**Purpose and Overview**

The goal is to develop a **simple bank application** that allows users (customers) to manage their bank accounts and perform transactions. Unlike a traditional monolithic approach, this application will leverage **Kafka** to implement an **event-driven** design for key operations such as deposits, withdrawals, and transfers.

The core features include:

* User registration and authentication
* Account creation and management
* Transaction handling (deposits, withdrawals, transfers)
* Transaction history viewing
* **Event-driven** processing via Kafka
* Optional admin features (view all accounts, suspend/close accounts, generate reports)
* Optional notifications and analytics based on consumed Kafka events

**1.2 User Roles**

1. **Customer**
   * Can create a user account (register).
   * Can log in and log out.
   * Can create/view bank accounts.
   * Can deposit and withdraw funds.
   * Can view transaction history.
   * Optional: Can transfer funds to other accounts.
2. **Administrator** (Optional, depending on project scope)
   * Can log in to an admin dashboard.
   * Can view all customer accounts.
   * Can suspend or close accounts.
   * Can generate reports (e.g., total funds in the system).

**1.3 Functional Requirements**

1. **User Registration and Authentication**
   * Users can register with username/email and password.
   * Passwords must be securely hashed (e.g., BCrypt).
   * Authentication can use JWT or session-based tokens.
   * Endpoints for registration, login, and logout.
2. **Account Management**
   * A registered user can create one or more bank accounts.
   * Each account has a unique account number, a display name (e.g., “Checking”), and a current balance.
   * Users can retrieve a list of their own accounts.
3. **Transactions**
   * **Deposit**: Increase an account’s balance.
   * **Withdraw**: Decrease an account’s balance (if sufficient funds).
   * **Transfer** (optional): Move funds between two accounts (could be the same user or different users).
   * Each transaction is recorded with details (type, amount, timestamp, account ID, etc.).
4. **Transaction History**
   * The user can retrieve a chronological list of transactions for any of their accounts.
   * Transactions are stored in the database for quick lookup.
   * (Optional) Provide filters by date range or transaction type.
5. **Event-Driven Architecture (Kafka Integration)**
   * **Transaction Events**: Whenever a deposit, withdrawal, or transfer occurs, publish a **transaction event** to a Kafka topic (e.g., “transactions”).
     + The event message should contain relevant data (transaction type, amount, account ID, timestamp, user ID).
   * **Consumers (Optional)**:
     + A separate microservice (or the same backend) could **consume** transaction events for:
       - **Notifications** (email or push alerts)
       - **Analytics** (building real-time dashboards or aggregated metrics)
       - **Auditing** (persist events in a separate store for compliance)
6. **Admin Features (Optional)**
   * Admins can view all accounts in the system.
   * Admins can suspend or close an account.
   * Admins can generate simple reports (e.g., total funds in all accounts).

**1.4 Non-Functional Requirements**

1. **Performance**
   * Must handle concurrent user requests with acceptable response times (under 2 seconds for standard operations).
   * Kafka throughput should handle the expected volume of transaction events without bottlenecks.
2. **Security**
   * All passwords must be hashed and salted.
   * All API endpoints that deal with user or account data must require authentication.
   * Use HTTPS/TLS to secure communication channels.
   * Optionally configure Kafka for secure communication (SASL, ACLs) if required.
3. **Reliability & Availability**
   * Design the system to handle database or Kafka failures gracefully (retry logic, circuit breakers, etc.).
   * Aim for minimal downtime during maintenance.
   * If using microservices, each service should be independently deployable.
4. **Maintainability**
   * Follow standard best practices and a layered architecture on the backend (controller, service, repository).
   * Organize React code cleanly (components, hooks, or Redux for state).
   * Write clear documentation and use logging to facilitate debugging.
5. **Scalability**
   * The system should be able to scale horizontally.
   * Kafka brokers can be added to handle higher message throughput.
   * Database can be scaled using replication or sharding if necessary.

**1.5 High-Level Architecture**

1. **Frontend (React)**
   * Single-page application developed with React.
   * Communicates with the backend via RESTful APIs (Axios or Fetch).
   * Manages state using React Context, Redux, or other state-management solutions.
   * Uses routing for pages like /login, /register, /dashboard, /account/:id.
2. **Backend (Java + Spring Boot or Similar)**
   * **API Layer**: Controllers exposing REST endpoints for registration, login, account CRUD, transactions, etc.
   * **Service Layer**: Business logic handling user authentication, transaction validation, etc.
   * **Repository Layer**: Database access, typically using JPA/Hibernate (or another ORM).
   * **Kafka Integration**:
     + **Producer**: Publishes “transaction events” to the “transactions” topic.
     + (Optional) **Consumer**: Consumes events for additional processing (notifications, analytics).
3. **Kafka**
   * A Kafka cluster with at least one topic named “transactions”.
   * Transaction events are published whenever a deposit, withdrawal, or transfer is processed.
   * Additional topics (optional) can be added for other events (e.g., account creation, account status changes).
4. **Database**
   * A relational database (MySQL, PostgreSQL) is recommended for storing user, account, and transaction records.
   * Alternatively, NoSQL (MongoDB, Cassandra) could be used depending on your requirements.
   * Each transaction is stored in the DB for quick reference and generating statements or history views.

**2. Tickets (Stories/Tasks)**

Below is a list of potential tickets for development. You can refine, split, or combine them further based on your team’s workflow.

**2.1 User Registration and Authentication**

1. **[BACKEND] Create User Entity and Repository**
   * **Description**: Implement a User entity with fields like id, username/email, hashedPassword, and roles.
   * **Acceptance Criteria**:
     + The user table is created in the database.
     + CRUD operations can be performed on the User entity.
2. **[BACKEND] Implement Password Hashing**
   * **Description**: Use a secure hashing algorithm (e.g., BCrypt) for storing user passwords.
   * **Acceptance Criteria**:
     + Plaintext passwords are never stored.
     + Validation checks the hash on login.
3. **[BACKEND] Implement JWT-based Authentication**
   * **Description**: Add a security configuration to issue JWTs upon successful login and validate them on protected endpoints.
   * **Acceptance Criteria**:
     + Valid JWT is returned on successful login.
     + Protected endpoints return 401 Unauthorized if no valid token is provided.
4. **[FRONTEND] Create Registration Page**
   * **Description**: A form that collects user data (email, password), calls the backend to register.
   * **Acceptance Criteria**:
     + On successful registration, user receives a confirmation.
     + Handle validation errors gracefully (e.g., “password too short”).
5. **[FRONTEND] Create Login Page**
   * **Description**: A form that collects credentials and obtains a JWT from the backend.
   * **Acceptance Criteria**:
     + Stores JWT in local or session storage.
     + Redirects to dashboard on successful login.
     + Shows an error message for invalid credentials.

**2.2 Account Management**

1. **[BACKEND] Create Account Entity and Repository**
   * **Description**: Implement an Account entity with fields like id, accountNumber, accountName, balance, userId.
   * **Acceptance Criteria**:
     + Account table is created in the DB.
     + Relationship with User is established (one-to-many or many-to-one).
2. **[BACKEND] Implement Create Account Endpoint**
   * **Description**: Endpoint for authenticated users to create a new account (e.g., “Checking”, “Savings”).
   * **Acceptance Criteria**:
     + Returns the newly created account with a unique account number.
     + User must be authenticated to create an account.
3. **[FRONTEND] Dashboard Page – List Accounts**
   * **Description**: After login, shows a user’s accounts with balances.
   * **Acceptance Criteria**:
     + Displays account name, account number, and current balance.
     + Clicking an account navigates to an account detail page.

**2.3 Transactions (With Kafka Publishing)**

1. **[BACKEND] Transaction Entity and Repository**
   * **Description**: Implement a Transaction entity with fields like id, accountId, type (deposit, withdrawal, transfer), amount, timestamp.
   * **Acceptance Criteria**:
     + Transaction table is created in the DB.
     + Basic CRUD operations possible.
2. **[BACKEND] Deposit Endpoint**
   * **Description**: Allows a user to deposit an amount to a specific account.
   * **Acceptance Criteria**:
     + Validates the request and updates the account balance.
     + Creates a transaction record in the DB (type = deposit).
     + Publishes a **transaction event** to the Kafka “transactions” topic.
3. **[BACKEND] Withdraw Endpoint**
   * **Description**: Allows a user to withdraw an amount from an account (if sufficient balance).
   * **Acceptance Criteria**:
     + Checks if the account has enough balance.
     + Creates a transaction record (type = withdrawal).
     + Publishes a **transaction event** to the Kafka “transactions” topic.
     + Returns an error if insufficient funds.
4. **[BACKEND] Transfer Endpoint (Optional)**
   * **Description**: Moves funds between two accounts, possibly belonging to different users.
   * **Acceptance Criteria**:
     + Decreases source account, increases destination account.
     + Creates two transaction records (transfer-out, transfer-in).
     + Publishes corresponding event(s) to Kafka with necessary info.
5. **[FRONTEND] Account Details Page – Deposit/Withdrawal Forms**
   * **Description**: Page displaying an account’s details, along with forms to deposit or withdraw.
   * **Acceptance Criteria**:
     + Users can input an amount and submit a deposit/withdraw request.
     + Shows updated balance after a successful transaction.
     + Displays error on insufficient funds or invalid input.
6. **[FRONTEND] Transaction History Table**
   * **Description**: Shows a list of recent transactions for an account.
   * **Acceptance Criteria**:
     + Displays date, transaction type, amount, new balance.
     + Sorted in descending order by date.
7. **[FRONTEND] Transfer Funds Page (Optional)**
   * **Description**: Form to initiate a transfer between two accounts.
   * **Acceptance Criteria**:
     + Validates source and destination accounts, as well as the amount.
     + On success, displays confirmation and updates balances.

**2.4 Kafka Integration**

1. **[DEVOPS/BACKEND] Set Up Kafka Environment**
   * **Description**: Stand up a local or remote Kafka cluster (and Zookeeper if needed) for development.
   * **Acceptance Criteria**:
     + Kafka is running and reachable.
     + Topics (e.g., “transactions”) are created or can be auto-created.
2. **[BACKEND] Add Kafka Configuration**
   * **Description**: In the Java application (e.g., Spring Boot), configure producer settings (bootstrap servers, key/value serializers, etc.).
   * **Acceptance Criteria**:
     + Application connects to Kafka successfully.
     + Producer config can be injected via application properties or environment variables.
3. **[BACKEND] Implement Kafka Producer for Transaction Events**
   * **Description**: After each transaction (deposit, withdrawal, transfer), publish a message to the “transactions” topic.
   * **Acceptance Criteria**:
     + Message includes key data (accountId, userId, transactionType, amount, timestamp).
     + Handle success/failure (logging, retry if needed).
4. **[BACKEND / Separate Service] Implement Kafka Consumer (Optional)**
   * **Description**: A consumer service that subscribes to the “transactions” topic to process events for:
     + Notifications, analytics, auditing, etc.
   * **Acceptance Criteria**:
     + Consumes messages reliably (handles duplicates, errors).
     + Processes or stores event data (e.g., in a separate DB for analytics).
5. **[BACKEND / Separate Service] Notification Service (Optional)**
   * **Description**: Consumes transaction events and sends emails or push notifications to users.
   * **Acceptance Criteria**:
     + Subscribes to “transactions” topic.
     + Sends notifications for deposits, withdrawals, transfers.
     + Errors are logged, possibly retried.

**2.5 Admin Dashboard (Optional)**

1. **[BACKEND] Admin User Role & Permissions**
   * **Description**: Extend user entity to differentiate admin vs. customer.
   * **Acceptance Criteria**:
     + Admin-only endpoints are secured.
     + Attempted access by non-admin returns 403 Forbidden.
2. **[BACKEND] View All Accounts Endpoint**
   * **Description**: Admin can fetch a list of all accounts and balances.
   * **Acceptance Criteria**:
     + Only accessible to admin users.
     + Returns relevant account and user details.
3. **[BACKEND] Suspend/Close Account**
   * **Description**: Admin can change an account’s status to suspended or closed.
   * **Acceptance Criteria**:
     + Suspended/closed accounts block further transactions.
     + DB updates reflect new status.
4. **[FRONTEND] Admin Dashboard Page**
   * **Description**: Displays a list of user accounts and allows admin actions (suspend, close, etc.).
   * **Acceptance Criteria**:
     + Only reachable by admin.
     + Shows account details and user info (where relevant).
5. **[BACKEND] Admin Reporting (Optional)**
   * **Description**: Provide endpoints that return aggregated data (e.g., total funds, number of active accounts).
   * **Acceptance Criteria**:
     + Query the database for aggregated data.
     + Return results in JSON for display on admin dashboard.

**2.6 Infrastructure & DevOps**

1. **[DEVOPS] Containerization**
   * **Description**: Create Dockerfiles for the backend (Java) and frontend (React). Consider Docker Compose for Kafka + Zookeeper.
   * **Acceptance Criteria**:
     + Images build successfully.
     + Containers can be run locally with environment variables for Kafka brokers, database connections, etc.
2. **[DEVOPS] CI/CD Setup**
   * **Description**: Configure a pipeline (GitHub Actions, Jenkins, etc.) to build and test the code on each commit.
   * **Acceptance Criteria**:
     + Automatic builds, tests, and reports on pull requests.
     + Optionally, integration tests against a local Kafka instance.
3. **[DEVOPS] Deployment Scripts**
   * **Description**: Scripts or Infrastructure-as-Code (Terraform, CloudFormation) to deploy the application to a cloud environment (AWS, Azure, GCP), including managed or self-managed Kafka.
   * **Acceptance Criteria**:
     + Environment variables are properly managed (database URLs, Kafka broker addresses, etc.).
     + Automated deployment pipeline.