Spring Transaction

Spring Framework - Transaction

- Spring Framework has
  - Comprehensive transaction support
  - Provides a consistent abstraction for transaction management
    - consistent programming model across different transaction APIs such as JTA, JDBC, Hibernate, JPA, and JDO.
    - Supports declarative transaction management.
    - Provides a simpler API for programmatic transaction management than a number of complex transaction APIs such as JTA.
    - Integrates very well with Spring's various data access abstractions

Spring Transaction Model

- The Spring Framework provides both declarative and programmatic transaction management
- Declarative transaction management is preferred by most users, and is recommended in most cases

Programmatic Transaction

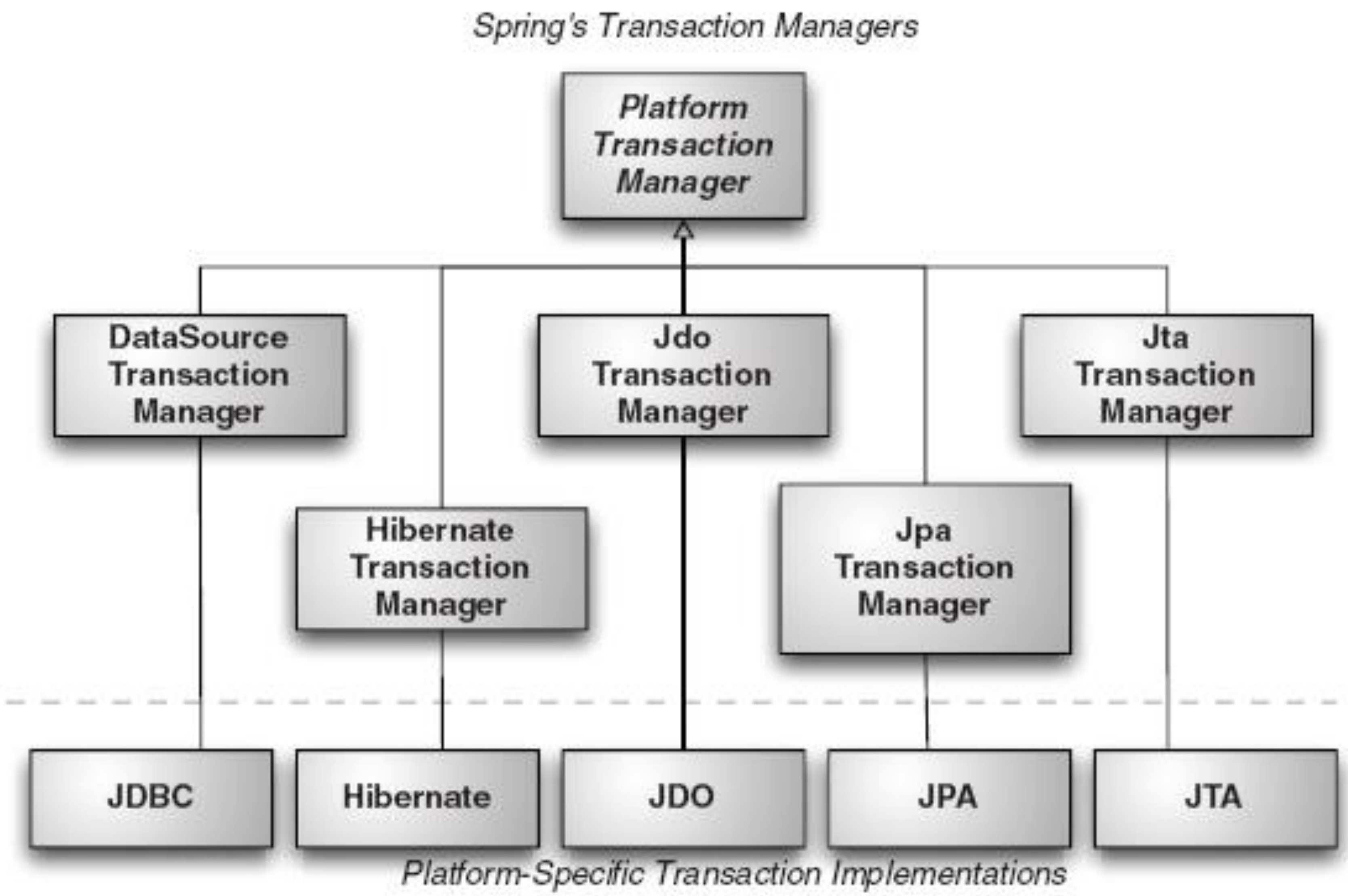
- Developers work with the Spring Framework transaction abstraction
- Can run over any underlying transaction infrastructure

Declarative Transaction

- Developers typically write little or no code related to transaction management
- Don't depend on the Spring Framework's transaction API (or indeed on any other transaction API).

PlatformTransactionManager

PlatformTransactionManager


Spring's Transaction Managers
Platform Transaction Manager
DataSource Transaction Manager
Jdo Transaction Manager
Jta Transaction Manager
Hibernate Transaction Manager
Jpa Transaction Manager
JDBC
Hibernate
JDO
JPA
JTA
Platform-Specific Transaction Implementations

Annotation Driven Transaction management

Using *@Transactional*

- The functionality offered by the ***@Transactional*** annotation and the support classes is only available to you if you are using at least Java 5 (Tiger)

Using *@Transactional*

```
<!-- the service class that we want to make transactional
   -->
@Transactional
public class DefaultFooService implements FooService {
    Foo getFoo(String fooName);
    Foo getFoo(String fooName, String barName);
    void insertFoo(Foo foo);
    void updateFoo(Foo foo);
}
```

Using *@Transactional*

- When the above POJO is defined as a bean in a Spring IoC container, the bean instance can be made transactional by adding merely *one* line of XML configuration

```
<!-- this is the service object that we want to make transactional -->
  <bean id="fooService" class="x.y.service.DefaultFooService"/>
  <!-- enable the configuration of transactional behavior based on annotations --
     >
  <tx:annotation-driven transaction-manager="txManager"/>
  <!-- a PlatformTransactionManager is still required -->
  <bean id="txManager"
     class="org.springframework.jdbc.datasource.DataSourceTransactionManag
     er">
    <!-- (this dependency is defined somewhere else) -->
    <property name="dataSource" ref="dataSource"/>
  </bean>
    <!-- other <bean/> definitions here -->
</beans>
```

Using *@Transactional*

- The @Transactional annotation may be placed before an interface definition, a method on an interface, a class definition, or a *public* method on a class
- However, the mere presence of the @Transactional annotation is not enough to actually turn on the transactional behavior - the @Transactional annotation *is simply metadata* that can be consumed by something that is @Transactional-aware and that can use the metadata to configure the appropriate beans with transactional behavior
- In the case of the above example, it is the presence of the <tx:annotation-driven/> element that *switches on* the transactional behavior

Using *@Transactional*

- method in the same class takes precedence over the transactional settings defined in the class level annotation.

```
@Transactional(readOnly = true)
public class DefaultFooService implements FooService {
    public Foo getFoo(String fooName) {
        // do something
    }
    // these settings have precedence for this method
    @Transactional(readOnly = false, propagation =
  Propagation.REQUIRES_NEW)
    public void updateFoo(Foo foo) {
        // do something
    }
}
```

@Transactional settings

- The default @Transactional settings
  - The propagation setting is PROPAGATION_REQUIRED
  - The isolation level is ISOLATION_DEFAULT
  - The transaction is read/write
  - The transaction timeout defaults to the default timeout of the underlying transaction system, or none if timeouts are not supported
  - Any RuntimeException will trigger rollback, and any checked Exception will not

Programmatic transaction management

Programmatic Transaction

- Spring provides two means of programmatic transaction management:
  - Using the TransactionTemplate
  - Using a PlatformTransactionManager implementation directly
- The Spring team generally recommend the first approach (i.e. using the TransactionTemplate)
- The second approach is similar to using the JTA UserTransaction API (although exception handling is less cumbersome).

Using the TransactionTemplate

- Adopts the same approach as other Spring *templates* such as JdbcTemplate and HibernateTemplate
- Uses a callback approach
- A TransactionTemplate instance is threadsafe

```
Object result = tt.execute(new TransactionCallback() {


    public Object doInTransaction(TransactionStatus
  status) {
       updateOperation1();
       return resultOfUpdateOperation2();
    }
});
```

Using the *TransactionTemplate*

- If there is no return value, use the convenient TransactionCallbackWithoutResult class via an anonymous class

```
tt.execute(new TransactionCallbackWithoutResult() {
protected void doInTransactionWithoutResult(
                           TransactionStatus status) {
        updateOperation1();
        updateOperation2();
    }
});
```

Code within the callback can roll the transaction back by calling the setRollbackOnly() method on the supplied TransactionStatus object

Using the `TransactionTemplate`

- Application classes wishing to use the TransactionTemplate must have access to a PlatformTransactionManager
  - which will typically be supplied to the class via dependency injection
  - It is easy to unit test such classes with a mock or stub PlatformTransactionManager
    - There is no JNDI lookup or static shenanigans here: it is a simple interface. As usual, you can use Spring to greatly simplify your unit testing

PlatformTransactionManager

- ***PlatformTransactionManager*** can be directly used to manage transaction
  - Simply pass the implementation of the PlatformTransactionManager you're using to your bean via a bean reference
  - Then, using the TransactionDefinition and TransactionStatus objects you can initiate transactions, rollback and commit

PlatformTransactionManager

```
DefaultTransactionDefinition def = new
   DefaultTransactionDefinition();
def.setPropagationBehavior(TransactionDefinition.PROPAGATI
   ON_REQUIRED);


TransactionStatus status = txManager.getTransaction(def);
try {
    // execute your business logic here
}
catch (MyException ex) {
    txManager.rollback(status);
    throw ex;
}
txManager.commit(status);
```

Summary

- Spring Transaction Support
- Different Transaction Managers
- Declarative Transaction Management
  - Using xml based configuration
  - Using @Transactional
- Programmatic Transaction Management
  - Using TransactionTemplate
  - Using PlateformTransactionManager

Thank You!

Resources:

Spring Framework Reference Documentation