Wallet Backend

Project Overview

This is a **Personal Finance Management System** built as a class project for **COMP 4411 - Programming Languages** at Lakehead University. The project demonstrates various programming paradigms including Object-Oriented, Procedural, and Concurrent programming using Kotlin and the Ktor framework.

DEVELOPMENT DOCS: documentation.md

Please refer to documentation.md

This project was created using the Ktor Project Generator.

🚀 Setup & Installation

Prerequisites

Before running this project, ensure you have the following installed:

1. **Java Development Kit (JDK) 21**
    - Download from Oracle or use OpenJDK
    - Verify installation: `java -version`
2. **Docker Desktop**
    - Download and install Docker Desktop
    - Ensure Docker is running before proceeding
3. **Gradle** (Optional - project includes Gradle Wrapper)
    - The project uses Gradle Wrapper (`./gradlew`), so manual installation is not required

Step 1: Clone the Repository

```
git clone <repository-url>
cd wallet-backend
```

Step 2: Start PostgreSQL Database with Docker

This project uses PostgreSQL running in a Docker container for consistent development environments.

Start the PostgreSQL container:

```
docker run --name my-wallet-db -e POSTGRES_PASSWORD=mysecretpassword -p
5432:5432 -d postgres
```

Verify the container is running:

```
docker ps
```

You should see `my-wallet-db` in the list of running containers.

Important Database Configuration:

- **Host:** `localhost`
- **Port:** `5432`
- **Database:** `postgres`
- **Username:** `postgres`
- **Password:** `mysecretpassword`

> **Note:** These credentials are configured in `src/main/kotlin/Databases.kt`. Update them if you use different credentials.

Step 3: Configure Database Connection

The database connection is configured in `Databases.kt`:

```
object Database {
    fun connect(): Connection {
        val url = "jdbc:postgresql://localhost:5432/postgres"
        val user = "postgres"
        val password = "mysecretpassword"
        return DriverManager.getConnection(url, user, password)
    }
}
```

If you need to change credentials:

1. Open `src/main/kotlin/Databases.kt`
2. Update the `url`, `user`, and `password` values
3. Ensure they match your Docker PostgreSQL configuration

Step 4: Build the Project

```
./gradlew build
```

This will:

- Download all dependencies
- Compile Kotlin code

- Run tests
- Generate build artifacts

Step 5: Run the Application

```
./gradlew run
```

The server will start on **http://localhost:8080**

Expected console output:

```
2024-11-XX XX:XX:XX.XXX [main] INFO  Application - Application started in
0.303 seconds.
2024-11-XX XX:XX:XX.XXX [main] INFO  Application - Responding at
http://0.0.0.0:8080
```

Step 6: Database Initialization (Automatic)

On first startup, the application will:

1. **Create custom PostgreSQL types:**
    - `transaction_type` ENUM (`'expense'`, `'income'`)
    - `period_type` ENUM (`'daily'`, `'weekly'`, `'monthly'`, `'yearly'`)
2. **Create tables if they don't exist:**
    - `users` - User accounts with authentication
    - `transactions` - Financial transaction records
    - `budgets` - Budget limits and tracking
3. **Seed demo data** (10 demo users with transactions and budgets)

Verify database tables:

Access PostgreSQL interactive terminal:

```
docker exec -it my-wallet-db psql -U postgres
```

List all tables:

```
\dt
```

Exit PostgreSQL terminal:

```
\q
```

Step 7: Test the API

Health check:

```
curl http://localhost:8080/
```

Expected response: `Hello World!`

Get transactions for user 1:

```
curl http://localhost:8080/transactions/1
```

Using Postman or any API client:

- Import the OpenAPI documentation from: `http://localhost:8080/openapi`
- Test all CRUD endpoints for Users, Transactions, and Budgets

---

🐋 Docker Commands Reference

Command	Description
`docker ps`	List running containers
`docker stop my-wallet-db`	Stop the database container
`docker start my-wallet-db`	Start the database container
`docker rm my-wallet-db`	Remove the container (must be stopped first)
`docker exec -it my-wallet-db psql -U postgres`	Access PostgreSQL CLI
`docker logs my-wallet-db`	View database logs

⚙️ System Requirements

- **OS:** macOS, Linux, or Windows
- **RAM:** Minimum 4GB (8GB recommended)
- **Disk Space:** ~500MB for dependencies and Docker images
- **Java:** JDK 21
- **Docker:** Latest stable version

🔧 Troubleshooting

Port 5432 Already in Use

If you have PostgreSQL already running locally:

```
# Stop local PostgreSQL service (macOS)
brew services stop postgresql

# Or use a different port for Docker
docker run --name my-wallet-db -e POSTGRES_PASSWORD=mysecretpassword -p
5433:5432 -d postgres
```

Then update the port in `Databases.kt` to `5433`.

Database Connection Failed

1. Verify Docker container is running: `docker ps`
2. Check container logs: `docker logs my-wallet-db`
3. Ensure credentials in `Databases.kt` match your Docker setup
4. Try restarting the container: `docker restart my-wallet-db`

Expected Errors on First Run

You may see error messages about types already existing:

```
Error executing SQL statement: CREATE TYPE transaction_type...
Error executing SQL statement: CREATE TYPE period_type...
```

This is normal! The application uses `CREATE TYPE IF NOT EXISTS` logic. On subsequent runs, these types already exist, causing harmless errors that are caught and logged.

Gradle Build Issues

```
# Clean and rebuild
./gradlew clean build

# If Gradle wrapper fails to download
chmod +x gradlew
./gradlew wrapper --gradle-version=8.5
```

Technology Stack

Backend Framework & Language

- **Kotlin** - Primary programming language
- **Ktor 3.0+** - Asynchronous web framework for building REST APIs
- **Gradle** - Build automation and dependency management

Database

- **PostgreSQL** - Relational database management system
- **JDBC** - Direct database connectivity without ORM

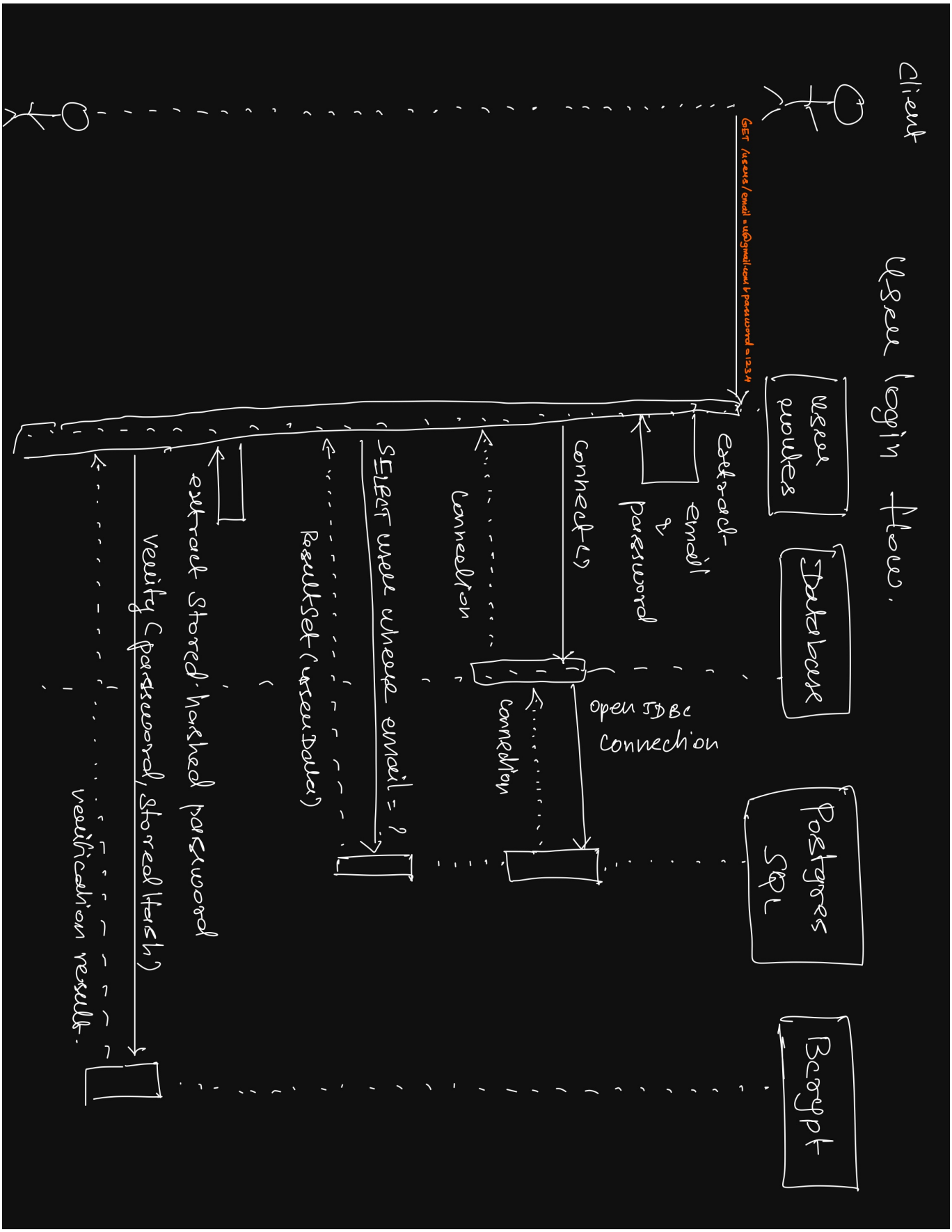
Security & Utilities

- **BCrypt** (at.favre.lib:bcrypt:0.10.2) - Password hashing
- **kotlinx.serialization** - JSON serialization/deserialization
- **Logback** - Logging framework

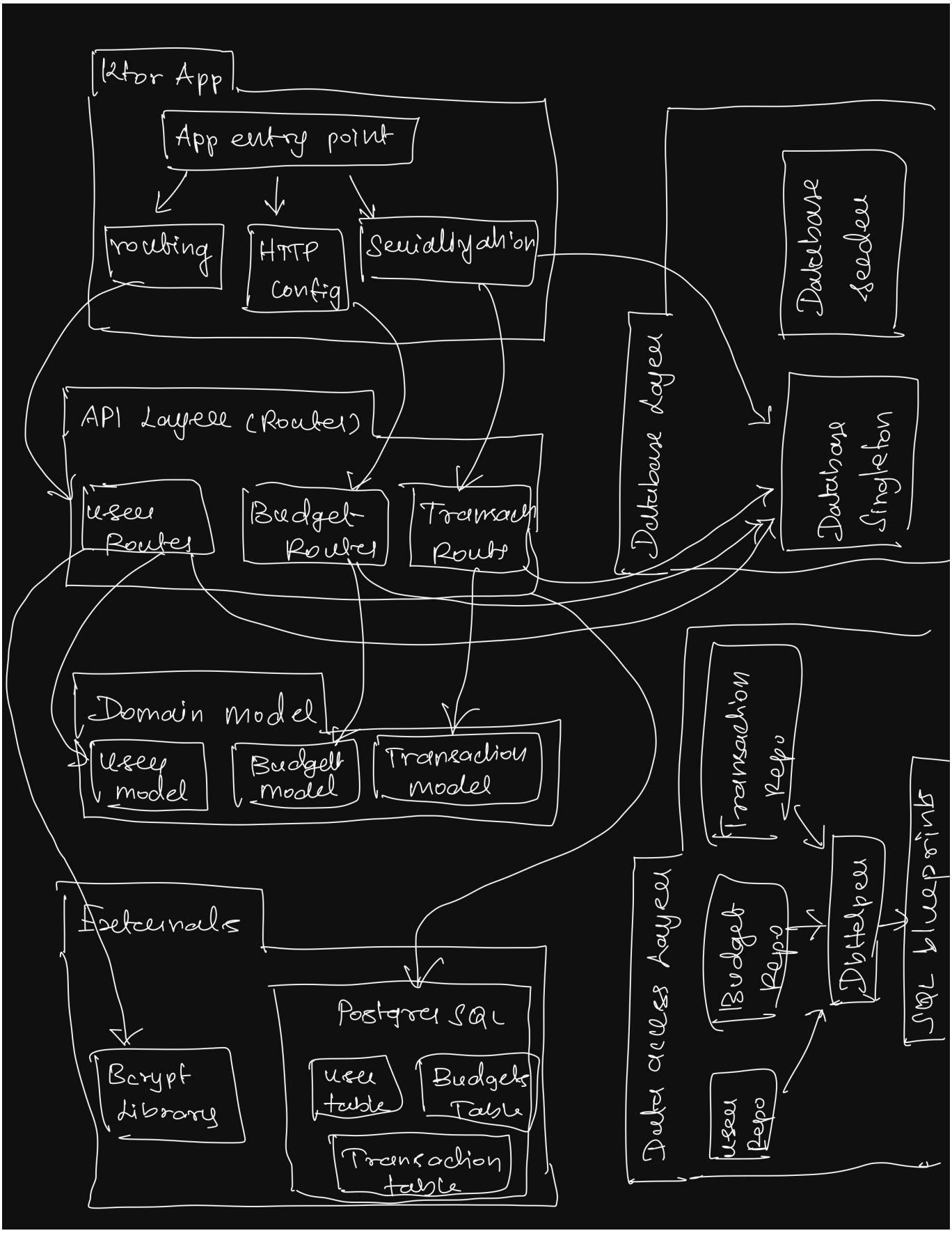
Server

- **Netty** - High-performance asynchronous event-driven network application framework

system design architecture and flow


Client
User login flow.
GET /users/email=ug@gmail.com&password=1234
User Handler
Database
Postgres SQL
Bcrypt
extract-email & password
connect()
Connection
open JDBC Connection
connection
SELECT user where email = ?
ResultSet (userData)
extract Stored Hashed password
Verify (password, stored Hash)
verification result.

Create Transaction Flow. (Similar to Budgets).
client
TransactionRoutes
Transaction Model
Database
PostgreSQL cluster
POST
receive<Transaction>
Deserialize JSON
connect()
open conn
connection
connection
prepare SQL insert stmt.
Bind ?? params to prepared Stmt.
execute capable
insert into table
Rows affected
close connection
201
client



Programming Paradigms Demonstrated

1. Object-Oriented Programming (OOP)

The project uses OOP principles extensively with data classes, singleton objects, and encapsulation.

Example - Data Models:

```
// User.kt - Data class with properties
@Serializable
data class User(
    val userId: Int? = null,
    val firstName: String,
    val lastName: String,
    val email: String,
    val password: String,
    val createdAt: String? = null,
    val updatedAt: String? = null,
)

// Transaction.kt - Domain model
@Serializable
data class Transaction(
    val transactionId: Int? = null,
    val userId: Int,
    val title: String,
    val category: String,
    val transactionType: String,
    val amount: String,
    val date: String
)
```

Example - Repository Pattern:

```
// UserRepository.kt - Encapsulation of data access logic
class UserRepository {
    fun createUser(first: String, last: String, email: String,
hashedPassword: String): Int {
        return DbHelper.query(
            sql = SqlBlueprints.INSERT_USER,
            prepare = { ps ->
                ps.setString(1, first)
                ps.setString(2, last)
                ps.setString(3, email)
                ps.setString(4, hashedPassword)
            },
            map = { rs ->
                rs.next()
                rs.getInt("user_id")
            }
        )
    }
}
```

Example - Singleton Object Pattern:

```
// Database.kt - Singleton object for database connection
object Database {
    fun connect(): Connection {
        val url = "jdbc:postgresql://localhost:5432/postgres"
        val user = "postgres"
        val password = "mysecretpassword"
        return DriverManager.getConnection(url, user, password)
    }

    fun init() {
        // Initialize database tables
    }
}
```

2. Procedural Programming

The project uses procedural approaches for sequential operations and data processing.

Example - Routing Configuration:

```
// Routing.kt - Procedural route setup
fun Application.configureRouting() {
    routing {
        userRouting()
        transactionRouting()
        budgetRouting()

        get("/") {
            call.respondText("Hello World!")
        }
    }
}
```

Example - Database Operations:

```
// TransactionRoutes.kt - Procedural database insert
post {
    val transaction = call.receive<Transaction>()
    val connection = Database.connect()

    val sql = """
        INSERT INTO transactions (user_id, title, category,
transaction_type, amount, date)
        VALUES (?, ?, ?, ?::transaction_type, ?, ?::timestamp with time
zone)
    """.trimIndent()

    connection.use { conn ->
```

```
        val statement = conn.prepareStatement(sql)
        statement.setInt(1, transaction.userId)
        statement.setString(2, transaction.title)
        statement.setString(3, transaction.category)
        statement.setString(5, transaction.transactionType)
        statement.setBigDecimal(6, BigDecimal(transaction.amount))
        statement.executeUpdate()
    }

    call.respond(HttpStatusCode.Created, "Transaction stored
successfully")
}
```

3. Concurrent Programming

The project demonstrates thread-based concurrency with proper synchronization mechanisms.

Example - Thread-Safe Demo Data Generation:

```
// DemoDataSeeder.kt - Concurrent data seeding with thread safety
object DemoDataSeeder {
    // Concurrency primitives
    private val userInsertLock = ReentrantLock()
    private val userIdListRWLock = ReentrantReadWriteLock()
    private val sharedUserIdList = mutableListOf<Int>()

    // Thread-safe write operation
    private fun addUserIdThreadSafely(newId: Int) {
        userIdListRWLock.writeLock().withLock {
            sharedUserIdList.add(newId)
        }
    }

    // Thread-safe read operation
    private fun safelyGetAllUserIdsSnapshot(): List<Int> {
        userIdListRWLock.readLock().withLock {
            return sharedUserIdList.toList()
        }
    }

    // Creating concurrent worker threads
    private fun createConcurrentWorkerThreadsForAllUsers(userIds:
List<Int>) {
        val allThreads = mutableListOf<Thread>()

        for (userId in userIds) {
            // One transaction thread per category
            for (category in categories) {
                val t = Thread({
                    generateTransactions(userId, category)
                }, "Tx-User$userId-$category")
```

```
                t.start()
                allThreads.add(t)
            }

            // One budget thread per user
            val b = Thread({
                generateBudgets(userId)
            }, "Budget-User$userId")
            b.start()
            allThreads.add(b)
        }

        // Wait for all threads to complete
        allThreads.forEach { it.join() }
    }
}
```

4. Functional Programming Elements

Kotlin's functional programming features are used throughout the project.

Example - Higher-Order Functions:

```
// DbHelper.kt - Generic query function using lambdas
fun <T> query(
    sql: String,
    prepare: (PreparedStatement) -> Unit,
    map: (ResultSet) -> T
): T {
    Database.connect().use { connection ->
        connection.prepareStatement(sql).use { ps ->
            prepare(ps)
            ps.executeQuery().use { rs ->
                return map(rs)
            }
        }
    }
}
```

Key Features

- **User Management**: Registration, login with BCrypt password hashing, profile updates
- **Transaction Tracking**: Create, read, update, and delete financial transactions
- **Budget Management**: Set and monitor spending budgets by category
- **RESTful API**: Clean REST endpoints for all operations
- **Thread-Safe Demo Data**: Concurrent data seeding with proper synchronization
- **PostgreSQL Integration**: Direct JDBC usage with custom SQL queries
- **CORS Support**: Cross-origin resource sharing for frontend integration

API Endpoints

Users

- `POST /users` - Create new user
- `GET /users/login` - Authenticate user
- `PUT /users/{id}` - Update user profile

Transactions

- `POST /transactions` - Create transaction
- `GET /transactions/{userId}` - Get all transactions for a user
- `PUT /transactions/{id}` - Update transaction
- `DELETE /transactions/{id}` - Delete transaction

Budgets

- `POST /budgets` - Create budget
- `GET /budgets/{userId}` - Get all budgets for a user
- `PUT /budgets/{id}` - Update budget
- `DELETE /budgets/{id}` - Delete budget

Database Schema

The application uses PostgreSQL with the following custom types and tables:

Custom ENUM Types

- `transaction_type`: `'expense' | 'income'`
- `period_type`: `'daily' | 'weekly' | 'monthly' | 'yearly'`

Tables

- **users**: User accounts with authentication
- **transactions**: Financial transaction records
- **budgets**: Budget limits and tracking

Useful Links

- Ktor Documentation
- Ktor GitHub page
- The Ktor Slack chat. You'll need to request an invite to join.

Features

Here's a list of features included in this project:

Name	Description
Routing	Provides a structured routing DSL

Name	Description
OpenAPI	Serves OpenAPI documentation
CORS	Enables Cross-Origin Resource Sharing (CORS)
kotlinx.serialization	Handles JSON serialization using kotlinx.serialization library
Content Negotiation	Provides automatic content conversion according to Content-Type and Accept headers
Postgres	Adds Postgres database to your application

Building & Running

To build or run the project, use one of the following tasks:

Task	Description
`./gradlew test`	Run the tests
`./gradlew build`	Build everything
`./gradlew run`	Run the server

If the server starts successfully, you'll see the following output:

```
2024-12-04 14:32:45.584 [main] INFO  Application - Application started in
0.303 seconds.
2024-12-04 14:32:45.682 [main] INFO  Application - Responding at
http://0.0.0.0:8080
```