Spring Data

CS544: Enterprise Architecture

Overview

- In this module we will first discuss what exactly spring data is, and how it works
- Then we will look at finder methods
  - Basics, general features, and more advanced

Spring Data

ABOUT SPRING DATA

What is Spring Data

- Spring Data offers a flexible abstraction for working with data access frameworks
- Purpose is to unify and ease the access to different kinds of persistence stores
- Both relational db systems and NoSQL data stores
- Addresses common difficulties developers face when working with databases in applications
- Spring Data is an umbrella project that provides you with easy to use data access technologies for all kinds of relational and non-relational DBs

Spring Data Modules

Modules supports:

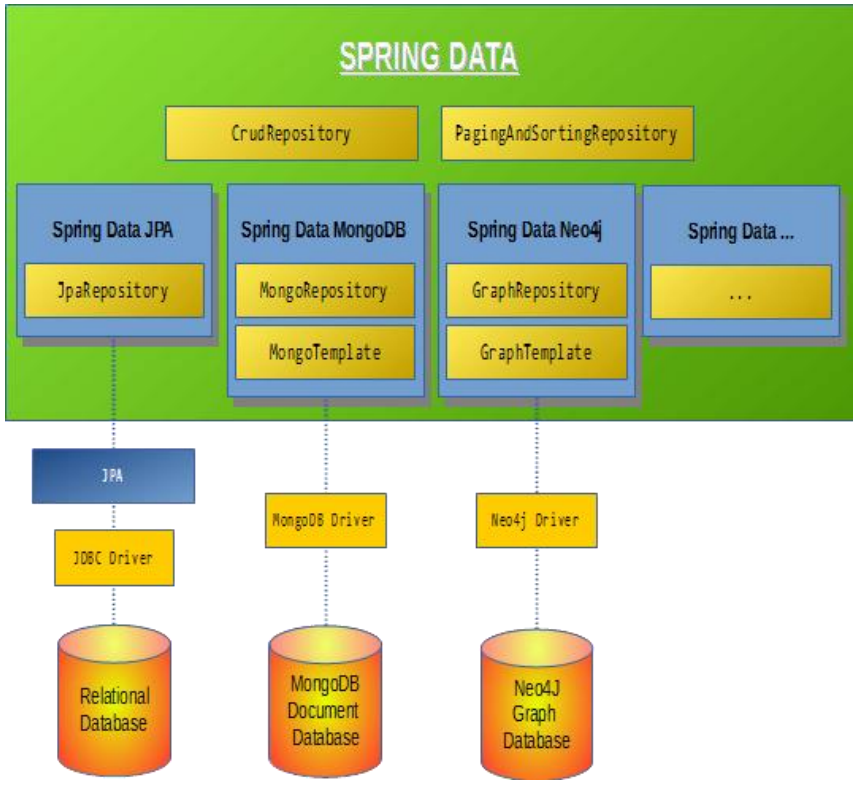
- Templating
- Object/Datastore mapping
- Repository support

Every repository offers:

- CRUD operations
- Finder Methods
- Sorting and Pagination

Spring data module repositories provide generic interfaces:

- CrudRepository
- PagingAndSortingRepository



Mapping

- JPA uses Object Relational Mapping
- Spring Data extends this concept to NoSQL datastores
- All Spring Data modules provide an Object to datastore mapping
- Because these data store structures can be so different, there is no common API
- Each type of datastore has it's own set of annotations to map between objects and datastore structures.

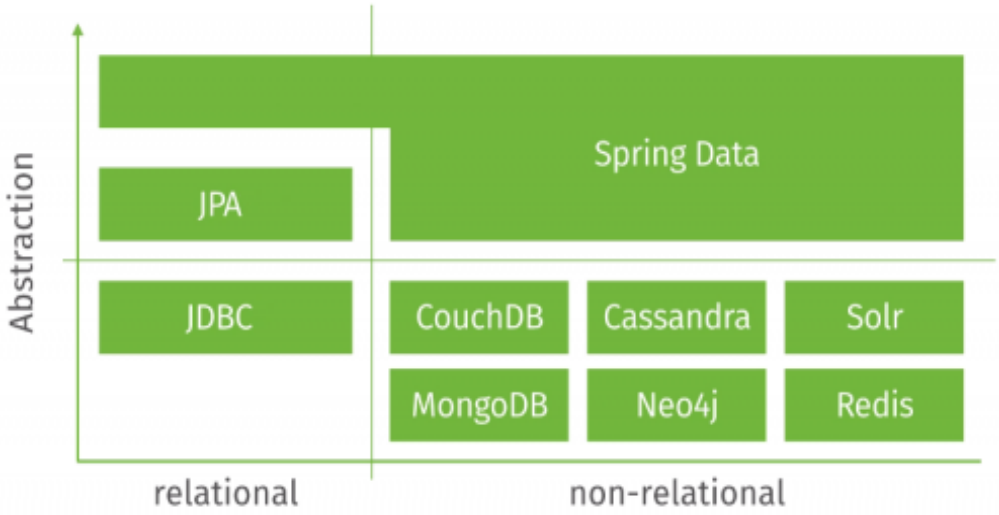
Mapping Examples

JPA	MongoDB	Neo4J

`@Entity`
`@Table(name="USR")`
`public class User {`
 	
`  @Id`
`  private String id;`
 	
`  @Column(name="fn")`
`  private String name;`
 	
`  private Date lastLogin;`
 	
`...`
`}`
	`@Document(`
	`collection="usr")`
	`public class User {`
	
	`  @Id`
	`  private String id;`
	
	`  @Field("fn")`
	`  private String name;`
	
	`  private Date lastLogin;`
	
	`  ...`
	`}`
		`@NodeEntity`
		`public class User {`
		
		`  @GraphId`
		`  Long id;`
		
		`  private String name;`
		
		`  private Date lastLogin;`
		
		`...`
		`}`

Spring Data Templates

- Each type of DB has its own template that:
  - Connection Management
  - CRUD operations, Queries, Map/Reduce jobs
  - Exception Translation to DataAccessExceptions
- No template for JPA
  - It provides most of these already (is abstraction)



MongoDB Template Example

```
<!-- Connection to MongoDB server -->
<mongo:db-factory host="localhost" port="27017" dbname="test" />

<!-- MongoDB Template -->
<bean id="mongoTemplate"
   class="org.springframework.data.mongodb.core.MongoTemplate">
  <constructor-arg name="mongoDbFactory" ref="mongoDbFactory"/>
</bean>
```

DAOs aka Repositories

- DAOs (also known as repositories) are classes that implement CRUD operations and finder methods for each Entity

- Spring Data generates DAOs that use its templates. These DAOs:
  - Same basic interface regardless of data store
  - Provide a common way of querying for data
  - Provide a CRUD interface (if reasonable for DS)

`CrudRepository` Interface

- Spring Data JPA is instructed to scan package
- DAOs are generated for any **interface** that extends **`Repository<T, id>`**
  - **`JpaRepository extends PagingAndSortingRepostirory extends CrudRepository extends Repository`**

count()	saveAll(Iterable<S> entities)
exists(ID id)	save(S entity)
findAll()	delete(ID id)
findAllById(Iterable<ID> ids)	delete(T entity)
findById(id)	deleteAll(Iterable<? extends T> entities)
getOne(ID id)	deleteAll()

Example

See exercise COV1

Springconfig.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jpa="http://www.springframework.org/schema/data/jpa"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/data/jpa
       http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

    ...

    <jpa:repositories base-package="cs544.cov1.dao"/>

</beans>
```

Also needs EntityManagerFactory, not SessionFactory!

ContactDao.java

```
public interface ContactDao extends JpaRepository<Contact, Long> {
}
```

ContactService.java

```
@Service
@Transactional(propagation = Propagation.REQUIRED)
public class ContactService {

    @Resource
    private ContactDao contactDao;

    ...
}
```

Spring Data

FINDER METHODS

Adding Better Methods

- The basic provided methods may be enough for a small application
  - Real application need more specific methods

- You can add finder methods by using the name of the property

```
public interface PersonDAO extends JpaRepository<Person, Integer> {

    List<Person> findByName(String name);

}
```

Assuming our Person entity has a name property

Finder Method Prefixes

- You can use the following introducing prefixes:
  - **`findBy`**, **`readBy`**, **`queryBy`**, **`getBy`**, and **`countBy`**

- These can contain further expressions such as:
  - **`Distinct`**, **`Top10`**, **`First5`**

```
List<Person> findDistinctPersonByLastname(String lastname);
List<Person> findPersonDistinctByLastname(String lastname);

Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);
Slice<User> findTop3ByLastname(String lastname, Pageable pageable);
List<User> findFirst10ByLastname(String lastname, Sort sort);
List<User> findTop10ByLastname(String lastname, Pageable pageable);
```

Criteria

- The first By acts as a delimiter to indicate the start of the actual criteria

```
List<Person> findDistinctPersonByLastname(String lastname);
```

- You can combine criteria with **And** and **Or**

```
List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);
List<Person> findByLastnameOrFirstname(String lastname, String firstname);
```

- Typical operators are also supported (up next)

Spring Data

FINDER METHOD FEATURES

List of Keywords

And	After	Containing
Or	Before	OrderBy
Is,Equals	IsNull	Not
Between	IsNotNull, NotNull	In
LessThan	Like	NotIn
LessThanEqual	NotLike	True
GreaterThan	StartingWith	False
GreaterThanEqual	EndingWith	IgnoreCase (AllIgnoreCase)

Examples:

```
findByFirstname,findByFirstnameIs,findByFirstnameEquals // all the same
findByStartDateBetween // expects two parameters
findByAgeLessThanEqual
findByAgeIsNotNull, findByAgeNotNull // same
findByFirstnameLike // first parameter matched as Like (including %’s)
findByAgeIn(Collection<Age> ages) // or subclass of collection
```

IgnoreCase

- You can ignore case for a single property

```
List<Person> findByFirstNameAndLastnameIgnoreCase(String firstname, String lastname);
```

Case is only ignored for lastName

- Or ignore case for all properties

```
List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);
```

PagingAndSortingRepository

- The PagingAndSortingRepository interface adds methods to sort and paginate entities
  - `findAll(Pageable pageable), findAll(Sort sort)`

- Enables you to use the OrderBy keyword

```
public List<Student> findByLastNameOrderByLastNameAsc(String lastName);
public List<Student> findByLastNameOrderByLastNameDesc(String lastName);
```

- You can request a page using **`PageRequest`**

```
PageRequest pageRequest = new PageRequest(0, 10);
Page<Student> page = studentDao.findAll(pageRequest);
Slice<Student> slice = studentDao.findByFirstName("Lisa", pageRequest);
```

Page, Slice, and Sort

- Page
  - To obtain the total number of element and pages available, the pages will do a *select count(*) derived from the query you triggered*

```
Page<Student> findByLastname(String lastname, Pageable pageable);
```

- Slice
  - Only knows if there is a next page
  - It selects an additional row from the PageRequest's size to determine if there are more records to retrieve

```
Slice<Student> findByLastname(String lastname, Pageable pageable);
```

- Sort
  - The sorting options are handled through the Pageable instance

```
List<Student> findByLastname(String lastname, Sort sort);
```

Limiting Results

- The keywords first and top will limit the amount of rows that will be returned
  - Optional numeric value van be added

```
User findFirstByOrderByLastnameAsc();
User findTopByOrderByAgeDesc();
Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);
Slice<User> findTop3ByLastname(String lastname, Pageable pageable);
List<User> findFirst10ByLastname(String lastname, Sort sort);
List<User> findTop10ByLastname(String lastname, Pageable pageable);
```

- If pagination or slicing is applied, it is done within the limited results

Spring Data

ADVANCED METHODS

Property Expressions

- Similar to HQL you can traverse properties:

```
List<Student> findByAddressZipCode(ZipCode zipCode);// x.address.zipCode
```

- What if Student has a property called "AddressZipCode"?

```
List<Student> findByAddress_ZipCode(ZipCode zipCode);// x.address.zipCode
```

Optional y underscores can be used to separate properties

Java methods should use CamelCase, there should not be any conflicts caused by this

@Query

- But what if you don't like typing long names, or need to have a really complicated query?

```
public interface UserRepository extends JpaRepository<User, Long> {

  @Query("select u from User u where u.emailAddress = ?1")
  User findByEmailAddress(String emailAddress);
}
```

- Or want to write SQL instead of HQL?

```
public interface UserRepository extends JpaRepository<User, Long> {

  @Query(value = "SELECT * FROM USERS WHERE EMAIL_ADDRESS = ?0", nativeQuery = true)
  User findByEmailAddress(String emailAddress);
}
```

Custom Functionality

- You can also write your own custom methods
  - Allowing you to do whatever you want

Step 1: create an interface

```
interface UserDaoCustom {
  public void someCustomMethod(User user);
}
```

Step 2: create an implementation

```
class UserDaoImpl implements UserDaoCustom {

  public void someCustomMethod(User user) {
    // Your custom implementation
  }
}
```

Step 3: add the interface to where you want to add the functionality

```
interface UserDao extends JpaRepository<User, Long>, UserDaoCustom {

  // Declare query methods here
}
```

Active Learning

- Write a finder method that retrieves a student based on his firstname or lastname

- Write another finder method to Retrieve students whose age is less than 25 ordered by studentId descending

Summary

- Spring data allows you to generate DAOs independent of the type of data store you use
- The default methods of the generated DAOs are good for simple applications
- It’s easy to add additional Finder methods using nothing but the method names
- Custom queries or complete method implementations are also possible

Main Point

- Spring Data can generate our DAO's, and if we want finder methods we can just specify what we want.
- *Science of Consciousness*: Do less and accomplish more