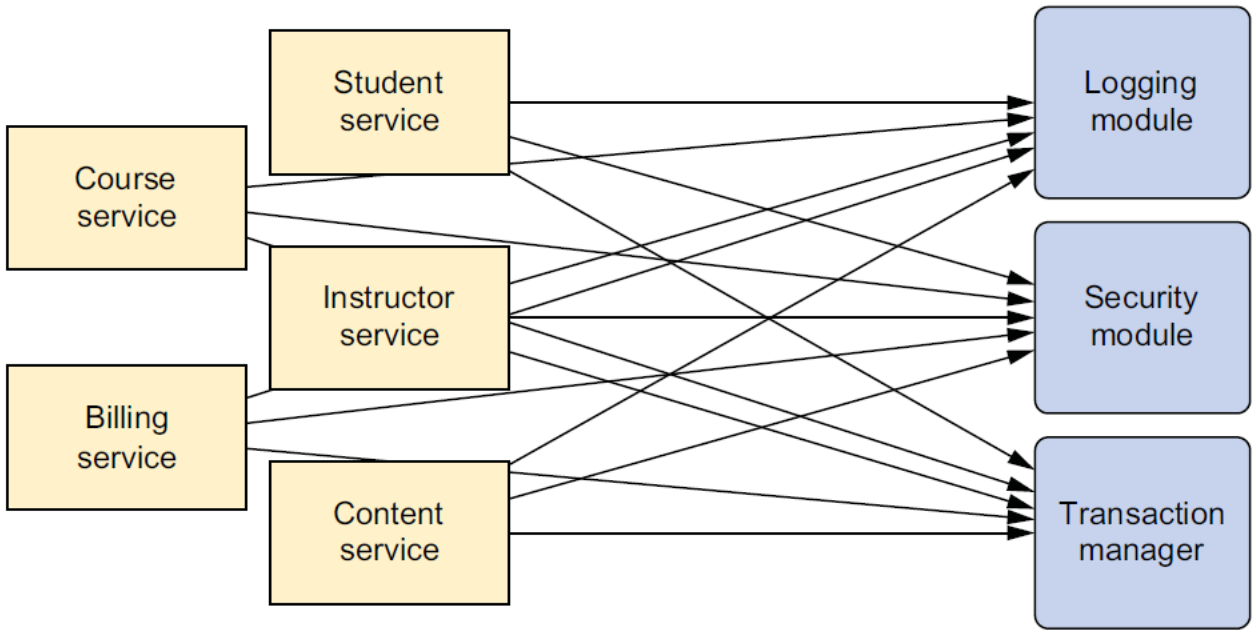
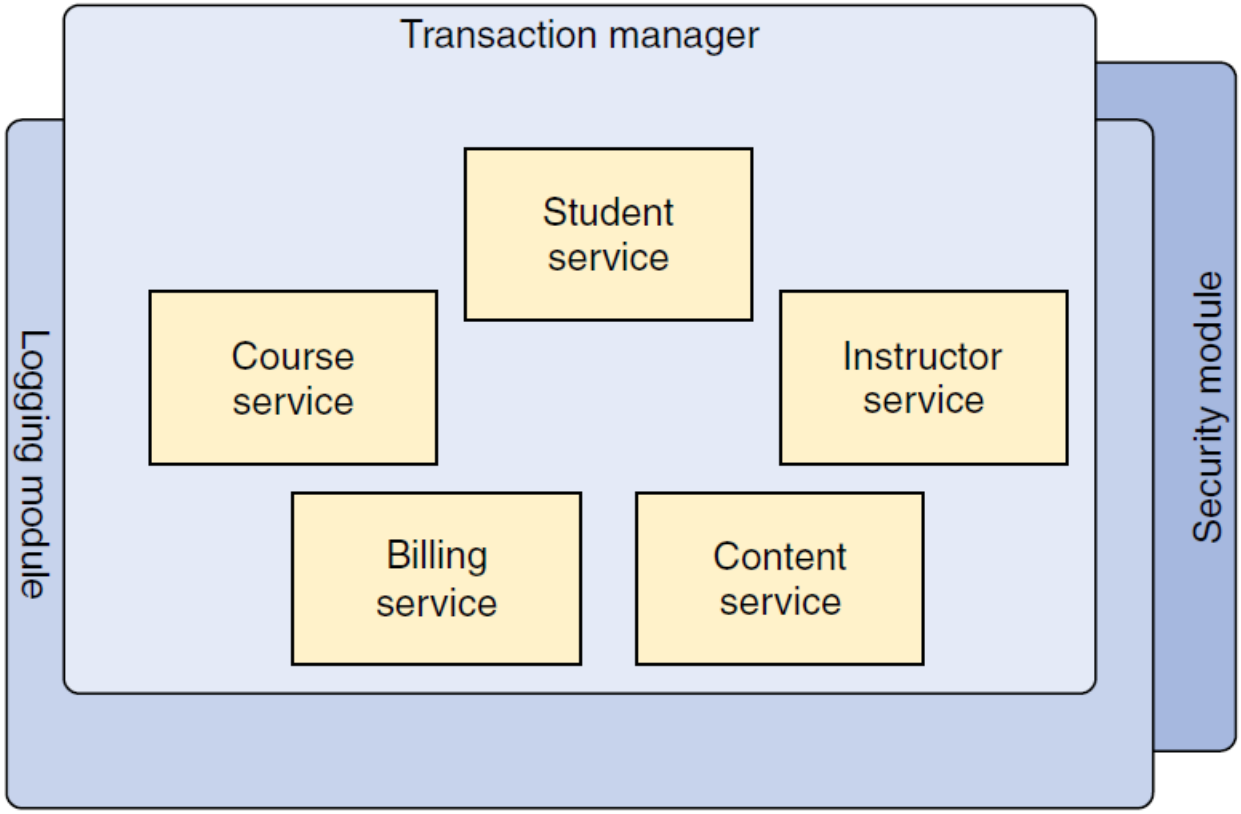
Working with data sources

- ✓ AOP – Aspect Oriented Programming
- ✓ Data sources
- ✓ JDBC connections
- ✓ Transactions
- ✓ Spring and JPA
- ✓ Second Level Cache

AOP – Aspect Oriented Programming




Transaction manager
Student service
Course service
Instructor service
Billing service
Content service
Logging module
Security module

Advice

Before—The advice functionality takes place before the advised method is invoked.

After—The advice functionality takes place after the advised method completes, regardless of the outcome.

Advice

After-returning—The advice functionality takes place after the advised method successfully completes.

After-throwing—The advice functionality takes place after the advised method throws an exception.

Around—The advice wraps the advised method, providing some functionality before and after the advised method is invoked.

Weaving

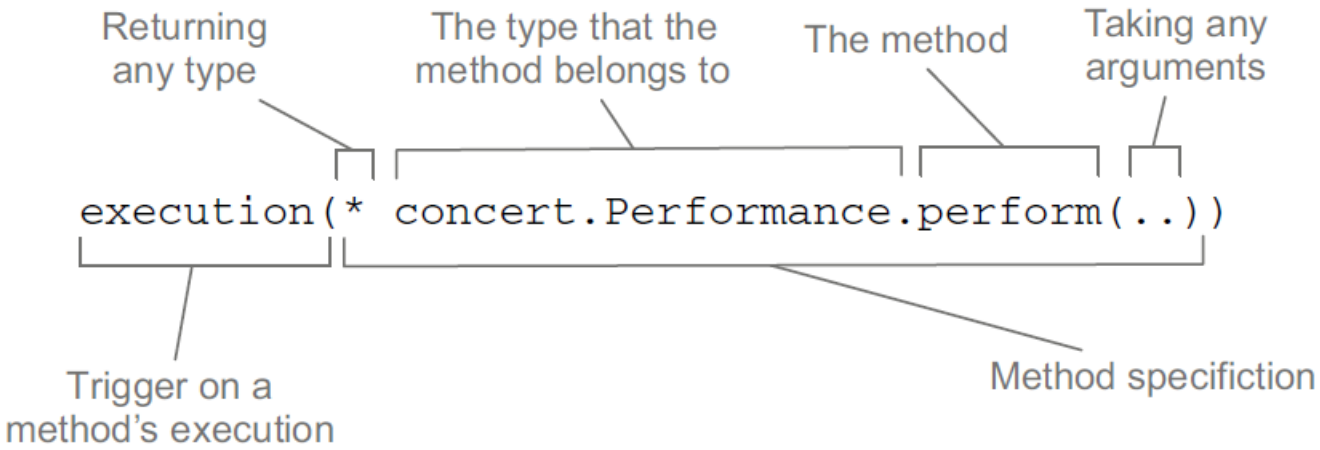
Compile time —Aspects are woven in when the target class is compiled. This requires a special compiler.

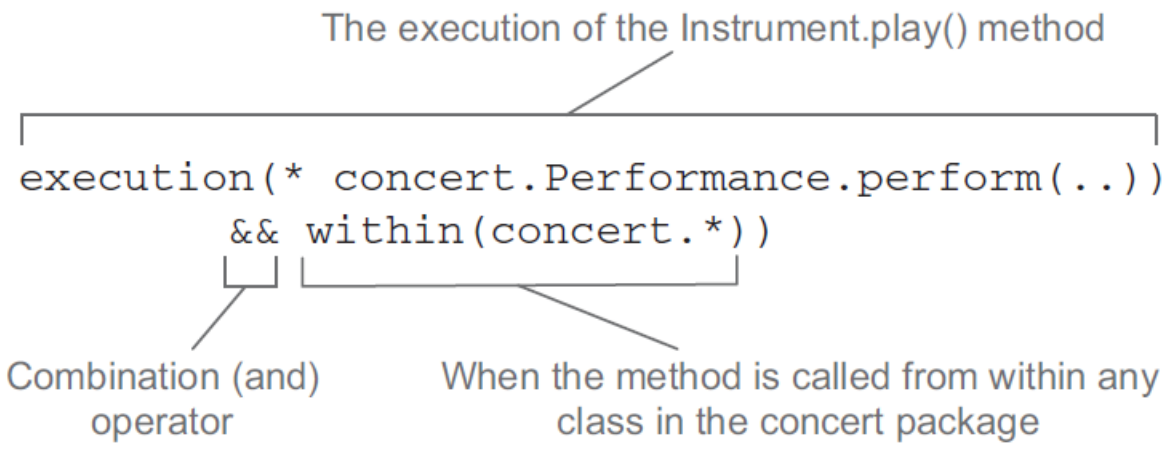
Class load time—Aspects are woven in when the target class is loaded into the JVM. This requires a special ClassLoader that enhances the target class's bytecode before the class is introduced into the application

Runtime—Aspects are woven in sometime during the execution of the application. **This is how Spring AOP aspects are woven.**

Pointcuts

AspectJ designator	Description
`args()`	Limits join-point matches to the execution of methods whose arguments are instances of the given types
`@args()`	Limits join-point matches to the execution of methods whose arguments are annotated with the given annotation types
`execution()`	Matches join points that are method executions
`this()`	Limits join-point matches to those where the bean reference of the AOP proxy is of a given type
`target()`	Limits join-point matches to those where the target object is of a given type
`@target()`	Limits matching to join points where the class of the executing object has an annotation of the given type
`within()`	Limits matching to join points within certain types
`@within()`	Limits matching to join points within types that have the given annotation (the execution of methods declared in types with the given annotation when using Spring AOP)
`@annotation`	Limits join-point matches to those where the subject of the join point has the given annotation


Returning any type
The type that the method belongs to
The method
Taking any arguments
execution(* concert.Performance.perform(..))
Trigger on a method's execution
Method specifiction


The execution of the Instrument.play() method
execution(* concert.Performance.perform(..))
&& within(concert.*))
Combination (and) operator
When the method is called from within any class in the concert package

```
execution(* concert.Performance.perform())
        and bean('woodstock')
```

```
execution(* concert.Performance.perform())
        and !bean('woodstock')
```

@Aspect

```
@Aspect
public class AspectClass{
        // aspect definition
}
```

```
@Before("execution(** cat.Cat.sayMeow(..))")
public void method(){}
```

```
@After ("execution(** cat.Cat.sayMeow(..))")
public void method(){}
```

```
@AfterReturning("execution(** cat.Cat.sayMeow(..))")
public void method(){}
```

```
@AfterThrowing("execution(** cat.Cat.sayMeow(..))")
public void method(){}
```

Define your own pointcut

```
@Pointcut("execution(** cat.Cat.sayMeow(..))")
public void myPointcut() {}

@Before("myPointcut()")
public void method() {}
```

Enabling aspects

```
@Configuration
@EnableAspectJAutoProxy
public class ProjectConfig{}
```

```
<aop:aspectj-autoproxy />
```

Around pointcut

```
@Around("execution(** cat.Cat.sayMeow(..))")
public void method( ProceedingJoinPoint jp ) {
        //do something here
        jp.proceed();
        // do something here
}
```

Using parameters in advice

```
@Pointcut("execution(* cat.Cat.sayMeow(String)) && args(name)")
public void myPointcut( String name ) {}
```

```
@Before(" myPointcut( name ) ")
public void countTrack(String name) {}
```

Aspects declared with XML

Enable aspects:

<aop:aspectj-autoproxy />

AOP configuration element	Purpose
`<aop:config>`	The top-level AOP element. Most `\<aop:\*\>` elements must be contained within `\<aop:config\>`.
`<aop:declare-parents>`	Introduces additional interfaces to advised objects that are transparently implemented.
`<aop:pointcut>`	Defines a pointcut.

Advices in XML

AOP configuration element	Purpose
`<aop:advisor>`	Defines an AOP advisor.
`<aop:after>`	Defines an AOP after advice (regardless of whether the advised method returns successfully).
`<aop:after-returning>`	Defines an AOP after-returning advice.
`<aop:after-throwing>`	Defines an AOP after-throwing advice.
`<aop:around>`	Defines an AOP around advice.
`<aop:aspect>`	Defines an aspect.
`<aop:aspectj-autoproxy>`	Enables annotation-driven aspects using `@AspectJ`.
`<aop:before>`	Defines an AOP before advice.

```
<aop:config>
    <aop:aspect ref="myBean">
        <aop:before
            pointcut="execution(** cat.Cat.sayMeow(..))"
            method="method1"/>

        <aop:after-returning
            pointcut="execution(** cat.Cat.sayMeow(..))"
            method=" method3"/>

        <aop:after-throwing
            pointcut="execution(** cat.Cat.sayMeow(..))"
            method=" method4"/>
    </aop:aspect>
</aop:config>
```

Define your own pointcut in XML

```
<aop:pointcut
        id=“myPointcut“
        expression="execution(** cat.AspectClass.myPointcut(..))"
/>


<aop:before
        pointcut-ref="myPointcut"
        method=“methodToBeCalledFromAspectClass“
/>
```

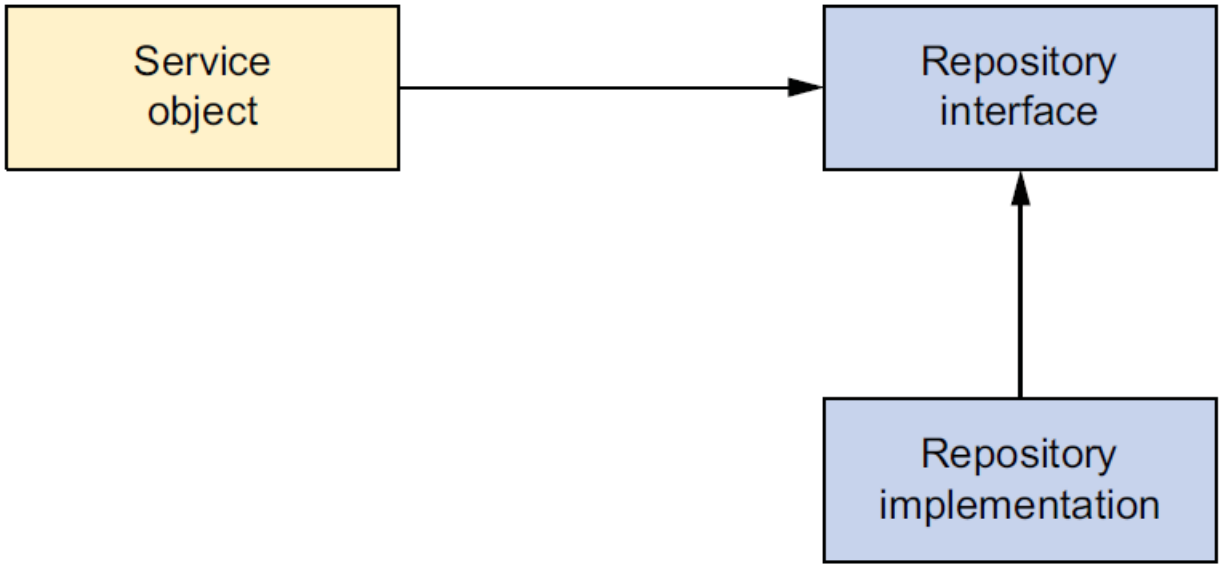
Around advice in XML

```
<aop:around
    pointcut-ref="myPointcut"
    method="methodToBeCalled"
/>
```

```
public void methodToBeCalled( ProceedingJoinPoint jp ) {
        //do something here
        jp.proceed();
        // do something here
}
```

Databases and how to persist data

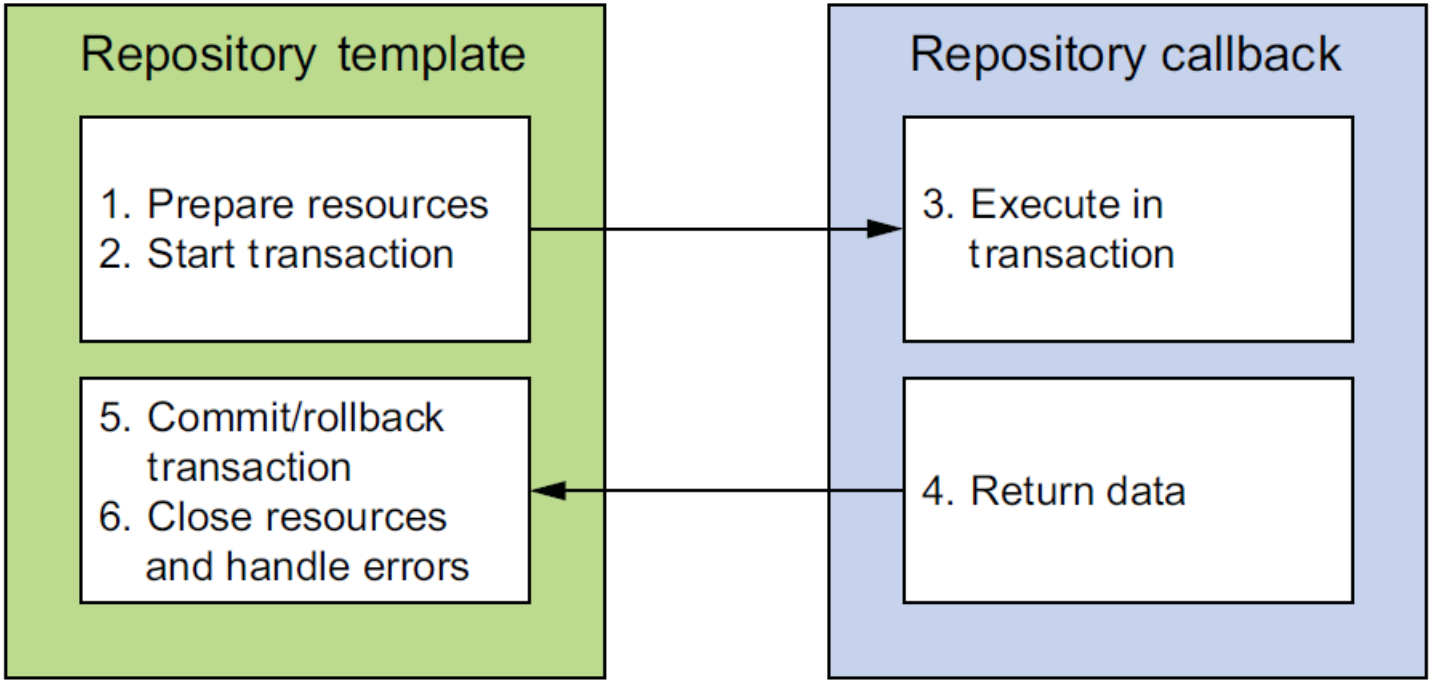
- Data Access Objects (DAO) / Repositories
- Services / Business Controllers



JDBC's exceptions	Spring's data-access exceptions
BatchUpdateException	BadSqlGrammarException
DataTruncation	CannotAcquireLockException
SQLException	CannotSerializeTransactionException
SQLWarning	CannotGetJdbcConnectionException
	CleanupFailureDataAccessException
	ConcurrencyFailureException
	DataAccessException
	DataAccessResourceFailureException
	DataIntegrityViolationException
	DataRetrievalFailureException
	DataSourceLookupApiUsageException
	DeadlockLoserDataAccessException
	DuplicateKeyException
	EmptyResultDataAccessException
	IncorrectResultSizeDataAccessException
	IncorrectUpdateSemanticsDataAccessException
	InvalidDataAccessApiUsageException
	InvalidDataAccessResourceUsageException
	InvalidResultSetAccessException
	JdbcUpdateAffectedIncorrectNumberOfRowsException

DataAccessException

BUT! …… All of them are unchecked!


Repository template
1. Prepare resources
2. Start transaction
5. Commit/rollback transaction
6. Close resources and handle errors
Repository callback
3. Execute in transaction
4. Return data

Data Sources

Data sources that are defined by a JDBC driver

Data sources that are looked up by JNDI

Data sources that pool connections

Registering a JDBC driver based data source

DriverManagerDataSource - Returns a new connection every time a connection is requested.

SimpleDriverDataSource - Works much the same as DriverManagerDataSource except that it works with the JDBC driver directly;

SingleConnectionDataSource - Returns the same connection every time a connection is requested.

Registering a JDBC driver based data source

In XML:

```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource"
p:driverClassName=“com.mysql.jdbc.Driver"
p:url=" jdbc:mysql://localhost/myDatabase "
p:username=“user1"
p:password=“12345" />
```

Registering a JDBC driver based data source

In Java configuration:

```
@Bean
public DataSource dataSource() {
    DriverManagerDataSource ds = new DriverManagerDataSource();
    ds.setDriverClassName(“com.mysql.jdbc.Driver");
    ds.setUrl(" jdbc:mysql://localhost/myDatabase ");
    ds.setUsername(“user1");
    ds.setPassword(“12345");
    return ds;
}
```

Registering a pooled data source

In XML:

```
<bean id="dataSource"
        class="org.apache.commons.dbcp.BasicDataSource"
        p:driverClassName=“com.mysql.jdbc.Driver"
        p:url="jdbc:mysql://localhost/myDatabase"
        p:username=“user1"
        p:password=“12345"
        p:initialSize=“10"
        p:maxActive=“20" />
```

Registering a pooled data source

In Java configuration:

```
@Bean
public BasicDataSource dataSource() {
    BasicDataSource ds = new BasicDataSource();
    ds.setDriverClassName(" com.mysql.jdbc.Driver ");
    ds.setUrl(" jdbc:mysql://localhost/myDatabase ");
    ds.setUsername(“user1");
    ds.setPassword(“12345");
    ds.setInitialSize(10);
    ds.setMaxActive(20);
    return ds;
}
```

Registering a JNDI data source

In XML:

```
<jee:jndi-lookup id="dataSource"
        jndi-name="jndiName"
        resource-ref="true" />
```

Registering a JNDI data source

In Java configuration:

```
@Bean
public JndiObjectFactoryBean dataSource() {
    JndiObjectFactoryBean jndiObjectFB =
                        new JndiObjectFactoryBean();

    jndiObjectFB.setJndiName(“jndiName");
    jndiObjectFB.setResourceRef(true);
    jndiObjectFB.setProxyInterface(javax.sql.DataSource.class);

    return jndiObjectFB;
}
```

Template class (`org.springframework.*`)	Used to template . . .
`jca.cci.core.CciTemplate`	JCA CCI connections
`jdbc.core.JdbcTemplate`	JDBC connections
`jdbc.core.namedparam.NamedParameterJdbcTemplate`	JDBC connections with support for named parameters
`jdbc.core.simple.SimpleJdbcTemplate`	JDBC connections, simplified with Java 5 constructs (deprecated in Spring 3.1)
`orm.hibernate3.HibernateTemplate`	Hibernate 3.*x*+ sessions
`orm.ibatis.SqlMapClientTemplate`	iBATIS SqlMap clients
`orm.jdo.JdoTemplate`	Java Data Object implementations
`orm.jpa.JpaTemplate`	Java Persistence API entity managers

JdbcTemplate

```
@Bean
@Autowired
public JdbcTemplate jdbcTemplate( DataSource dataSource ) {
        return new JdbcTemplate(dataSource);
}
```

NamedParamterJdbcTemplate

```
@Bean
@Autowired
public NamedParameterJdbcTemplate jdbcTemplate(
                                        DataSource dataSource ) {

        return new NamedParameterJdbcTemplate);

}
```

Transactions

Characteristics of transactions in Spring:

- Atomic
- Two-step commit
- Default commit
- Rollback for runtime exceptions

Enabling transactions

```
@Configuration
@EnableTransactionManagement
public class ProjectConfig {

    //…

    @Bean
    @Autowired
    public PlatformTransactionManager txManager(DataSource dataS) {
        return new DataSourceTransactionManager(dataS);
    }
}
```

Enabling transactions

```
<tx:annotation-driven/>

<bean id="dataSource"
        class="com.vendor.VendorDataSource"/>

<bean  id="transactionManager"
        class="org.sfwk...DataSourceTransactionManager">

        <constructor-arg ref="dataSource"/>

</bean>
```

@Transactional

```
@Transactional
public void myServiceMethod(){
        // do something
}


@Transactional(Propagation.REQUIRED)
public void myServiceMethod(){
        // do something
}
```

Propagation levels

Enum Constant Summary

Enum Constants

Enum Constant and Description
`MANDATORY` Support a current transaction, throw an exception if none exists.
`NESTED` Execute within a nested transaction if a current transaction exists, behave like PROPAGATION_REQUIRED else.
`NEVER` Execute non-transactionally, throw an exception if a transaction exists.
`NOT_SUPPORTED` Execute non-transactionally, suspend the current transaction if one exists.
`REQUIRED` Support a current transaction, create a new one if none exists.
`REQUIRES_NEW` Create a new transaction, suspend the current transaction if one exists.
`SUPPORTS` Support a current transaction, execute non-transactionally if none exists.

Exceptions handling in transactions

1) Spring transactions are **default rolled back** for Runtime Exceptions

2) Spring transactions are **NOT default rolled back** for checked exceptions

```
@Transactional( rollbackFor={MyCheckedException})
public void myServiceMethod(){

}


@Transactional( noRollbackFor={MyRuntimeException})
public void myServiceMethod(){

}
```

Spring and the Java Persistence API

- **Application managed**
  - ✓ LocalEntityManagerFactoryBean
- **Container managed**
  - ✓ LocalContainerEntityManagerFactoryBean

LocalEntityManagerFactoryBean

```
@Bean
public LocalEntityManagerFactoryBean emf() {
        LocalEntityManagerFactoryBean emfb
                        = new LocalEntityManagerFactoryBean();

        emfb.setPersistenceUnitName(“persistenceUnitName");

        return emfb;
}
```

JpaTransactionManager

```
@Bean
@Autowired
public PlatformTransactionManager
        transactionManager(EntityManagerFactory emf) {
    JpaTransactionManager tm= new JpaTransactionManager();

    tm.setEntityManagerFactory(emf);

    return tm;
}
```

```
@Repository
public class MyRepository{

        @PersistenceContext
        public EntityManager em;

        public void create(MyEntityObject o){
                em.persist(o);
        }
}
```

Second level cache

- SimpleCacheManager
- NoOpCacheManager
- ConcurrentMapCacheManager
- CompositeCacheManager
- **EhCacheCacheManager**
- RedisCacheManager
- GemfireCacheManager

```
@Configuration
@EnableCaching
public class CachingConfig {

        //… configuration
}
```

```
@Bean
public EhCacheCacheManager cacheManager(CacheManager cm) {
        return new EhCacheCacheManager(cm);
}
```

```
@Bean
public EhCacheManagerFactoryBean ehcache() {
    EhCacheManagerFactoryBean ehCacheFactoryBean =
                                    new EhCacheManagerFactoryBean();

    ehCacheFactoryBean.setConfigLocation(
        new ClassPathResource("eh-cache.xml"));

    return ehCacheFactoryBean;
}
```

ehcache-config.xml

```
<ehcache>
    <cache name="spittleCache"
           maxBytesLocalHeap="50m"
           timeToLiveSeconds="100">
    </cache>
</ehcache>
```