Module 4
Java Web Applications
Spring Data Access

Spring and Enterprise Data Access

- Provide comprehensive data access support
  - Resource management
  - Connection management
  - Transaction management
  - Exception handling
- Enable a layered application architecture
  - To isolate business logic from the complexity of data access
- Support is provided for all major data access technologies
  - JDBC
  - JBoss Hibernate and JPA
  - Apache iBatis

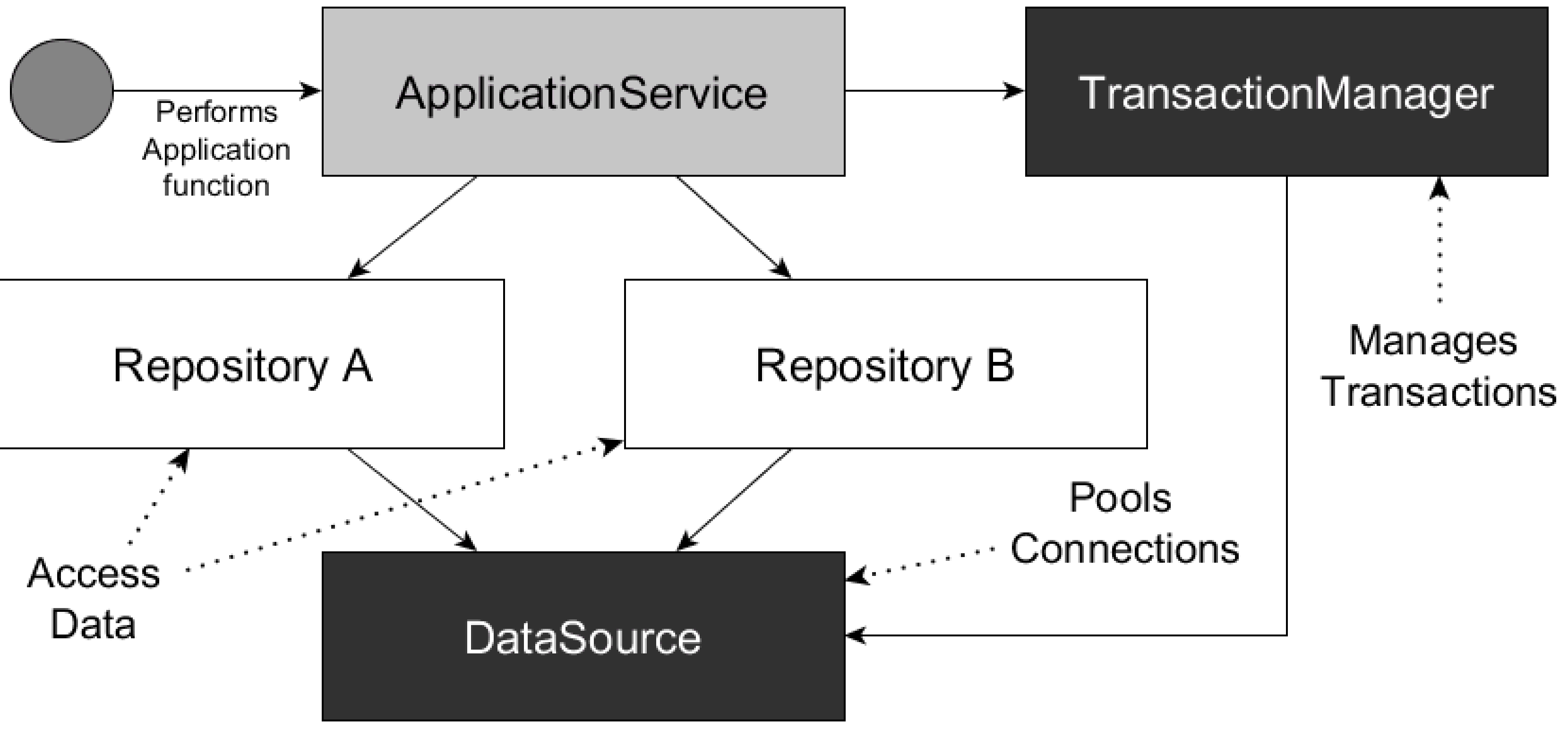
Transaction Management Overview

- Spring provides a transaction management abstraction
- Implementation for all major data access technologies
  - Java Transaction API (JTA)
  - JDBC
  - Hibernate
  - Java Persistence API (JPA)
  - Java Data Objects (JDO)
- Declarative and Programmatic transaction management
- Integration with Spring's data access abstractions

Data Source

- Spring uses a standard JDBC data source for acquiring connections
- It can integrate JEE container managed data source

Spring Data Access Integration



Database Access In Java

- Two specifications:
  - JDBC (Java Database Connectivity)
  - JPA (Java Persistence API)
- JDBC is low level
  - Access to everything from the DB
  - A lot of Java code needed even for simple things
- JPA is high level
  - Object-Relational mapping (ORM)
  - Automatic conversion between objects and database tables (views, stored procedures, cursors)

Spring JDBC

- Spring abstracts JDBC in order to
  - Reduce error prone code
  - Eliminate boilerplate code
  - Ease result sets processing
  - Handle exceptions properly

JDBC Abstraction Added Value

- Spring's JDBC abstraction added value

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	

JdbcTemplate

- JdbcTemplate is central class in the Spring JDBC core

```
List results = jdbcTemplate.query(someSql,
new RowMapper(){
 public Object mapRow(ResultSet rs, int row) throws SQLException{
    // map the current row to an object
  }
});
```

```
class JdbcTemplate {
  public List query(String sql, RowMapper rowMapper){
       try{
            // acquire connection
            // prepare statement
            // execute statement
            // for each row in the result set
               results.add(rowMapper.mapRow(rs,rowNumber));
             return results;
        } catch (SQLException e){
            // convert to root cause exception
        } finally{
            // release connection
        }
  }
```

Create a JdbcTemplate

- Requires a DataSource

```
public class JdbcBookRepository implements BookRepository {

    private JdbcTemplate jdbcTemplate;

    public JdbcBookRepository(DataSource dataSource) {
        this.jdbcTemplate=new JdbcTemplate(dataSource);
    }
…
}
```

- Create a template once and re-use it
  - Thread safe after construction

Querying with JdbcTemplate

- JdbcTemplate can query for
  - Simple types (int, long, String)
  - Generic Maps
  - Domain Objects
- Can return each row of a `ResultSet` as `Map`
- When expecting a single row
  - Use `queryForMap(..)`
- When expecting multiple rows
  - Use `queryForList(..)`

RowMapper

- Generic interface for mapping a single row of ResultSet to an object
  - Can be used for both single and multiple row queries

```
public interface RowMapper<T> {
       T mapRow(ResultSet rs, int rowNum) throws SQLException;
}
```

```
 class BookRowMapper implements RowMapper<Book>{

 @Override
 public Book mapRow(ResultSet rs, int rowNum) throws SQLException {
    return mapBook(rs);
 }
}
```

RowCallbackHandler

- You can use when there is no return object
  - Streaming rows to a file
  - Converting rows to XML
  - Filtering rows before adding to a Collection (SQL is preferable)

```
public class JdbcBorrowingsRepository {
 public void generateReport(Writter out){
   JdbcTemplate.query(query, new BorrowingsReportWritter(out);
 }
}
```

```
 class BorrowingsReportWritter implements RowcallbackHandler{
 @Override
 public void processRow(ResultSet rs) throws SQLException {
    // parse current row from ResultSet and stream to output
 }
}
```

ResultSetExtractor

- Use for processing an entire ResultSet at once
  - You are responsible for iterating the ResultSet
  - e.g. for mapping entire ResultSet to a single data object

```
public class JdbcBorrowingsRepository {
 public Borrowing findByCustomer(String cartId){
   return JdbcTemplate.query(query,
                                     new BorrowingExtractor(),cartId);
 }
}
```

```
 class BorrowingExtractor implements ResultSetExtractor<Borrowing>{
 @Override
 public Borrowing extractData(ResultSet rs) throws SQLException {
    // create Borrowing object from ResultSet
 }
}
```

Insert and Update

```
private int insert(Book book){
 String sql = "insert into BOOK (ID, CATID, TITLE, PUBLISHER, AUTHOR, ISBN, GENRE)"
             + " values (?,?,?,?,?,?,?)";
 return jdbcTemplate.update(sql,book.getId(),
                               book.getCatId(),
                               book.getTitle(),
                               book.getPublisher(),
                               book.getAuthor(),
                               book.getIsbn(),
                               book.getGenre());
}
```

```
private int update(Book book){
 String sql = "update BOOK
              set CATID=?, TITLE=?, PUBLISHER=?, AUTHOR=?, ISBN=?, GENRE=?
              where ID=?";
 return jdbcTemplate.update(sql, book.getCatId(),
                        book.getTitle(),
                        book.getPublisher(),
                        book.getAuthor(),
                        book.getIsbn(),
                        book.getGenre(),
                        book.getId());
}
```

SQLException Handling

- SQLExceptions are not explicitly caught in most cases
- Can create a leaky abstraction
  - Must propagate if it is not caught
    - Catch and Wrap
    - Being swallowed
  - Bad portability
  - Testability suffers
- JdbcTemplate ensures that SQLExceptions are handled in a consistent and portable fashion
  - Resources are always released properly
  - Generic SQLExceptions are translated to DataAccessExceptions

    (refer `org.springframework.dao` package)

JDBC Namespace

- Provides support for embedded databases
- Useful for testing
- Supports H2, HSQL and Derby

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
…
<jdbc:embedded-database id="dataSource">
    <jdbc:script location="classpath:schema.sql"/>
    <jdbc:script location="classpath:test-data.sql"/>
</jdbc:embedded-database>
…
</beans>
```

Java Persistence API (JPA)

- API for the management of persistence and object/relational mapping
- Build on top of JDBC
- POJO Persistence mode
- Part of the JEE specification
- Can be used outside JEE
- Vendor Independent

JPA General Concepts

- `PersistenceUnit` describes group of persistent classes
  - You can have multiple units per application
  - Defines JPA providers (Hibernate, Eclipselink,...)
  - Defines Transaction types (local, JTA)
- `EntityManager` represents a unit of work
  - Known as Persistent context
  - Corresponds at a higher-level to a connection
  - Manages persistent objects within unit of work
- `EntityManagerFactory` represents a single data source
  - Shareable and thread safe
  - Provides access to transactional EntityManager

JPA Providers

- Several major JPA spec implementations available
  - Jboss Hibernate
  - EclipseLink
  - Apache OpenJPA

JPA Mapping

- JPA requires metadata for mapping Java classes/properties to database tables/columns
  - Preferable is to use Java annotations
  - XML configuration also available
- JPA uses convention over configuration
  - No need to provide metadata for obvious
  - Metadata relies on defaults
- Support for relations mappings
  - Collections of entities
- Support for entities inheritance

JPA Annotations

```
@Entity
@Table(name=“BOOK_TABLE”)
public class Book {
    @Id
    private Long id;
    @Column(name=”TITLE”)
    private String title;
    private String author;

    ...
}
```

JPA Querying

- Several options for querying JPA entities
  - Retrieve objects by primary key
  - Retrieve objects with Java Persistence Query Language (JPQL)
  - Retrieve objects using native SQL

Setting Up EntityManagerFactory

- Spring supports three ways to setup EntityManagerFactory
  - LocalEntityManagerFactoryBean
  - LocalContainerEntityManagerFactoryBean
  - JNDI lookup
- Requires persistence.xml configuration file
  - Stored in META-INF directory
  - Specifies Persistence Unit
  - JPA provider

persistence.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
             http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">
  <persistence-unit name="libraryMaster"/>
</persistence>
```

LocalEntityManagerFactoryBean

- Creates an EntityManagerFactory suitable for simple deployment environments where the application uses only JPA
- Useful for standalone applications and integration testing
  - Cannot specify a DataSource
  - No support for global transactions

```
<beans>
 <bean id="myEmf"
       class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
    <property name="persistenceUnitName" value="libraryMaster"/>
 </bean>
</beans>
```

LocalContainerEntityManagerFactoryBean

- Provides full JPA capabilities
  - You can use with containers (Tomcat)
  - Standalone applications and integration tests
- Integrates with existing DataSources
- Useful when fine-grained customization needed

```
<beans>
 <bean id="myEmf"
       class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="libraryMasterDataSource"/>
  <property name="loadTimeWeaver">
   <bean class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver"/>
  </property>
 </bean>
</beans>
```

JNDI Lookup

- Use this option when deploying to a Java EE server

```
<beans>
 <jee:jndi-lookup id="myEmf" jndi-name="persistence/myPersistenceUnit"/>
</beans>
```

Spring Transactions and EntityManager

- To transparently participate in Spring Managed transactions
  - Configure FactoryBeans for building an EntityManagerFactory
  - Inject an EntityManager reference using @PersistenceContext anotation
- Define a Transaction Manager
  - JpaTransactionManager
  - JtaTransactionManager

JPA Dao Example

- Notice there is no Spring dependency

```
public class JpaBookRepository implements BookRepository {

    @PersistenceContext
    private EntityManager entityManager;

    @Override
    public List<Book> findAll() {
        // use entityManager
    }
...
}
```

JPA Configuration

```
<beans>
 <bean id="entityManagerFactory"
       class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="libraryMasterDataSource"/>
  ...
 </bean>

 <bean id="transactionManager"
       class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="entityManagerFactory"/>
 </bean>

 <bean id=”bookRepository” class=”ite.librarymaster.dao.JpaBookRepository”/>

 <context:annotation-config/>
</beans>
```

Transparent Exception Translation

- Spring can translate `PersistenceException`s into `DataAccessException` out of the box
  - Annotate DAO class with @Repository annotation
  - Define a Spring-provided BeanPostProcessor

```
@Repository
public class JpaBookRepository implements BookRepository {
...
}
```

```
<bean class=”org.springframework.dao.annotation.
             PersistenceExceptionTranslationPostProcessor”/>
```