Complex Systems and DevOps: Deliverable 2

Date: November 25, 2024

Course Details

Course Name: Complex Systems and DevOps **Course Code:** 62582 **Semester:** Fall 2024

Team Members

Name	Student Number
Christoffer Fink	*s205449*
Kasper Falch Skov	*s205429*
Johan Søgaard Jørgensen	*s224324*
Henrik Lynggaard Skindhøj	*s205464*
Kevin Wang Højgaard	*s195166*
Sebastian Halfdan Lauridsen	*s215769*

Link to GitHub Project

Table of Contents

4. Backend Development
    - Quarkus Framework Implementation
    - REST API Design with Siren Hypermedia
    - Database Integration
    - Business Logic Implementation
    - Security Implementation
        - JWT Authentication
    - Testing Strategy
        - JUnit Implementation
        - REST-assured Testing
    - OpenAPI Documentation
5. Frontend Development
    - React Application Structure
    - TypeScript Integration
    - Vite Build Tool Implementation
    - State Management
    - Component Architecture
    - Security Features
        - Token Security Implementation
    - Package Management
6. DevOps Implementation
    - Version Control Practices
        - Git Workflow
        - GitHub Integration
    - Continuous Integration/Continuous Deployment
        - GitHub Actions Configuration
        - Build Server Setup
        - Testing Pipeline
    - Containerization
        - Docker Implementation
        - Container Registry
    - Cloud Deployment
        - Google Cloud Setup
        - Netlify Frontend Deployment
    - Monitoring and Maintenance
7. Conclusion
    - Project Outcomes
    - Future Improvements
    - Lessons Learned

+– csdg8 +– auth | +– AuthController.java | +– AuthResource.java | +– AuthService.java | +– dto | | +– CreateTokenRequest.java | | +– CreateTokenResponse.java | | +– GetAuthResponse.java | | +– RefreshAccessTokenRequest.java | | +– RefreshAccessTokenResponse.java | +– TokenService.java +– user | +– dto | | +– CreateUserRequest.java | | +– GetCollectionUserResponse.java | | +– GetUserResponse.java | +– UserController.java | +–

User.java | +– UserResource.java | +– UserService.java | ...

```
*Snippet of the backend file structure showing its organization by the domains "auth" and "u

To further simplify new features or changes our general class hierarchy is as follows

1. Resource
2. Controller
3. Service
4. Business Logic

The resource classes only have direct interaction with a single controller in the same domai

Another feature of this model is our separation of DTOs from implementation classes. None of

A further improvement to this structure would be, in the same vein as our DTOs, shielding th

#### Security Implementation
*quarkus security with rolesallowed, 401 vs 403.*

To secure our API we use Quarkus Security Jakarta Persistance. Our User entity's variables h

```java
@Entity
@Table(name = "app-user")
@UserDefinition
public class User extends PanacheEntity {

    @Username
    public String username;

    @Password
    public String password;

    @Roles
    public Set<String> role;

    //...
}
```

The `@UserDefinition` tells our application that this entity is a source of identity information, whilst the accompanying `@Username` indicates that this field is a username, the `@Password` indicates that this field is a hashed password and finally the `@Roles` indicates that this field is a collection (a `Set` in our case to force unique roles) of roles.

To secure a specific endpoint we use the `@RolesAllowed` annotation with a
```

single or multiple specified roles. In the following example users accessing the `/auth/token/refresh` endpoint must have either the "user" or "admin" role.

```
@POST
@RolesAllowed({ "user", "admin" })
@Path("/token/refresh")
@Consumes(MediaType.APPLICATION_JSON)
public Entity refreshAccessToken(RefreshAccessTokenRequest request) throws Siren4JException
    //...
}
```

Simplified endpoint which uses the `@RolesAllowed` annotation to limit access.

The annotation can also be used for more fine-grained control on service methods if a part of the application must be extra secure, however we have not utilized this functionality yet.

JWT Authentication For authentication we decided to roll our own JWT tokens via the MicroProfile JWT RBAC specification. Quarkus conveniently provides a library for this named `quarkus-smallrye-jwt`. In short, a JWT token is a server provided signed token which anyone can verify was signed by the server, meaning the contents and access which it grants are valid. To accomplish this JWT tokens are signed with a *private key* by the server and can be verified with the linked *public key*. Contrary to a regular encrypted transaction where the *public key* signs some data which can then only be unlocked with the *private key*. In our self-rolled JWT implementation, when a registered user sends their username and password in a POST request to the `/auth/token` endpoint, the credentials are validated and a JWT token is generated via the *private key*.

```
public String generateAccessToken(User user) {
    return Jwt.issuer(this.issuer)
        .upn(user.getUsername())
        .subject(user.id.toString())
        .groups(user.getRole())
        .expiresIn(Duration.ofMinutes(5))
        .sign();
}
```

Our self-rolled JWT token generation.

Of note here is the `upn` which is our main unique ID in our JWT tokens, but we also have a `subject` claim which we use to identify a user in our backend.

Once the JWT token is generated and signed it is returned to the user. When the user then sends a request to a locked endpoint (recall the `@RolesAllowed` annotation) the token is automatically verified by the Quarkus framework with the *public key* and verified if it was actually signed by the server. If it is valid and if the user has the required roles the request may proceed.

As mentioned earlier we utilize a access-token (JWT) and refresh-token pair. With the access-token being short-lived and the refresh-token being long lived. The refresh-token is a simple generated UUID and the backend keeps track of which UUIDs it generates. A future improvement will be storing the list of these generated UUIDs in persistent storage instead of in-memory as it is now. If the server terminates for whatever reason the list of valid refresh-tokens is lost and all users must re-login to get a new refresh-token. The access-token should not be persisted as it is assumed, because it is cryptographically signed, that it is always valid if it is not expired. To re-emphasize, a JWT token can only come from the server as it is the only one with the *private key*.

Testing Strategy for UserService

The **testing strategy** for the `UserService` class ensures that core functionalities related to user management—such as user validation, adding users, fetching users, and handling user roles and balances—work as expected. It also addresses edge cases and error scenarios, ensuring the service remains reliable and resilient under various conditions.

The strategy includes both unit tests and integration tests to confirm that the system's behavior aligns with business requirements.

---

1. JUnit Implementation for User Validation

- **Valid User Credentials**: Ensures users with correct credentials are validated successfully.
  - **Test**: `shouldValidateUserWithCorrectCredentials`
  - **Strategy**: Verify that the system returns the correct user when provided with valid credentials using JUnit.
- **Invalid User Credentials**: Ensures the system handles invalid credentials gracefully.
  - **Test**: `shouldNotValidateUserWithIncorrectCredentials`
  - **Strategy**: Verify that incorrect credentials do not authenticate the user using JUnit.

---

2. JUnit Implementation for User Role Management

- **Fetching Roles for Existing Users**: Confirms that user roles can be retrieved for valid users.
  - **Test**: `shouldGetUserRoleForExistingUser`
  - **Strategy**: Verify that the correct roles are returned for valid users using JUnit.
- **Fetching Roles for Non-Existent Users**: Ensures the service does not return roles for non-existent users.

  - **Test**: `shouldNotGetUserRoleForNonExistentUser`
  - **Strategy**: Verify that a non-existent user returns an empty set or a proper error using JUnit.

---

3. JUnit Implementation for User Management (CRUD Operations)

- **Adding New Users**: Ensures the system correctly handles user creation.
  - **Test**: `shouldAddUserSuccessfully`
  - **Strategy**: Verify that a new user is added successfully with the correct username and role using JUnit.
- **Handling Existing Users**: Ensures the system throws an appropriate exception when adding an existing user.
  - **Test**: `shouldThrowUserAlreadyExistsExceptionWhenAddingExistingUser`
  - **Strategy**: Verify that the service throws a `UserAlreadyExistsException` for duplicate users using JUnit.
- **Invalid Usernames or Passwords**: Tests edge cases with invalid credentials.
  - **Tests**: `shouldThrowInvalidCredentialsExceptionForInvalidUsername`, `shouldThrowInvalidCredentialsExceptionForInvalidPassword`
  - **Strategy**: Verify that invalid credentials trigger appropriate exceptions using JUnit.
- **Fetching User by Username**: Ensures users can be retrieved based on their username.
  - **Test**: `shouldGetUserByUsername`
  - **Strategy**: Verify that a user can be found and retrieved correctly by username using JUnit.
- **Handling Non-Existent User Fetching**: Ensures the system handles fetching non-existent users appropriately.
  - **Test**: `shouldThrowUserNotFoundExceptionForNonExistentUser`
  - **Strategy**: Verify that fetching a non-existent user results in a `UserNotFoundException` using JUnit.
- **Fetching All Users**: Validates that the system returns all users correctly.
  - **Test**: `shouldGetAllUsers`
  - **Strategy**: Verify that all users in the system are returned in a list using JUnit.

---

4. JUnit Implementation for User Balance Management

- **Adding Balance to User**: Ensures user balances can be incremented correctly.
  - **Test**: `shouldAddBalanceToUser`
  - **Strategy**: Verify that adding balance to a user's account works correctly using JUnit.

- **Subtracting Balance from User**: Ensures balances can be deducted correctly.
  - **Test**: `shouldSubtractBalanceFromUser`
  - **Strategy**: Verify that balance subtraction functions correctly using JUnit.

---

5. JUnit Implementation for Edge Cases and Error Scenarios

- **User Already Exists Exception**: Ensures the service prevents duplicate user creation.
  - **Test**: `shouldThrowUserAlreadyExistsExceptionWhenAddingExistingUser`
  - **Strategy**: Verify that the system throws `UserAlreadyExistsException` for duplicate users using JUnit.
- **Invalid Credentials**: Confirms that invalid credentials trigger appropriate error handling.
  - **Tests**: `shouldThrowInvalidCredentialsExceptionForInvalidUsername`, `shouldThrowInvalidCredentialsExceptionForInvalidPassword`
  - **Strategy**: Verify that invalid credentials are handled properly using JUnit.
- **Non-Existent User Fetching**: Ensures requests for non-existent users are handled appropriately.
  - **Test**: `shouldThrowUserNotFoundExceptionForNonExistentUser`
  - **Strategy**: Verify that the service handles invalid user requests with proper exception handling using JUnit.

---

6. Transactional and Database Integrity

- **Database Transactions**: Ensures changes made during tests are rolled back after each test.
  - **Strategy**: Verify that database changes during tests are rolled back to maintain isolation between tests using JUnit.

---

7. REST-assured Testing for API Endpoints

- **User Validation Endpoint**: Ensures the API correctly validates users.
  - **Test**: `shouldValidateUserEndpoint`
  - **Strategy**: Use REST-assured to verify that the user validation endpoint returns the correct response for valid and invalid credentials.
- **User Role Management Endpoint**: Ensures the API correctly handles user roles.
  - **Test**: `shouldManageUserRolesEndpoint`

  - **Strategy**: Use REST-assured to verify that the user role management endpoint returns the correct roles for valid users and handles errors for non-existent users.
- **User Management Endpoints**: Ensures the API correctly handles CRUD operations for users.
  - **Tests**: `shouldAddUserEndpoint`, `shouldFetchUserEndpoint`, `shouldUpdateUserEndpoint`, `shouldDeleteUserEndpoint`
  - **Strategy**: Use REST-assured to verify that the user management endpoints correctly handle adding, fetching, updating, and deleting users.
- **User Balance Management Endpoint**: Ensures the API correctly handles user balances.
  - **Tests**: `shouldAddBalanceEndpoint`, `shouldSubtractBalanceEndpoint`
  - **Strategy**: Use REST-assured to verify that the user balance management endpoints correctly handle adding and subtracting balances.

---

8. Test Automation and Continuous Integration

- **Automated Test Suites**: Ensures the user service works as expected across different scenarios using JUnit and REST-assured.
  - **Strategy**: Integrate automated tests into a CI pipeline to catch issues early.
- **Regression Testing**: Ensures new changes or features do not break existing functionality.
  - **Strategy**: Rerun tests regularly to maintain application stability.

OpenAPI Documentation

The **OpenAPI Documentation** provides a comprehensive specification for the API endpoints that interact with the `UserService`. It outlines the available operations for managing users, validating credentials, handling user roles, and updating user balances. This section ensures that the API is clearly defined, making it easy for developers to integrate with the service, understand the request/response formats, and know the expected behavior.

The following is the structure of the OpenAPI documentation for the endpoints related to the `UserService`:

1. **Validate User**

   - **Endpoint**: `/api/users/validate`
   - **Method**: `POST`
   - **Description**: Validates the credentials of a user by checking the provided username and password.

   Request:

```
{
  "username": "string",
  "password": "string"
}
```

Response:

- `200 OK`: User is validated successfully.

  ```
  {
    "username": "string",
    "roles": ["string"]
  }
  ```

- `401 Unauthorized`: Invalid credentials provided.

  ```
  {
    "error": "Invalid credentials"
  }
  ```

2. **Add User**

   - **Endpoint**: `/api/users`
   - **Method**: `POST`
   - **Description**: Creates a new user in the system with the provided username, password, and roles.

   Request:

   ```
   {
     "username": "string",
     "password": "string",
     "roles": ["string"]
   }
   ```

   Response:

   - `201 Created`: User successfully added.

     ```
     {
       "username": "string",
       "roles": ["string"]
     }
     ```

   - `409 Conflict`: User already exists.

     ```
     {
       "error": "User already exists"
     }
     ```

   - `400 Bad Request`: Invalid credentials or malformed request.

```
{
  "error": "Invalid credentials"
}
```

3. **Get User Roles**

   - **Endpoint**: `/api/users/{username}/roles`
   - **Method**: `GET`
   - **Description**: Retrieves the roles associated with a specific user.

   Request:

   - **Path Parameter**:
     - `username` (string): The username of the user whose roles are to be retrieved.

   Response:

   - `200 OK`: Returns the roles associated with the user.

     ```
     {
       "roles": ["string"]
     }
     ```

   - `404 Not Found`: User not found.

     ```
     {
       "error": "User not found"
     }
     ```

4. **Get User by ID**

   - **Endpoint**: `/api/users/{userId}`
   - **Method**: `GET`
   - **Description**: Fetches the details of a user based on their unique user ID.

   Request:

   - **Path Parameter**:
     - `userId` (integer): The unique ID of the user to retrieve.

   Response:

   - `200 OK`: Successfully retrieves the user information.

     ```
     {
       "id": "integer",
       "username": "string",
       "roles": ["string"],
       "balance": "integer"
     }
     ```

   - `404 Not Found`: User not found.

```
{
  "error": "User not found"
}
```

5. **Update User Balance**

   - **Endpoint**: `/api/users/{userId}/balance`
   - **Method**: `PATCH`
   - **Description**: Updates the balance of a specific user by adding or subtracting a specified amount.

   Request:

   - **Path Parameter**:
     - `userId` (integer): The unique ID of the user whose balance is to be updated.
   - **Body**:

     ```
     {
       "amount": "integer"
     }
     ```

   Response:

   - `200 OK`: Balance updated successfully.

     ```
     {
       "id": "integer",
       "username": "string",
       "balance": "integer"
     }
     ```

   - `404 Not Found`: User not found.

     ```
     {
       "error": "User not found"
     }
     ```

6. **Remove User Balance**

   - **Endpoint**: `/api/users/{userId}/balance`
   - **Method**: `DELETE`
   - **Description**: Removes a specified amount from the user's balance.

   Request:

   - **Path Parameter**:
     - `userId` (integer): The unique ID of the user whose balance is to be updated.
   - **Body**:

```
{
  "amount": "integer"
}
```

Response:

- `200 OK`: Balance updated successfully.

```
{
  "id": "integer",
  "username": "string",
  "balance": "integer"
}
```

- `404 Not Found`: User not found.

```
{
  "error": "User not found"
}
```

Response Codes Summary

- `200 OK`: Request successfully completed.
- `201 Created`: Resource successfully created.
- `400 Bad Request`: Malformed request or invalid input.
- `401 Unauthorized`: Invalid credentials provided.
- `404 Not Found`: Resource (user) not found.
- `409 Conflict`: Resource already exists (e.g., attempting to create a user that already exists). ### 5. Frontend Development #### React Application Structure

The project is a frontend application built using React with TypeScript, designed to provide an engaging and interactive user experience. It features a landing page with multiple sections, including clear navigation to different game pages. The application incorporates secure authentication with login and registration pages. Each game is implemented on its own dedicated page, leveraging a modular and structured approach. This design ensures scalability, allowing for seamless project expansion or simplification, while maintaining clean and maintainable code.

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.

```
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. Further down 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 the 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`.

```
ReactDOM.createRoot(document.getElementById('root') as HTMLElement).render(
  <React.StrictMode>
      <Router>
        <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 minimizes the amount of work developers have to focus on configuring build pipelines.

State Management In this project, React's built in state management hooks are used. UseNavigation from react-router-dom is used to handle the navigation state and allows 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. To clarify, we also sanitize and validate inputs in the backend, but catching validation errors at the frontend enhances the overall system by reducing invalid requests sent 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 In our project, we utilize npm (Node Package Manager) as the primary package manager for frontend development. npm is a powerful tool that simplifies the process of managing dependencies and libraries required for the project. By specifying the necessary packages in the package.json file, npm allows us to efficiently install, update, and manage all dependencies in a structured and repeatable manner.

One of the key advantages of npm is its ability to define and execute custom scripts. These scripts streamline various development and build tasks. For example:

npm run dev: This command is configured to start a development server, providing a live-reloading environment that makes it easier to test and iterate on frontend code during development. npm run build: This command specifies the steps required to build the production-ready version of the frontend application. It typically compiles, minifies, and optimizes the code for deployment. Additionally, npm enhances the security of our project by providing the npm audit tool. This tool scans the project dependencies for known vulnerabilities and security threats. By running npm audit, we can:

- Detect security issues in real-time across all installed libraries.
- Receive detailed reports on the nature of vulnerabilities, their severity, and potential fixes.
- Apply automated patches for certain vulnerabilities using npm audit fix.

By regularly auditing our dependencies, we ensure that our project remains secure and compliant with best practices, reducing the risk of threats from third-party libraries. As a future improvement, we could set up a GitHub Action to automatically create issues whenever vulnerabilities are detected during the auditing process. This automation would help us maintain a proactive approach to dependency management, ensuring timely resolution of potential security concerns.

In addition to managing dependencies and automating tasks, npm also enables us to run custom test suites. For instance, by specifying test commands in package.json, such as npm run test, we can execute different testing frameworks or tools to ensure the quality and stability of our code. This feature presents an opportunity for future improvement, as we plan to leverage it when implementing a comprehensive testing strategy for the frontend.

By leveraging npm for package management, task automation, and security auditing, we ensure that our frontend development workflow is consistent, efficient, and secure. It simplifies collaboration across the team, as all required dependencies and commands are easily accessible, while safeguarding the project against potential vulnerabilities. #### Framework Selection Rationale

Our project uses **React** with **TypeScript** as the primary framework and language for the frontend, paired with **Vite** as the build tool. This choice is informed by the features we have discussed previously and is further supported by the following specific points:

React:

- Component-based architecture: Simplifies the development of reusable, modular, and scalable UI elements, enabling consistent design and functionality across the application.
- Robust ecosystem: With a large developer community, React provides abundant resources, libraries, and third-party integrations to address common challenges efficiently.

- Extensibility: React integrates seamlessly with tools such as react-router-dom for routing and state management libraries like Redux or Context API, making it versatile for various project needs.

TypeScript:

- Static type-checking: Reduces runtime errors by catching issues during development, ensuring more robust code.
- Enforces stricter coding standards: Enables explicit definitions for props, states, and API responses, fostering better collaboration and reducing ambiguity in the codebase.
- Enhanced maintainability: Improves code readability, simplifies debugging, and makes the project easier to scale.

Vite:

- Lightning-fast development server: Delivers near-instant hot module replacement (HMR), significantly boosting development speed and productivity.
- Ease of configuration: Requires minimal setup, letting developers focus on building the application rather than configuring the tooling.
- Optimized builds: Produces lightweight, high-performance builds that outperform traditional tools like Webpack, ensuring a smoother user experience.

This combination of React, TypeScript, and Vite provides a well-rounded stack that prioritizes development efficiency, scalability, and code quality. It is tailored to meet the demands of a modern, interactive frontend application, making it an ideal choice for our project.

Project Structure Our project structure is meticulously designed to adhere to React best practices, ensuring maintainability, scalability, and readability of the codebase. We follow a component-based architecture, where each UI element is encapsulated in its own reusable and self-contained component. This approach promotes reusability and makes it easier to manage changes or updates.

We also emphasize clear separation of concerns by organizing files logically. For example, each component has its own directory containing its Typescript file, CSS (or module CSS for scoped styling), and any relevant assets. This structure ensures that related code is grouped together, making the project easier to navigate.

Another key best practice we follow is state management. For local state, we use React's useState and useReducer hooks, while global state is managed with Context API or third-party libraries, depending on the project requirements. For instance, in our authentication flow, we use Context API to share user and authentication states across the app, avoiding unnecessary prop drilling.

Finally, we focus on writing clean, readable code by adhering to consistent

formatting rules enforced by tools like Prettier and ESLint. This ensures that all developers contribute code that follows a unified style, reducing friction during code reviews.

GitHub Workflow

- **ci.yml**: This file defines the GitHub workflow for automating key stages of our development pipeline, including building and testing the project before deployment. The workflow is triggered by specific events, such as pushing to the main branch or opening a pull request, ensuring that our codebase remains stable and free of regressions. By automating these processes, we reduce manual effort, maintain high code quality, and accelerate development cycles.

Further details about the setup, configuration, and implementation of the ci.yml file, as well as how it integrates with other DevOps practices, are discussed in the DevOps section of this report. ###### Source Directory (`src`) Our project's directory structure is carefully designed to promote clarity, maintainability, and scalability, adhering to established best practices in React development - **Components**: This folder contains individual React components, each in its own folder with associated files like .tsx. By isolating components, we ensure reusability, making it easier to share UI elements across pages. This modular approach also simplifies debugging and updating specific parts of the UI. - **Pages**: Complete pages are created by assembling multiple components. This separation of "components" and "pages" enforces a clean hierarchy, where pages focus on layout and orchestration while components handle granular functionality. This approach improves readability and helps maintain a clear distinction between reusable pieces of UI and higher-level structures. - **State**: The State folder centralizes state management, using Context API. This setup avoids scattering state-related code across the application, making it easier to debug, scale, and extend the application's logic. - **Styling**: All .css files are grouped under this directory to maintain consistency in styling and enforce a separation of concerns. Using a dedicated folder allows developers to quickly locate and modify styles, whether they are global or scoped to specific components. - **Tests**: Housing all test files in a single directory ensures that testing remains a first-class citizen in the project. It helps maintain an organized structure, where tests are easy to locate and run, and it encourages consistent testing practices across the team.

6. DevOps Implementation

Version Control Practices

Git Workflow

GitHub Integration

Continuous Integration/Continuous Deployment

GitHub Actions Configuration

Build Server Setup

Testing Pipeline

Containerization

Docker Implementation

Container Registry

Cloud Deployment

Google Cloud Setup

Netlify Frontend Deployment In our project, we utilize Netlify as the platform for hosting and deploying our frontend application. Netlify offers an automated and seamless deployment workflow by integrating directly with our GitHub repository. Every time changes are pushed to the main branch, Netlify automatically pulls the latest code, runs the specified build command (npm run build), and deploys the application to its globally distributed Content Delivery Network (CDN).

As part of this automated process, Netlify provides real-time feedback on the deployment status. If the npm run build command succeeds, the application is immediately deployed, and the live site is updated. If the build fails, Netlify generates detailed logs, making it easy to identify and resolve any issues. After deployment, Netlify confirms whether the site was successfully deployed, ensuring full transparency in the deployment process.

In addition, for every pull request, Netlify runs a Lighthouse audit to evaluate the performance, accessibility, best practices, and SEO of the site. This feature provides a clear score and actionable insights, helping us continuously optimize the application before merging changes into the main branch. By catching potential issues at the pull request level, we ensure that only high-quality updates make it into production.

We have also implemented GitHub Actions as part of our CI/CD pipeline. These actions complement Netlify by automating additional checks, such as running unit tests, linting, and verifying configurations before deployment. GitHub Actions provide an extra layer of validation, ensuring that the codebase adheres to predefined quality standards and that potential issues are identified early in the development cycle.

Netlify's integration with GitHub and the use of Github Actions significantly streamline our deployment process. Features like automatic builds, deployment

status notifications, and Lighthouse audits on pull requests, we are able to maintain a high-quality frontend application. Its global CDN ensures that the deployed site loads quickly for users worldwide, further enhancing the user experience. By leveraging Netlify, we have a reliable, efficient, and developer-friendly solution for managing our frontend deployment pipeline.

Another of the key advantages of using Netlify is that it offers its core features for free, including automated builds, deployments, and a globally distributed CDN. This makes it an excellent choice for small teams or projects, allowing us to deploy a high-quality frontend application without incurring additional costs.

Monitoring and Maintenance

7. Conclusion

Project Outcomes

Future Improvements

Lessons Learned