CRC Cards

ContractCard.jsx

Parent Classes: ContractHistory
Sub-classes: N/A

Responsibilities:
Displays information about a user contract.
Collaborators:
N/A

ContractHistory.jsx

Parent Classes: N/A
Sub-classes: N/A

Responsibilities:
Sends HTTP request to fetch all contracts for the current user. It then dynamically renders all the contracts in a grid format.

Collaborators:
ContractCard.jsx, Express, mongoDB, Node

notFound.jsx

Parent Classes: N/A
Sub-classes: N/A

Responsibilities:
Indicates to the user if they have accessed an incomplete or non-existent page
Collaborators:
N/A

nameChangeForm.jsx

Parent Classes: N/A
Sub-classes:

Responsibilities:
Form that the user can use to change the name associated with their account
Collaborators:
footer.jsx, navBar.jsx, mongoDB

aboutUs.jsx

Parent Classes: N/A
Sub-classes: N/A

Responsibilities:
Gives a brief overview on what Pactify is about.
Collaborators:
login.jsx, footer.jsx

footer.jsx

Parent Classes: N/A
Sub-classes: N/A

Responsibilities:
Provides access to contact info, privacy policy and contains our copyright notice
Collaborators:
N/A

login.jsx

Parent Classes: N/A
Sub-classes: N/A

Responsibilities:
Contains the fields required for entering user information to log in. Sends user to home page if user data exists in database
Collaborators:
footer.jsx, aboutUs.jsx, mongoDB

navBar.jsx

Parent Classes: N/A
Sub-classes: N/A

Responsibilities:
Provides an organized and intuitive way of navigating through our product.
Collaborators:
accountInfo.jsx, home.jsx

home.jsx

Parent Classes: N/A
Sub-classes: N/A

Responsibilities:
Show the user's contracts and recently viewed.
Collaborators:
footer.jsx, navBar.jsx, mongoDB

signup.jsx

Parent Classes: N/A
Sub-classes: N/A

Responsibilities:
Displays a form for users to fill in their information and create an account. Does error-handling by providing feedback to users indicating empty fields, non-matching password and re-entered password, and minimum password length of 6 characters.
Collaborators:
footer.jsx

forgotPassword.jsx

Parent Classes: N/A
Sub-classes: N/A

Responsibilities:
Allows the user to change their password
Collaborators:
footer.jsx, mongoDB

accountInfo.jsx

Parent Classes: N/A
Sub-classes: N/A

Responsibilities:
Displays the info of the currently logged in user. Provides the option to edit the fields
Collaborators:
footer.jsx, navBar.jsx, mongoDB

System Interactions/Assumptions

Operating System

Dependencies/Assumptions: The system is assumed to run on a Unix-like OS, such as Linux or macOS, although it can also be configured to run on Windows. The development environment is usually on Linux or macOS, but the production environment can be any OS that is supported by Node.js and MongoDB.

Programming Languages and Compilers

JavaScript/JSX: The primary programming language used for both our front-end and back-end development. We use JSX for our React components.

Node.js: The JavaScript runtime environment used for executing server-side code. The Node.js version should be compatible with the packages and dependencies used in the app.

Databases

MongoDB: The database for our application, which is hosted on MongoDB Atlas. Assumptions include having a stable connection to the database and proper indexing for efficient querying.

Mongoose: An ODM (Object Data Modeling) library for MongoDB and Node.js, providing a schema-based solution to model application data. We use it to define our User and Contract models.

Network Configuration

For local development, Node.js, MongoDB, and the React App are on localhost.

Environment Variables: Configuration settings (e.g., database URIs, JWT secret encryption strings) are managed through environment variables, stored in a .env file.

Development and Build Tools

Prettier: A code formatter to ensure consistent code style.

Assumptions

- Node.js and npm must be installed on the system to run the server and manage dependencies.
- The user must know how to run both the server and application.
- The system assumes a stable network connection for accessing the database.
- Sufficient system resources (CPU, memory) to handle the application load in both development and production environments.

System Architecture

From the most abstract view, our application is that of a three-tier architecture. By using React components, we send requests to the database via HTTP requests to fetch data and render the response to the user.
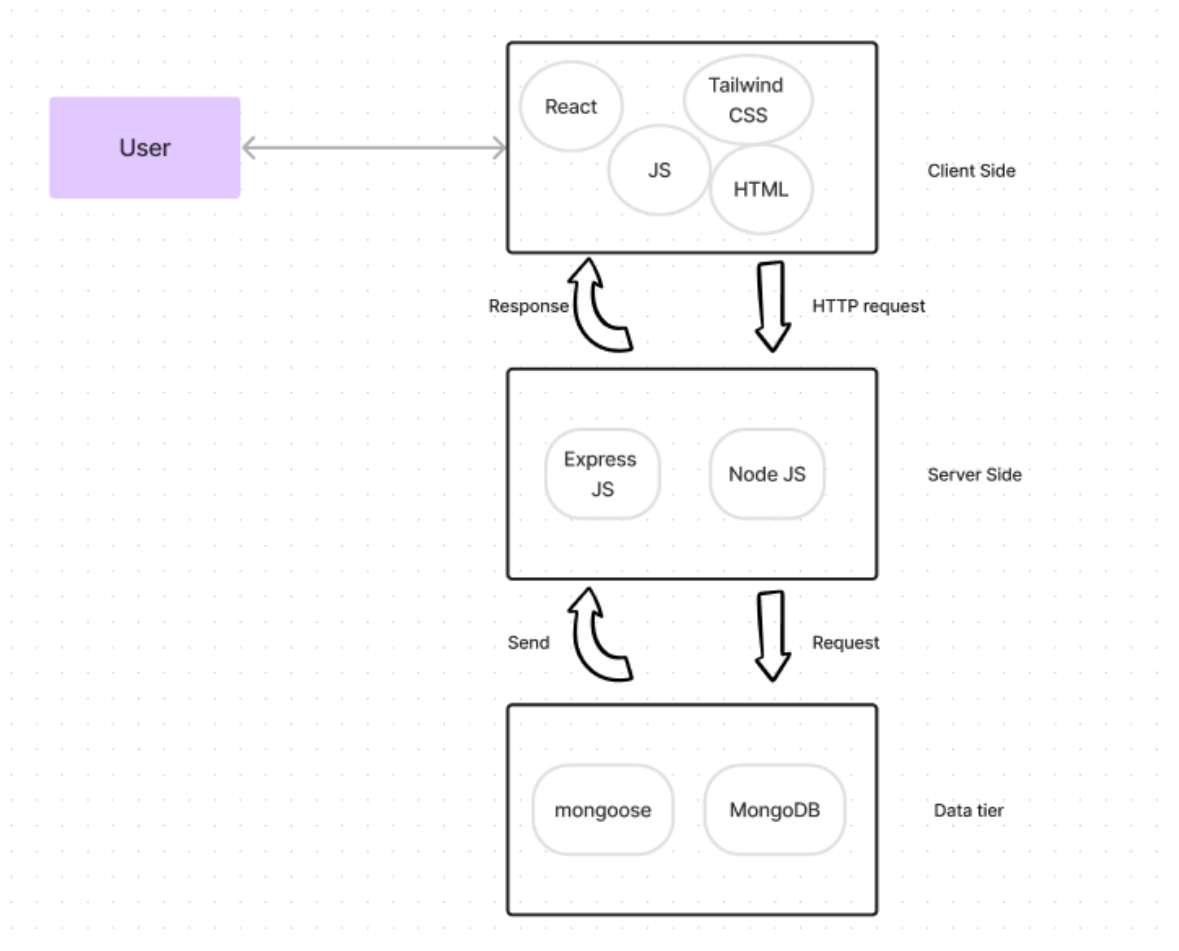
System Decomposition

1. User Interface (React, Tailwind CSS, HTML)

- **Role:** Provides the front-end of the application where users interact. Developed with React.js and Tailwind CSS.
- **Components:**
  - **React Components:** Modular and reusable components for different parts of the UI (e.g., navigation bar, contract objects).
  - **Routing:** Handled using React Router to navigate between different pages.

2. Front-End Logic (JS)

- **Role:** Manages client-side logic and communicates with the back-end server.
- **Components:**
  - **API Services:** JavaScript functions that make HTTP requests to the back-end using Axios API.
  - **Form Validation:** Client-side validation to ensure user inputs are correct before sending them to the server.

3. Server-Side Logic (Express + Node.js)

- **Role:** Handles business logic, processes client requests, interacts with the database, and sends responses back to the client.
- **Components:**
  - **Express Routes:** Define the endpoints for the API and handle HTTP requests.
  - **Controllers:** Contain the logic for handling requests and generating responses.
  - **Middleware:** Functions for processing requests (e.g., authentication).

4. Database Layer (MongoDB + Mongoose)

- **Role:** Manages data storage, retrieval, and manipulation.
- **Components:**
  - **Mongoose Models:** Define the schema and data validation rules for MongoDB collections.
  - **Database Connection:** Configuration and management of the connection to the MongoDB instance.

Error and Exceptional Case Handling Strategy

1. Invalid User Input

- **Front-End Validation:**
    - **Strategy:** Use form validation to ensure that inputs meet the required format and constraints before submission.
    - **Response:** Display error messages next to the form fields to inform the user of the specific issues.
- **Back-End Validation:**
    - **Strategy:** Validate inputs on the server-side using Mongoose validation or custom validation logic.
    - **Response:** Send a 400 Bad Request response with details about the validation errors in JSON.

2. Database Errors

- **Strategy:** Handle database connection errors, query failures, and data integrity issues.
- **Response:**
    - **Connection Errors:** Attempt to reconnect to the database and log the error. Send a 500 Internal Server Error response if the connection cannot be re-established.
    - **Query Failures:** Log the error and send a 500 Internal Server Error response with a generic error message.

3. External System Failures

- **Strategy:** Handle failures when interacting with external services (e.g., third-party APIs).
- **Response:**
    - **Fallback:** Provide fallback mechanisms or alternative flows when the external service is unavailable.
    - **Notification:** Inform the user of the issue and suggest actions they can take.

4. Unhandled Exceptions

- **Strategy:** Capture unhandled exceptions using global error handlers in both the front-end and back-end.
- **Response:**
    - **Front-End:** Show a generic error message and log the error for further investigation.
    - **Back-End:** Log the error details and send a 500 Internal Server Error response with a generic message.

Summary of Error Responses

- **400 Bad Request:** For validation errors and invalid user inputs.

- **401 Unauthorized / 403 Forbidden:** For authentication and authorization failures.
- **404 Not Found:** For requests to non-existent resources.
- **500 Internal Server Error:** For unexpected server errors, database issues, and unhandled exceptions.
- **503 Service Unavailable:** For issues with external systems or services.