Pivotal

Transaction Management with Spring

Transactional Proxies and @Transactional

spring by Pivotal

Objectives

After completing this lesson, you should be able to

- Explain why Transactions are used
  - And how Java supports them in different ways
- Describe and use Spring Transaction Management
- Configure Transaction Propagation
- Setup Rollback rules
- Use Transactions in Tests

Pivotal

Agenda

- **Why use Transactions?**
- Java Transaction Management
- Spring Transaction Management
- Transaction Propagation
- Rollback rules
- Testing
- Lab
- Advanced topics

Pivotal

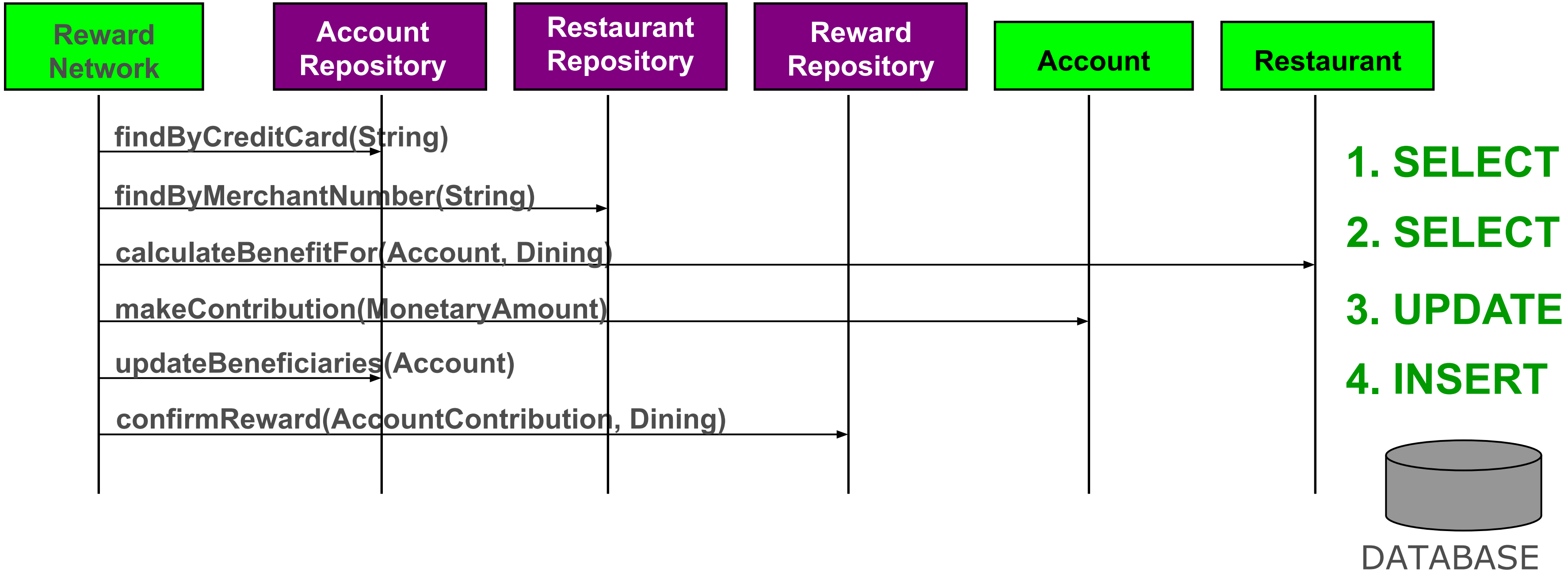
What is a Transaction?

Enable ***concurrent*** access to a ***shared*** resource

- A set of tasks which take place as a single, indivisible action
  - **A**tomic
    - Each unit of work is an all-or-nothing operation
  - **C**onsistent
    - Database integrity constraints are never violated
  - **I**solated
    - Isolating transactions from each other
  - **D**urable
    - Committed changes are permanent

Transactions in the RewardNetwork

- The *rewardAccountFor(Dining)* method represents a unit-of-work that should be atomic

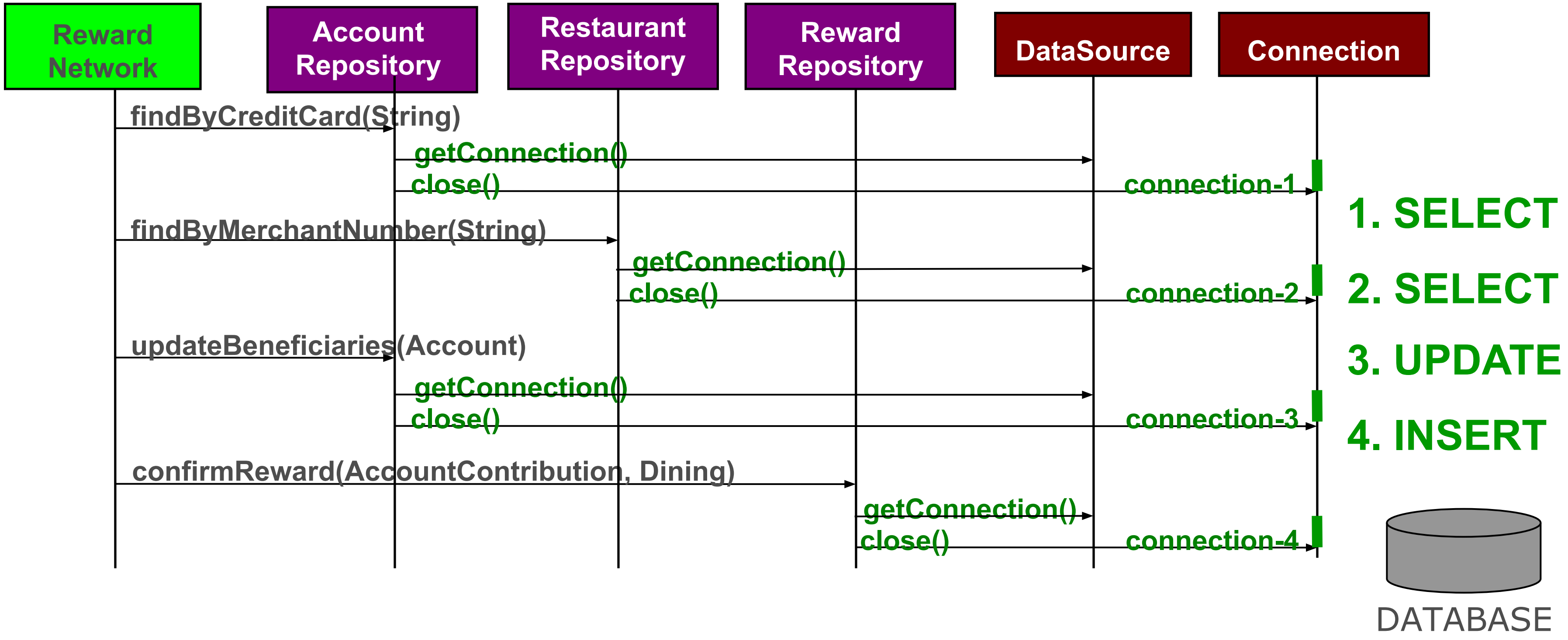

Reward Network
Account Repository
Restaurant Repository
Reward Repository
Account
Restaurant
findByCreditCard(String)
findByMerchantNumber(String)
calculateBenefitFor(Account, Dining)
makeContribution(MonetaryAmount)
updateBeneficiaries(Account)
confirmReward(AccountContribution, Dining)
1. SELECT
2. SELECT
3. UPDATE
4. INSERT
DATABASE

Pivotal

Naïve Approach

- Connection per Data Access Operation
  - This unit-of-work contains 4 data access operations
    - Each acquires, uses, and releases a distinct Connection
  - The unit-of-work is *non-transactional*

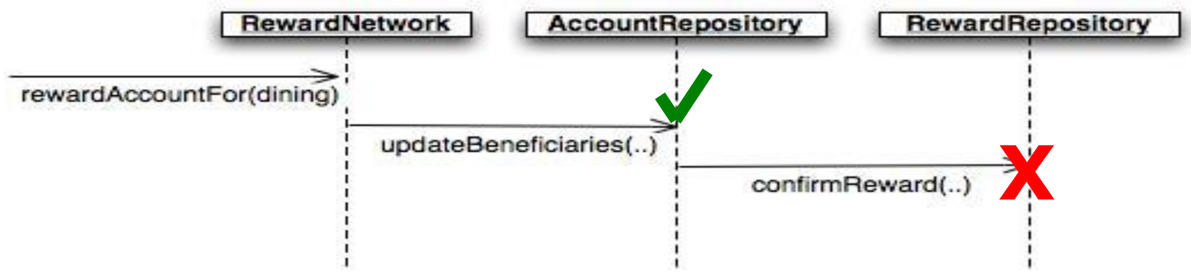
Running non-Transactionally



Pivotal

Partial Failures (in non-Transactional operation)

- Suppose an Account is being rewarded



- If the beneficiaries are updated…
- But the reward confirmation fails…
- There will be no record of the reward!

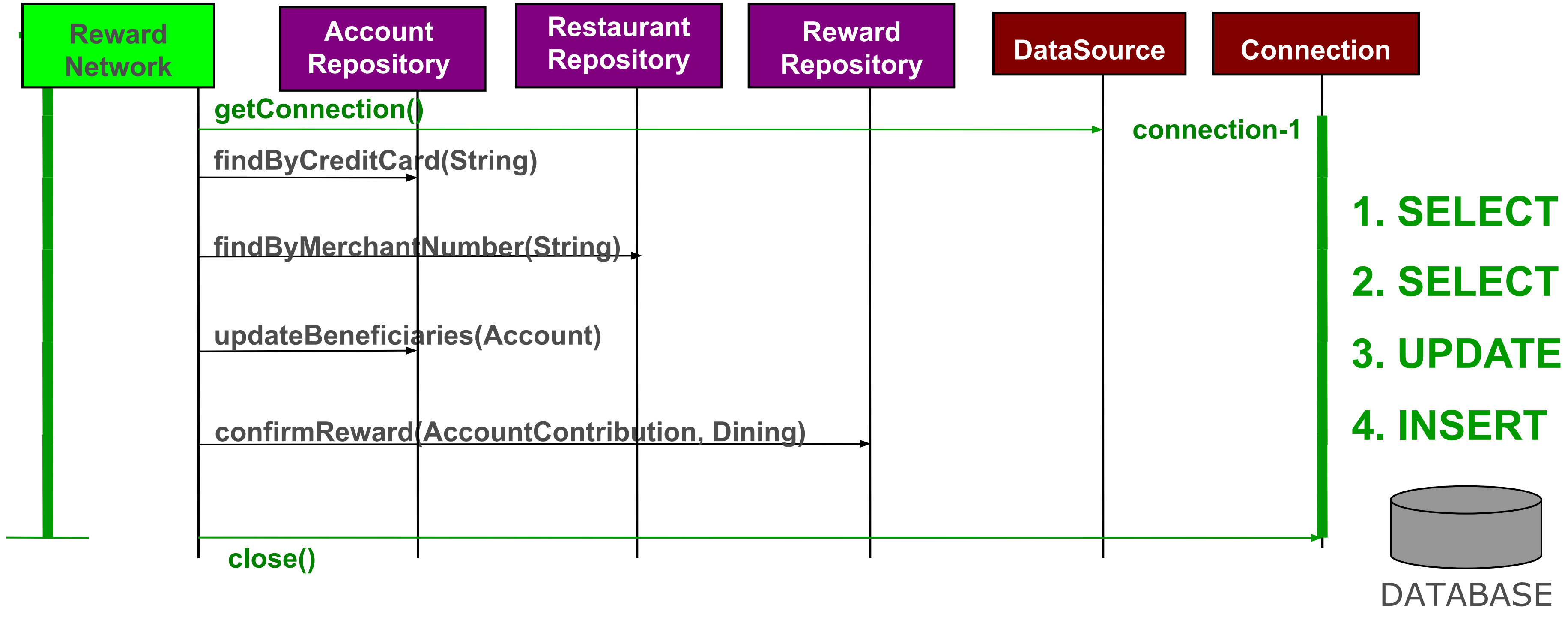
The unit-of-work is **not** *atomic*

Pivotal

Correct Approach

- Connection per Unit-of-Work
  - More efficient
    - Same Connection reused for each operation
  - Operations complete as an atomic unit
    - Either all succeed or all fail
  - The unit-of-work can run in a *transaction*

Running in a Transaction



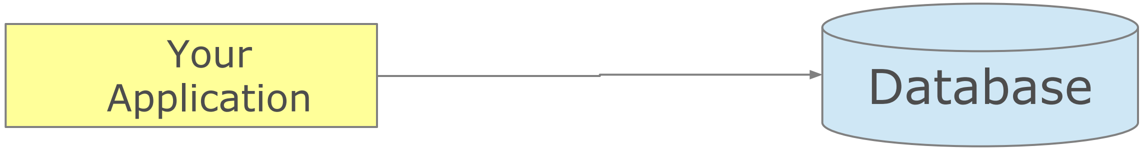
Pivotal

Agenda

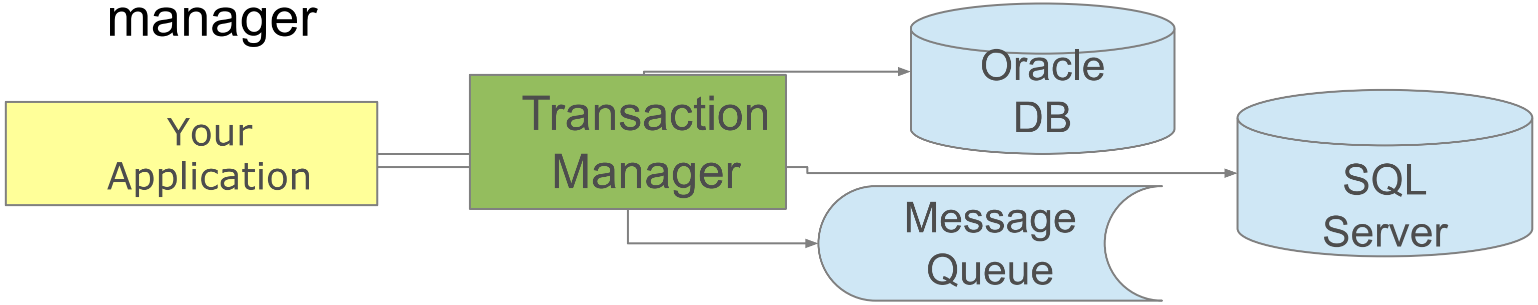
- Why use Transactions?
- **Java Transaction Management**
- Spring Transaction Management
- Transaction Propagation
- Rollback rules
- Testing
- Lab
- Advanced topics

Local and Global Transaction Management

- Local Transactions – Single Resource
  - Transactions managed by underlying resource



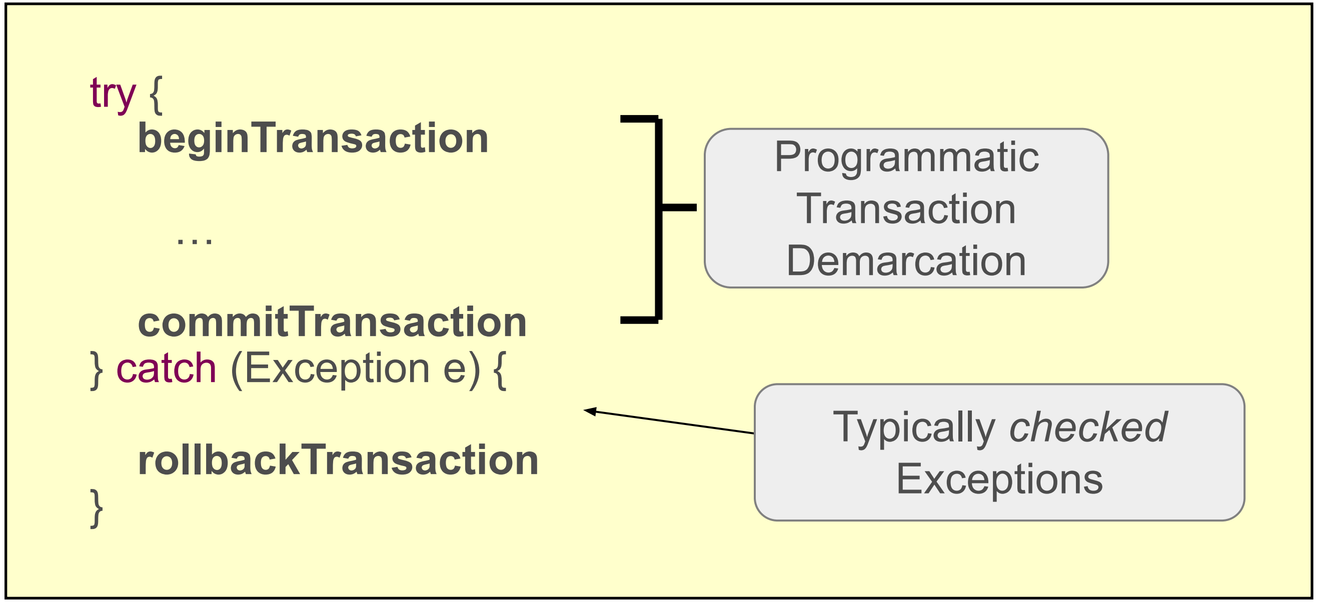
- Global (distributed) Transactions – Multiple Resources
  - Transaction managed by separate, dedicated transaction manager



Pivotal

Transactional Code Pattern

- Many different APIs, but a common pattern
  - Implemented using code
  - Classic cross-cutting concern



```
try {
    beginTransaction

        …

    commitTransaction
} catch (Exception e) {

    rollbackTransaction
}
```

Programmatic Transaction Demarcation
Typically *checked* Exceptions

Java API Transaction Examples

API	Begin Transaction	End Transaction

JDBC	conn = dataSource.getConnection()
	conn.setAutoCommit(false)
		conn.commit()
		conn.rollback()

JMS	session = connection
	.createSession (true, 0)
		session.commit()
		session.rollback()

JPA	Transaction tx =
	entityManager.getTransaction();
	tx.begin();
		tx.commit()
		tx.rollback()

Hibernate	Transaction tx =
	session.beginTransaction();
		tx.commit()
		tx.rollback()

Local transactions only:

- *Code cannot 'join' a transaction already in progress*
- *Code cannot be used with global transaction*

Pivotal

Global Transactions in Java

Java Transaction API (JTA)

```
try {
        UserTransaction ut =
            (UserTransaction) new InitialContext()
                    .lookup("java:comp/UserTransaction");
        ut.begin();
        …


        ut.commit();
} catch (Exception e) {
        ut.rollback();
        …
}
```

JTA Specific API

Checked Exceptions

Programmatic Transaction Demarcation

Requires a JTA implementation:

- *"Full" application server (WebSphere, WebLogic, JBoss, etc.)*
- *Standalone implementation (Atomikos, JTOM, etc.)*

Pivotal

Problems with Java Transaction Management

- Multiple APIs for different local resources
- Programmatic transaction demarcation
  - Typically performed in the service layer but we don't want data-access code in the service-layer (separation of concerns)
  - Usually repeated (cross-cutting concern)
- Orthogonal concerns
  - Transaction demarcation should be independent of transaction implementation

Agenda

- Why use Transactions?
- Java Transaction Management
- **Spring Transaction Management**
- Transaction Propagation
- Rollback rules
- Testing
- Lab
- Advanced topics

Spring Transaction Management – 1

- Spring separates transaction *demarcation* from transaction *implementation*
  - Demarcation expressed declaratively via AOP
    - Programmatic approach also available
  - PlatformTransactionManager abstraction hides implementation details.
    - Several implementations available
- Spring uses the same API for global vs. local.
  - Change from local to global is minor
    - Just change the transaction manager

Spring Transaction Management – 2

- There are only 2 steps
  - Declare a **`PlatformTransactionManager`** bean
  - Declare the transactional methods
    - Using Annotations, Programmatic
    - Can mix and match

PlatformTransactionManager Implementations

- Spring's **PlatformTransactionManager** is the base interface for the abstraction
- Several implementations are available
  - **`DataSourceTransactionManager`**
  - **`JmsTransactionManager`**
  - **`JpaTransactionManager`**
  - **`JtaTransactionManager`**
  - **`WebLogicJtaTransactionManager`**
  - **`WebSphereUowTransactionManager`**

Spring allows you to configure whether you use JTA or not.
It does not have *any* impact on your Java classes

Pivotal

Deploying the Transaction Manager

- Create the required implementation
  - Just like any other Spring bean
    - Configure it as appropriate
  - Here is the manager for a DataSource

```
@Bean
public PlatformTransactionManager
                 transactionManager(DataSource dataSource) {
   return new DataSourceTransactionManager(dataSource);
}
```

A DataSource bean must be defined elsewhere

Bean id "*transactionManager*" is recommended name. See Advanced slides (5) for detailed explanation on naming this bean.

Pivotal

Accessing a JTA Transaction Manager

- Use a JNDI lookup for container-managed DataSource

```
@Bean
public PlatformTransactionManager transactionManager() {
   return new JtaTransactionManager();
}

@Bean
public DataSource dataSource(@Value("${db.jndi}" String jndiName) {
   JndiDataSourceLookup lookup = new JndiDataSourceLookup();
   return lookup.getDataSource(jndiName);
}
```

   – Or use container-specific subclasses:
      - `WebLogicJtaTransactionManager`
      - `WebSphereUowTransactionManager`

@Transactional Configuration

In your code

```
public class RewardNetworkImpl implements RewardNetwork {
  @Transactional
  public RewardConfirmation rewardAccountFor(Dining d) {
    // atomic unit-of-work
  }
}
```

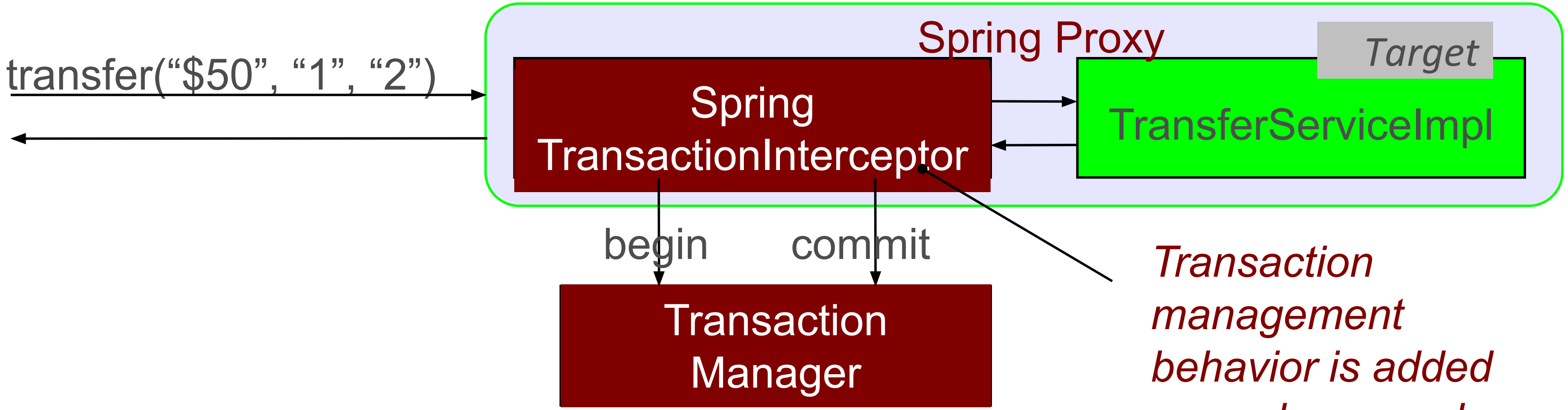
```
@Configuration
@EnableTransactionManagement
public class TxnConfig {
  @Bean
  public PlatformTransactionManager transactionManager(DataSource ds) {
     return new DataSourceTransactionManager(ds);
  }
```

Defines a Bean Post-Processor – proxies @Transactional beans

In your Spring configuration

Declarative Transaction Management

- Target service wrapped in a proxy
  - Uses an "*around*" advice



Transaction management behavior is added around your code

@Transactional: What Happens Exactly?

- Proxy implements the following behavior
  - Transaction started before entering the method
  - Commit at the end of the method
  - Rollback if method throws a **`RuntimeException`**
    - Default behavior
    - Can be overridden (see later)
    - Checked exceptions do not cause Rollback
- All controlled by *configuration*

Spring Proxy
Reward NetworkImpl

Pivotal

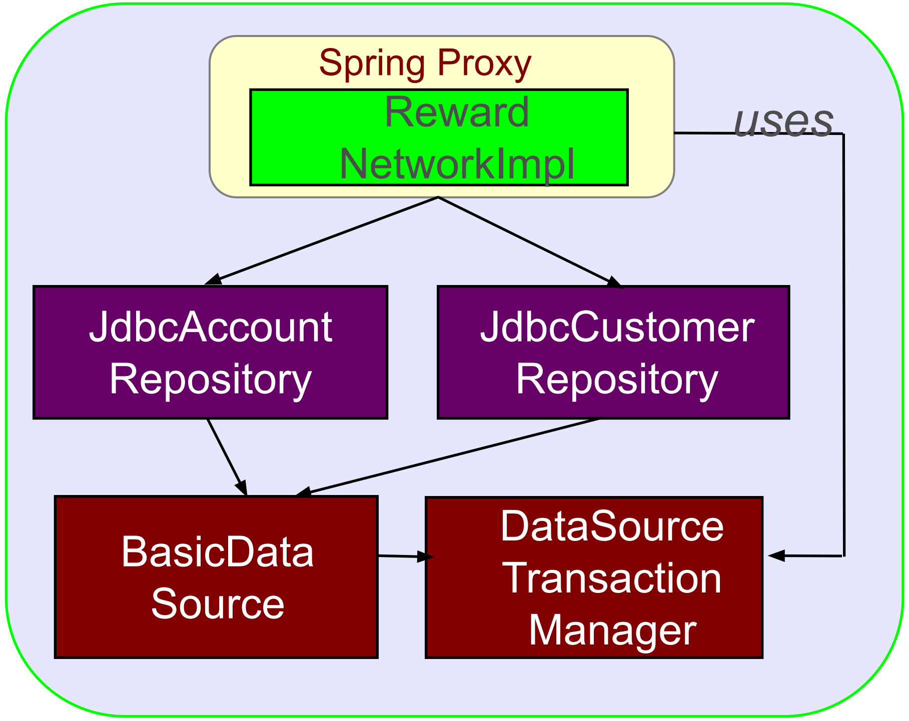
Transaction Bound to Current Thread

- Transaction context bound to current thread
  - Holds the underlying JDBC connection
  - Hibernate sessions, JTA (Java EE) work similarly
- `JdbcTemplate` used in an `@Transactional` method
  - Uses that connection automatically
- You can access it manually

```
DataSourceUtils.getConnection(dataSource)
```

Pivotal

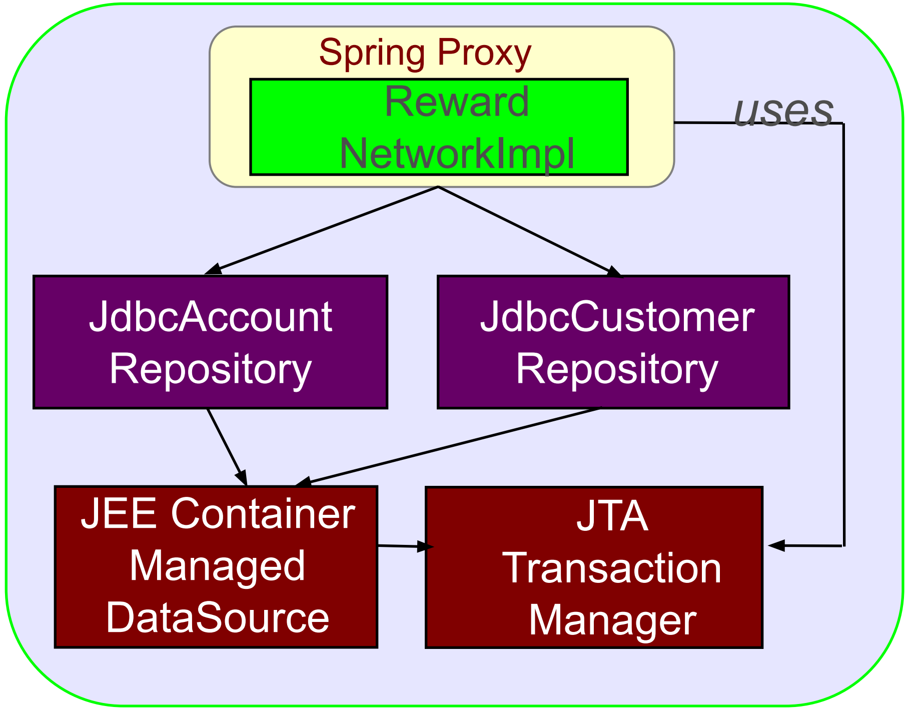
Local JDBC Configuration



- How?
  - Define and use local data source
  - Use DataSource Transaction Manager
- Purpose
  - Integration testing and/or Production
  - Deploy to Tomcat or other servlet container

JDBC Java EE Configuration

No code changes
Just configuration



- How?
  - Use container-managed datasource (JNDI)
  - Use JTA Transaction Manager
- Purpose
  - Deploy to Java EE container

Pivotal

@Transactional – Class Level

- Applies to all methods declared by the interface(s)

```
@Transactional
public class RewardNetworkImpl implements RewardNetwork {

   public RewardConfirmation rewardAccountFor(Dining d) {
      // atomic unit-of-work
   }

   public RewardConfirmation updateConfirmation(RewardConfirmantion rc) {
      // atomic unit-of-work
   }
}
```

Alternatively *@Transactional* can be declared on the interface instead
– since Spring Framework 5.0

@Transactional – Class *and* method levels

- Combining class and method levels

```
@Transactional(timeout=60)
public class RewardNetworkImpl implements RewardNetwork {

   public RewardConfirmation rewardAccountFor(Dining d) {
      // atomic unit-of-work
   }
   @Transactional(timeout=45)
   public RewardConfirmation updateConfirmation(RewardConfirmantion rc) {
      // atomic unit-of-work
   }
}
```

default settings

override attributes at method level

Java's @Transactional

- Java also has an annotation
  - **`javax.transaction.Transactional`**
- Also supported by Spring
  - Fewer options
  - Not used in these examples
  - Be careful when doing the lab
    - Use Spring's **`@Transactional`**

Agenda

- Why use Transactions?
- Java Transaction Management
- Spring Transaction Management
- **Transaction Propagation**
- Rollback rules
- Testing
- Lab
- Advanced topics

Pivotal

Understanding Transaction Propagation

- What should happen if **`ClientServiceImpl`** calls **`AccountServiceImpl`**?
  - Single transaction?
  - Two separate transactions?

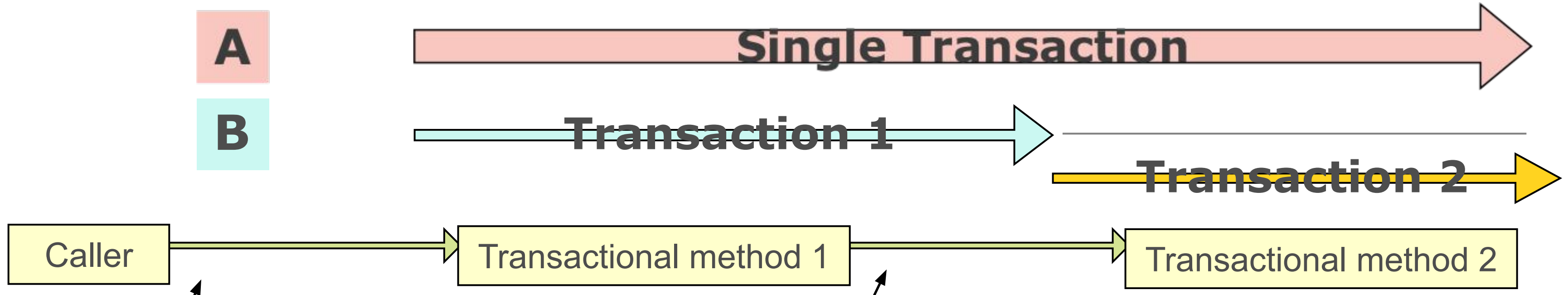
```
public class ClientServiceImpl
            implements ClientService {
  @Autowired
  private AccountService accountService;

  @Transactional
  public void updateClient(Client c) {
    // …
    this.accountService.update(c.getAccounts());
  }
}
```

```
public class AccountServiceImpl
            implements AccountService {

  @Transactional
  public void update(List <Account> accs)
   { // … }
}
```

Understanding Transaction Propagation



Transaction created, committed or rolled back as needed

Is there an existing transaction? Should method 2 execute in that transaction (A)

or should a new transaction be created (B), with outer transaction suspended?

Behavior driven by metadata for the method

Pivotal

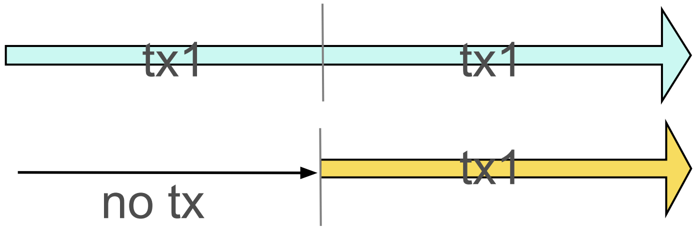
Transaction Propagation with Spring

- 7 levels of propagation
- The following examples show *REQUIRED* and *REQUIRES_NEW*
  - *Check the documentation for other levels*
- Can be used as follows:

```
@Transactional( propagation=Propagation.REQUIRES_NEW )
```

REQUIRED

- Default value
- Execute within a current transaction, create a new one if none exists

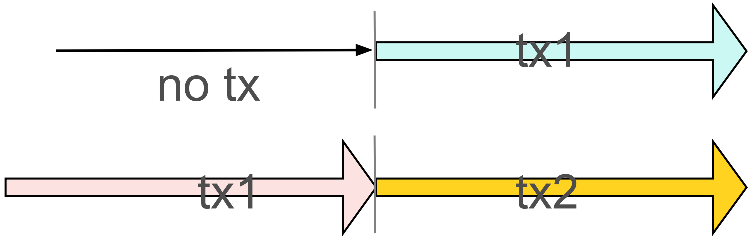


```
@Transactional(propagation=Propagation.REQUIRED)
```

Pivotal

REQUIRES_NEW

- Create a new transaction, suspending the current transaction if one exists



```
@Transactional(propagation=Propagation.REQUIRES_NEW)
```

Propagation Rules Are Enforced by a Proxy

- In the example below, the 2nd propagation rule does not get applied because the call does not go through a proxy

```
public class ClientServiceImpl implements ClientService {
        @Transactional(propagation=Propagation.REQUIRED)
        public void update1() {
            update2();
        }
        @Transactional(propagation=Propagation.REQUIRES_NEW)
        public void update2() {
        }
    }
}
```

Does not get applied because the call is internal

Agenda

- Why use Transactions?
- Java Transaction Management
- Spring Transaction Management
- Transaction Propagation
- **Rollback Rules**
- Testing
- Lab
- Advanced topics

Default Behavior

- By default, a transaction is rolled back only if a *RuntimeException* has been thrown
  - Could be any kind of *RuntimeException*: *DataAccessException*, *HibernateException* etc.

```
public class RewardNetworkImpl implements RewardNetwork {
   @Transactional
   public RewardConfirmation rewardAccountFor(Dining d) {
      // ...
      throw new RuntimeException();

   }
}
```

Triggers a rollback

rollbackFor and noRollbackFor

- Default settings can be overridden with *rollbackFor* and/or *noRollbackFor* attributes

```
public class RewardNetworkImpl implements RewardNetwork {

   @Transactional(rollbackFor=MyCheckedException.class,
                           noRollbackFor={JmxException.class, MailException.class})
   public RewardConfirmation rewardAccountFor(Dining d) throws Exception {
         // ...
   }

}
```

Agenda

- Why use Transactions?
- Java Transaction Management
- Spring Transaction Management
- Transaction Propagation
- Rollback Rules
- **Testing**
- Lab
- Advanced topics

Pivotal

@Transactional within Integration Test

- Annotate test method (or class) with @Transactional
  - Runs test methods in a transaction
  - Transaction will be *rolled back* afterwards
    - No need to clean up your database after testing!

```
@SpringJUnitConfig(RewardsConfig.class)
public class RewardNetworkTest {
    @Test @Transactional
    public void testRewardAccountFor() {
        ...
    }
}
```

This test is now transactional

Pivotal

Controlling Transactional Tests

```java
@SpringJUnitConfig(RewardsConfig.class)
@Transactional
public class RewardNetworkTest {

    @Test
    @Commit
    public void testRewardAccountFor() {
        ...    // Whatever happens here will be committed
    }
}
```

Make *all* tests transactional

Commit transaction at end of test

Lab: Managing Transactions Declaratively using Spring Annotations

Lab project
28-transactions

Anticipated Lab time:
20 Minutes

Optional Topics: Programmatic transactions, read-only and multiple transactions, global transactions, propagation options

Pivotal

Agenda

Advanced Topics

(1) Programmatic Transactions
(2) Read-only Transactions
(3) More on Transactional Tests
(4) Multiple and Global Transactions
(5) Transaction Manager bean name
(6) Global Transactions
(7) Propagation Options

Pivotal

1. Programmatic Transactions with Spring

- Declarative transaction management is highly recommended
  - Clean code
  - Flexible configuration
- Spring does enable programmatic transaction
  - Works with local or JTA transaction manager
  - `TransactionTemplate` plus callback

Can be useful inside a technical framework that would not rely on external configuration

Pivotal

Programmatic Transactions: example

```
public RewardConfirmation rewardAccountFor(Dining dining) {
  ...
  return  new TransactionTemplate(txManager).execute( (status) -> {
      try {
        ...
        accountRepository.updateBeneficiaries(account);
        confirmation = rewardRepository.confirmReward(contribution, dining);
      }
      catch (RewardException e) {
        status.setRollbackOnly();
        confirmation = new RewardFailure();
      }
      return confirmation;
    }
  );
}
```

Method *not* @Transactional

Lambda syntax

Method no longer throws exception, using status to perform *manual* rollback

```
public interface TransactionCallback<T> {
  public T doInTransaction(TransactionStatus status)
            throws Exception;
}
```

2. Read-only Transactions – Faster

- Why use transactions if you're only planning to read data?
  - Performance: allows Spring to optimize the transactional resource for read-only data access

```
public void rewardAccount1() {
    jdbcTemplate.queryForList(…);
    jdbcTemplate.queryForInt(…);
}

@Transactional(readOnly=true)
public void rewardAccount2() {
    jdbcTemplate.queryForList(…);
    jdbcTemplate.queryForInt(…);
}
```

Two connections

One single connection

Pivotal

Read-only Transactions – Isolation

- Why use transactions if you're only planning to read data?
  - With a high isolation level, a read-only transaction prevents data from being modified until the transaction commits

```
@Transactional(readOnly=true, isolation=Isolation.REPEATABLE_READ)
public void myAccounts(long userId) {
   List accounts = jdbcTemplate.queryForList
               ("SELECT * FROM Accounts WHERE user = ?", userId);
   process(accounts);
   int nAccounts = jdbcTemplate.queryForInt
               ("SELECT count(*) FROM Accounts WHERE user = ?", userId);
   assert accounts.size() == nAccounts;
 }
```

3. Transactional Tests

@BeforeEach vs @BeforeTransaction

```
@SpringJUnitConfig(RewardsConfig.class)
public class RewardNetworkTest {

  @BeforeTransaction
  public void verifyInitialDatabaseState() {…}

  @BeforeEach
  public void setUpTestDataInTransaction() {…}

  @Test  @Transactional
  public void testRewardAccountFor() { … }
```

Run *before* transaction is started

Run *within* the transaction

@AfterEach and *@AfterTransaction* work in same way as *@BeforeEach* and *@BeforeTransaction*

Pivotal

@Sql and Transaction Control

- Transaction control options
  - ***ISOLATED***: Uses *own* txn, a PTM *must* exist
  - ***INFERRED***: If PTM exists, txn started using default propagation (same txn as test method)
    otherwise a DataSource *must* exist (used with *no* txn)
  - ***DEFAULT***: Whatever @Sql defines at class level, *INFERRED* otherwise

```
@Sql(  scripts = "/test-user-data.sql",
       config = @SqlConfig
           ( transactionMode = TransactionMode.ISOLATED,
             transactionManager = "myTxnMgr",
             dataSource= "myDataSource" )
```

Optionally specify bean ids

PTM = PlatformTransactionManager, txn = transaction

4. Multiple Transaction Managers

- Configuration – mark *one* as primary

Java Config

```
@Bean
public PlatformTransactionManager myOtherTransactionManager() {
    return new DataSourceTransactionManager(dataSource1());
}

@Bean
@Primary
public PlatformTransactionManager transactionManager() {
    return new DataSourceTransactionManager(dataSource2());
}
```

XML

```
<bean id="transactionManager" primary="true" … > … </bean>
```

@Transactional with Multiple Managers

- @Transactional can declare the id of the transaction manager that should be used

```
@Transactional("myOtherTransactionManager")
public void rewardAccount1() {
    jdbcTemplate.queryForList(…);
    jdbcTemplate.queryForInt(…);
}

@Transactional
public void rewardAccount2() {
    jdbcTemplate.queryForList(…);
    jdbcTemplate.queryForInt(…);
}
```

Uses the bean with id "*myOtherTransactionManager*"

Defaults to use the bean annotated as the *primary*

Important: Separate transaction managers = separate transactions!

Pivotal

5. Transaction Manager Naming

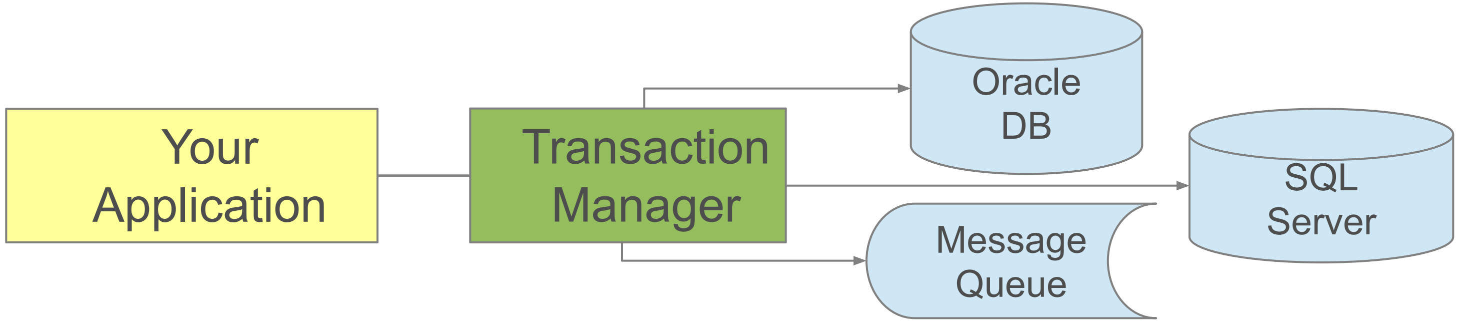
- **`@EnableTransactionManagement`**
  - Expects a bean called **`txManager`**
  - Or looks for **`PlatformTransactionManager`** by *type*
- **`Spring Boot`**
  - Creates a bean called **`transactionManager`** by default
- **`@Transactional`**
  - Looks for *primary* transaction manager if exists
  - Or looks for singleton **`PlatformTransactionManager`**
  - Or bean called **`transactionManager`** by default

Recall: bean id "*transactionManager*" is recommended name and **`@EnableTransactionManagement`** will find it by type.

Pivotal

6. Global Transactions

- Also called *distributed* transactions
- Involve multiple dissimilar resources:



- Global transactions typically require JTA and specific drivers (XA drivers)
  – Two-phase commit protocol

Global Transactions → Spring Integration

- Many possible strategies
  - Spring allows you to switch easily from a non-JTA to a JTA transaction policy
  - Just change the type of the transaction manager
- Reference:
  - *“Distributed transactions with Spring, with and without XA”* by Dr. Dave Syer

http://www.javaworld.com/javaworld/jw-01-2009/jw-01-spring-transactions.html

7. Propagation Levels and their Behaviors

Propagation Type	If NO current transaction (txn) exists	If there IS a current transaction (txn)
MANDATORY	Throw exception	Use current txn
NEVER	Don't create a txn, run method without a txn	Throw exception
NOT_SUPPORTED	Don't create a txn, run method without a txn	Suspend current txn, run method without a txn
SUPPORTS	Don't create a txn, run method without a txn	Use current txn
REQUIRED (default)	Create a new txn	Use current txn
REQUIRES_NEW	Create a new txn	Suspend current txn, create a new independent txn
NESTED	Create a new txn	Create a new nested txn