# 25장

# 스프링 트랜잭션 기능 사용하기

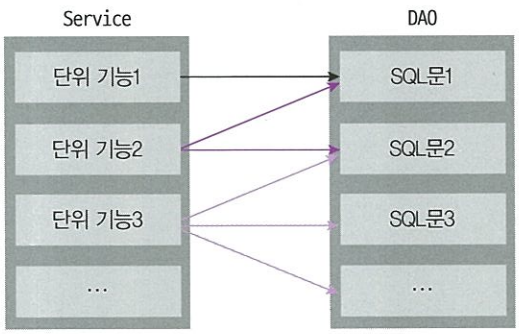
# 25.1 트랜잭션 기능

스프링은 트랜잭션 기능을 마이바티스 기능과 연동해서 사용합니다. 트랜잭션 기능은 XML 파일에서 설정하는 방법과 애너테이션을 이용하는 방법이 있습니다. XML로 설정하는 방법은 설정 파일이 복잡해지면 불편하므로 현재는 애너테이션으로 트랜잭션을 적용하는 방법을 더 선호합니다. 따라서 이 장에서는 애너테이션을 이용해 트랜잭션 기능을 구현해 보겠습니다.

트랜잭션(Transaction)은 여러 개의 DML 명령문을 하나의 논리적인 작업 단위로 묶어서 관리하는 것으로, All 또는 Nothing 방식으로 작업 단위가 처리됩니다. 즉, SQL 명령문들이 모두 정상적으로 처리되었다면 모든 작업의 결과를 데이터베이스에 영구 반영(commit)하지만 그중 하나라도 잘못된 것이 있으면 모두 취소(rollback)합니다. 일단 웹 애플리케이션의 구조와 기능이 실행되는 과정을 보면 트랜잭션이 실제로 어떻게 동작하는지를 쉽게 이해할 수 있습니다.

그림 25-1에 일반적인 애플리케이션 Service 클래스의 메서드 구조를 나타내었습니다.

❤ 그림 25-1 일반적인 웹 애플리케이션의 Service 클래스와 DAO 클래스 구조




Service 클래스의 각 메서드가 애플리케이션의 단위 기능을 수행합니다. 단위 기능1은 DAO 클래스의 SQL문 하나로 기능을 수행하는 반면에 단위 기능2나 단위 기능3은 여러 개의 SQL문을 묶어서 작업을 처리합니다. 그런데 묶어서 작업을 처리할 때 어느 하나의 SQL문이라도 잘못되면 이전에 수행한 모든 작업을 취소해야만 작업의 일관성이 유지됩니다.

따라서 트랜잭션은 각 단위 기능 수행 시 이와 관련된 데이터베이스 연동 작업을 한꺼번에 묶어서 관리한다는 개념입니다.

보통 웹 애플리케이션에서 묶어서 처리하는 단위 기능은 다음과 같습니다.

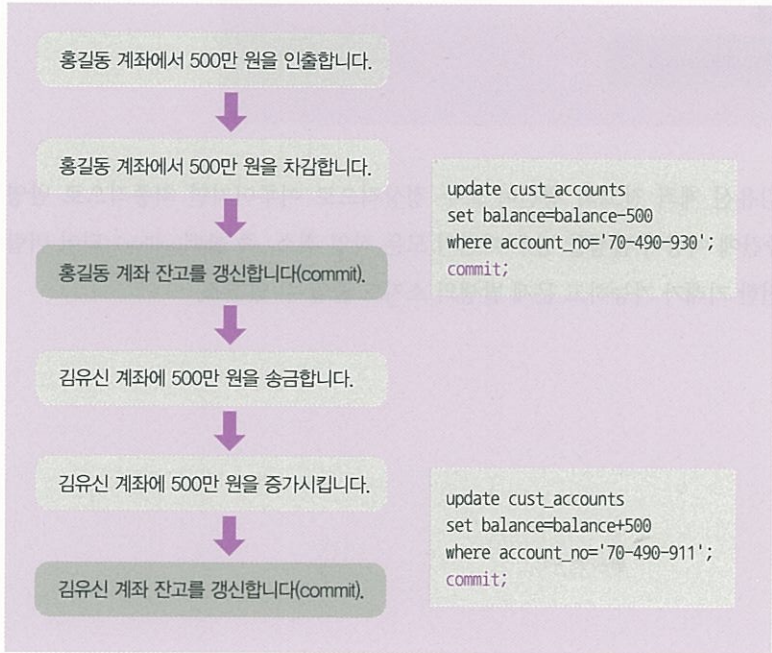
- 게시판 글 조회 시 해당 글을 조회하는 기능과 조회 수를 갱신하는 기능
- 쇼핑몰에서 상품 주문 시 주문 상품을 테이블에 등록하는 기능과 주문자의 포인트를 갱신하는 기능
- 은행에서 송금 시 송금자의 잔고를 갱신하는 기능과 수신자의 잔고를 갱신하는 기능

JAVA WEB

## 25.2 은행 계좌 이체를 통한 트랜잭션 기능

트랜잭션의 개념을 좀 더 이해하기 쉽게 두 예금자 사이의 계좌 이체 과정을 예로 들어 살펴보겠습니다. 그림 25-2는 트랜잭션을 적용하지 않고 홍길동이 김유신에게 500만 원을 인터넷 뱅킹으로 계좌 이체하는 과정을 나타낸 것입니다.

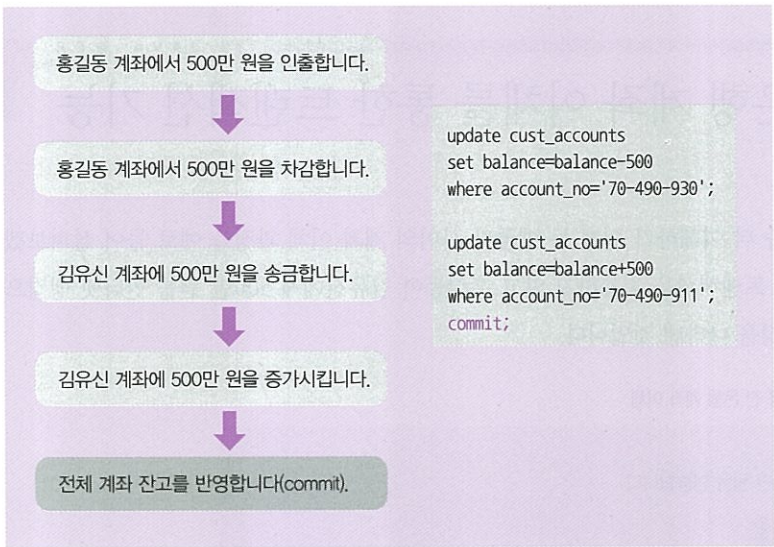
❯ 그림 25-2 트랜잭션 적용 전 은행 계좌 이체



그런데 홍길동의 계좌에서 500만 원이 인출되고 커밋한 후 김유신 계좌의 잔고(balance)를 갱신하려고 할 때 시스템에 이상이 생기면 어떻게 될까요? 김유신의 계좌가 갱신되지 않으면 다음 날 김유신은 홍길동에게 자신의 계좌에 500만 원이 입금되지 않았다고 말하겠지요. 그런데 이미 홍길동의 계좌는 반영이 되었으므로 홍길동은 김유신에게 자신은 분명히 500만 원을 송금했다고 말할 것입니다. 즉, 두 사람 사이에 엄청난 문제가 발생하겠죠.

그림 25-3은 트랜잭션을 적용한 계좌 이체 과정입니다.

❤ 그림 25-3 트랜잭션 적용 후 은행 계좌 이체




이번에는 홍길동과 김유신 계좌 잔고의 갱신이 모두 정상적으로 이루어지면 최종적으로 반영(commit)합니다. 즉, 중간에 이상이 발생할 경우 이전의 모든 작업 취소, 즉 롤백(rollback)되어 버립니다. 앞에서보다 안전한 거래가 가능하고 문제 발생의 소지도 줄일 수 있겠죠.

# 25.3 스프링의 트랜잭션 속성 알아보기

스프링에서 사용하는 트랜잭션 기능의 속성은 표 25-1과 같습니다. 지금은 일단 내용을 읽어 보기만 하고 이후 실습을 통해 확실하게 이해하도록 합시다.

❤ 표 25-1 스프링의 여러 가지 트랜잭션 속성들

| 속성 | 기능 |
|---|---|
| propagation | 트랜잭션 전파 규칙 설정 |
| isolation | 트랜잭션 격리 레벨 설정 |
| readOnly | 읽기 전용 여부 설정 |
| rollbackFor | 트랜잭션을 롤백(rollback)할 예외 타입 설정 |
| norollbackFor | 트랜잭션을 롤백하지 않을 예외 타입 설정 |
| timeout | 트랜잭션 타임아웃 시간 설정 |

표 25-2와 표 25-3은 각각 propagation 속성과 isolation 속성에 관련된 값을 나타낸 것입니다.

❤ 표 25-2 propagation 속성이 가지는 값

| 값 | 의미 |
|---|---|
| REQUIRED | • 트랜잭션 필요. 진행 중인 트랜잭션이 있는 경우 해당 트랜잭션 사용<br>• 트랜잭션이 없으면 새로운 트랜잭션 생성. 디폴트 값 |
| MANDATORY | • 트랜잭션 필요<br>• 진행 중인 트랜잭션이 없는 경우 예외 발생 |
| REQUIRED_NEW | • 항상 새로운 트랜잭션 생성<br>• 진행 중인 트랜잭션이 있는 경우 기존 트랜잭션을 일시 중지시킨 후 새로운 트랜잭션 시작<br>• 새로 시작된 트랜잭션이 종료되면 기존 트랜잭션 계속 진행 |
| SUPPORTS | • 트랜잭션 필요 없음<br>• 진행 중인 트랜잭션이 있는 경우 해당 트랜잭션 사용 |
| NOT_SUPPORTED | • 트랜잭션 필요 없음<br>• 진행 중인 트랜잭션이 있는 경우 기존 트랜잭션을 일시 중지시킨 후 메서드 실행<br>• 메서드 실행이 종료되면 기존 트랜잭션 계속 진행 |
| NEVER | • 트랜잭션 필요 없음<br>• 진행 중인 트랜잭션이 있는 경우 예외 발생 |

| 값 | 의미 |
|---|---|
| NESTED | • 트랜잭션 필요<br>• 진행 중인 트랜잭션이 있는 경우 기존 트랜잭션에 중첩된 트랜잭션에서 메서드 실행<br>• 트랜잭션이 없으면 새로운 트랜잭션 생성 |

표 25-3 isolation 속성이 가지는 값

| 속성 | 기능 |
|---|---|
| DEFAULT | 데이터베이스에서 제공하는 기본 설정 사용 |
| READ_UNCOMMITED | 다른 트랜잭션에서 커밋하지 않은 데이터 읽기 가능 |
| READ_COMMITED | 커밋한 데이터만 읽기 가능 |
| REPEATABLE_READ | 현재 트랜잭션에서 데이터를 수정하지 않았다면 처음 읽어온 데이터와 두 번째 읽어온 데이터가 동일 |
| SERIALIZABLE | 같은 데이터에 대해 한 개의 트랜잭션만 수행 가능 |

JAVA WEB

# 25.4 스프링 트랜잭션 기능 적용해 계좌 이체 실습하기

이번에는 계좌 이체 기능을 스프링의 트랜잭션 기능을 적용하여 실습해 보겠습니다.

먼저 SQL Developer로 예금자 계좌 정보를 저장하는 테이블을 생성합니다. 그리고 예금자의 계좌 정보를 다음과 같이 추가합니다.

**코드 25-1** 계좌 테이블 생성 코드

```
create table cust_account(
  accountNo varchar2(20)  primary key,      계좌 번호
  custName varchar2(50),                    예금자
  balance  number(20,4)                     계좌 잔고
);

insert into cust_account(accountNo,custName,balance)
values('70-490-930','홍길동',10000000);
```

'홍길동'과 '김유신'의 계좌 정보를 생성합니다.

```
insert into cust_account(accountNo,custName,balance)
values('70-490-911','김유신',10000000);

commit;
```

insert문 실행 후 반드시 커밋을 해야 합니다.

```
select * from cust_account;
```

다음은 계좌 정보를 조회한 결과입니다. 모든 예금자의 계좌 잔고는 10,000,000원입니다.

❯ 그림 25-4 테이블에 저장된 예금자들의 잔고 현황

| | ACCOUNTNO | CUSTNAME | BALANCE |
|---|---|---|---|
| 1 | 70-490-930 | 홍길동 | 10000000 |
| 2 | 70-490-911 | 김유신 | 10000000 |

## 25.4.1 트랜잭션 관련 XML 파일 설정하기

1. 스프링과 연동해 트랜잭션 기능을 구현하는 데 필요한 XML 파일들을 다음과 같이 준비합니다. web.xml은 24장의 것을 복사해 붙여 넣습니다.

❯ 그림 25-5 실습 파일 위치


2. action-servlet.xml에서는 뷰 관련 빈과 각 URL 요청명에 대해 호출될 메서드들을 설정합니다.

코드 25-2 pro25/WebContent/WEB-INF/action-servlet.xml

```
...
  <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass"
              value="org.springframework.web.servlet.view.JstlView"/>
```

```
    <property name="prefix" value="/WEB-INF/views/account/" />
    <property name="suffix" value=".jsp"/>
  </bean>

  <bean id="accController
        class="com.spring.account.AccountController">
    <property name="methodNameResolver">
      <ref local="methodResolver"/>
    </property>
    <property name="accService" ref="accService"/>
  </bean>

  <bean id="methodResolver"
          class="org.springframework.web.servlet.mvc.multiaction.
                                  PropertiesMethodNameResolver" >
    <property  name="mappings" >
      <props>
        <prop key="/account/sendMoney.do" >sendMoney</prop>
      </props>
    </property>
  </bean>
  <bean id="urlMapping"
        class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
      <props>
        <prop key="/account/*.do">accController</prop>
      </props>
    </property>
  </bean>
</beans>
```

accService 빈을 주입합니다.

/account/sendMoney.do로 요청 시 sendMoney 메서드를 호출합니다.

/account/*.do로 요청 시 accController 빈을 실행합니다.

3. action-mybatis.xml을 다음과 같이 작성합니다. 스프링의 DataSourceTransactionManager 클래스를 이용해 트랜잭션 처리 빈을 생성한 후 DataSource 속성에 dataSource 빈을 주입하여 데이터베이스 연동 시 트랜잭션을 적용합니다. 그리고 txManager 빈에 <tx:annotation-driven> 태그를 설정해 애너테이션을 적용할 수 있게 합니다.

**코드 25-3** pro25/WebContent/WEB-INF/config/action-mybatis.xml

```
...
  <bean id="accDAO" class="com.spring.account.AccountDAO">
    <property name="sqlSession" ref="sqlSession" />
  </bean>
```

```
<bean id="txManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource" />
</bean>
```

DataSourceTransactionManager 클래스를 이용해 dataSource 빈에 트랜잭션을 적용합니다.

```
<tx:annotation-driven transaction-manager="txManager" />
</beans>
```

애너테이션을 사용하여 트랜잭션을 적용하기 위해 txManager 빈을 설정합니다.

4. action-service.xml에서는 AccountService의 accDAO 속성에 accDAO 빈을 주입하도록 구현합니다.

**코드 25-4** pro25/WebContent/WEB-INF/config/action-service.xml

```
...
<bean id="accService" class="com.spring.account.AccountService">
  <property name="accDAO" ref="accDAO"/>
</bean>
</beans>
```

accService 빈의 속성에 accDAO 빈을 주입합니다.

## 25.4.2 마이바티스 관련 XML 파일 설정하기

이번에는 계좌 이체 기능을 SQL문으로 구현한 매퍼 파일을 설정해 보겠습니다.

1. 다음과 같이 매퍼 파일인 account.xml을 준비합니다.

❤ 그림 25-6 매퍼 파일 위치


2. 매퍼 파일에서는 두 개의 update문으로 두 명의 계좌 잔고를 갱신합니다.

**코드 25-5** pro25/src/mybatis/mappers/account.xml

```
...
<mapper namespace="mapper.account">
  <update id="updateBalance1">
  <![CDATA[
    update cust_account
```

```
    set balance=balance-5000000
    where
      accountNo = '70-490-930'
  ]]>
  </update>

  <update id="updateBalance2">
    <![CDATA[
      update cust_account
      set balance=balance+5000000
      where
        accountNo = '70-490-911'
      ]]>
  </update>
</mapper>
```

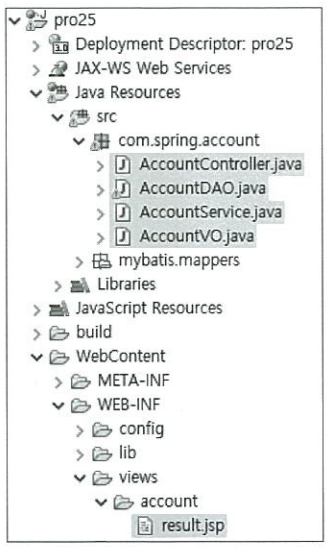
잔고를 5000000원 감액합니다.

잔고를 5000000원 증액합니다.

## 25.4.3 트랜잭션 관련 자바 클래스와 JSP 파일 구현하기

1. 계좌 이체 기능에 필요한 자바 파일과 JSP 파일들을 다음과 같이 준비합니다.

그림 25-7 실습 파일 위치



2. 컨트롤러에서는 속성 accService에 빈을 주입하기 위해 setter를 구현합니다. /account/sendMoney.do로 요청 시 sendMoney() 메서드를 호출해 계좌 이체 작업을 수행합니다.

**코드 25-6** pro25/src/com/spring/account/AccountController.java

```
package com.spring.account;
...
public class AccountController  extends MultiActionController  {
  private AccountService accService ;
  public void setAccService(AccountService accService){
    this.accService = accService;
  }

  public ModelAndView sendMoney(HttpServletRequest request,
                                HttpServletResponse response) throws Exception {
    ModelAndView mav=new ModelAndView();
    accService.sendMoney();
    mav.setViewName("result");
    return mav;
  }
}
```

속성 accService에 빈을 주입하기 위해 setter를 구현합니다.

금액을 이체합니다.

3. AccountService 클래스를 다음과 같이 작성합니다. 서비스 클래스의 메서드는 단위 기능을 수행하므로 @Transactional 애너테이션을 서비스 클래스에 적용해 메서드별로 트랜잭션을 적용합니다.

**코드 25-7** pro25/src/com/spring/account/AccountService.java

```
package com.spring.account;
...
@Transactional(propagation=Propagation.REQUIRED)
public class AccountService {
  private AccountDAO accDAO;
  public void setAccDAO(AccountDAO accDAO) {
    this.accDAO = accDAO;
  }

  public void sendMoney() throws Exception {
    accDAO.updateBalance1();
    accDAO.updateBalance2();
  }
}
```

@Transactional을 이용해 AccountService 클래스의 모든 메서드에 트랜잭션을 적용합니다.

속성 accDAO에 빈을 주입하기 위해 setter를 구현합니다.

sendMoney() 메서드 호출 시 accDAO의 두 개의 SQL문을 실행합니다.

4. AccountDAO 클래스에서는 각 예금자 계좌를 갱신하는 메서드를 구현합니다.

**코드 25-8** pro25/src/com/spring/account/AccountDAO.java

```
package com.spring.account;
...
public class AccountDAO {
  private SqlSession sqlSession;
  public void setSqlSession(SqlSession sqlSession) {
    this.sqlSession = sqlSession;
  }

  public void updateBalance1() throws DataAccessException {
    sqlSession.update("mapper.account.updateBalance1");
  }

  public void updateBalance2() throws DataAccessException {
    sqlSession.update("mapper.account.updateBalance2");
  }
}
```

속성 sqlSession에 빈을 주입하기 위해 setter를 구현합니다.

첫 번째 update문을 실행해 홍길동 계좌에서 5000000원을 차감합니다.

두 번째 update문을 실행해 김유신 계좌에서 5000000원을 증액합니다.

5. 트랜잭션을 적용하지 않은 경우와 적용한 경우의 실행 결과를 각각 확인해 보겠습니다. 먼저 http://localhost:8090/pro25/account/sendMoney.do로 요청하여 정상적으로 계좌 이체가 이루어진 경우의 결과를 확인합니다.

❤ 그림 25-8 실행 결과


6. SQL Developer로 조회하면 홍길동의 계좌에서 김유신의 계좌로 5,000,000만 원이 이체된 것을 확인할 수 있습니다.

❤ 그림 25-9 정상 송금 후 계좌 잔고

| | ACCOUNTNO | CUSTNAME | BALANCE |
|---|---|---|---|
| 1 | 70-490-930 | 홍길동 | 5000000 |
| 2 | 70-490-911 | 김유신 | 15000000 |

7. 이번에는 트랜잭션을 적용하지 않은 경우의 실행 결과를 보겠습니다. AccountService.java에서 다음 부분을 주석 처리합니다.

**코드 25-9** pro25/src/com/spring/account/AccountService.java

```
...
/*@Transactional(propagation=Propagation.REQUIRED) */
public class AccountService {
    ...
}
```

8. account.xml의 두 번째 SQL문에 일부러 문법 오류를 발생시킵니다.

**코드 25-10** pro25/src/mybatis/mappers/account.xml

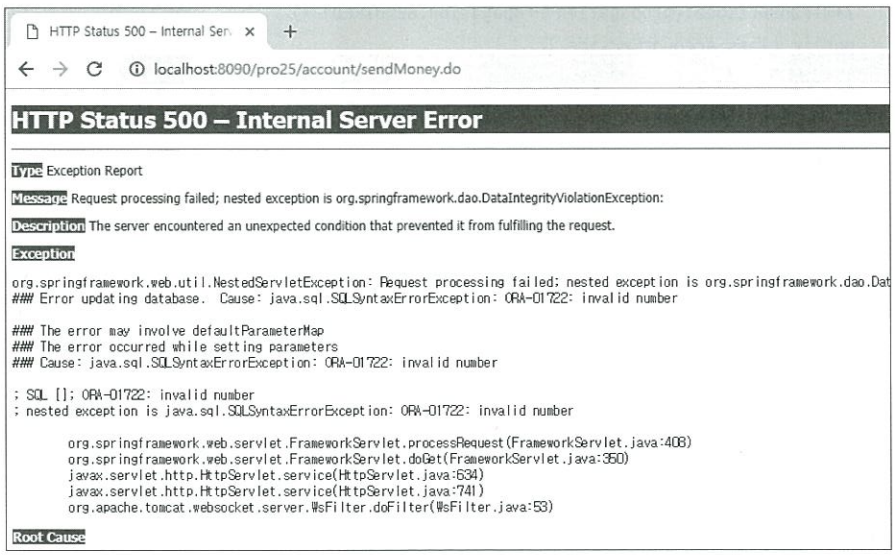
```
...
  <update id="updateBalance1" >
    <![CDATA[
      update cust_account
      set balance=balance-5000000
      where  accountNo = '70-490-930'
    ]]>
  </update>

  <update id="updateBalance2" >
  <![CDATA[
    update cust_account
    set balance=balance+5000000
    where  accountNo = 70-490-911
  ]]>
  </update>
...
```

계좌 번호의 양쪽 작은따옴표(')를 삭제하여 오류를 발생시킵니다.

9. SQL Developer에서 예금자들의 잔고를 원래대로 되돌린 후, 즉 10,000,000원으로 갱신한 후 브라우저에서 http://localhost:8090/pro25/account/sendMoney.do로 요청하면 다음과 같은 오류가 발생합니다.

❤ 그림 25-10 이체 요청 시 오류 발생




10. SQL Developer로 각 계좌 잔고를 조회해 보면 홍길동의 잔고는 5,000,000원이 감소했으나 김유신의 잔고는 10,000,000원 그대로인 것을 확인할 수 있습니다.

❤ 그림 25-11 오류 발생 후 계좌 잔고

| | ACCOUNTNO | CUSTNAME | BALANCE |
|---|---|---|---|
| 1 | 70-490-930 | 홍길동 | 5000000 |
| 2 | 70-490-911 | 김유신 | 10000000 |

11. 트랜잭션을 적용한 후 브라우저에서 요청한 결과를 확인하기에 앞서 원래대로 주석을 해제합니다. 그리고 SQL Developer로 다시 예금자들의 잔고를 10,000,000원으로 변경합니다.

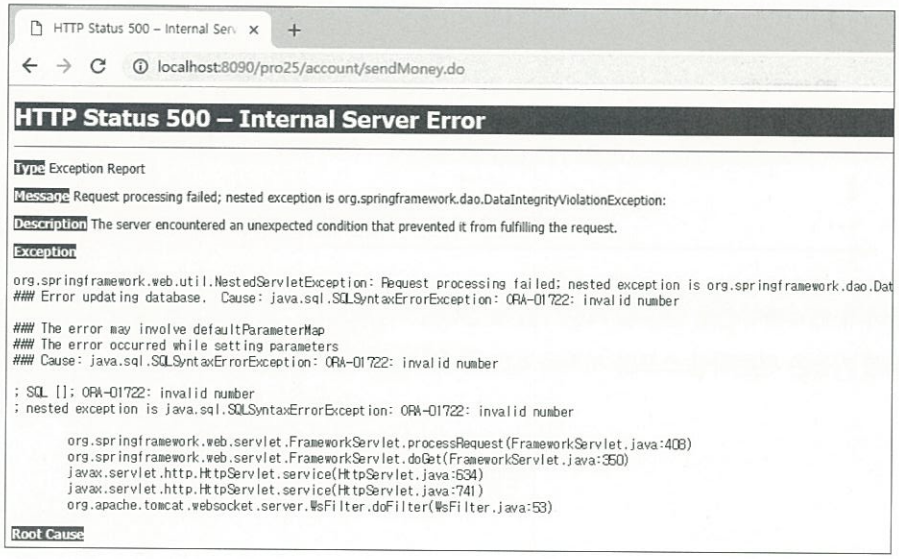
**코드 25-11** pro25/src/com/spring/account/AccountService.java

```
...
@Transactional(propagation=Propagation.REQUIRED)
public class AccountService {
  ...
}
```

주석을 해제합니다.

12. http://localhost:8090/pro25/account/sendMoney.do로 요청하면 또다시 오류가 발생합니다.

❯ 그림 25-12 송금 요청 시 오류 발생




13. SQL Developer로 각 계좌 잔고를 조회합니다. 이번에는 트랜잭션이 적용되었으므로 김유신의 잔고는 물론이고 오류가 발생하지 않은 홍길동의 잔고도 원래의 금액으로 롤백이 됩니다.

❯ 그림 25-13 오류 발생 후 계좌 잔고

| | ACCOUNTNO | CUSTNAME | BALANCE |
|---|---|---|---|
| 1 | 70-490-930 | 홍길동 | 10000000 |
| 2 | 70-490-911 | 김유신 | 10000000 |

대부분의 애플리케이션에서는 이처럼 Service 클래스에 트랜잭션을 적용합니다. 사실 우리가 24장에서 구현한 회원 기능 프로그램에서 MemberServiceImpl 클래스에도 트랜잭션 애너테이션을 적용했었습니다.

코드 25-12 pro24/src/com/spring/member/service/MemberServiceImpl.java

```
...
@Transactional(propagation=Propagation.REQUIRED)
public class MemberServiceImpl  implements MemberService{
  private MemberDAO memberDAO;
  public void setMemberDAO(MemberDAO memberDAO){
    this.memberDAO = memberDAO;
  }

  @Override
  public List listMembers() throws DataAccessException {
```

```
    List membersList = null;
    membersList = memberDAO.selectAllMemberList();
    return membersList;
  }

  @Override
  public int addMember(MemberVO memberVO) throws DataAccessException {
    return memberDAO.insertMember(memberVO);
}
...
```

지금까지 애너테이션을 이용해 기본적인 트랜잭션 기능을 알아봤습니다. 더 세부적인 스프링 트랜잭션 기능은 전문적인 스프링 서적을 참고하기 바랍니다.