CSCC01
System Design Document
Group: The Algoholics

<u>Table of Contents</u>

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

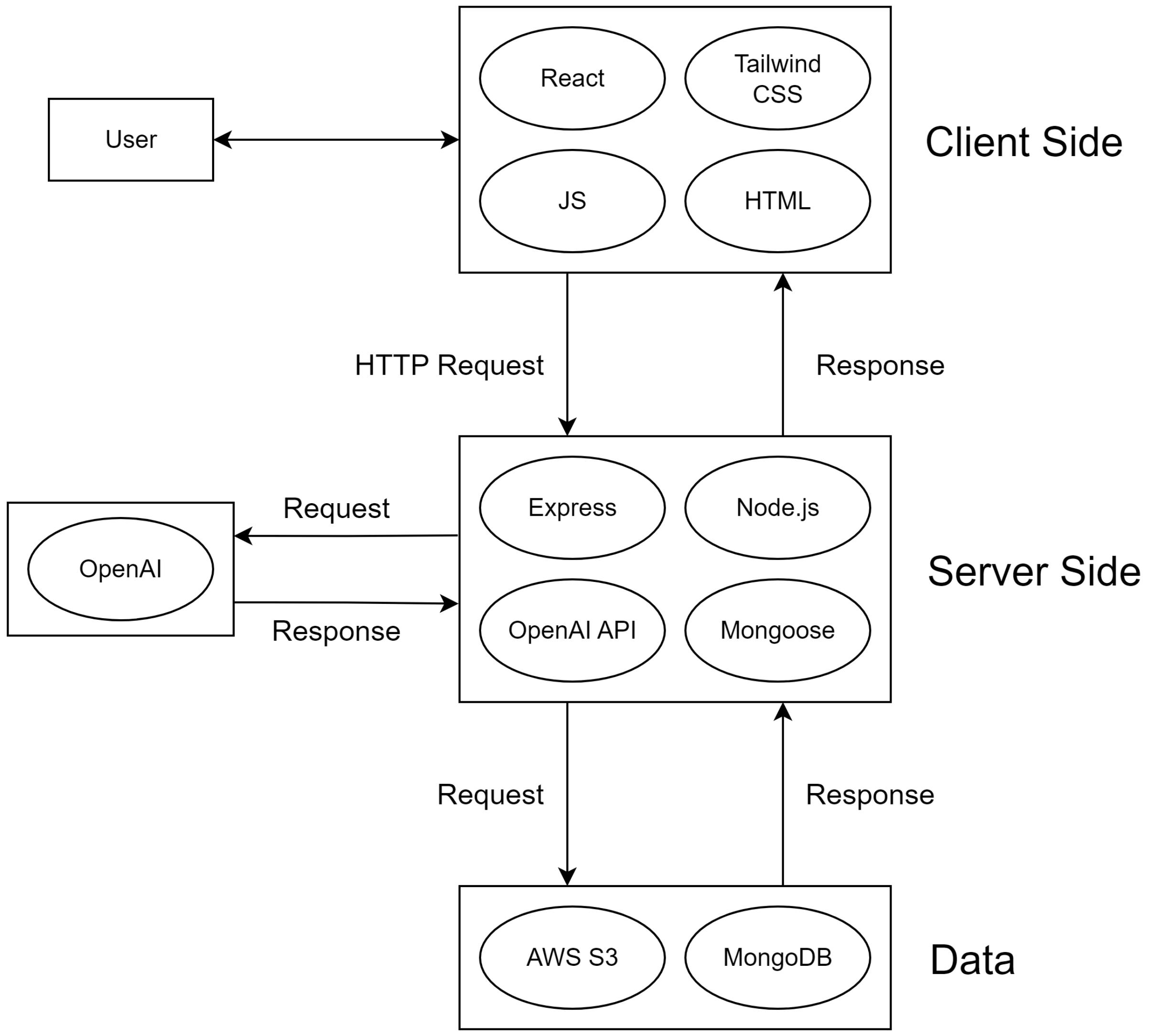
Prettier: A code formatter to ensure consistent code style.

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.
    - **AWS S3:** Stores contract information, with a link to MongoDB.

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.

Pactify API Documentation

Path: /api/user/getUserData

Method: GET

Body: No body

Description: Acquires information about the currently logged-in user using cookies.

Responses:

200:

Description: Successful request

Example:

```
{
    "firstName": "John",
    "lastName": "Doe",
    "email": "john.doe@example.com",
    "password":
    "$2a$10$I.Ic9xfQg3HWXbGBpF05SudqweVYUud0OoiBzuNkN6N
    v3O7.8j62RJH9Su"
}
```

404:

Description: User not found

Example:

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

500:

Description: Error retrieving user data

Example:

```
{
    "message": "Error retrieving user data",
    "error": "error.message"
}
```

Path: /api/user/updateUserName

Method: PATCH

Body:

firstName (String): First name of the user
lastName (String): Last name of the user

Description: Updates the name of the currently logged-in user using cookies.

Responses:

200:

Description: Successful request
Example:

```
{
    "_id": "666e4c9238d1f2a8e1d4482a",
    "email": "email@gmail.com",
    "role": "Basic",
    "firstName": "Sohil",
    "lastName": "Chanana",
    "createdAt": "2024-06-16T02:23:14.072Z",
    "updatedAt": "2024-07-16T06:10:29.065Z",
    "__v": 0
}
```

404:

Description: User not found
Example:

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

500:

Description: Error updating user data
Example:

```
{
    "message": "Error updating user data",
    "error": "error.message"
}
```

Path: /api/user/getUserContracts

Method: GET

Body: No body

Description: Fetches all contracts for the currently logged-in user using cookies.

Responses:

200:

Description: Successful request

Example:

```
{
    "contracts": [
    // List of contract objects
    ]
}
```

500:

Description: Error fetching contracts

Example:

```
{
    "message": "Error fetching contracts",
    "error": "error.message"
}
```

Path: /api/user/updateUserEmail

Method: PATCH

Body:

email (String): New email of the user

Description: Updates the email of the currently logged-in user using cookies.

Responses:

200:

Description: Successful request

Example:

```
{
    "_id": "666e4c9238d1f2a8e1d4482a",
    "email": "newemail@gmail.com",
    "password":
    "$2a$10$KcIY/LmgTRD8yGdIT6C8NuJ9A892U71CZ7ebST/1IGXI
    KguYKJQx2",
    "role": "Basic",
    "firstName": "Sohil",
    "lastName": "Chanana",
    "createdAt": "2024-06-16T02:23:14.072Z",
    "updatedAt": "2024-07-16T06:10:29.065Z",
    "__v": 0
}
```

404:

Description: User not found

Example:

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

500:

Description: Error updating user data

Example:

```
{
    "message": "Error updating user data",
    "error": "error.message"
}
```

Path: /api/user/deleteUser

Method: DELETE

Body: No body

Description: Deletes the currently logged-in user using cookies from the database.

Responses:

- **200:**
  - **Description:** Successful request
  - **Example:**

```
{
        "message": "User successfully deleted",
        "userId": "req.user.id"
}
```

- **400:**
  - **Description:** Error deleting user
  - **Example:**

```
{
        "message": "An error occurred",
        "error": "error.message"
}
```

Path: /api/user/getUserSignature

Method: GET

Body: No body

Description: Retrieves the signature of the currently logged-in user using cookies.

Responses:

200:

Description: Successful request

Example:

```
{
    "signature": "user's signature"
}
```

404:

Description: User not found

Example:

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

500:

Description: Error retrieving signature

Example:

```
{
    "message": "Server error",
    "error": "error.message"
}
```

Path: /api/user/updateUserSignature

Method: PATCH

Body:

signature (String): New signature of the user

Description: Updates the signature of the currently logged-in user using cookies.

Responses:

200:

Description: Successful request

Example:

```
{
    "message": "Signature updated successfully",
    "signature": "new user's signature"
}
```

404:

Description: User not found

Example:

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

500:

Description: Error updating signature

Example:

```
{
    "message": "Server error",
    "error": "error.message"
}
```

Path: /api/auth/register

Method: POST

Body:

email (String): User's email
password (String): User's password (must be at least 6 characters)
firstName (String): User's first name
lastName (String): User's last name

Description: Registers a new user.

Responses:

200:

Description: User successfully created
Example:

```
{
    "message": "User successfully created",
    "user": {
        "email": "newuser@example.com",
        "firstName": "First",
        "lastName": "Last",
        "password": "$2a$10$hashedpassword"
    }
}
```

400:

Description: Password too short
Example:

```
{
    "message": "Password less than 6 characters."
}
```

Description: Email already in use
Example:

```
{
    "message": "Email already in use!"
}
```

Description: User not successfully created
Example:

```
{
    "message": "User not successful created",
    "error": "error.message"
}
```

Path: /api/auth/login

Method: POST

Body:

email (String): User's email
password (String): User's password

Description: Logs in an existing user.

Responses:

200:

Description: User successfully logged in
Example:
```
{
    "message": "User successfully logged in",
    "user": "user._id",
    "token": "jwt_token"
}
```
400:

Description: Email or password missing
Example:
```
{
    "message": "Email or password missing"
}
```
Description: Incorrect credentials
Example:
```
{
    "message": "Login failed, incorrect credentials",
    "error": "User not found"
}
```
Description: Password incorrect
Example:
```
{
    "message": "Login failed, incorrect credentials",
    "error": "Password not found"
}
```
Description: Error during login
Example:
```
{
    "message": "An error occurred",
    "error": "error.message"
}
```

Path: /api/chatGPT

Method: POST

Body:

```
context(String)
message(String)
conversation(Array)(Optional)
    role(String)
    content(String)
```

Description: Makes a request using the chatGPT API with the provided context, message, and conversation history

Responses:

200:

Description: Successful request

Example:

```
{
   "message": [
      {
         "role": "user",
         "content": "Create a simple contract for an NDA"
      },
      {
         "role": "assistant",
         "content": "(Sample GPT Response)"
      }
   ]
}
```

400:

Description: Invalid request

Example:

```
{
        "message": "Invalid request"
}
```

500:

Description: Issue with request to GPT API

Example:

```
{
        "message": "Error generating response"
}
```

Path: /api/uploadFile

Method: POST

Body:

fileName(String)
content(String)

Description: Uploads a file to AWS S3 and creates a new contract entry in the database.

Responses:

200:

Description: File uploaded successfully
Example:

```
{
  "message": "File uploaded successfully",
  "data": {
        "ETag": "\"etag\"",
        "Location":
"https://bucket-name.s3.region.amazonaws.com/folderPrefix/fileName",
        "key": "folderPrefix/fileName",
        "Bucket": "bucket-name"
  }
}
```

500:

Description: Error uploading file
Example:

```
{
  "message": "Error uploading file",
  "error": "error.message"
}
```