**Book Exchange Platform**

**Vidhya**

**2023TM93607**

1. Full application architecture (frontend-backend interaction, API endpoints, database schema).

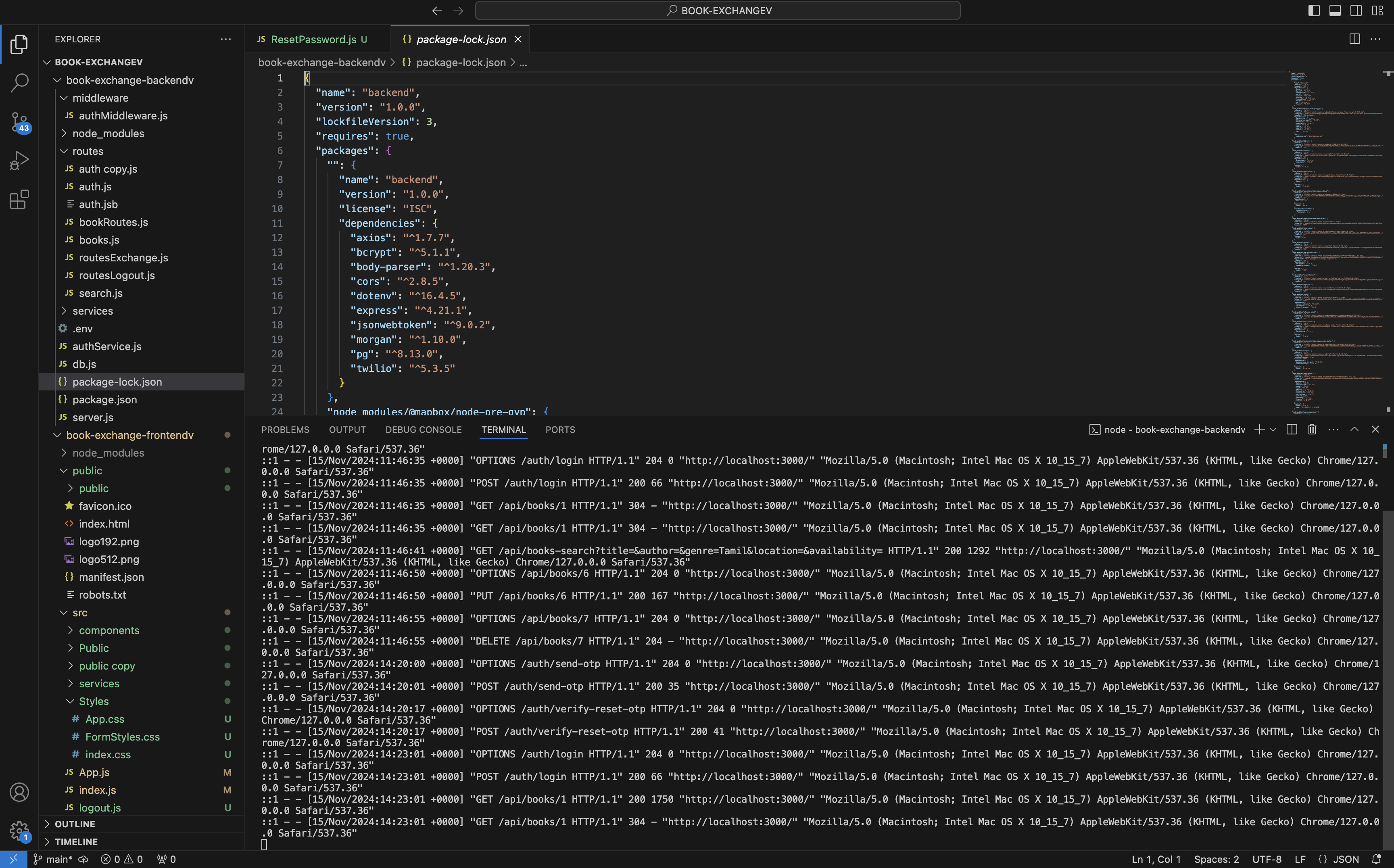
A diagram of a software application

Description automatically generated

TWILIO

OTP,VERITY-OTP,PASSWORD RESET

Successful request-response captured debugging logs for add books/delete books/book search/login/logout/password reset.



1. UI/UX wireframes for the platform’s core features.



1. A Detailed explanation of your approach to implementing scalability and modularity.  
   I have done this setup with a **React frontend**, **Node.js/Express backend**, and **PostgreSQL database**.

**1. Frontend (React)**

* **Component-Based Modularity**:
  + **Reusable Components**: Used React components to modularize the UI. Components like **BookCard**, **SearchBar**, **BookDetails**, and **UserProfile** are reused and tested independently. This keeps the codebase organized and allows rapid updates or new feature additions.
  + **Separation of Concerns**: Kept UI components (presentation) separate from data-fetching components (container). Containers handle API calls to the backend, while presentational components focus on displaying the data, simplifying maintenance.
* **State Management**:
  + **React Query**: For data-fetching and caching, consider **React Query**. It optimizes API calls by caching responses and refreshing only when needed, reducing backend load and improving user experience.

**2. Backend (Node.js/Express)**

* **Layered Architecture**:
  + Organized backend code into a layered architecture: **controllers** for request handling, **services** for business logic, and **models** for database interactions. This separation ensures that each layer has a single responsibility, making the codebase more modular and easier to scale.
  + **API Routes**: Define routes for each feature (e.g., **/books**, **/requests**, **/messages**). Each route handles a specific feature, which allows adding or removing features without affecting the entire codebase.
* **Express Middleware**:
  + Used middleware for common tasks like **authentication**, **logging**, and **error handling**. This modular approach keeps your routes and controllers clean and focuses them on core functionality.
* **Caching with Redis**:
  + Future Implementation plan to use Redis as a caching layer for frequently accessed data, like book listings or user profiles, to reduce database queries and speed up response times. Redis can also cache session data, ensuring fast access to user session details.
* **Token Management:** Uses JSON Web Tokens (JWT) to create, validate, and refresh user sessions. Improves API security

**Purpose of CORS Configuration**

1. **Allow Specific Origins:**
   * **allowedOrigins defines the list of frontend URLs that can access the backend server.**
   * **Example: A React app running on http://localhost:3000 needs to fetch data from the backend. The server checks if this origin is in the allowedOrigins list before allowing the request.**
2. **Enable Credentials:**
   * **Setting credentials: true allows cookies or authentication tokens (like JWT) to be sent with cross-origin requests.**
3. **Custom Handling:**
   * **If the origin of the request matches an item in allowedOrigins, the callback allows the request.**
   * **If the origin is not on the list, the callback returns an error, blocking the request.**

**const allowedOrigins = ['http://localhost:3000', 'http://localhost:3001', 'http://localhost:3002'];**

**app.use(cors({**

**origin: function(origin, callback) {**

**if (!origin || allowedOrigins.indexOf(origin) !== -1) {**

**return callback(null, true);}**

**return callback(new Error('Not allowed by CORS'));**

**},**

**credentials: true,**

**}));**

**Flow of OTP-Based Password Reset through Twilio- 3step Process: Enable Twilio account:Sandbox account(free trial)**

Now, let’s look at the specific flow and files involved when a user requests an OTP-based password reset:

**1. User Requests Password Reset**

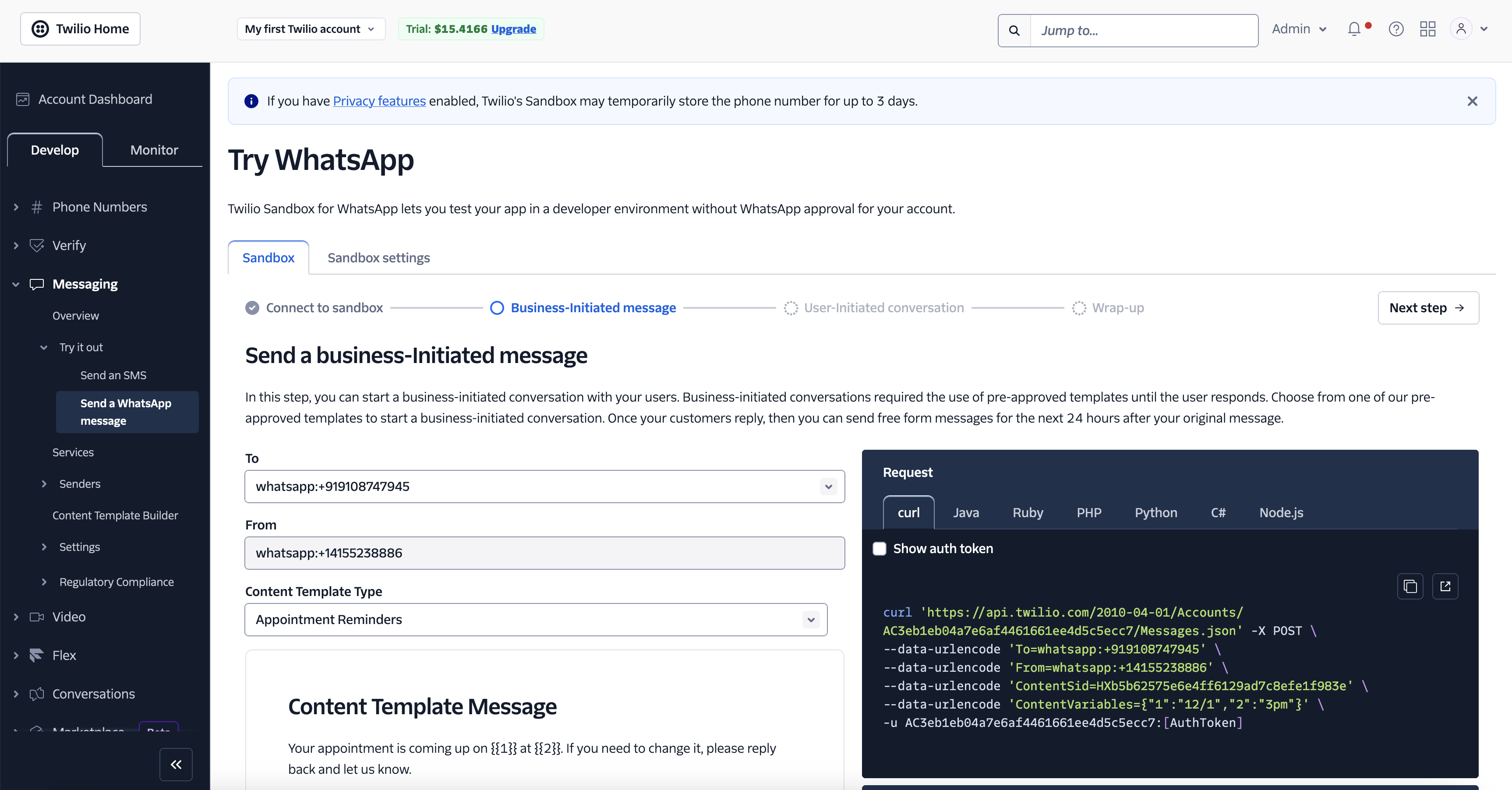
* **Frontend:**
  + ResetPassword.js: The user enters their email and clicks a "Request OTP" button.
  + Service Call: authService.js (frontend) makes a POST request to the backend’s /auth/request-otp endpoint, sending the user's email.
* **Backend:**
  + **authService.js (backend): Handles the request to /auth/request-otp.**
    - Step 1: Verifies if the email is associated with a registered user by querying the PostgreSQL database.
    - Step 2: Generates an OTP (a random numeric code) and stores it temporarily (in memory, cache, or the database) along with an expiry time.
    - Step 3: Sends the OTP to the user's phone number using Twilio.
  + **Twilio Integration:**
    - Setup: The backend uses Twilio’s Node.js SDK to send SMS messages. The Twilio account credentials are stored securely in environment variables.
    - Sending OTP: Twilio’s messages.create() method is called to send the OTP SMS to the user’s registered phone number.

**2. User Enters OTP to Verify and Reset Password**

* **Frontend:**
  + ResetPassword.js: The user enters the OTP they received and the new password.
  + Service Call: authService.js (frontend) makes a POST request to the backend’s /auth/verify-otp endpoint, sending the email, OTP, and new password.
* **Backend:**
  + **authService.js (backend): Processes the /auth/verify-otp request.**
    - Step 1: Checks the OTP and email in the temporary storage (or database) to verify the OTP’s validity and expiry.
    - Step 2: If the OTP is valid, it hashes the new password and updates the user’s record in the PostgreSQL database.
    - Step 3: Sends a response to the frontend indicating a successful password reset.

**3. User Logs In with New Password**

* **Frontend:**
  + Login.js: After resetting the password, the user can log in using their new password. They enter their credentials and click the login button.
  + Service Call: authService.js (frontend) sends a POST request to /auth/login on the backend with the user’s email and new password.
* **Backend:**
  + authService.js (backend): Handles the login request by verifying the credentials.
    - Step 1: Checks the user’s email and password hash in the PostgreSQL database.
    - Step 2: If verified, generates a JSON Web Token (JWT) for the session and sends it back to the frontend.
* **Token Storage:**
  + Login.js: The frontend receives the JWT, stores it in localStorage, and updates the isAuthenticated state.
  + Protected Routes: With isAuthenticated set to true, the user can access protected routes like /books.



A screenshot of a computer screen

Description automatically generated

**3.Database (PostgreSQL)**

* **Database Schema Design**:Organize the database into **normalized tables** (e.g., **Users**, **Books**, **Requests**, **Messages**) to maintain data integrity. Relationships between tables are defined with foreign keys (e.g., **user\_id** in the **Books** table to link books to users).
* **Indexing**: Used indexes on frequently queried fields like title, author, and genre in the **Books** table. Indexing optimizes read-heavy queries, making it ideal for scalable applications with search functionality.
* **Read Replicas for Scalability**: As your user base grows for future, I would consider setting up **read replicas** for handling read-heavy traffic. This allows the primary database to handle only write requests, improving overall performance.
* **Connection Pooling**: In Future I will Use a connection pool in PostgreSQL to manage connections from the Node.js backend efficiently, ensuring each user request doesn’t create a new connection. This approach improves performance and avoids overwhelming the database.

1. **Deployment and Scaling Stratergy.(Future Plans)**

* Containerization with Docker
* Load Balancer
* Horizontal Scaling for Frontend
* Environment Variables for Configurations

**Flow Summary of Files and Functional Responsibilities**

Here’s a summary of how files work together to support the OTP-based password reset functionality:

1. **Frontend**
   * **ResetPassword.js**: Renders the OTP request and verification forms, handles input for email, OTP, and new password, and interacts with authService.js.
   * **authService.js (frontend)**: Manages API calls related to authentication and password reset, including requests for OTP and new password setting.
   * **Login.js**: Allows the user to log in with their new password after resetting it, stores the JWT token in localStorage.
2. **Backend**
   * **server.js**: Sets up the Express server, connects to PostgreSQL, and includes middleware for parsing requests.
   * **authService.js (backend)**: Manages password reset logic, including OTP generation, sending OTP via Twilio, and password update in PostgreSQL.
   * **Twilio Integration**: Configured in the backend using Twilio’s Node.js SDK to send the OTP to the user’s phone.

**Application Flow from Startup to Password Reset Completion**

1. **Startup**:
   * The frontend starts with npm start, loading index.js, which renders App.js into index.html.
   * The backend starts with node server.js, setting up routes and connecting to PostgreSQL.
2. **Password Reset Flow**:
   * The user requests an OTP, handled by ResetPassword.js on the frontend and authService.js on the backend.
   * The backend generates an OTP, sends it via Twilio, and temporarily stores it.
   * The user verifies the OTP, enters a new password, and submits.
   * The backend checks the OTP, updates the password in PostgreSQL, and completes the reset process.
   * The user logs in with the new password, and the application is fully functional for the user with the updated credentials.

This setup ensures a secure password reset process using OTP verification and Twilio SMS for delivery, providing a complete, streamlined user experience. By organizing project around these files, React application becomes more modular, maintainable, and scalable.