Spring Boot and Spring Data for Backing Stores

Simplifying JPA setup and implementation using Spring Boot and Spring Data Repositories

1.18.5

Objectives

After completing this lesson, you should be able to do the following

- Implement a Spring JPA application using Spring Boot
- Create Spring Data Repositories for JPA

Agenda

- **Spring JPA using Spring Boot**
- Spring Data – JPA
- Lab
- Advanced Topics

Spring JPA “Starter” Dependencies

- Everything you need to develop a Spring JPA application

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
</dependencies>
```

Resolves

spring-boot-starter.jar
spring-boot-starter-jdbc.jar
spring-boot-starter-aop.jar
spring-data-jpa.jar
hibernate-core
javax.transaction-api
…

Spring Boot and JPA

- If JPA is on classpath, Spring Boot automatically
  - Auto-configures a **`DataSource`**
  - Auto-configures an **`LocalContainerEntityManagerFactoryBean`**
  - Auto-configures a **`JpaTransactionManager`**

- You can customize (will be covered in subsequent slides)
  - **`LocalContainerEntityManagerFactoryBean`**
  - **`JpaTransactionManager`**

LocalContainerEntityManagerFactoryBean Setup *without* Spring Boot

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {

        HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
        adapter.setShowSql(true);
        adapter.setGenerateDdl(true);
        adapter.setDatabase(Database.HSQL);

        Properties props = new Properties();
        props.setProperty("hibernate.format_sql", "true");

        LocalContainerEntityManagerFactoryBean emfb =
                        new LocalContainerEntityManagerFactoryBean();
        emfb.setDataSource(dataSource);
        emfb.setPackagesToScan("rewards.internal");
        emfb.setJpaProperties(props);
        emfb.setJpaVendorAdapter(adapter);

        return emfb;
}
```

Boot can implement this for us
– so how do we customize it?

Customize EntityManagerFactoryBean - Entity Locations

- Where to find entities?
  - By default, Boot looks in same package as class annotated with **`@EnableAutoConfiguration`**
    - And all its sub-packages
  - You can override using **`@EntityScan`**

```
@SpringBootApplication
@EntityScan("rewards.internal")
public class Application {
    //...
}
```

Customize EntityManagerFactoryBean - Configuration Properties

- You can specify vendor-provider properties

```
# Leave blank - Spring Boot will try to select dialect for you
# Set to 'default' - Hibernate will try to determine it
spring.jpa.database=default

# Create tables automatically? Default is:
#    Embedded database: create-drop
#    Any other database: none (do nothing)
# Options: validate | update | create | create-drop
spring.jpa.hibernate.ddl-auto=update

# Show SQL being run (nicely formatted)
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true

# Any hibernate property 'xxx'
spring.jpa.properties.hibernate.xxx=???
```

application.properties

JPA Configuration without Spring Boot

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
        ...
    return entityManagerFactoryBean;
}


@Bean
public PlatformTransactionManager
                        transactionManager(EntityManagerFactory emf)  {
   return new JpaTransactionManager(emf);
}


@Bean
public DataSource dataSource()  {   /* Lookup via JNDI or create locally */  }
```

JPA Configuration with Spring Boot

```
@Bean
public LocalContainerEntityManagerFactoryBean entityMa[illegible]Factory() {
        ...
    return entityManagerFactoryBean;
}


@Bean
public PlatformTransactionMana[illegible]
                        transacti[illegible]nager(EntityManagerFactory emf)  {
   return new JpaTransac[illegible]anager(emf);
}


@Bean
public [illegible]Source dataSource()  {   /* Lookup via JNDI or create locally */  }
```

No longer needed!

Replaced By ..

- One annotation

Application.java

```
@SpringBootApplication
@EntityScan("rewards.internal")
public class Application {
    //...
}
```

- Some properties

application.properties

```
# Show SQL being run (nicely formatted)
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format-sql=true
spring.datasource…
spring.sql.init…
```

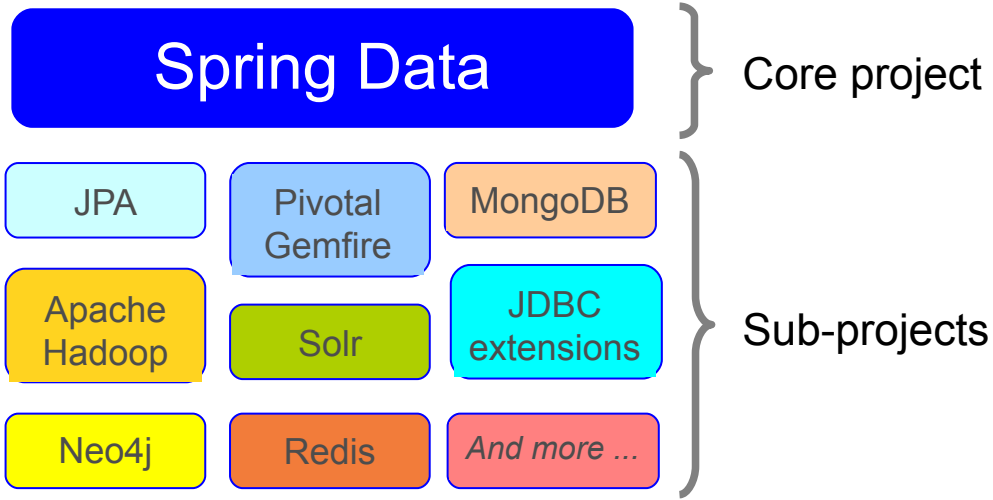
- And *lots* of defaults

Agenda

- Spring JPA using Spring Boot
- **Spring Data – JPA**
- Lab
- Advanced Topics

What is Spring Data?

- Reduces boiler plate code for data access
- Works in many environments



Instant Repositories

SPRING DATA

- How?
  - **Step 1:** Annotate domain class
    - define keys & enable persistence
  - **Step 2:** Define your repository as an *interface*

- Spring Data will implement it at run-time
  - Scans for interfaces extending Spring Data Common Repository<T, K>
  - CRUD methods auto-generated if using CrudRepository<T, K>
  - Paging, custom queries and sorting supported
  - Variations exist for most Spring Data sub-projects

Step 1: Annotate Domain Class

Here we are using JPA

SPRING DATA

- Annotate JPA Domain class - standard JPA

```
@Entity
@Table(...)
public class Customer {

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;
  private Date orderDate;
  private String email;

  // Other data-members and getters and setters omitted
}
```

Domain Class

Note: Key is a *Long*

Domain Classes: Other Data Stores

- Spring Data provides similar annotations to JPA for other Data stores
  - *@Document*, *@Region*, *@NodeEntity* …

MongoDB – map to a JSON document

```
@Document
public class Account {
  ...
```

```
@NodeEntity
public class Account {
  @GraphId
   Long id;
  ...
```

Neo4J – map to a graph

Gemfire – map to a region

```
@Region
public class Account {
  ...
```

Step 2a: Choose a Repository Interface to extend

```
public interface Repository<T, ID> { }
```

Marker interface: it does not have any methods

```
public interface CrudRepository<T, ID extends Serializable>
        extends Repository<T, ID> {

    public long count();
    public <S extends T> S save(S entity);
    public <S extends T> Iterable<S> save(Iterable<S> entities);

    public Optional<T> findById(ID id);
    public Iterable<T> findAll();
    public Iterable<T> findAllById(Iterable<ID> ids);

    public void deleteAll(Iterable<? extends T> entities);
    public void delete(T entity);
    public void deleteById(ID id);
    public void deleteAll();
}
```

`PagingAndSortingRepository<T, K>`
- adds `Iterable<T> findAll(Sort)`
- adds `Page<T> findAll(Pageable)`

Step 2b: Define your Repository Interface - Option #1

- You can add finders - must obey naming convention
  - **find(First)By*<DataMemberOfClass><Op>***
  - <Op> can be GreaterThan, NotEquals, Between, Like …

id

```
public interface CustomerRepository extends CrudRepository<Customer, Long> {

    public Customer findFirstByEmail(String someEmail);     // No <Op> for Equals
    public List<Customer> findByOrderDateLessThan(Date someDate);
    public List<Customer> findByOrderDateBetween(Date d1, Date d2);

    @Query("SELECT c FROM Customer c WHERE c.email NOT LIKE '%@%'")
    public List<Customer> findInvalidEmails();
}
```

Custom query uses query-language of underlying product (here JPQL)

Step 2b: Define your Repository interface - Option #2

Extend `Repository` and build your own interface – all using conventions.

```
import org.springframework.data.repository.Repository;
import org.springframework.data.jpa.repository.Query;

public interface CustomerRepository extends Repository<Customer, Long> {

    <S extends Customer> save(S entity);   // Definition as per CrudRepository
    Optional<Customer> findById(Long i);    // Definition as per CrudRepository

    Customer findFirstByEmailIgnoreCase(String email);  // Case insensitive search

    @Query("select u from Customer u where u.emailAddress = ?1")
    Customer findByEmail(String email);      // ?1 replaced by method param
}
```

Finding Your Repositories

- Spring Boot automatically scans for repository interfaces
  - Starts in package of `@SpringBootApplication` class
    - Scans all sub-packages

- Or you can control scanner manually

Specify packages to scan

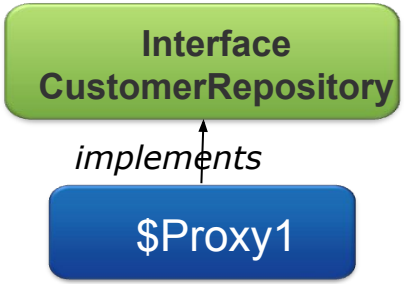
```
@Configuration
@EnableJpaRepositories(basePackages="com.acme.repository")
public class CustomerConfig { … }
```

Internal Behavior – Another Spring Proxy

- Spring Data implements your repositories at run time
  - Creates instances as Spring Beans

- *Before startup*

- *After startup*



Accessing the Repository - It is a regular bean during runtime

```
@Configuration
@EnableJpaRepositories(basePackages="com.acme.repository")
public class CustomerConfig {


    @Bean
    public CustomerService customerService(CustomerRepository repo) {
        return new CustomerService( repo );
    }
}
```

You can inject your repository bean as a dependency

Summary

- Spring Boot significantly simplifies setup for data access
  - Will set up most of JPA for you
- Similarly, Spring Data simplifies Repositories
  - Just define an interface with your finder methods