Introduction to Data Access with Spring

A look at Data Access in Spring from simple SQL queries to Spring Data JPA and NoSql implementations

Who am I

- Frank Moley
- Internet Architect, Garmin International
- Languages: Java, C#, C++, SQL, Python, PHP, VB
- SpringSource Certified Spring Professional
- Social
  - Twitter: @fpmoles
  - GitHub: fpmoles
  - Web: www.frankmoley.com

Agenda

- Introduction to Project
- Data Access with SQL and Spring JdbcTemplate
- Introduction to Spring Data
- Data Access for Relational Databases using JPA/ Hibernate and Spring-Data-JPA
- Data Access for NoSql Databases Spring-Data-MongoDb
- Summary

“Git” the Materials

https://github.com/fpmoles/data-bound-spring

Introduction to Project

- Provide a system to store Personally Identifiable Information
- Provide a simple RESTful interface to serve and collect the data
- We will focus on the data access in this presentation
- Storage of such information as names, email address(es), phone number(s), and address(es)

Data Access with Spring JdbcTemplate

Why Show This?

- To get the full picture of what Spring-Data abstracts for the developer, I would like to start with the most rudimentary version using Spring
- JdbcTemplate handles the connection open, commit, rollback, and closing
- Also handles ResultSet mapping to objects
- Simple enough, but there is a lot of code

Spring JdbcTemplate Infrastructure

```
public abstract class PiiRepository {

    protected NamedParameterJdbcTemplate jdbcTemplate;
    protected Properties sqlProperties;
    protected final static String INSERT = "insert";
    protected final static String GET = "get";
    protected final static String UPDATE = "update";
    protected final static String DELETE = "delete";
    protected final static String FIND_ALL = "get_all";

    PiiRepository(DataSource dataSource, Properties sqlProperties){
        super();
        this.jdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
        this.sqlProperties = sqlProperties;
    }
```

Spring JdbcTemplate Infrastructure (cont)

```java
public class PersonEntityRepository extends PiiRepository {

    private final static String PARAM_PERSON_ID = "personId";
    private final static String PARAM_PREFIX = "prefix";
    private final static String PARAM_FIRST_NAME = "firstName";
    private final static String PARAM_MIDDLE_NAME = "middleName";
    private final static String PARAM_LAST_NAME = "lastName";
    private final static String PARAM_SUFFIX = "suffix";

    public PersonEntityRepository(DataSource dataSource, Properties sqlProperties) {
        super(dataSource, sqlProperties);
    }
```

Spring JdbcTemplate

```java
@Transactional(propagation = Propagation.REQUIRED)
public PersonEntity addPerson(PersonEntity model){
    String personId = UUID.randomUUID().toString();
    String sql = this.sqlProperties.getProperty(INSERT);
    Map<String, Object> namedParameters = new HashMap<>(6);
    namedParameters.put(PARAM_PERSON_ID, personId);
    namedParameters.put(PARAM_PREFIX, StringUtils.trimToNull(model.getPrefix()));
    namedParameters.put(PARAM_FIRST_NAME, StringUtils.trimToNull(model.getFirstName()));
    namedParameters.put(PARAM_MIDDLE_NAME, StringUtils.trimToNull(model.getMiddleName()));
    namedParameters.put(PARAM_LAST_NAME, StringUtils.trimToNull(model.getLastName()));
    namedParameters.put(PARAM_SUFFIX, StringUtils.trimToNull(model.getSuffix()));
    jdbcTemplate.update(sql, namedParameters);
    return this.getPerson(personId);
}
```

Configuration

```
<context:property-placeholder location="classpath*:/piiDataServices.properties"/>
<context:component-scan base-package="com.frankmoley.services.pii"/>
<mvc:annotation-driven />

<jdbc:embedded-database id="piiDataSource" type="H2">
    <jdbc:script location="classpath*:/databaseCreate.sql"/>
    <jdbc:script location="classpath*:/preloadData.sql"/>
</jdbc:embedded-database>

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <constructor-arg name="dataSource" ref="piiDataSource"/>
</bean>

<tx:annotation-driven transaction-manager="transactionManager"/>

<bean id="piiRepository" class="com.frankmoley.services.pii.data.repository.PiiRepository" abstract="true">
    <constructor-arg name="dataSource" ref="piiDataSource"/>
</bean>

<bean id="personRepository" class="com.frankmoley.services.pii.data.repository.PersonEntityRepository" parent="piiRepository">
    <constructor-arg name="sqlProperties">
        <util:properties location="classpath:/com.frankmoley.services.pii.data.repository/person.properties"/>
    </constructor-arg>
</bean>
```

Introduction to Spring-Data

Why Spring-Data

- Provides common interface for most data access technologies
- Allows the developer to swap out data sources with limited code changes
- Focus on the business logic, not on the data access technology

Repository

- Core object of Spring Data is the Repository
- Interface based on the Repository design pattern
- Get a piece of data, come back and get the next piece
- No massive join/aggregation paths in complex DAO methods

Crud Repository

- The Crud Repository is the primary “implementation” of the Repository Pattern in Spring Data - it is an interface that Spring proxies with an implementation
- Extends the Repository marker interface
- Provides the expected operations; save, findOne, delete. Also supports findAll, exists, save (in batch mode), count, and delete (in batch mode)

PagingAndSortingRepository

- Extension of CrudRepository adds the ability to sort or page results from the findAll method
- Very useful in Restful Repository based micro services among other locations

Extending Repository

- You can use standard language to add dynamic functionality to your repository
- Uses reflection based on bean notation to create queries
- Supported syntax is based on datastore technology, but all follows same syntax

Requirements

- Repository is based on generics, each definition requires two object definitions
  - Entity definition
  - Id definition
- The major difference between technologies at this point comes from the Entity definition

Example Entity

```java
@Entity
public class Foo implements Serializable {
    private static long serialVersionUID = 1L;

    @Id
    private long id;
    private String name;
    private long partNumber;
    private String serialNumber;
    private boolean active;

    public long getId() { return id; }

    public void setId(long id) { this.id = id; }

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public long getPartNumber() { return partNumber; }

    public void setPartNumber(long partNumber) { this.partNumber = partNumber; }

    public String getSerialNumber() { return serialNumber; }

    public void setSerialNumber(String serialNumber) { this.serialNumber = serialNumber; }

    public boolean isActive() { return active; }

    public void setActive(boolean active) { this.active = active; }
}
```

Example Repository

```
@Repository
public interface FooRepository extends PagingAndSortingRepository<Foo, Long> {

    Foo findByName(String name);

    List<Foo> findByNameLike(String name);

    Foo findByPartNumber(long partNumber);

    List<Foo> findByNameOrSerialNumber(String name, String serialNumber);

    List<Foo> findByActiveIsTrueOrderByPartNumberAsc();
}
```

Benefits

- By utilizing the Repository pattern and interfaces, creating data access is similar if not identical across several different data storage engines
- Swapping out datasources becomes trivial, as we will see

Spring-Data-JPA

ORM Abstraction

- Builds off of the JPA model to handle all aspects of EntityManager
- Transactional boundaries still managed in your code (or preferred through transactional proxies)
- No need to interact with Hibernate or any other JPA implementation, Spring handles the dependencies

First Class Entity Support

- Support of relationship models: @OneToOne, @OneToMany etc
- EntityManager injection through Spring allows for the container to manage the EntityManager as well as associated caches
- Full transaction support through EntityManager and Transactional Proxies

XML Config

```
<jpa:repositories base-package="com.frankmoley.services.pii.data.repository"/>

<jdbc:embedded-database id="piiDataSource" type="H2">
    <jdbc:script location="classpath*:/databaseCreate.sql"/>
    <jdbc:script location="classpath*:/preloadData.sql"/>
</jdbc:embedded-database>

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

<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="piiDataSource"/>
    <property name="packagesToScan" value="com.frankmoley.services.pii.data.entity"/>
    <property name="jpaVendorAdapter">
        <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"/>
    </property>
</bean>

<tx:annotation-driven transaction-manager="transactionManager"/>
```

Entity Example

```java
@Entity(name = "PERSON")
public class PersonEntity {

    @Id
    @Column(name = "PERSON_ID")
    private String id;
    @Column(name="PREFIX")
    private String prefix;
    @Column(name="FIRST_NAME")
    private String firstName;
    @Column(name = "MIDDLE_NAME")
    private String middleName;
    @Column(name="LAST_NAME")
    private String lastName;
    @Column(name="SUFFIX")
    private String suffix;

    public PersonEntity(){
        super();
        //becaue I want to do pure JPA, I need to generate the id here instead of using the Hibernate generator
        if(null==id){
            this.id = UUID.randomUUID().toString();
        }
    }
```

Repository Example

```
@Repository
@Transactional
public interface PersonEntityRepository extends CrudRepository<PersonEntity, String> {
    //marker interface for now

}
```

Spring Data MongoDB

NoSql Support

- MongoDB is just an example, most NoSql data packages are the same with respect to Spring
- Support for Neo4j, CouchDB, Gemfire, Hadoop, ElasticSearch, and others
- Reduces need to learn or remember the driver details, just implement it with Spring Data

Same Paradigm

- Still just uses a repository and an entity
- Query syntax may require some special considers with the datasource, for instance gemFire requires the entity to be on the server for OQL
- Mongo supports the ability to index fields on the document

XML Config

```
<mongo:db-factory id="mongoFactory" host="127.0.0.1" port="27017" dbname="piidemo"/>

<bean id="mongoTemplate" class="org.springframework.data.mongodb.core.MongoTemplate">
    <constructor-arg name="mongoDbFactory" ref="mongoFactory"/>
    <property name="writeConcern">
        <util:constant static-field="com.mongodb.WriteConcern.SAFE"/>
    </property>
</bean>

<mongo:repositories base-package="com.frankmoley.services.pii.data.repository"
                    mongo-template-ref="mongoTemplate"/>
```

Entity

```
*/
@Document(collection = "PERSON")
public class PersonEntity {

    @Id
    private String id;
    private String prefix;
    private String firstName;
    private String middleName;
    private String lastName;
    private String suffix;

    public PersonEntity(){
        super();
        if(null==id){
            this.id = UUID.randomUUID().toString();
        }
    }
```

Repository

```
@Repository
public interface PersonEntityRepository extends MongoRepository<PersonEntity, String> {
    //marker interface for now

}
```

Summary

Simple Data Access

- Spring Data allows for the most simple form of data access, zero boiler plate code

- Config is show for XML, learn Java Config however, it is the future

- RestRepositories allow you to build micro services with zero effort for data access. Spring boot makes this even faster.

Questions?