Front-End Implementation Documentation

1. Introduction

In this section, we will explore the front-end implementation of our project, designed to connect learners and mentors effectively.

Our front-end is developed using React.js, leveraging its component-based architecture to create a dynamic and interactive user interface. We will delve into the project's structure, design patterns, state management and routing. Additionally, we'll discuss the styling approach, API integration, performance optimization, deployment strategies used to bring this application to life and Challenges During Development.

2. Technology Stack

In developing the front-end of our project, a diverse range of modern technologies and libraries were employed to ensure a robust, interactive, and high-performing application. Each tool and library was chosen to fulfil specific aspects of the development process. Below is a detailed overview of the technology stack used:

2.1 Core Development Tools:

- **React.js**: React.js serves as the foundation of our front-end development. This JavaScript library is essential for building user interfaces, providing a component-based structure that allows for the creation of reusable and maintainable UI components.

- **Redux Toolkit**: We use Redux Toolkit for efficient state management. It simplifies the configuration and usage of Redux with tools like createSlice and createAsyncThunk, facilitating the handling of global state across the application.

- **React Router Dom**: This library is crucial for managing the navigation and routing within our application. React Router Dom plays a key role in developing Single Page Applications (SPA), where the app loads a single

HTML page and dynamically updates the content as users interact with it. By defining routes for various components and views, React Router Dom allows seamless transitions between different parts of the app without reloading the page. This enhances the user experience by providing fast, responsive navigation and maintaining the state of the application across different sections. It also supports route-based access control, enabling us to define routes specific to different user roles or conditions, such as guests, authenticated users, or users who need to complete initial setup steps.

2.2 User Interface and Styling:

- **Material-UI**: To streamline UI development, Material-UI provides a collection of pre-designed components and styling solutions based on Material Design principles. This helps in quickly developing consistent and responsive interfaces.

- **CSS Modules**: CSS Modules ensure that styles are scoped locally to components, preventing conflicts and making it easier to maintain the styles for each component separately.

- **React Big Calendar**: Used for displaying and managing events, React Big Calendar provides a comprehensive calendar interface that supports multiple views like day, week, and month, ideal for scheduling and organizing tasks.

- **React Draggable**: This library adds drag-and-drop functionality to our components, enhancing the user experience by allowing intuitive interactions through draggable elements.

- **Swiper**: Swiper is utilized for creating engaging sliders and carousels. It supports touch and mouse interactions, offering a modern and responsive way to display content in a slider format.

- **Notistack**: This library is used for displaying notifications. It allows for stacking and positioning of notifications, providing users with real-time feedback and alerts.

2.3 Form Management and Validation:

- **Formik and Yup**: These libraries work together to manage form state and validate data. Formik handles form creation, state management, and submission, while Yup provides schema-based validation to ensure data integrity.

- **React Images Uploading**: Facilitates the process of uploading and managing images. It provides a user-friendly interface for users to upload, preview, and handle image files efficiently.

2.4 Real-Time Communication and Interaction:

- **Socket.io Client**: Essential for implementing real-time features like chat and notifications, Socket.io Client enables bidirectional communication between the client and server, ensuring that updates and interactions are instantaneous.

- **Simple Peer**: For enabling voice and video calls, Simple Peer simplifies the implementation of WebRTC-based peer-to-peer connections, allowing users to communicate directly without the need for server intermediaries for media streams.

- **Firebase**: Firebase Cloud Messaging (FCM) is integrated to handle push notifications.

2.5 Additional Libraries:

- **Moment.js**: We use Moment.js for parsing, validating, manipulating, and displaying dates and times. It supports various formats and locales, simplifying date and time management within the app.

- **React Copy to Clipboard**: This library simplifies the action of copying text to the clipboard, making it easy for users to share links, codes, or any textual content with a single click.

2.6 Deployment:

- **Vercel**: Our choice for deploying the front-end application, Vercel offers seamless integration with GitHub for continuous deployment. It provides scalable hosting solutions and is optimized for performance and reliability.

3. Project Structure

The project follows a well-organized directory structure to maintain modularity and clarity. Below is a snapshot of the key folders and their roles:

- **src/**: Root directory for source files.
  - **assets/**: Contains static assets like images, icons, audios and static validation messages.
  - **callStore/**: Manages state related to call functionalities and all voice/video call functionality.
  - **components/**: Houses reusable components used across the application.

    Barrel Design Pattern: To maintain clean and manageable imports, the components directory utilizes the barrel pattern. This means each subfolder has an index.js file that re-exports its modules. This allows for more concise import statements, like:

```
// Instead of
import Card from './components/Card';

import Chip from './components/Chip';
import PopUpCard from './components/PopUpCard

// We use
import { Card, Chip, PopUpCard } from './components/Ut';
```

  - **FCM/**: Firebase Cloud Messaging integration.
  - **helpers/**: Utility functions and helpers.
  - **hooks/**: Custom hooks for various functionalities.
  - **imagesStore/**: A repository for image files.

- **pages/**: Main views and pages of the application.
- **routes/**: Manages the routing setup for the application.
- **sockets/**: Handles main socket connection.
- **store/**: Global state management setup using Redux.
- App.css: CSS styling file for manage app styling and colours, font sizes, z-index, and border-radius variables
- App.js: Wrap routes components and general layout and handle general logic.
- config.js: Manage API endpoints modules variables.
- index.css: Global CSS styling.
- index.js: Render App and manage providers components.

3.1 Page-Based Structure

In our project, we follow a **Page-Based Structure** to organize the application. Each major view or page is encapsulated in its own directory under the pages/ folder. This structure simplifies development and maintenance by grouping related components, styles, and logic together.

For example:

- **src/pages/login/**: Folder Contains all files related to the login page.
  - **Login.jsx**: The container component manages state, logic and login API.
  - **LoginUi.jsx**: The Presentational component manages UI of the login page.
  - **LoginInputsData.js**: Data about the login form inputs.
  - **LoginValidationSchema.js**: Validation schema for the login form using Yup.
  - **Login.module.css**: Styles specific to the login page.

  Note: The Login page follows the **Presentational and Container** design pattern, which separates UI components from logic handlers. This pattern is discussed in detail in the Architectural Patterns section.

- **src/pages/notes**: Folder Manages the components and logic for the notes functionality.
  - **components/**: Contains components specific to the notes page.
  - **hooks/**: Custom hooks related to notes functionalities.

- **styles/**: CSS modules and styles for the notes page.
- **Notes.jsx**: The main component for the notes page.

Note: The **notes** page follows the Facade pattern, as it aggregates complex functionalities related to notes page into a single component that provides a simplified interface. This pattern is discussed in detail in the Architectural Patterns section.

This approach allows us to manage each page's complexity independently, making it easier to handle routing and state management.

4. Design Patterns

In building the front-end of our application, we leveraged several key design patterns to enhance modularity, maintainability, and clarity. Here's a closer look at the patterns used:

4.1 Presentational and Container Design Pattern

Overview: The Presentational and Container pattern is a design approach that separates the rendering logic (UI) from the business logic (data handling and state management). This division of concerns simplifies both the development and testing of components.

Presentational Components: These components are concerned with how things look. They receive data and callbacks exclusively via props and do not have their own state or side effects.

Example: In the Login page, LoginUi.jsx is a presentational component responsible for rendering the login form UI elements.

Code Example:

```
const LoginUi = (props) =>
{
    const {
        handleLogin,
        isLoadingLogin,
    } = props;

    return (
        <div
```

```
            className='height-100vh  center-y center-x'
        >
            <FormCard
                size="small"
            >
                <div className={`center-x ${classes.logo}`}>
                    <Logo />
                </div>
                <HeaderText>
                    Login
                </HeaderText>
                <Paragraph>
                    Please, Enter your details
                </Paragraph>
                <FormikContainer
                    initialValues={loginInitialValues}
                    validationSchema={loginValidationSchema}
                    onSubmit={handleLogin}
                >
                    <LoopOnInputs
                        inputs={loginInputs}
                        disabled={isLoadingLogin}
                    />
                    <div
                        className={classes.forgetPasswordContainer}
                    >
                        <Link
                            to="/forgot-password"
                        >
                            Forgot password?
                        </Link>
                    </div>
                    <Btn
                        type="submit"
                        isLoading={isLoadingLogin}
                    >
                        Login
                    </Btn>
                    <Divider className={classes.divider}>Or</Divider>
                    <TextAndLink type="login" />
                </FormikContainer>
            </FormCard>
            <IllustrationSection
                type="login"
                size="big"
            />
        </div>
```

```
    )}
```

Container Components: These components are concerned with how things work. They provide data and behaviour to presentational components, managing the state and handling user interactions. Container components connect to Redux or other state management systems to fetch and manage data.

Example: Login.jsx in the Login page acts as a container component. It manages the form submission logic, handles login API, handles state for the input fields, and passes the necessary data and callbacks to LoginUi.jsx.

Code Example:

```
const Login = () =>
{
    const dispatch = useDispatch();
    const navigate = useNavigate();
    const {
        isLoading: isLoadingLogin,
        sendRequest: login
    } = useHttp();

    const handleLogin = async (values) =>
    {
        const deviceToken = await requestNotificationPermission();

        values.deviceToken = deviceToken;

        const getResponse = ({ message, token, data, profileDetails }) =>
        {
            if (message === "success")
            {
                dispatch(authActions.login({
                    token: token,
                    userData: { ...profileDetails, ...data }
                }))
                navigate("/", { replace: true });
            }
        };
```

```
        await login(
            {
                url: `${userModulePath}/login`,
                method: "post",
                body: values,
            },
            getResponse
        );
    }


    return (
        <LoginUi
            handleLogin={handleLogin}
            isLoadingLogin={isLoadingLogin}
        />
    )
}
```

Benefits:

- **Reusability**: UI components can be reused across different parts of the application without being tightly coupled to specific data or behaviours.
- **Testability**: Separating UI from business logic makes it easier to write unit tests for each component.
- **Clarity**: This separation improves the clarity and readability of the code, making it easier to understand and maintain.

4.2 Facade Design Pattern

In our project, the **Facade Design Pattern** is utilized effectively in the **Notes** component. This pattern simplifies the interaction with complex subsystems by providing a unified and straightforward interface.

Notes Component Example: The **Notes** component is designed to manage various features related to note handling using custom hooks.

Code Example:

```
const Notes = () =>
{
    // reset notes slice state
    useResetNotesSlice();
```

```
    // get notes and store it in the notes store
    const {
        isLoadingGetNotes,
        lastElementRef,
    } = useGetNotes();

// notes list from store
    const notes = useSelector(state => state.notes.notes);
    // pin note
    const { handlePinNote } = usePinNote();
    // move note to trash
    const { handleMoveNoteToTrash } = useMoveNoteToTrash();
    // notes socket listeners
    useNoteListeners();

    return (
        <>
            <NotesMenu />
            <NotesList
                notes={notes}
                lastElementRef={lastElementRef}
                pinNote={handlePinNote}
                moveToTrash={handleMoveNoteToTrash}
            />


            {isLoadingGetNotes && <LoadingCenter />}

            {(!notes.length && !isLoadingGetNotes) && (
                <VectorAndText
                    isBig={true}
                    fullScreen={true}
                    img={noNotesImg}
                    h="No Notes yet"
                    p={
                        <>
                            No notes are available yet. Start creating.
                            <br />
                            your first note.
                        </>
                    }
                />
            )}

            <Outlet />
        </>
```

```
    )
}
```

Here’s how it works:

- **Custom Hooks**:
  - useGetNotes: Manages the fetching and storage of notes in the state.
  - usePinNote: Handles the logic for pinning notes.
  - useMoveNoteToTrash: Manages the operation of moving notes to the trash.
  - useNoteListeners: Sets up listeners for real-time updates and changes in notes.
- **Centralized Management**:
  - The **Notes** component aggregates these hooks and integrates their functionalities. It manages the lifecycle and state of the notes feature internally, providing a clean and simplified interface for rendering the notes list and handling user interactions.
- **Simplified Interaction**:
  - The hooks are used within the **Notes** component to perform operations like fetching notes, pinning, and deleting. The complexities of these operations are hidden, presenting a unified interface to the **NotesList** component and other child components.

By employing the Facade Design Pattern in the **Notes** component, we achieve a streamlined and maintainable approach to managing complex functionality, enhancing the overall architecture of our front-end application.

5. State Management

State management is handled using Redux Toolkit, which simplifies the process of managing and updating the global state. Each major module has its own slice, which includes the state and reducers for actions related to that module. For example:

- **Authentication Slice**: Manages user login data, user main data and token.
- **Notes Slice**: Manages Notes data and sockets related to it.

- **Tasks Slice**: Manages **Tasks** data and sockets related to it.
- Ui Slice: Manages modals and pop menus states.

Code example on Ui Slice:

```
import { createSlice } from '@reduxjs/toolkit';

const initialUiState = {
    isPopMenuOpened: {},
    isModalOpened: {},
}

const uiSlice = createSlice({
    name: 'ui',
    initialState: initialUiState,
    reducers: {
        openPopMenu(state, action)
        {
            state.isPopMenuOpened[action.payload] = true;
        },
        closePopMenu(state, action)
        {
            state.isPopMenuOpened[action.payload] = false;
        },
        openModal(state, action)
        {
            state.isModalOpened[action.payload] = true;
        },
        closeModal(state, action)
        {
            state.isModalOpened[action.payload] = false;
        },
    }
})

export const uiActions = uiSlice.actions

export default uiSlice.reducer;
```

For modules involving real-time communication, we use createAsyncThunk to handle asynchronous actions, particularly for socket events.

Code Example:

```
// sockets:
// add task
export const emitAddTask = createAsyncThunk(
    'tasks/emitAddTask',
    async (payload) =>
    {
        socket.emit('addTask', payload, () => { });
    }
);
// listen to get task
export const listenToGetTask = createAsyncThunk(
    "tasks/listenToGetTask",
    async (_, thunkAPI) =>
    {
        socket.on('getTask', (data) =>
        {
            //  update state
            thunkAPI.dispatch(
                tasksActions.addTask(data.data)
            );
        });
    }
)
```

6. Routing

Routing is managed using react-router-dom. The routing structure is organized in the routes directory, with specific files for different user roles or application states. For instance:

- **Guest.js**: Routes accessible to non-logged-in users.
- **FirstTime.js**: Routes for first time login users.
- **User.js**: Routes for regular authenticated users.
- **Mentor.js**: Routes for m**entors**.
- **Admin.js**: Routes for admin functionalities.
- **IndexRoutes.js**: Wrap all routes files.

This modular routing approach helps in enforcing access control and simplifying the navigation structure.

7. Styling

Styling is primarily done using CSS Modules, ensuring that styles are scoped locally to components and avoiding conflicts. Each page or component directory contains its own CSS module file or a styles folder if multiple style files are needed.

Global styles and theme settings are managed in App.css, which includes variables for colours, font sizes, and common style properties.

8. API Integration

API calls are handled through custom hook useHttp, which abstracts the fetch logic and error handling.

8.1 useHttp custom hook:

```
import { useState, useCallback } from 'react'
import { useSnackbar } from 'notistack'
import { trimObject } from '../helpers/trimObject'
import { useSelector } from 'react-redux/es/hooks/useSelector'
import { backendUrl } from '../config'

const useHttp = () =>
{

    const [isLoading, setIsLoading] = useState(false)
    const { enqueueSnackbar: popMessage } = useSnackbar()
    const token = useSelector((state) => state.auth.token)



    const sendRequest = useCallback(
        async (requestConfig, applyData) =>
        {
            setIsLoading(true)
            const requestData =
                requestConfig?.contentType === 'form-data'
                    ? {
                        method: requestConfig?.method ?
                            requestConfig.method : 'GET',
                        headers: {
                            Authorization: 'Bearer ' + token,
                        },
```

```
                    body: requestConfig?.body,
                }
                : {
                    method: requestConfig?.method ?
                        requestConfig.method : 'GET',
                    headers: {
                        Authorization: 'Bearer ' + token,
                        'Content-Type': 'application/json',
                    },
                    body: requestConfig.body
                        ? JSON.stringify(trimObject(requestConfig.body))
                        : null,
                }
        try
        {
            const response = await fetch(
                `${requestConfig.baseUrl ?
                    requestConfig.baseUrl : backendUrl}
                    ${requestConfig.url}`,
                requestData
            )

            const data = await response.json()
            applyData(data)

            if (!response.ok)
            {
                throw new Error(data.message)
            }

            let message = data.message
            if (message)
            {
                message = message.toLowerCase()
                if (
                    !message.includes('success') &&
                    // for not make error when no messages in chat
                    !message.includes('no') &&
                    !message.includes('yet') &&
                    //  for not make error when push notification failed
                    !message.includes('notification') &&
                    data.statusCode !== 500 &&
                    !data.success
                )
                {
                    popMessage(message ||
                        'Something went wrong',
```

```
                            { variant: 'error' })
                    }
                }
            } catch (error)
            {
                setIsLoading(false)

                //  for not make error when push notification failed
                if (
                    !error.message.includes('notification') &&
                    !error.message.includes('unexpected error !') &&
                    // for not make error when No tasks yet !
                    !error.message.includes('No tasks yet !')
                )
                    popMessage(error.message ||
                        'Something went wrong', {
                        variant: 'error',
                    })
            }
            setIsLoading(false)
        },
        [popMessage, token]
    )

    return {
        isLoading,
        sendRequest,
    }
}

export default useHttp
```

9. Performance Optimization

To ensure optimal performance, we employ various techniques, including:

- **Code Splitting**: Using React.lazy to load components only when needed.
- **Callback Optimization**: Using useCallback to maintain function references and avoid re-creation on every render.

10. Deployment

The front-end application is deployed on Vercel, which offers seamless integration with GitHub for continuous deployment. The deployment process includes building the application and setting up necessary environment variables and configurations.

11. Challenges During Development

Throughout our front-end development, we tackled significant challenges including integrating real-time functionalities with sockets. We centralized socket management and adopted a modular approach using createAsyncThunk in Redux slices for efficient event handling.

Organizing our codebase with a Page-Based Structure and employing design patterns like Presentational and Container and Facade was crucial, separating UI from business logic for improved scalability.

Implementing reliable video and voice calls with Simple Peer, and Socket.io ensured smooth real-time communication despite network complexities.

These strategies enabled us to build a robust, scalable application connecting learners and mentors seamlessly, delivering an engaging user experience.