Tasks 1-5 Documentation

This document provides beginner-friendly documentation for Tasks 1-5 in the Midas Core project. Each task builds upon the previous one, implementing features for a transaction processing system using Spring Boot, Kafka, and JPA.

Task 1: Project Setup

Objective

Set up the Spring Boot project with required dependencies for building a transaction processing system using Kafka, JPA, and H2 database.

Requirements

- Add Spring Boot starters for web, data JPA, Kafka
- Add H2 database for in-memory storage
- Add testing dependencies including Spring Kafka test and Testcontainers
- Ensure basic Maven project structure
- Run tests to verify setup

Steps

1. Update pom.xml with necessary dependencies
2. Verify project structure follows Maven conventions
3. Run tests to ensure everything is configured correctly

Files to Edit/Create

- pom.xml: Add dependencies for Spring Boot (web, data-jpa), Spring Kafka, H2, testing libraries

Testing

To run all tests: `./mvnw test`

To run individual test for Task 1: `./mvnw test -Dtest=TaskOneTests`

Verification: Ensure tests pass without errors, indicating dependencies are correctly added and project is set up.

Complete Code Implementation

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.2.5</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.jpmc</groupId>
    <artifactId>midas-core</artifactId>
    <version>1.0.0</version>
    <name>midas-core</name>
    <description>Midas Core</description>
    <properties>
        <java.version>17</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
            <version>3.2.5</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
            <version>3.2.5</version>
            <version>3.2.5</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.kafka</groupId>
            <artifactId>spring-kafka</artifactId>
            <version>3.1.4</version>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <version>2.2.224</version>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <version>3.2.5</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.kafka</groupId>
            <artifactId>spring-kafka-test</artifactId>
            <version>3.1.4</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.testcontainers</groupId>
            <artifactId>kafka</artifactId>
            <version>1.19.1</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

application.yml

```yaml
 # application.yml
server:
  port: 33400
spring:
  kafka:
    kafka-topic: midas-topic
    producer:
      # Key Serializer (usually String or Long)
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
      # Value Serializer MUST be set to the Spring Kafka JSON Serializer
      value-serializer: org.springframework.kafka.support.serializer.JsonSerializer
    consumer:
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer: org.springframework.kafka.support.serializer.JsonDeserializer
      properties:
        spring.json.trusted.packages: com.jpmc.midascore.foundation
  datasource:
    url: jdbc:h2:mem:testdb
    driver-class-name: org.h2.Driver
  jpa:
    hibernate:
      ddl-auto: update
```

MidasCoreApplication.java

```java
 package com.jpmc.midascore;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
public class MidasCoreApplication {

    public static void main(String[] args) {
        SpringApplication.run(MidasCoreApplication.class, args);
    }

    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }

}
```

Task 2: Implement Transaction Processing via Kafka

Objective

Implement a Kafka listener component that processes incoming transaction messages, validates them, calculates incentives via an external API, updates user balances, and records transactions in the database.

Requirements

- Listen to Kafka topic for transaction messages
- Validate sender and recipient exist and sender has sufficient balance
- Call external incentive API to get transaction incentive
- Update sender and recipient balances (sender deducts amount, recipient adds amount + incentive)
- Save transaction record to database
- Log balance updates for user "wilbur" for debugging

Steps

1. Create a `@Component` class `TransactionKafkaListener`

2. Annotate a method with `@KafkaListener` for the transaction topic
3. Inject required repositories and `RestTemplate`
4. Implement validation logic
5. Make HTTP POST to incentive API
6. Update balances and save entities
7. Add logging for wilbur's balance changes

Files to Edit/Create

- src/main/java/com/jpmc/midascore/component/TransactionKafkaListener.java (create)

Code Snippet

```
@Component
public class TransactionKafkaListener {
    // Inject dependencies

    @KafkaListener(topics = "${spring.kafka.kafka-topic}", groupId = "midas-group")
    public void listenForTransactions(Transaction transaction) {
        // Validation
        // API call
        // Balance updates
        // Save records
        // Logging
    }
}
```

Complete Code Implementation

```
package com.jpmc.midascore.component;
import com.jpmc.midascore.foundation.Transaction;
import com.jpmc.midascore.foundation.Incentive;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Component;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.client.RestTemplate;
import com.jpmc.midascore.repository.UserRepository;
import com.jpmc.midascore.repository.TransactionRecordRepository;
import com.jpmc.midascore.entity.UserRecord;
import com.jpmc.midascore.entity.TransactionRecord;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@Component
public class TransactionKafkaListener {
    private static final Logger logger = LoggerFactory.getLogger(TransactionKafkaListener.class);

    @Autowired
    private UserRepository userRepository;

    @Autowired
    private TransactionRecordRepository transactionRecordRepository;

    @Autowired
    private RestTemplate restTemplate;

    @KafkaListener(topics = "${spring.kafka.kafka-topic}", groupId = "midas-group")
    public void listenForTransactions(Transaction transaction) {
        UserRecord sender = userRepository.findById(transaction.getSenderId());
        UserRecord recipient = userRepository.findById(transaction.getRecipientId());
        if (sender != null && recipient != null && sender.getBalance() >= transaction.getAmount()) {
            try {
                Incentive incentive = restTemplate.postForObject("http://localhost:8080/incentive", transaction, Incentive.class);
                TransactionRecord tr = new TransactionRecord();
                tr.setSender(sender);
                tr.setRecipient(recipient);
                tr.setRecipient(recipient);
                tr.setAmount(transaction.getAmount());
                tr.setIncentive(incentive.getAmount());
                transactionRecordRepository.save(tr);
                sender.setBalance(sender.getBalance() - transaction.getAmount());
                recipient.setBalance(recipient.getBalance() + transaction.getAmount() + incentive.getAmount());
                userRepository.save(sender);
                userRepository.save(recipient);
                // Log balance updates for wilbur
                if ("wilbur".equals(sender.getName())) {
                    logger.info("Wilbur (sender) balance after transaction: {}", sender.getBalance());
                }
                if ("wilbur".equals(recipient.getName())) {
                    logger.info("Wilbur (recipient) balance after transaction: {}", recipient.getBalance());
                }
            } catch (Exception e) {
                logger.error("Failed to process transaction", e);
                // If API call fails, discard the transaction
            }
        } else {
            logger.warn("Transaction discarded: sender={}, recipient={}, amount={}, senderBalance={}",
                sender != null ? sender.getName() : "null",
                recipient != null ? recipient.getName() : "null",
                transaction.getAmount(),
                sender != null ? sender.getBalance() : "null");
        }
        // If conditions fail, discard the transaction
    }
}
```

Testing

Run individual test: `./mvnw test -Dtest=TaskTwoTests`

Verification: Observe logs for wilbur's balance updates after transactions. Ensure transactions are processed correctly and balances are updated.

Note: Ensure the transaction-incentive-api.jar is running on port 8080 for the external API call.

Task 3: Implement User Population

Objective

Create components to populate the database with user data from a file, and ensure transaction processing correctly updates balances for multiple users.

Requirements

- Load user data from a file (name, initial balance)
- Save users to database
- Transactions should update balances correctly
- Query specific user's balance after transactions

Steps

1. Create `DatabaseConduit` component to save users
2. Create `UserPopulator` component to load and parse user data
3. Ensure `UserRepository` has `findByName` method
4. Run transactions and verify balance updates

Files to Edit/Create

- src/main/java/com/jpmc/midascore/component/DatabaseConduit.java (create)
- src/test/java/com/jpmc/midascore/UserPopulator.java (create)
- src/main/java/com/jpmc/midascore/repository/UserRepository.java (edit: add findByName method)

Code Snippets

DatabaseConduit.java

```
@Component
public class DatabaseConduit {
    private final UserRepository userRepository;

    public DatabaseConduit(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public void save(UserRecord userRecord) {
        userRepository.save(userRecord);
    }
}
```

UserPopulator.java

```
@Component
public class UserPopulator {
    // Inject FileLoader and DatabaseConduit

    public void populate() {
        String[] userLines = fileLoader.loadStrings("/test_data/lkjhgfdsa.hjkl");
        for (String userLine : userLines) {
            String[] userData = userLine.split(", ");
            UserRecord user = new UserRecord(userData[0], Float.parseFloat(userData[1]));
            databaseConduit.save(user);
        }
    }
}
```

UserRepository.java

```
public interface UserRepository extends CrudRepository<UserRecord, Long> {
    UserRecord findById(long id);
    UserRecord findByName(String name);
}
```

Complete Code Implementation

DatabaseConduit.java

```
package com.jpmc.midascore.component;

import com.jpmc.midascore.entity.UserRecord;
import com.jpmc.midascore.repository.UserRepository;
import org.springframework.stereotype.Component;

@Component
public class DatabaseConduit {
    private final UserRepository userRepository;

    public DatabaseConduit(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public void save(UserRecord userRecord) {
        userRepository.save(userRecord);
    }

}
```

UserPopulator.java

```
 package com.jpmc.midascore;

import com.jpmc.midascore.component.DatabaseConduit;
import com.jpmc.midascore.entity.UserRecord;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class UserPopulator {
    @Autowired
    private FileLoader fileLoader;

    @Autowired
    private DatabaseConduit databaseConduit;

    public void populate() {
        String[] userLines = fileLoader.loadStrings("/test_data/lkjhgfdsa.hjkl");
        for (String userLine : userLines) {
            String[] userData = userLine.split(", ");
            UserRecord user = new UserRecord(userData[0], Float.parseFloat(userData[1]));
            databaseConduit.save(user);
        }
    }
}
```

UserRepository.java

```
 package com.jpmc.midascore.repository;

import com.jpmc.midascore.entity.UserRecord;
import org.springframework.data.repository.CrudRepository;

public interface UserRepository extends CrudRepository<UserRecord, Long> {
    UserRecord findById(long id);
    UserRecord findByName(String name);
}
```

Testing

Run individual test: `./mvnw test -Dtest=TaskThreeTests`

Verification: Check that users are populated from file, transactions processed, and waldorf's final balance is queried correctly.

Note: Ensure the transaction-incentive-api.jar is running on port 8080 for the external API call.

Task 4: Implement Balance Querying by Name

Objective

Add functionality to query a user's balance by their name, including logging the floored balance value.

Requirements

- Query user by name from repository
- Return balance, handle non-existent users
- Log both exact and floored balance

Steps

1. Ensure `UserRepository` has `findByName` method (already done in Task 3)
2. In test, query wilbur's balance and log it

Files to Edit/Create

- None (uses existing UserRepository)

Code Snippet

```java
UserRecord wilbur = userRepository.findByName("wilbur");
if (wilbur != null) {
    double balance = wilbur.getBalance();
    int flooredBalance = (int) Math.floor(balance);
    logger.info("Wilbur's final balance: {}", balance);
    logger.info("Floored balance: {}", flooredBalance);
}
```

Complete Code Implementation

The balance query logic is implemented in the test method as follows:

```java
// Query wilbur's final balance
UserRecord wilbur = userRepository.findByName("wilbur");
if (wilbur != null) {
    double balance = wilbur.getBalance();
    int flooredBalance = (int) Math.floor(balance);
    logger.info("----------------------------------------------------------");
    logger.info("----------------------------------------------------------");
    logger.info("----------------------------------------------------------");
    logger.info("Wilbur's final balance: {}", balance);
    logger.info("Floored balance: {}", flooredBalance);
    logger.info("----------------------------------------------------------");
    logger.info("----------------------------------------------------------");
    logger.info("----------------------------------------------------------");
} else {
    logger.error("Wilbur not found!");
}
```

Testing

Run individual test: `./mvnw test -Dtest=TaskFourTests`

Verification: Observe logs showing wilbur's final balance and floored balance after transactions.

Task 5: Implement Balance REST Controller

Objective

Create a REST endpoint to query user balances by ID, returning a structured response.

Requirements

- GET endpoint `/balance` with `userId` parameter
- Return `Balance` object with user's balance
- Handle non-existent users (return balance 0)
- Support querying multiple users

Steps

1. Create `Balance` foundation class
2. Create `BalanceController` with GET mapping
3. Inject `UserRepository`
4. Implement balance retrieval logic

Files to Edit/Create

- src/main/java/com/jpmc/midascore/foundation/Balance.java (create)
- src/main/java/com/jpmc/midascore/controller/BalanceController.java (create)

Code Snippets

Balance.java

```java
public class Balance {
    private double amount;

    public Balance(double amount) {
        this.amount = amount;
    }

    // getters, setters, toString
}
```

BalanceController.java

```java
@RestController
public class BalanceController {
    private final UserRepository userRepository;

    public BalanceController(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @GetMapping("/balance")
    public Balance getBalance(@RequestParam long userId) {
        UserRecord user = userRepository.findById(userId);
        if (user != null) {
            return new Balance(user.getBalance());
        } else {
            return new Balance(0);
        }
    }
}
```

Complete Code Implementation

Balance.java

```java
package com.jpmc.midascore.foundation;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

@JsonIgnoreProperties(ignoreUnknown = true)
public class Balance {
    private float amount;

    public Balance() {
    }

    public Balance(float amount) {
        this.amount = amount;
    }

    public float getAmount() {
        return amount;
    }

    public void setAmount(float amount) {
        this.amount = amount;
    }

    @Override
    public String toString() {
        return "Balance {amount=" + amount + "}";
    }
}
```

BalanceController.java

```
 package com.jpmc.midascore.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import com.jpmc.midascore.repository.UserRepository;
import com.jpmc.midascore.foundation.Balance;
import com.jpmc.midascore.entity.UserRecord;

@RestController
public class BalanceController {

    private final UserRepository userRepository;

    public BalanceController(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @GetMapping("/balance")
    public Balance getBalance(@RequestParam long userId) {
        UserRecord user = userRepository.findById(userId);
        if (user != null) {
            return new Balance(user.getBalance());
        } else {
            return new Balance(0);
        }
    }
}
```

Testing

Run individual test: `./mvnw test -Dtest=TaskFiveTests`

Verification: Ensure the REST endpoint returns correct balances for queried user IDs (0-12), with 0 for non-existent users.