5. Frontend Development

React Application Structure The project features a React application for the frontend using Typescript. This is designed to provide and engaging and interactive experience for the users. It has a landing page with multiple sections, and navigation to each implemented game. Furthermore it is developed in a modular way, ensuring that scaling the project up and down can be done easily.

Key sections and features The landing page is the first page of the application and where the user will start when they enter. It is built up by using several components that form the landing page when put together.

```
import Header from './components/Header';
import HeroSection from './components/HeroSection';
import GamesSection from './components/GamesSection';
import StartSection from './components/StartSection';
import Footer from './components/Footer';

function LandingPage() {
  return (
    <div className="min-h-screen flex flex-col">
      <Header />
      <HeroSection />
      <GamesSection />
      <StartSection />
      <Footer />
    </div>
  );
}

export default LandingPage;
```

The `header` is at the top of the landing page, and serves as a navigation bar. It has the logo of the application and then links to different sections of the application: `login`,`register`etc. Below that is the `HeroSection`, where you would typically have a banner, welcoming the user onto the page and inviting them to navigate around the application to explore the different features. Below that is the `GamesSection` where the different implemented games are listed along with buttons that navigate to said games. Below that there is a "Start" section, which is component that invites the user to sign up in order to play the games. Lastly a `Footer` is implemented, which is a generic footer containing copyright information and links to resources such as terms of service and privacy policy. The application also has seperate game pages, which when navigated to, shows each of the games.

Navigation and Routing For routing in the application react-router-dom is used to enable navigation between different pages. The routes are defined

in the index.tsx file, and currently contains routes for `blackjack`, `coinflip`, `register` and the `landing page`.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import LandingPage from './Landingpage';
import PokerPage from './Pages/Poker';
import CoinFlipPage from './Pages/CoinFlip';
import BlackjackPage from './Pages/Blackjack';
import RegisterPage from './Pages/RegisterPage';
//import RoulettePage from './Pages/Roulette';
import { BrowserRouter as Router, Route,Routes } from 'react-router-dom'
import { UserProvider } from './state/userState/userContext';
import { AuthProvider } from './state/authState/authContext';
//import { LoginPage } from './components/LoginPage';

ReactDOM.createRoot(document.getElementById('root') as HTMLElement).render(
  <React.StrictMode>
      {/*Router is used to navigate between pages in the app :O*/}
      <Router>
        {/*gives access to the users state in all inclosed components :D*/}
        <AuthProvider>
          <UserProvider>
          <Routes>
            <Route path="/" element={<LandingPage />} />
            <Route path="/coinflip" element={<CoinFlipPage />} />
            <Route path="/poker" element={<PokerPage />} />
            <Route path="/blackjack" element={<BlackjackPage />} />
            <Route path="/register" element={<RegisterPage />} />

          </Routes>
          </UserProvider>
        </AuthProvider>
    </Router>
  </React.StrictMode>
);
```

This setup makes it easy if new games are added, because they can just be added to the index file.

TypeScript Integration The project is built with TypeScript, which makes it possible to do static type checking. This is super helpful in order to catch errors during development instead of having to catch them at runtime. Typescript and the static type checking is best used when defining interfaces and types for components props and states. This limits the data passed between

componenets such that it has to be structured properly for the component to accept it.

Vite Build Tool Implementation for building the application this project uses Vite. This is a popular build tool with some advantages mainly aimed at ease of use for the developers. The advantages include a really fast development server providing instant updates when changes are made to the code. It supports TypeScript out of the box, which minimized the amount of work developers have to focus on configuring build pipelines.

State Management In this project, Reacts built in state management hooks are used. UseNavigation is used from react-router-dom is used to handle the navigation state and allowing the user to navigate to different pages. Local states are also being used within the components with UseState. UseState can be used to handle specific UI states, for example when users are interacting with the application, or when dynamic content rendering is used.

In our application, we manage state efficiently using React's useContext in combination with a Provider. This approach centralizes state management, making the application easier to scale, maintain, and extend as the complexity of features grows. With this setup: State stores the current application data. Dispatch handles updates and actions, ensuring predictable and controlled state transitions. By wrapping our components in context Providers, any part of the application can access and modify the shared state without the need for cumbersome prop drilling, enhancing the overall developer experience. We have implemented specific contexts and providers tailored to key areas of our application: Authentication Context: Manages authentication tokens, login status, and user session data. User Context: Handles user-specific information, such as profile details or preferences. Blackjack Context: Manages the state for our Blackjack game, including game logic, player actions, and dealer interactions. This modular approach ensures that each context is focused on its specific domain, improving code organization and maintainability. By leveraging useContext and the Provider, we create a robust and scalable foundation for managing complex state transitions and interactions across the application while adhering to React's functional component paradigm.

Security Features

Token Security Implementation In our frontend application, we have chosen to store authentication tokens securely in cookies. This decision strikes a balance between usability and security, ensuring sensitive token data is protected while providing a seamless user experience. The main reason cookies are optimal for token storage is their ability to include security features like the HttpOnly flag. This flag makes cookies inaccessible to JavaScript, significantly reducing the risk of XSS (Cross-Site Scripting) attacks. Additionally, the Secure flag

ensures cookies are transmitted only over HTTPS, preventing exposure over unencrypted connections. For token expiration, we enforce short lifetimes to limit the risk of misuse if a cookie is compromised. While the expiration mechanism is managed on the backend, it adds another layer of security by ensuring tokens do not remain valid indefinitely. We store both the refresh token and access token in cookies. The frontend includes functionality to refresh the access token as its expiration approaches, ensuring uninterrupted user sessions. This mechanism allows users to remain logged in as long as they keep the site open. Moreover, cookies enable persistent login between visits. If the user closes the webpage and later reopens it, the application automatically checks if valid tokens exist in the cookies. If the tokens are not expired and can still be used, the user is logged back into the site seamlessly. By leveraging cookies for token storage and refresh management, we provide a secure and user-friendly authentication experience. ##### Input Validation and Sanitation We perform input validation and sanitization in the frontend to protect our backend from malicious inputs and to catch invalid data early in the process. This proactive approach enhances security while improving the user experience by providing immediate feedback if the input does not meet our predefined standards. For instance, we enforce specific requirements for fields such as passwords and usernames, ensuring they comply with our security and usability guidelines before they are submitted to the server. ##### Secure Communication (HTTPS) Our frontend exclusively uses HTTPS to ensure that all data transmitted between the client and the server is encrypted. This protects sensitive information, such as authentication tokens and user data, from interception or tampering during transmission, maintaining the integrity and confidentiality of the communication.

Package Management

Framework Selection Rationale The project uses **React** with **TypeScript** as the primary framework and language for the frontend, paired with **Vite** as the build tool. This selection is based on the following considerations:

1. **React**:
   - A component-based architecture makes it easy to build reusable and scalable UI elements.
   - A large developer community ensures abundant resources and libraries for solving common challenges.
   - Seamless integration with state management tools and routing libraries like `react-router-dom`.
2. **TypeScript**:
   - Provides static type-checking, reducing runtime errors.
   - Helps enforce stricter coding standards by defining types for props, states, and API responses.
   - Facilitates better code readability and maintainability.
3. **Vite**:

- Offers a fast development server with near-instant hot module replacement (HMR).
- Simplifies configuration, allowing developers to focus more on building the application.
- Produces optimized builds with faster performance compared to traditional tools like Webpack.

This combination was chosen for its ability to balance development speed, scalability, and code quality, making it an ideal stack for a modern, interactive frontend application.

Project Structure Our project structure is meticulously designed to adhere to React best practices.

GitHub Workflow

- **ci.yml**: This file defines the GitHub workflow for building and testing the project before deployment.

Source Directory (`src`)

- **Components**: Contains individual React components, each in its own folder.
- **Pages**: Combines various components into complete pages.
- **State**: Manages the application's state.
- **Styling**: Includes `.css` files for styling the application.
- **Tests**: Contains various test files.

Special Files

- **Landingpage.tsx**: A standalone file that imports different components to create the landing page.
- **Main**: Primarily handles the routing for the application.