplin.io/en/articles/6577298-getting-started-with-zeplin-for-developers. Accessed 27.10.2023.

Zustand, 2023. *Official docs*. Available at https://github.com/pmndrs/zustand. Accessed 27.10.2023.

Most used web frameworks among developers worldwide, as of 2023.
– By Statista, Software development - statistics & facts.

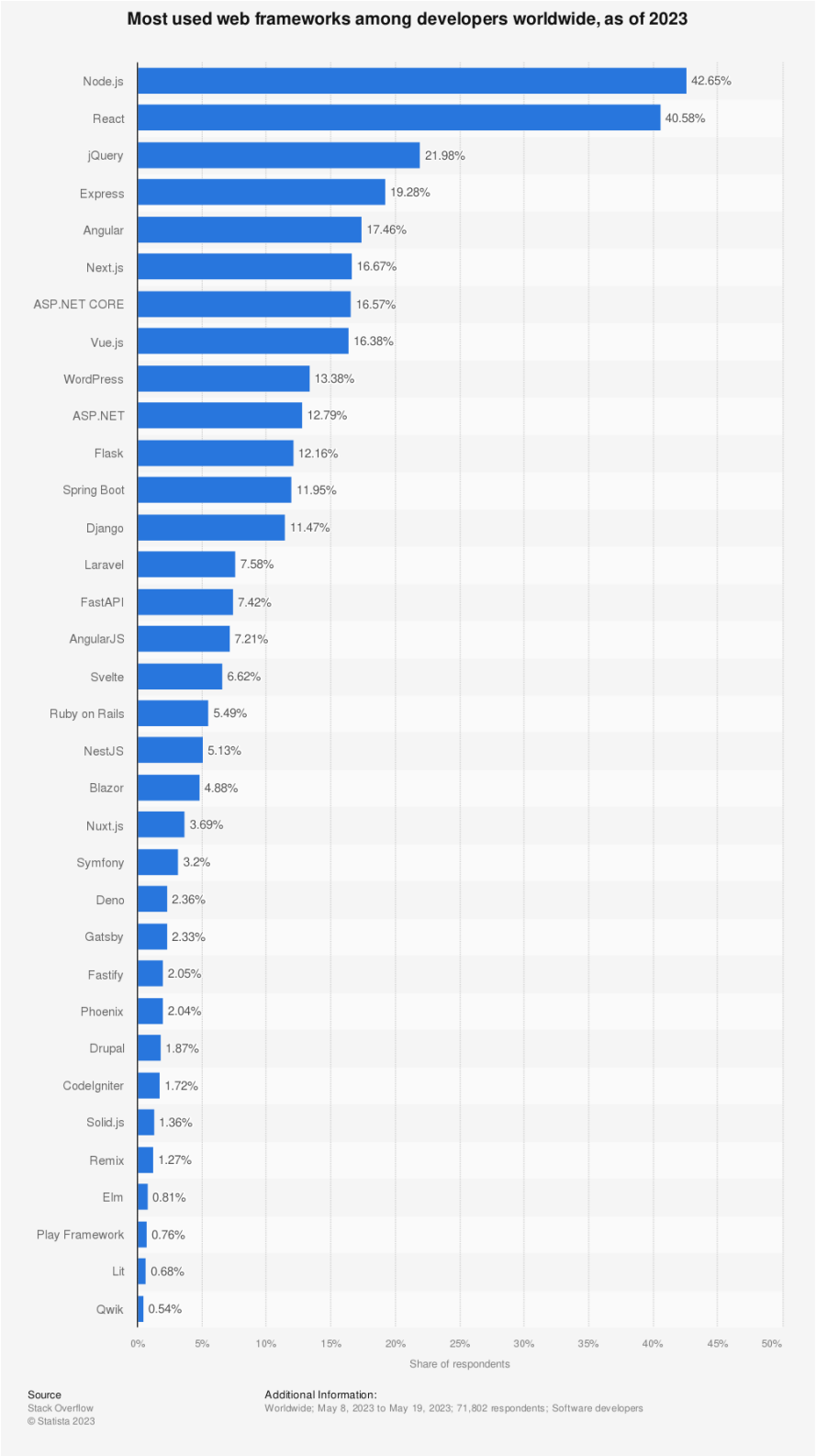


TABLE 1. React UI Library comparison in December 2023 (Shahab 2023).

React UI library	GitHub stars	Functions	Newness
Material UI	89.3K	Google's Material design; Pre-built components; Accessibility; Theme support.	10 years
Ant Design	88K	Enterprise-level components: data tables, form, charts; Professional look; Support type safety.	9 years
Chakra UI	34.7K	Simple, modular, and accessible UI Components	5 years
Headless UI	22.5K	A set of unstyled UI components; Accessibility; Well-documented	4 years
React Bootstrap	22K	Implemented from Bootstrap CSS; Mobile-first and responsive; Well-documented	11 years
Mantine	21.9K	Theme engine; Hooks library; Customized components; Styles overriding; Flexible theming.	3 years
Next UI	16.3K	Build on top of React & Tailwind CSS; Custom themes; Accessibility; Dark mode; Easy Customizations.	3 years
Semantic UI React	13.1K	jQuery-free, declarative API, beautifully styled React components for enterprise-class UI; Theming; Extensibility.	9 years
Grommet	8.3K	Styled with CSS-in-JS; Theming; Accesibility.	9 years

TABLE 2. Node.js Build-in Modules (Node.js 2023)

Module name	Description
buffer	buffer module can be used to create Buffer class.
console	console module is used to print information on stdout and stderr.
dns	dns module is used to do actual DNS lookup and underlying o/s name resolution functionalities.
domain	domain module provides way to handle multiple different I/O operations as a single group.
fs	fs module is used for File I/O.
net	net module provides servers and clients as streams. Acts as a network wrapper.
os	os module provides basic o/s related utility functions.
path	path module provides utilities for handling and transforming file paths.
process	process module is used to get information on current process.
http	creates an HTTP server in Node.js.
assert	set of assertion functions useful for testing.
querystring	utility used for parsing and formatting URL query strings.

TABLE 3. Express middleware (Express.js 2023)

Middleware module	Description
body-parser	Parse HTTP request body. See also: body, co-body, and raw-body.
compression	Compress HTTP responses.
connect-rid	Generate unique request ID.
cookie-parser	Parse cookie header and populate req.cookies. See also cookies and keygrip.
cookie-session	Establish cookie-based sessions.
cors	Enable cross-origin resource sharing (CORS) with various options.
errorhandler	Development error-handling/debugging.
method-override	Override HTTP methods using header.
morgan	HTTP request logger.
multer	Handle multi-part form data.
response-time	Record HTTP response time.
serve-favicon	Serve a favicon.
serve-index	Serve directory listing for a given path.
serve-static	Serve static files.
session	Establish server-based sessions (development only).
timeout	Set a timeout period for HTTP request processing.
vhost	Create virtual domains.
cls-rtracer	Middleware for CLS-based request id generation. An out-of-the-box solution for adding request ids into your logs.
connect-image-optimus	Optimize image serving. Switches images to .webp or .jxr, if possible.
express-debug	Development tool that adds information about template variables (locals), current session.
express-partial-response	Filters out parts of JSON responses based on the fields query-string; by using Google API's Partial Response.
express-simple-cdn	Use a CDN for static assets, with multiple host support.
express-slash	Handles routes with and without trailing slashes.
express-stormpath	User storage, authentication, authorization, SSO, and data security.
express-uncapitalize	Redirects HTTP requests containing uppercase to a canonical lowercase form.
helmet	Helps secure your apps by setting various HTTP headers.
join-io	Joins files on the fly to reduce the requests count.
passport	Authentication using "strategies" such as OAuth, OpenID and many others. See http://passportjs.org/ for more information.
static-expiry	Fingerprint URLs or caching headers for static assets.
view-helpers	Common helper methods for views.
sriracha-admin	Dynamically generate an admin site for Mongoose.

TABLE 4. Core APIs

No	Method	Path	Description
1	GET	/	Check server status
2	POST	/api/v1/auth/signup	Create new user
3	POST	/api/v1/auth/signin	Login user
4	POST	/api/v1/auth/signout	User log out
5	POST	/api/v1/auth/refreshToken	Get new accessToken
6	GET	/api/v1/auth/verifyAccount	Verify user email
7	POST	/api/v1/auth/resendVerify	Resend email for verify email
8	POST	/api/v1/auth/resetPassword	Reset user password
9	POST	/api/v1/auth/recoverPassword	Request recover password
10	GET	/api/v1/questions/	Get questions by page, limit
11	GET	/api/v1/questions/statistic	Get question statistic
12	GET	/api/v1/tests/	Get tests by page, limit
13	GET	/api/v1/tests/free	Get free tests
14	GET	/api/v1/user/	Get user details
15	POST	/api/v1/user/	Update user metadata
16	GET	/admin	Admin dashboard
17	POST	/api/v2/admin/users/:id	Update user
18	GET	/api/v2/admin/users/:id	Get user info
19	POST	/api/v2/admin/questions/:id	Update question by id
20	POST	/api/v2/admin/tests/:id	Update tests by id
21	DELETE	/api/v2/admin/tests/:id	Delete test by id
22	DELETE	/api/v2/admin/questions/:id	Delete question by id

CODE 24. A hook function to calculate the time left

```
const useCountdown = (targetDate: Date) => {
  const countDownDate = new Date(targetDate).getTime();
  const countDownTime = countDownDate - new Date().getTime();
  const [countDown, setCountDown] = useState(countDownTime);

  useEffect(() => {
    const time = new Date(targetDate).getTime() - new Date().getTime();
    setCountDown(time);
    const interval = setInterval(() => {
      const time1 = new Date(targetDate).getTime() - new Date().getTime();
      if (time1 <= 0) {
        clearInterval(interval);
      }
      setCountDown(time1);
    }, 1000);
    return () => clearInterval(interval);
  }, [targetDate]);
  return getReturnValues(countDown);
};
const getReturnValues = (countDown: number) => {
  // calculate time left …
  return [days, hours, minutes, seconds];
};
```

CODE 25. Using countdown Hook inside a component

```
const CoundownView = ({
  hour,
  minutes,
  seconds,
}: {
  hour: number;
  minutes: number;
  seconds: number;
}) => {
  // calculate display time …
  return (
    <Text>
      {translate('common.time_left')} {displayedTime}
    </Text>
  );
};

const ExamControl = (props: ExamControlProps) => {
  const {
    countdownDate,
  } = props;
  const [, , minutes, seconds] = useCountdown(countdownDate);
```

```
  return (
    <CoundownView hour={0} minutes={minutes} seconds={seconds} />
  );
};
```

CODE 26. Sample for Single Responsibility in React Hook

```
// File useDarkode.ts
import {useEffect, useState} from 'react';
import AsyncStorage from '@react-native-async-storage/async-storage';

type ThemeType = 'light' | 'dark';

export const useDarkMode = () => {
  const [theme, setTheme] = useState<ThemeType>('light');

  const setMode = async (mode: ThemeType) => {
    await AsyncStorage.setItem('theme', mode);
    setTheme(mode);
  };

  const themeToggler = async () => {
    theme === 'light' ? await setMode('dark') : await setMode('light');
  };

  useEffect(() => {
    const loadTheme = async () => {
      try {
        const localTheme = (await AsyncStorage.getItem('theme')) as ThemeType;
        if (localTheme !== null) {
          setTheme(localTheme);
        }
      } catch (e) {
        setMode('light');
      }
    };
    loadTheme();
  }, []);

  return [theme, themeToggler];
};
```

CODE 27. Sample for Interface Segregation principle

```
interface IUserEmail {
  email: string;
  submitEmail: (email: string) => Promise<boolean>;
```

```
}

interface IUserPhone {
  phone: string;
  submitPhone: (phone: string) => Promise<boolean>;
}

interface IUserSaveLogin {
  checked: boolean;
  toggle: () => void;
}

interface UserProfileViewProps extends IUserEmail, IUserPhone, IUserSaveLogin {
  signOut: () => void;
}

const UserEmail = (props: IUserEmail) => {
  return <Text>{props.email}</Text>;
};

const UserPhone = (props: IUserPhone) => {
  return <Text>{props.phone}</Text>;
};

export const UserProfileView = (props: UserProfileViewProps) => {
  const emailProps: IUserEmail = {
    email: props.email,
    submitEmail: props.submitEmail,
  };

  return (
    <>
      <UserEmail {...emailProps} />

      <UserPhone {...props} />
    </>
  );
};
```

CODE 28. Text To Speech interface

```
export interface ITextToSpeech {
  language: string;
  isSupport: boolean;
  processCallback?: TextToSpeechCallback;
  endCallback?: TextToSpeechCallback;
  init: (process?: TextToSpeechCallback, end?: TextToSpeechCallback) => void;
  setDefaultLanguage: (lang: string) => void;
  speak: (text: string) => void;
  cleanup: () => void;
```

```
}
```

CODE 29. Speaker implementation with Tts module

```
// File speaker.ts using for React Native
import Tts from 'react-native-tts';
// More imports …
export class Speaker implements ITextToSpeech {
  processCallback?: TextToSpeechCallback | undefined = undefined;
  endCallback?: TextToSpeechCallback | undefined = undefined;
  language: string = 'en-US';
  isSupport: boolean = CONFIG.supportTextToSpeech;

  init = (process?: TextToSpeechCallback, end?: TextToSpeechCallback) => {
    this.processCallback = process;
    this.endCallback = end;
    // Implementation …

  };

  // Other Implementation …
}
const speaker = new Speaker();
speaker.setDefaultLanguage(solveTtsLanguage());
export default speaker;
```

CODE 30. Speaker implementation with ‘speechSynthesis’

```
// File speaker.web.ts
// More imports …
const synth = window.speechSynthesis;
export class Speaker implements ITextToSpeech {
  processCallback?: TextToSpeechCallback | undefined = undefined;
  endCallback?: TextToSpeechCallback | undefined = undefined;
  language: string = 'en-US';
  isSupport: boolean = CONFIG.supportTextToSpeech && synth !== undefined;

  // Other Implementation …

  speak = (text: string) => {
    const utterance = new SpeechSynthesisUtterance(text);
    // Implementation …
    synth.speak(utterance);
  };
}
const speaker = new Speaker();
speaker.setDefaultLanguage(solveTtsLanguage());
export default speaker;
```

CODE 31. Use speaker in a component

```
export interface RefQuestionView {
  isBusy: boolean;
  startFirstAudio: () => void;
  cancelAudio: () => void;
}

const QuestionView = React.forwardRef<RefQuestionView, QuestionViewProp>(
  (props, ref) => {
    // Other code…
    const cancelAudio = useCallback(() => {
      speaker.cleanup();
    }, []);

    React.useImperativeHandle(ref, () => {
      return {
        isBusy: stage !== 'final',
        startFirstAudio,
        cancelAudio,
      };
    });

    const handlePlayAudio = useCallback(async () => {
      speaker.speak(initLabel);
    }, [initLabel]);

    // Handle displayLabel match with Audio
    const handleTtsProcess = useCallback(
      (charIndex: number, charLength: number) => {
        if (stage !== 'final') {
          const text = initLabel.substring(0, charIndex + charLength);
          setDisplayLabel(text);
        }
      },
      [initLabel, stage],
    );

    const handleTtsEnd = useCallback(() => {
      setStage('final');
      setDisplayLabel(initLabel);
      onPlayAudioFinish && onPlayAudioFinish();
    }, [initLabel, onPlayAudioFinish]);

    // Effect
    useEffect(() => {
      switch (stage) {
        case 'waiting':
          setDisplayLabel('');
          break;
```

```
        case 'init':
          speaker.init(handleTtsProcess, handleTtsEnd);
          handlePlayAudio();
          setStage('playing');
          break;
        case 'playing':
          break;
        case 'final':
          setDisplayLabel(initLabel);
          break;
      }
    }, [handlePlayAudio, handleTtsEnd, handleTtsProcess, initLabel, stage]);
    // Other code…
}
```

CODE 32. Reset state of QuestionView with key

```
const Exam = (props: RootParamsScreenProps) => {
  // Other implementation...
  const [questionNo, setQuestioNo] = useState(0);
  const {
    data: question,
    isLoading: isLoadingQuestion,
    error: errorQuestion,
    refetch: refetchQuestion,
  } = useQuestion(questionIds[questionNo]);
  return (
    <ScrollView>
      {question && (
        <FlexView>
          // Other implementation...
          <View _flex={3}>
            <QuestionView
              key={question.id}
              ref={refMainQuestion}
              isMain
              initQuestionNo={question.id}
              initLabel={solveLanguage(question.main_question)}
              initStage={questionStages[questionNo][0]}
              onPlayAudioFinish={() => handlePlayAudioFinish(0)}
            />
          </View>
        </FlexView>
      )}
    </ScrollView>
  );
};
```

CODE 33. Run script for React Native Web in ‘package.json’ file

```
"scripts": {
    // Other scripts
    "web": "NODE_ENV=development webpack-dev-server --mode development",
    "build:web": "webpack",
},
```

CODE 34. File index.web.js

```
import {AppRegistry} from 'react-native';
import App from './src/App';
import packageInfo from './app.json';

// Create stylesheet
// Another config …
AppRegistry.registerComponent(packageInfo.name, () => App);
AppRegistry.runApplication(packageInfo.name, {
  rootTag: document.getElementById('react-app'),
});

if (module.hot) {
  module.hot.accept();
}
```

CODE 35. Resolving alias in webpack config

```
resolve: {
    alias: {
      'react-native': 'react-native-web',
      'react-native-linear-gradient': 'react-native-web-linear-gradient',
      'lottie-react-native': 'react-native-web-lottie',
    },
    extensions: [
      '.web.tsx', '.tsx',
      '.web.ts',
      '.ts',
      '.web.jsx',
      '.jsx',
      '.web.js',
      '.js',
      '.css',
      '.json',
    ],
    mainFields: ['browser', 'main'],
  },
```

CODE 36. Install Pods dependencies for building iOS project

```
// Cleanup Pods folder if needed
remove -rf ios/Pods

// Install pods
npx pod install ios
```

CODE 37. Update tsconfig.json file

```
{
  "extends": "@tsconfig/react-native/tsconfig.json"
}
```

CODE 38. Props with TypeScript

```
export interface ITest {
  id: string;
  initDate?: string;
  updateDate?: string;
  score?: number;
  questions: number[];
}
export type AnswerType = boolean | undefined;
export type TestMode = 'exam' | 'practice';
export type QuestionStage = 'waiting' | 'init' | 'playing' | 'final';
// Other types…

// Use in component
const initialAnswers = Array<AnswerType>(4).fill(undefined);
const initialStages = Array<QuestionStage>(5).fill(
  speaker.isSupport ? 'waiting' : 'final',
);

const [stagesMap, setStagesMap] = useState({[questionNo]: initialStages});
const [answersMap, setAnswersMap] = useState({[questionNo]: initialAnswers});
```

CODE 39. Install styled-components packages with yarn

```
yarn add styled-components
yarn add -D babel-plugin-styled-components
```

CODE 40. Styles, spacing definition

```
export const palette = {
  primary: '#7b1e4d',
  primaryShadow: '#ff9acd',
  secondary: '#9c2243',
  // Other colors …
};
const spacing = {
  xxs: 4,
  xs: 8,
  s: 12,
  m: 16,
  ml: 24,
  l: 28,
  xl: 40,
  xxl: 60,
};

const textSize = {
  xxs: Platform.OS === 'web' ? 'xx-small' : '8px',
  xs: Platform.OS === 'web' ? 'x-small' : '10px',
  s: Platform.OS === 'web' ? 'small' : '12px',
  m: Platform.OS === 'web' ? 'medium' : '14px',
  // Other size …
};

// Sample use
const testButtonStyle = {margin: spacing['m'], padding: spacing['s']};
```

CODE 41. Dark/light theme definition

```
export const lightTheme = {
  darkMode: false,
  spacing,
  textSize,
  colors: {
    primary: palette.primary,
    primaryShadow: palette.primaryShadow,
    secondary: palette.secondary,
    error: palette.red,
    uptrend: palette.green,
    downtrend: palette.red100,
    // Other colors …
    safeArea: palette.white,
  },
};
export const darkTheme = {
  darkMode: true,
  spacing,
```

```
  textSize,
  colors: {
    ...lightTheme.colors,
    value: palette.white100, // Customize list colors …
    background: palette.dark,
    // Customize colors
  },
};
```

CODE 42. Based styled-components view

```
export const View = styled(BasedView)<IViewStyle>`
  ${props => {
    if (props._gap) {
      return css`
        gap: ${props.theme.spacing[props._gap]}px;
      `;
    }
  }}
  // Other customize options …
  ${props => props._fullWidth && 'width: 100%;'}
  ${props => props._flex && `flex: ${props._flex};`};
  ${props =>
    props._backgroundColor && `background-color: ${props._backgroundColor}`};
`;

export const BottomView = styled(View)`
  position: absolute;
  bottom: 0px;
  left: 0px;
  right: 0px;
`;
```

CODE 43. Styled components in used

```
export const ExamHeader = (props: ExamHeaderProps) => {
  const {testNo, onAbort} = props;
  return (
    <HStack
      _marginH="s"
      _marginV="xxs"
      _justifyContent="space-between"
      _alignItems="center">
      <Text _bold _fontSize="xl">
        {translate('common.exam_no')} {testNo}
      </Text>
      <Spacer />
      <Button
        label={translate('common.abort')}
```

```
        onPress={() => onAbort && onAbort()}
      />
    </HStack>
  );
};

export default ExamHeader;
```

CODE 44. Install i18next packages

```
yarn add i18next react-i18next

// Using in React components
import {useTranslation} from 'react-i18next';
const {t} = useTranslation();
```

CODE 45. I18N Configuration & use

```
import i18n, {t} from 'i18next';
import {initReactI18next} from 'react-i18next';
import en from './resources/en.json';
import vi from './resources/vi.json';
import dk from './resources/dk.json';
import {pickedLanguage} from '../../ui/pages/Home/common';

const resources = {
  en: {
    translation: en,
  },
  vi: {
    translation: vi,
  },
  dk: {
    translation: dk,
  },
};

i18n.use(initReactI18next).init({
  resources,
  lng: pickedLanguage,
  fallbackLng: 'en',
  compatibilityJSON: 'v3',
  interpolation: {
    escapeValue: false,
  },
});
```

```
export const translate = (
  key: string,
  params?: {[key: string | number]: string},
): string => {
  return t(key, params);
};

// translate function in use
const testString = translate('common.feature_not_available_yet');
```

CODE 46. Question interface

```
export interface ILang {
  en?: string;
  vi?: string;
  dk?: string;
}
export interface IQuestionOptions {
  true: ILang;
  false: ILang;
}
export interface ISubQuestion {
  sub_question: ILang;
  sub_hint: ILang;
  sub_anw_option: number;
}
export interface IQuestion {
  id: number;
  image_question: string;

  main_question: ILang;
  main_audio?: ILang;

  sub_questions: ISubQuestion[];
}
export interface IQuestionsStatistic {
  totalQuestion: number;
  totalTests: number;
}
```

CODE 47. Generated questions result from using ChatGPT

```
"questions": [
    {
      "id": 0,
      "image_question": "",
      "main_question": {
```

```
        "dk": "Du holder stille ved kantstenen og vil gerne sette igang. Hvad vil du
gore?",
        "vi": "Bạn đứng yên ở lề đường và muốn bắt đầu. Bạn muốn làm gì?",
        "en": "You stand still at the curb and want to get started. What do you want to
do?"
      },
      "sub_questions": [
        {
          "sub_question": {
            "dk": "Jeg satter forsigtig igang med det samme?",
            "vi": "Tôi bắt đầu thận trọng ngay lập tức?",
            "en": "I start cautiously right away?"
          },
          "sub_hint": {
            "dk": "Nej, du skal kigge bagud til venstre for at tjekke din blinde vinkel.",
            "vi": "Không, bạn phải nhìn lại bên trái để kiểm tra điểm mù.",
            "en": "No, you have to look back to the left to check your blind spot."
          },
          "sub_anw_option": 0
        },
        {
          "sub_question": {
            "dk": "Jeg kigger bagud til venstre?",
            "vi": "Tôi đang nhìn lại bên trái?",
            "en": "I'm looking back left?"
          },
          "sub_anw_option": 1,
          "sub_hint": {
            "dk": "Ja.",
            "vi": "Đúng.",
            "en": "Yes."
          }
        },
        {
          "sub_question": {
            "dk": "Hvis der er frit bagude, setter jeg i gang?",
            "vi": "Nếu có thời gian rảnh phía sau, tôi có bắt đầu không?",
            "en": "If there is free time behind, do I start?"
          },
          "sub_anw_option": 1,
          "sub_hint": {
            "dk": "Ja.",
            "vi": "Đúng.",
            "en": "Yes."
          }
        }
      ]
    },
    // Other questions
  }
]
```

CODE 48. Install React-query

```
yarn add @tanstack/react-query

// Using @tanstack/react-query
import {QueryClient, QueryClientProvider} from '@tanstack/react-query';
<QueryClientProvider client={queryClient}>
      <ThemeProvider theme={isDarkMode ? darkTheme : lightTheme}>
        <SafeAreaProvider>
          <ToastProvider>
            <MainComponent />
          </ToastProvider>
        </SafeAreaProvider>
      </ThemeProvider>
    </QueryClientProvider>
```

CODE 49. Using QueryClientProvider from React Query

```
import {QueryClient, QueryClientProvider} from '@tanstack/react-query';
// Other imports
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      staleTime: CONFIG.staleTime,
      refetchOnMount: true,
    },
  },
});
const App = () => {
  const [theme] = useDarkMode();
  const themeMode = theme === 'light' ? lightTheme : darkTheme;
  return (
    <QueryClientProvider client={queryClient}>
      <ThemeProvider theme={themeMode}>
        <SafeAreaProvider>
          <NavigationContainer>
            <AppRouter/>
          </NavigationContainer>
        </SafeAreaProvider>
      </ThemeProvider>
    </QueryClientProvider>
  );
};
```

CODE 50. React Query in use

```
export const useFreeTests = () =>
  useQuery({
    queryKey: ['free-tests'],
    queryFn: fetchFreeTests,
  });

export const useTest = (id: string) =>
  useQuery({
    queryKey: ['test', id],
    queryFn: () => fetchTest(id),
  });

export const useAllTests = () =>
  useInfiniteQuery({
    initialPageParam: 1,
    queryKey: ['all-tests'],
    queryFn: ({pageParam = 1}) => fetchTestsPage(pageParam),

    getNextPageParam: (currentPage: ITest[], allPages: ITest[][]) => {
      const nextPage =
        currentPage.length === CONFIG.fetchItemsPerPage
          ? allPages.length + 1
          : undefined;
      return nextPage;
    },
  });
```

CODE 51. Custom error boundary component with TS

```
import React, {Component, ErrorInfo, ReactNode} from 'react';
import {Text} from './styled-components/text';
import {translate} from '../../common/i18n';

interface Props {
  children?: ReactNode;
}

interface State {
  hasError: boolean;
  error?: Error;
}

class ErrorBoundary extends Component<Props, State> {
  public state: State = {
    hasError: false,
  };
  public static getDerivedStateFromError(error: Error): State {
    return {hasError: true, error};
```

```
  }
  public componentDidCatch(error: Error, errorInfo: ErrorInfo) {
    console.error('Uncaught error:', error, errorInfo);
  }
  public render() {
    if (this.state.hasError && this.state.error) {
      return (
        <Text>
          {translate('Sorry! There was an error: {{error}}', {
            error: this.state.error.message,
          })}
        </Text>
      );
    }
    return this.props.children;
  }
}

export default ErrorBoundary;
```

CODE 52. Client store type definition

```
export interface ClientState {
  isAuthenticated: boolean;
  authToken?: string;
  isPermiumAccount: boolean;

  setting: ISetting;

  answers: IAnswer[];
  studyRecord: IStudyRecord[];
}

export interface SettingAction {
  toggleDarkMode: () => void;
  setPreLanguage: (lang: SupportedLanguage) => void;
  toogleTTS: () => void;
  setTTSRate?: (rate: number) => void;
  setTTSPitch?: (pitch: number) => void;
  setTTSVoice?: (voice: string) => void;
  setTTSIgnoreSilentSwitch: (type: IgnoreSilentSwitchType) => void;
}

export interface ClientAction {
  signIn: () => void;
  signOut: () => void;
  // Other actions…
  updateAnswers: (answers: IAnswer[]) => void;
  addStudyRecords: (records: IStudyRecord[]) => void;
```

```
}
```

CODE 53. Init Zustand data and actions

```
const initialSetting: ISetting = {
  pickedLanguage: 'vi',
  ttsConfig: {
    isEnable: true,
    rate: 1,
    pitch: 0.5,
    ignoreSilentSwitch: 'inherit',
  },
  isDarkMode: false,
};

const initialClient: ClientState = {
  isAuthenticated: true,
  isPermiumAccount: true,
  setting: initialSetting,
  answers: [],
  studyRecord: [],
};

export const initializer: StateCreator<
  ClientState & ClientAction & SettingAction,
  [['zustand/persist', unknown], ['zustand/immer', never]],
  []
> = (set, get) => ({
  ...initialClient,
  // Other actions…
  toggleDarkMode: () => {
    set(state => {
      state.setting.isDarkMode = !get().setting.isDarkMode;
    });
  },
  setPreLanguage: (lang: SupportedLanguage) => {
    set(state => {
      state.setting.pickedLanguage = lang;
      i18next.changeLanguage(lang);
    });
  },
});
```

CODE 54. Create useStore with Zustand

```
import {StoreApi, UseBoundStore} from 'zustand';
type WithSelectors<S> = S extends {getState: () => infer T}
  ? S & {use: {[K in keyof T]: () => T[K]}}
```

```
  : never;

export const createSelectors = <S extends UseBoundStore<StoreApi<object>>>(
  _store: S,
) => {
  let store = _store as WithSelectors<typeof _store>;
  store.use = {};
  for (let k of Object.keys(store.getState())) {
    (store.use as any)[k] = () => store(s => s[k as keyof typeof s]);
  }
  return store;
};
const useStoreBase = create<ClientState & ClientAction & SettingAction>()(
  devtools(
    persist(immer(initializer), {
      name: 'driver-license-dk',
      storage: createJSONStorage(() => AsyncStorage),
    }),
  ),
);

const useStore = createSelectors(useStoreBase);
```

CODE 55. Install json-server tool

```
yarn add json-server -D
```

CODE 56. Sample json data to run with json-server

```
{
  "notifications": [
    …
  ],
  "questions": [
    …
  ],
  "tests": [
    …
  ],
  …
  "questions-statistic": {"totalQuestion": 1000, "totalTests": 40}
]
```

CODE 57. Run json-server from default port 3000

```
json-server db.json -static ./public -delay 1000
```

CODE 58. Simple web server with JS, Node.js

```
const hostname = '127.0.0.1';
const port = 3000;
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});
server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

CODE 59. Run Node application with environment production

```
NODE_ENV=production node index.js
```

CODE 60. File .env

```
# .env
NODE_ENV='development'
PORT=3001

 # Auth
COOKIE_SECRET='some_secret_key'
AUTH_SECRET='some_secret_key'
```

CODE 61. Load modules in Node.js

```
import 'dotenv/config';
import express from 'express';
import cors from 'cors';
import path from 'path';

// Or with require() function
const path = require('path');
```

CODE 62. Use fs.stat() to get file metadata

```
try {
  const stats = await fs.stat('/Users/ducminh/test.txt');
  stats.isFile(); // true
  stats.isSymbolicLink(); // false
```

```
  stats.size; // 1024000 //= 1MB
} catch (err) {
  console.log(err);
}
```

CODE 63. Track file changes with watch function in module ‘fs’

```
fs.watchFile(
  'FILEPATH',
  {
    persistent: true,
  },
  function (data) {
    // Notify changes with socket io
    io.emit('server', { message: 'FileChanged' });
  },
);
```

CODE 64. Simple Node.js application with Express

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

app.get('/', function (req, res, next) {
  next();
})

app.listen(3000);
```

CODE 65. Sample of use express static

```
const options = {
  dotfiles: 'ignore',
  etag: false,
  extensions: ['htm', 'html'],
  index: false,
  maxAge: '1d',
  redirect: false,
  setHeaders(res, path, stat) {
    res.set('x-timestamp', Date.now());
  },
};

app.use(express.static('public', options));
```

CODE 66. Express Router in use

```
// src/routes/v1/tests.ts
import express from 'express';
import authJwt from '../../middlewares/authAccount';
import controller from '../../controllers/test.controller';
const router = express.Router();

router.get('/', [authJwt.verifyToken], controller.getTest);
router.get('/free', controller.getFreeTest);
export default router;

// src/routes/v1/tests.ts
import express from 'express';
import test from './test';

const router = express.Router();
router.use('/tests', tests);
export default router;
```

CODE 67. Sample express middleware

```
import { type RateLimitRequestHandler, rateLimit } from 'express-rate-limit';

const limiter = (limit: number, minutes?: number): RateLimitRequestHandler => {
  const minutesX = minutes ?? 15;
  const limitX = process.env.NODE_ENV === 'development' ? limit * 10 : limit;

  return rateLimit({
    windowMs: minutesX * 60 * 1000,
    limit: limitX,
    message: {
      message: `You have exceeded your ${limitX} request(s) per ${minutesX} minute(s)
limit.`,
      key: 'error.too_many_requests',
    },
    standardHeaders: 'draft-7', // draft-6: `RateLimit-*` headers; draft-7: combined
`RateLimit` header
    legacyHeaders: false, // Disable the `X-RateLimit-*` headers.
  });
};
export default limiter;

// Use limiter middleware in application-level
app.use(limiter(5, 1)); // Rate limit 5 request/minute
// Use limiter middleware in router-level
router.post('/resendVerify', [limiter(1, 15)], controller.resendVerifyAccount);
```

CODE 68. Implement User scheme using mongoose

```
import mongoose, { type Types, type Document } from 'mongoose';
import validator from 'validator';
import { type IRole } from './role.model';

export interface IUser extends Document {
  username: string;
  email: string;
  password: string;
  phone: string;
  roles: Types.ObjectId[];
  isVerified: boolean;
}

export interface IPopulatedUser extends Omit<IUser, 'roles'> {
  roles: IRole[];
}

const userScheme = new mongoose.Schema({
  username: {
    type: String,
    minLength: 4,
    required: true,
  },
  email: {
    type: String,
    required: true,
    validate: [validator.isEmail, 'invalid email'],
  },
  isVerified: {
    type: Boolean,
    default: false,
  },
  phone: {
    type: String,
    required: true,
    validate: {
      validator: (v: string) => {
        return validator.isMobilePhone(v, 'da-DK', { strictMode: false });
      },
      message: (props: { value: any }) =>
        `${props.value} is not a valid Danish phone number`,
    },
  },
  password: {
    type: String,
    required: true,
    validate: [validator.isStrongPassword, 'invalid strong password'],
  },
  roles: [
    {
```

```
      type: mongoose.Schema.Types.ObjectId,
      required: true,
      ref: 'Role',
    },
  ],
});

userScheme.set('toJSON', {
  transform: (doc, returnObject) => {
    delete returnObject._id;
    delete returnObject.__v;
    delete returnObject.password;
  },
});

const User = mongoose.model<IUser>('User', userScheme);
export default User;
```

CODE 69. Create new user via User scheme in mongoose

```
const user = new User({
  username: req.body.username,
  email: req.body.email,
  phone: req.body.phone,
  password: req.body.password,
});

user.password = bcrypt.hashSync(user.password);

if (req.body.roles !== null) {
  const roles = await Role.find({ name: { $in: req.body.roles } });
  if (roles !== null) {
    user.roles = roles.map((role: IRole) => role._id);
    await user.save();
  } else {
    throw new ApiError('invalid_role');
  }
} else {
  const role = await Role.findOne({ name: 'user' });
  if (role !== null) {
    user.roles = [role?._id];
    await user.save();
  } else {
    throw new ApiError('invalid_user_role');
  }
}
```

CODE 70. Connect Redis via redis package

```
import * as redis from 'redis';
import redisConfig from './config/redis.config';

const client = redis.createClient({
  username: redisConfig.username,
  password: redisConfig.pwd,
  socket: {
    host: redisConfig.HOST,
    port: redisConfig.PORT,
  },
});

client.on('error', (err) => {
  console.log('Redis Client Error', err);
});

export const connectRedis = async (): Promise<void> => {
  await client.connect();
};

export default client;
```

CODE 71. Verify JWT accessToken middleware with redis and ‘jwt’

```
const verifyToken = async (
  req: Request,
  res: Response,
  next: NextFunction,
): Promise<void> => {
  const token = req.headers.authorization;

  if (token == null) {
    const error = new ApiError('token_missing', 403);
    res
      .status(error.status)
      .send({ message: error.message, key: `error.${error.key}` });
    return;
  }

  const realToken = token.replace(/^Bearer\s/, '');

  try {
    const redisCheck = await redisClient.get(realToken);
    if (redisCheck == null) {
      const error = new ApiError('invalid_auth_token', 401);
      res
        .status(error.status)
        .send({ message: error.message, key: `error.${error.key}` });
```

```
      return;
    }

    const decoded = jwt.verify(realToken, authConfig.secret) as jwt.JwtPayload;
    if (decoded === null) {
      const error = new ApiError('invalid_auth_token', 401);
      res
        .status(error.status)
        .send({ message: error.message, key: `error.${error.key}` });
      await redisClient.del(realToken);
      await Token.findOneAndRemove({ token: realToken });
      return;
    }

    (req as CustomRequest).userId = decoded.id;
    next();
  } catch (error) {
    await redisClient.del(realToken);
    await Token.findOneAndRemove({ token: realToken });

    if (error instanceof jwt.TokenExpiredError) {
      const err = new ApiError('token_expired', 401);
      res.status(err.status).send({
        message: err.message,
        key: `error.${err.key}`,
      });
      return;
    }

    const err = new ApiError('undefined', 500);
    res.status(err.status).send({
      message: error instanceof Error ? error.message : err.message,
      key: `error.${err.key}`,
    });
  }
};
```

CODE 72. APIs version maintenance

```
// Routers
app.get('/', (_request, response) => {
  response.send('<h1>It works!</h1>');
});

app.use('/api/v1', apiV1);
app.use('/api/v2', apiV2);

app.use(errorHandler);
```

CODE 73. Config Nodemon.json

```
{
  "watch": ["src"],
  "ext": ".ts,.js",
  "ignore": [],
  "exec": "npx ts-node ./src/index.ts"
}
```

CODE 74. CI/CD Workflow for Node.js and Webpack

```
name: CI

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

  workflow_dispatch:

jobs:
  build:
    runs-on: ubuntu-latest

    if: github.ref == 'refs/heads/main'

    strategy:
      matrix:
        node-version: [14.x]

    steps:
      - uses: actions/checkout@v2

      - run: npm ci
      - run: npm run build

      - name: rsync deployments
        uses: burnett01/rsync-deployments@6.0.0
        with:
          switches: -crlt --delete --exclude='/*/.log' --omit-dir-times
          path: ./*
          remote_path: ${{ secrets.DEPLOY_PATH }}
          remote_host: ${{ secrets.DEPLOY_HOST }}
          remote_user: ${{ secrets.DEPLOY_USER }}
          remote_key: ${{ secrets.DEPLOY_KEY }}
```

CODE 75. Sign up new account function

```
const signup = asyncHandler(
  async (req: Request, res: Response, next: NextFunction) => {
    try {
      if (req.body.password === undefined) {
        throw new ApiError('password_missing', 400);
      }

      const pwdScore = validator.isStrongPassword(req.body.password, {
        returnScore: true,
      });

      if (pwdScore < 50) {
        throw new ApiError('weak_password');
      }

      const user = new User({
        username: req.body.username,
        email: req.body.email,
        phone: req.body.phone,
        password: req.body.password,
      });

      user.password = bcrypt.hashSync(user.password);

      if (req.body.roles !== null) {
        const roles = await Role.find({ name: { $in: req.body.roles } });
        if (roles !== null) {
          user.roles = roles.map((role: IRole) => role._id);
          await user.save();
        } else {
          throw new ApiError('invalid_role');
        }
      } else {
        const role = await Role.findOne({ name: 'user' });
        if (role !== null) {
          user.roles = [role?._id];
          await user.save();
        } else {
          throw new ApiError('invalid_user_role');
        }
      }

      const otp = await Token.createOTP(user._id);
      const link =
        'http://' +
        req.headers.host +
        '/api/v1/auth/verifyAccount?token=' +
        otp;
```

```
      await sendRequestActiveAccountMail(user.email, user.username, link);
      res.send({ message: 'User was registered successfully' });
    } catch (error) {
      next(error);
    }
  },
);
```