Custom Project Stage 2 and 3 Criteria

Stage 2. Back End

Project Rejected Without Review

- The API doesn't register new users when valid data is input.
- The application doesn't start upon using the `npm run dev` command once all required dependencies have been installed.
- There must not be code snippets that have been plagiarized.

Performance Criteria

- The repository contains all the necessary infrastructure files: **`/ 3.64`**
  - `package.json`
  - `.editorconfig`
  - `.eslintrc` is required for extending the configuration of `airbnb-base`
    - `package.json` contains the `devDependencies` needed for the linter to work correctly
    - An exception for `_id` is added
    - The following rules are forbidden: `eslint-disable`, `eslint-disable-line`, and `eslint-disable-next-line`
  - `.gitignore` should contain at least the `node_modules` folder
- There are no lint errors. When running `npx eslint .` they should be absent. **`/ 3.64`**
- The `scripts` section of the `package.json` file contains the following: **`/ 3.64`**
  - An `npm run start` command that starts the server on `localhost:3000`.
  - An `npm run dev` command that starts the server on `localhost:3000` with hot reloading.

- The following routes function as described: `/ 21.81`
  - A request to `GET /users/me` returns information about the user (email and name).
  - `POST /signup` creates a user with the data passed inside the request body.
  - `POST /signin` returns a JWT when the correct email and password are passed in the request body.
  - `GET` returns data saved by the user.
  - `POST` creates a data item with the data passed inside the request body.
  - `DELETE` deletes the saved data item using `_id`.
- Users can't delete saved data from other user profiles. `/ 3.64`
- All routes are protected with authorization, except for `/signin` and `/signup`. `/ 3.64`
- User routes and data-related routes are described in separate files. `/ 3.64`
- API errors are handled: `/ 7.28`
  - Error status codes are used: 400, 401, 403, 404, 409, 500.
  - If something is wrong with the request, the server returns a response with an error message and a corresponding status.
  - The error message matches its type.
  - Asynchronous handlers end with a `catch()` block.
  - The API does not return standard database or Node.js errors.
- Safe password storage has been implemented: `/ 3.64`
  - Passwords are stored in a hashed form.
  - The API does not return a password hash to the client.
- Validation has been implemented correctly: `/ 7.28`
  - Requests are validated before being passed to the controller. The body and (where applicable) headers and parameters are checked against the corresponding schemas. If a request doesn't match the schema, the processing is not passed to the controller and the client receives a validation error.

    - Data is validated before being added to the database.
- In production mode, the database address is taken from `process.env`. **/ 7.28**
- The server can be accessed via HTTPS using the domain specified in `README.md`. **/ 3.64**
- Storing the secret key for creating a JWT is implemented correctly: **/ 7.28**
    - In production (i.e., when deployed to the server), the secret key should be stored in a `.env` file, and this file should not be added to Git.
    - In development (i.e., when run on localhost), the code should run and function correctly, even though there is no `.env` file present.

Best Practices

- All routes are connected to the `index.js` file, which is located in the `routes` folder, and `app.js` contains one main route handled by `routes`. **/ 2.14**
- Asynchronous operations are implemented using promises or async/await. **/ 2.14**
- Validation is described in a separate module. **/ 2.14**
- Logging is set up: **/ 2.14**
    - All requests and responses are logged to the `request.log` file.
    - All errors are logged to the `error.log` file.
    - Log files aren't added to the git repository.
- Errors are handled by a centralized handler. **/ 2.14**
- Centralized error handling is described inside a separate module. **/ 2.14**
- The application API is located in an `/api` subdirectory or on an `api.` subdomain (not just name.zone): **/ 2.14**
    - Correct: domain-name.tk/api or api.domain-name.tk
    - Incorrect: domain-name.tk

Recommendations

- For API errors, classes have been created to extend the `Error` constructor. `/ 1.0`
- The Helmet module is used to set security-related headers. `/ 1.0`
- Configuration and constants are stored in separate files: `/ 1.0`
  - The Mongo server address and the private key for the JWT in development mode are stored inside a separate configuration file.
  - Application constants (response and error messages) are stored inside a separate file with constants.
- A rate limiter is set up: the number of requests from a single IP address is limited to a particular value in a given amount of time. `/ 1.0`
- The rate limiter is configured in a separate file and imported into `app.js`. `/ 1.0`

Stage 3. Authorization with React

Project Rejected Without Review

- The pull request was not sent for review.
- The markup was not ported into JSX.
- Data from the API doesn't load or appear.
- Any of the functionality required for this stage does not work (i.e., authorization/registration).
- There are errors when building or running the project.
- The application is not deployed to the server.
- There must not be code snippets that have been plagiarized.

Performance Criteria

- The project functionality is fully implemented according to the current stage's requirements: `/ 21.92`
  - **General**
    - All project links and buttons are functioning.

        - Both header states function correctly. If the user is not logged in, the header should have the "Sign in" button; and if the user is logged in, there should be no "Sign in" button. Log out button, should appear in its place.
        - If the user closes the tab and then returns to the site, data is taken from local storage upon mounting the `App` component.
    - **"Sign up" and "Sign in" pages**
        - When clicking on the "Sign up" button in the "Sign up" popup window, a request is sent to the `/signup` route, provided that all input fields have been filled in correctly. If the request is successful, a popup should appear, which informs the user that they are registered and offers to log them in.
        - When clicking on the "Sign in" button, provided that all input fields have been filled in correctly, a request is sent to the `/signin` route. If the request is successful, the popup is closed.
        - All forms are validated on the client side. The user cannot send a request with invalid data.
- Registration and authorization:
    - At least one route is protected using the `ProtectedRoute` HOC component. `/ 2.64`
    - When trying to access the protected route, unauthorized users are redirected to `/` with an open authorization popup window. `/ 2.64`
    - The `/` route is not protected. `/ 2.64`
    - If the user was logged in and closed the tab, they can return directly to any page of the application by the URL, except for the login and registration pages. `/ 2.64`
    - After a successful `onSignOut()` handler call, the user is redirected to `/`. `/ 2.64`
    - The `useHistory()` hook is used correctly. `/ 2.64`
    - The components `<Switch />`, `<Route />`, and `<Redirect />` are used correctly. `/ 2.64`
- The interaction with the JWT token works correctly:

    - The JWT token is stored in localStorage. / 2.64
    - The JWT token is validated by a request to the server, not just local storage. / 2.64
    - When you log out of the account, JWT is deleted. / 2.64
- A global state variable has been created that stores user data. / 2.64
- Components:
    - Hooks are not used inside conditional statements or loops. / 2.64
    - Hooks are called in a component's main function. / 2.64
    - For class components, effects are described inside the component lifecycle methods. / 2.64
    - For list items, a unique `id` is used instead of an array index. / 2.64
    - Components that use profile data are subscribed to the context. / 2.64
    - The context is embedded in the `App` component via `CurrentUserContext.Provider`. / 2.64
    - A state variable has been created in the root App component that stores user data. This variable is used as a value of the context provider. / 2.64
- Asynchronous API requests: / 5.28
    - Requests can be made through the Fetch API or by using XMLHttpRequest. Third-party libraries (such as axios or jQuery) are not used.
    - API requests are contained in a separate file.
    - The chain for processing promises ends with a `catch()` block.
    - The first `then()` handler returns `res.json`.
- The project complies with the following code style requirements: / 2.64
    - camelCase is used for function and variable names.
    - Only nouns are used as variable names.
    - Variable names clearly describe what is stored in them. If the project has several variables with similar data, then those variables have unique but descriptive names.

    - Descriptive names are used for functions, which reflect what they do.
    - Function names start with a verb.
    - JS classes and functional components are named using nouns and start with a capital letter.
    - Names must not include inappropriate or unclear abbreviations.
    - Custom hook names start with `use`.
- No third-party JavaScript libraries are used unless absolutely necessary. If third-party libraries are connected, then they are used appropriately. **/ 2.64**

Best Practices

- The initial state of state variables contains the correct data type. **/ 3**
- API requests are described inside the `App` component. **/ 3**
- There is no memory leak when hanging handlers. All handlers added with `addEventListener` are removed when the component is unmounted. **/ 3**
- API error handling: **/ 3**
    - The user receives a message in the case of an error.
- Non-variable values (hard-coded constants) are named in all capital letters and stored in a separate configuration file. **/ 3**

Recommendations

- Components from the `react-router` library are used for internal links in the application. **/ 1.66**
- Semantically correct blocks are used for components. No `<div>` or other unnecessary HTML tags are used for components that consist of single-level blocks. **/ 1.66**
- The code is clean and easy to understand: **/ 1.68**
    - The code is readable and clearly structured. Some parts of the code are explained with comments if needed.

- There is no extra code: for example, when a variable is declared but not used, or there is some kind of redundant logic.
- The code is formatted in the same way, and the indentation hierarchy is respected.