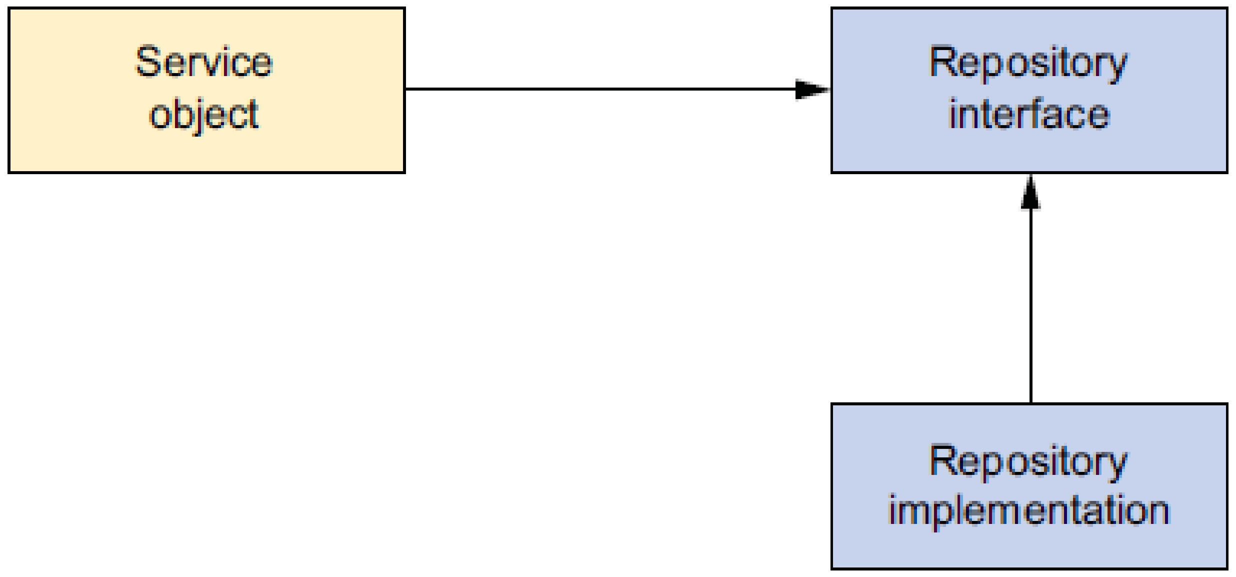
Data Access

- Spring provides support for following:
  - ✓ JDBC
  - ✓ Object Relational Mapping (ORM) - Hibernate and JPA
  - ✓ Data Access Objects (DAO)
  - ✓ NoSQL Databases
- Spring provides a convenient translation from technology-specific exceptions like SQLException to its own exception class hierarchy with the DataAccessException as the root exception.
- In addition to JDBC exceptions, Spring can also wrap Hibernate-specific exceptions, converting them from proprietary exception to a set of focused runtime exceptions (the same is true for JDO and JPA exceptions).

Repository Pattern



@Repository Annotation

```
@Repository
public class JpaMovieFinder implements MovieFinder {
   @PersistenceContext
   private EntityManager entityManager;
   ...
```

```
@Repository
public class HibernateMovieFinder implements MovieFinder {
    private SessionFactory sessionFactory;
    @Autowired
    public void setSessionFactory(SessionFactory sessionFactory) {
          this.sessionFactory = sessionFactory;
    }
    ...
```

```
@Repository
public class JdbcMovieFinder implements MovieFinder {
  private JdbcTemplate jdbcTemplate;
  @Autowired
  public void init(DataSource dataSource) {
     this.jdbcTemplate = new JdbcTemplate(dataSource);
  }
  ...
```

Spring data-access Templates

- JdbcTemplate
- NamesParameterJdbcTemplate
- SimpleJdbcTemplate
- HibernateTemplate
- JpaTemplate

DataAccessException

- All data access exception in Spring are subclasses of this exception.
- This is an unchecked exception. So you are not forced to handle data access exceptions in Spring.
- This is data access API agnostic.

Data Access - JDBC

Action	Spring	You
Define connection parameters.		X
Open the connection.	X	
Specify the SQL statement.		X
Declare parameters and provide parameter values		X
Prepare and execute the statement.	X	
Set up the loop to iterate through the results (if any).	X	
Do the work for each iteration.		X
Process any exception.	X	
Handle transactions.	X	
Close the connection, statement and resultset.	X	

JDBC driver-based data sources

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 to overcome class loading issues that may arise in certain environments, such as in an OSGi container.

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

Java Config for DataSource

```
@Bean
public DataSource dataSource() {
    DriverManagerDataSource ds = new DriverManagerDataSource();
    ds.setDriverClassName(“jdbc.oracle.driver.OracleDriver");
    ds.setUrl(“jdbc:oracle:thin:@localhost:1521:xe");
    ds.setUsername(“hr");
    ds.setPassword(“hr");
    return ds;
}
```

```
<bean id="dataSource"
  class="org.springframework.jdbc.datasource.DriverManagerDataSource"
  p:driverClassName=“jdbc.oracle.driver.OracleDriver”
  p:url="jdbc:oracle:thin:@localhost:1521:xe "
  p:username=“hr"
  p:password=“hr" />
```

JDBC classes

JdbcTemplate
The classic Spring JDBC approach and the most popular. This "lowest level" approach and all others use a JdbcTemplate under the covers.

NamedParameterJdbcTemplate
Wraps a JdbcTemplate to provide named parameters instead of the traditional JDBC "?" placeholders. This approach provides better documentation and ease of use when you have multiple parameters for an SQL statement.

SimpleJdbcInsert and SimpleJdbcCall
Optimize database metadata to limit the amount of necessary configuration. This approach simplifies coding so that you only need to provide the name of the table or procedure and provide a map of parameters matching the column names. This only works if the database provides adequate metadata. If the database doesn’t provide this metadata, you will have to provide explicit configuration of the parameters.

JdbcTemplate Methods

void execute(String sql)
Issue a single SQL execute, typically a DDL statement.

<T> List<T>query(String sql, ParameterizedRowMapper<T> rm,Object... args)
Query for a List of Objects of type T using the supplied ParameterizedRowMapper to the query results to the object.

<T> T queryForObject(String sql, Class<T> requiredType, Object... args)
Execute a query for a result object, given static SQL.

List<Map<String,Object>> queryForList(String sql, Object... args)
Execute the supplied query with the (optional) supplied arguments. Parameters are represented by ?.

<T> T queryForObject(String sql, ParameterizedRowMapper<T> rm, Object... args)
Query for an object of type T using the supplied ParameterizedRowMapper to the query results to the object.

int update(String sql, Object... args)
Executes the supplied SQL statement with (optional) supplied arguments.

JdbcTemplate Query Examples

```
int rowCount = this.jdbcTemplate.queryForObject("select
count(*) from  employees", Integer.class);


String lastName = this.jdbcTemplate.queryForObject
("select last_name from employees where employee_id = ?",
new Object[]{111}, String.class);


List<Actor> actors = this.jdbcTemplate.query
("select first_name, last_name  from employees",
  new RowMapper<Employee>() {
   public Employee mapRow(ResultSet rs, int rowNum)
                              throws SQLException {
     Employee e = new Employee();
     e.setFirstName(rs.getString("first_name"));
     e.setLastName(rs.getString("last_name"));
     return e;
   }
});
```

JdbcTemplate Examples

```
jdbcTemplate.update
("insert into jobs (job_id, job_title) values (?, ?)",
“IT_DBA", “Oracle DBA");

jdbcTemplate.update("delete from jobs where job_id = ?“,
“IT_DBA”));


this.jdbcTemplate.execute
("create table mytable (id integer, name varchar(100))");
```

SimpleJdbcInsert Examples

```
public class JdbcActorDao {
 private SimpleJdbcInsert insertJob;
 public void setDataSource(DataSource dataSource) {
   this.insertActor = new SimpleJdbcInsert(dataSource)
   .withTableName("jobs");
 }

 public void add(Job job) {
  Map<String, Object> parameters =
          new HashMap<String, Object>(2);
  parameters.put("job_id", job.getId());
  parameters.put("job_title", job.getTitle());
  insertJob.execute(parameters);
 }

}
```

Spring Transaction Management

- ❑ **Provides a 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.**
- ❑ **You write your code once, and it can benefit from different transaction management strategies in different environments**

Global vs. Local Transactions

- Global transactions enable you to work with multiple transactional resources, typically relational databases and message queues.
- The application server manages global transactions through the JTA
- EJB Container Managed Transaction (CMT) is a way to handle global transactions
- Usage of global transaction confines application to Application server as only Application server provides JTA
- A JDBC transaction is called local transaction
- They are easy to use but cannot work with multiple resources

Transaction Managers

- Spring provides PlatformTransactionManager interface
- Its methods throw unchecked exeption TransactionException

```
public interface PlatformTransactionManager {
  TransactionStatus getTransaction
        (TransactionDefinition definition)
                throws TransactionException;
  void commit(TransactionStatus status)
                throws TransactionException;
  void rollback(TransactionStatus status)
                throws TransactionException;
}
```

TransactionStatus

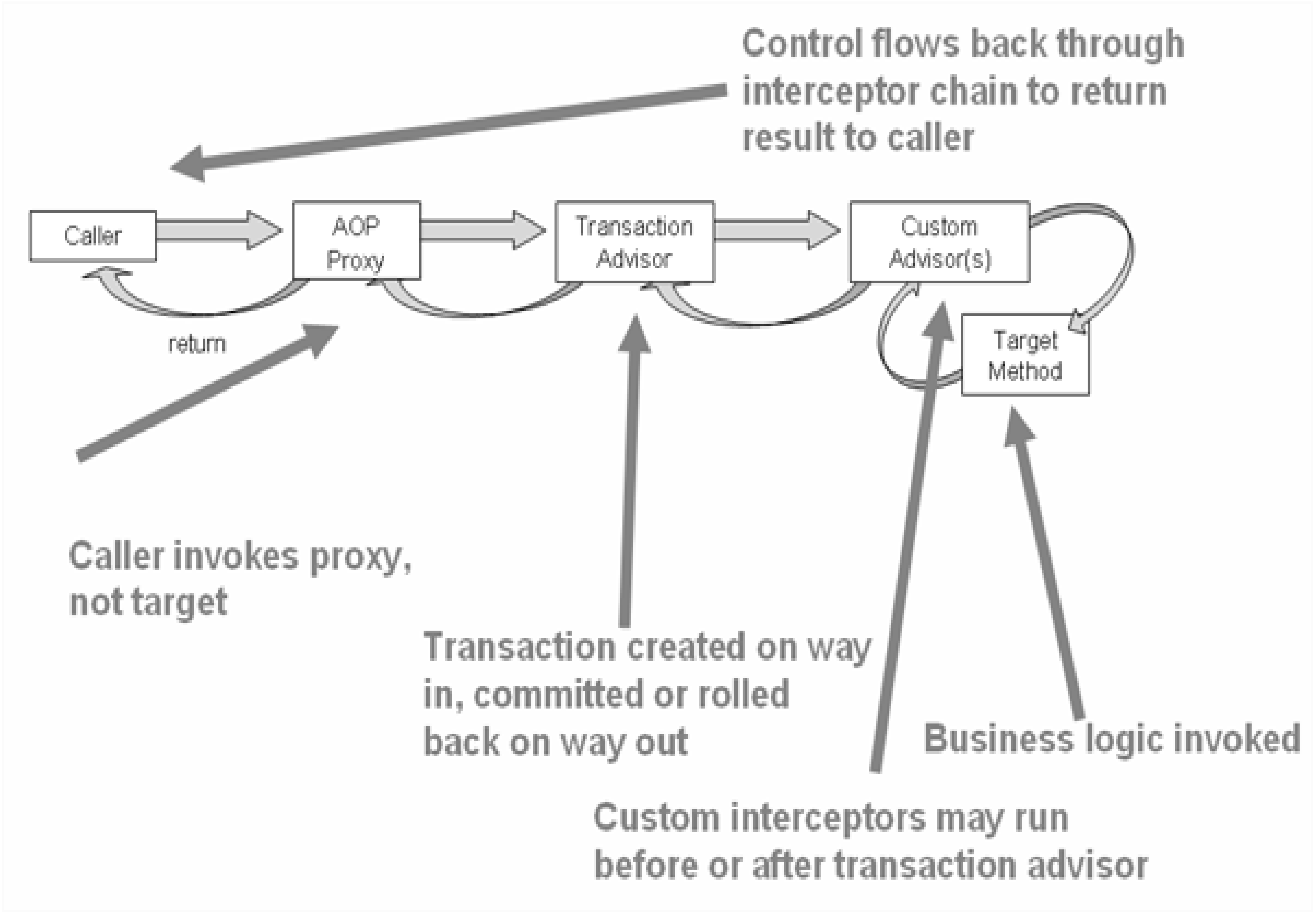
- TransactionStatus interface provides methods to control transaction and query transaction status

```
public interface TransactionStatus extends SavepointManager {
 boolean isNewTransaction();
 boolean hasSavepoint();
 void setRollbackOnly();
 boolean isRollbackOnly();
 void flush();
 boolean isCompleted();
}
```

Available Transaction Managers

- DataSourceTransactionManager
- JtaTransactionManager
- HibernateTransactionManager

Transaction Management


Control flows back through interceptor chain to return result to caller
Caller
AOP Proxy
Transaction Advisor
Custom Advisor(s)
Target Method
return
Caller invokes proxy, not target
Transaction created on way in, committed or rolled back on way out
Business logic invoked
Custom interceptors may run before or after transaction advisor

When a transaction is rolled back?

- To rollback work is to throw an Exception from code that is currently executing in the context of a transaction.
- In its default configuration, the Spring Framework's transaction infrastructure code *only marks a* transaction for rollback in the case of runtime, unchecked exceptions
- Checked exceptions that are thrown from a transactional method do *not result in rollback in the* default configuration.
- You can configure exactly which Exception types mark a transaction for rollback, including checked exceptions using configuration.

```
<tx:advice id="txAdvice" transaction-manager="txManager">
   <tx:attributes>
       <tx:method name="get*“  rollback-for="NoProductInStockException"/>
       <tx:method name="*"/>
  </tx:attributes>
</tx:advice>
```

Steps In Using Hibernate

- Create Entity classes
- Create mapping files
- Configure DataSource
- Configure Hibernate SessionFactory and provide Hibernate related properties
- Create a class to have SessionFactory property and inject SessionFactory into it.
- Get Session from SessionFactory and use it for Hibernate API

As of Spring 4.0 , Spring requires Hibernate 3.6 or later.

Hibernate Configuration

```
<bean id="mySessionFactory“
class="org.springframework.orm.hibernate4.LocalSessionFactoryBean“>
   <property name="dataSource" ref="oracledatasource"/>
   <property name="mappingResources">
     <list>
       <value>hibernate/job.hbm.xml</value>
     </list>
   </property>
   <property name="hibernateProperties">
     <value>
       hibernate.dialect=org.hibernate.dialect.OracleDialect
     </value>
   </property>
</bean>
```

Hibernate Example

```
public List<Job> getJobs() {
  return (List<Job>) sessionFactory.openSession()
             .createQuery("from Job").list();
}
```