**Step-by-Step Plan to Rebuild the Website Using Microservice Architecture**

**1. Understand the Current Website**

* Analyze existing PHP scripts and database schema:
  + Review ceb\_management (17).sql to understand database tables and relationships.
  + Identify the purpose of each PHP script (e.g., membership.php for member management, loan.php for loans, etc.).
  + Map user flows, business logic, and dependencies.

**2. Technologies to Use**

**Backend:**

* **Spring Boot**: For building modular, RESTful microservices.
* **MySQL**: For relational database management.
* **Spring Data JPA**: For ORM and database interaction.
* **Spring Security**: For authentication and role-based access control.
* **Swagger/OpenAPI**: For API documentation.
* **Spring Cloud**: For microservice communication and discovery.
  + Netflix Eureka (Service Discovery)
  + Spring Cloud Gateway (API Gateway)
* **RabbitMQ/Kafka**: For asynchronous communication between services (if needed).

**Frontend:**

* **React**: For building the user interface.
* **Redux**: For state management.
* **Axios**: For making HTTP requests to the backend.
* **React Router**: For client-side routing.
* **TailwindCSS/Bootstrap**: For UI styling (optional).

**DevOps & Deployment:**

* **Docker**: For containerization.
* **Kubernetes**: For orchestration and scaling (optional for advanced deployment).
* **Jenkins/GitHub Actions**: For CI/CD.
* **Postman**: For testing APIs.

**3. Define Microservices**

Break down the application into smaller, independent services based on core functionalities:

| **Microservice** | **Responsibilities** | **Database Tables** |
| --- | --- | --- |
| **User Service** | Handles authentication, user registration, and role management. | Users, Roles |
| **Membership Service** | Manages membership forms, requests, and approval processes. | Memberships, Members |
| **Loan Service** | Handles loan applications, processing, and status tracking. | Loans |
| **Savings Service** | Manages savings accounts, contributions, and balances. | Savings, Contributions |
| **Notification Service** | Sends notifications (emails, SMS) for loan approvals, membership updates, etc. | Notifications |
| **Report Service** | Generates reports (PDFs or HTML) for membership, savings, and loans. | (Fetches data from other services) |

**4. Database Design**

* Use MySQL to implement the database schema.
* Normalize tables to maintain data integrity.
* Create relationships using foreign keys and map them in JPA.

**5. Backend Implementation**

1. **Set Up a Spring Boot Project**:
   * Use Spring Initializr to generate projects for each microservice.
   * Add dependencies for Spring Web, JPA, MySQL, Spring Security, etc.
2. **Develop Each Microservice**:
   * Build REST APIs for CRUD operations in each service.
   * Implement DTOs, Entities, and Services layers.
   * Use Spring Data JPA for database interaction.
3. **Service Communication**:
   * Use **REST** for synchronous communication.
   * For asynchronous tasks (e.g., notifications), integrate **RabbitMQ** or **Kafka**.
4. **API Gateway**:
   * Set up a **Spring Cloud Gateway** to route requests to the appropriate services.
   * Centralize authentication and rate limiting.
5. **Service Discovery**:
   * Use **Eureka Server** for registering and discovering services dynamically.
6. **Authentication & Authorization**:
   * Implement **JWT-based authentication** in the User Service.
   * Use Spring Security for role-based access control.
7. **Testing**:
   * Write unit tests for each microservice (JUnit, Mockito).
   * Test APIs using Postman.

**6. Frontend Implementation**

1. **Set Up React**:
   * Use Create React App or Vite for setup.
   * Install necessary libraries (Redux Toolkit, Axios, React Router).
2. **Component Design**:
   * Break UI into reusable components (e.g., Navbar, LoanForm, MemberList).
   * Follow a modular folder structure:

css

Copy code

src/

├── components/

├── pages/

├── services/

├── store/

└── utils/

1. **State Management**:
   * Use Redux Toolkit for centralized state management.
   * Store user authentication state, API data (e.g., membership details).
2. **API Integration**:
   * Use Axios to consume backend APIs.
   * Handle authentication with JWT tokens in headers.
3. **UI Design**:
   * Use TailwindCSS or Bootstrap for responsive styling.
   * Ensure user-friendly navigation (e.g., sidebar, modals).
4. **Testing**:
   * Write unit tests with Jest.
   * Use Cypress for end-to-end testing.

**7. CI/CD Pipeline**

1. **Dockerize Services**:
   * Create a Dockerfile for each microservice and the frontend.
   * Use docker-compose to manage containers locally.
2. **Automate Builds**:
   * Set up Jenkins or GitHub Actions to automate builds, tests, and deployments.
3. **Deploy**:
   * Use AWS/GCP/Azure or on-premises servers for deployment.
   * Use Kubernetes for container orchestration if scaling is required.

**8. Documentation and Maintenance**

1. Document APIs using **Swagger/OpenAPI**.
2. Maintain code quality with linters and SonarQube.
3. Monitor services using tools like **Prometheus** and **Grafana**.

**1. User Service APIs**

Handles authentication, user management, and roles.

| **API** | **HTTP Method** | **Endpoint** | **Description** |
| --- | --- | --- | --- |
| Register User | POST | /api/users/register | Register a new user. |
| Login User | POST | /api/users/login | Authenticate user and return a JWT token. |
| Logout User | POST | /api/users/logout | Invalidate the user's session or token. |
| Get User Profile | GET | /api/users/profile | Get details of the currently logged-in user. |
| Update User Profile | PUT | /api/users/profile | Update user profile details. |
| Change Password | POST | /api/users/change-password | Allow the user to change their password. |
| List Users | GET | /api/users | List all registered users (admin-only). |
| Get User by ID | GET | /api/users/{id} | Fetch details of a specific user. |
| Delete User | DELETE | /api/users/{id} | Remove a user from the system (admin-only). |

**2. Membership Service APIs**

Manages membership applications, approvals, and status.

| **API** | **HTTP Method** | **Endpoint** | **Description** |
| --- | --- | --- | --- |
| Apply for Membership | POST | /api/memberships/apply | Submit a membership application. |
| Get Membership Status | GET | /api/memberships/status | Get the status of the logged-in user's membership. |
| Approve Membership | PUT | /api/memberships/{id}/approve | Approve a membership application (admin-only). |
| Reject Membership | PUT | /api/memberships/{id}/reject | Reject a membership application (admin-only). |
| List Membership Applications | GET | /api/memberships/applications | List all membership applications (admin-only). |
| Get Member Details | GET | /api/memberships/{id} | Fetch details of a specific member. |
| Update Membership Info | PUT | /api/memberships/{id} | Update a member's information. |

**3. Loan Service APIs**

Handles loan requests, approvals, and payments.

| **API** | **HTTP Method** | **Endpoint** | **Description** |
| --- | --- | --- | --- |
| Request Loan | POST | /api/loans/request | Submit a loan request. |
| Get Loan Details | GET | /api/loans/{id} | Get details of a specific loan. |
| List Loans for User | GET | /api/loans/user | List all loans for the logged-in user. |
| List All Loan Requests | GET | /api/loans | List all loan requests (admin-only). |
| Approve Loan | PUT | /api/loans/{id}/approve | Approve a loan request (admin-only). |
| Reject Loan | PUT | /api/loans/{id}/reject | Reject a loan request (admin-only). |
| Make Loan Payment | POST | /api/loans/{id}/pay | Make a payment towards a loan. |
| Get Loan Repayment History | GET | /api/loans/{id}/payments | Fetch payment history for a specific loan. |

**4. Savings Service APIs**

Handles savings contributions and balances.

| **API** | **HTTP Method** | **Endpoint** | **Description** |
| --- | --- | --- | --- |
| Add Savings Contribution | POST | /api/savings/contribute | Add a contribution to savings. |
| Get Savings Balance | GET | /api/savings/balance | Get the current savings balance for the user. |
| List Savings Transactions | GET | /api/savings/transactions | Get a list of all savings transactions for the user. |
| List All Savings Accounts | GET | /api/savings/accounts | List all savings accounts (admin-only). |

**5. Notification Service APIs**

Handles email or SMS notifications for loan approvals, membership updates, etc.

| **API** | **HTTP Method** | **Endpoint** | **Description** |
| --- | --- | --- | --- |
| Send Notification | POST | /api/notifications/send | Send a notification (email or SMS). |
| Get Notification History | GET | /api/notifications | List notifications sent to the user. |

**6. Report Service APIs**

Generates reports for admin users.

| **API** | **HTTP Method** | **Endpoint** | **Description** |
| --- | --- | --- | --- |
| Generate Membership Report | GET | /api/reports/membership | Generate a report of all members. |
| Generate Loan Report | GET | /api/reports/loans | Generate a report of all loans and their statuses. |
| Generate Savings Report | GET | /api/reports/savings | Generate a report of all savings accounts. |

**7. Audit Service APIs (Optional)**

Tracks actions and changes for compliance or monitoring purposes.

| **API** | **HTTP Method** | **Endpoint** | **Description** |
| --- | --- | --- | --- |
| Log Action | POST | /api/audit/log | Log a user or system action. |
| Get Audit Logs | GET | /api/audit/logs | Retrieve audit logs (admin-only). |

**8. Admin APIs**

For managing system-level operations.

| **API** | **HTTP Method** | **Endpoint** | **Description** |
| --- | --- | --- | --- |
| Health Check | GET | /api/admin/health | Check the health of the service. |
| Clear Cache | POST | /api/admin/cache/clear | Clear application-level caches. |

**Authentication & Security**

All APIs will require JWT authentication, with role-based access control for sensitive or admin-only endpoints. Here's an example:

1. **Public Endpoints**:
   * /api/users/register
   * /api/users/login
2. **Authenticated Endpoints**:
   * Most other endpoints require a valid JWT token in the Authorization header:

makefile

Copy code

Authorization: Bearer <JWT\_TOKEN>

1. **Role-Based Access**:
   * Use roles like USER, ADMIN, etc., to restrict access to certain APIs:
     + Example: Only users with ADMIN role can access /api/memberships/applications.