Pivotal

Spring Boot and Spring Data for Backing Stores

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

spring by Pivotal

Objectives

After completing this lesson, you should be able to

- 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-configure a **`DataSource`**
  - Auto-configure an **`EntityManagerFactoryBean`**
  - Auto-configure a **`JpaTransactionManager`**

- You can customize
  - **`EntityManagerFactoryBean`**
  - Transaction manager

EntityManagerFactory 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
  - Override using **`@EntityScan`**

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

```
setPackagesToScan("rewards.internal");
```

Customize EntityManagerFactoryBean

Configuration Properties

- Specifying 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 entityMan      actory() {
          ...
    return entityManagerFactoryBean;
}


@Bean
public PlatformTransactionMana
                            transacti      nager(EntityManagerFactory emf)  {
   return new JpaTransa      anager(emf);
}


@Bean
publi      aSource 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...
```

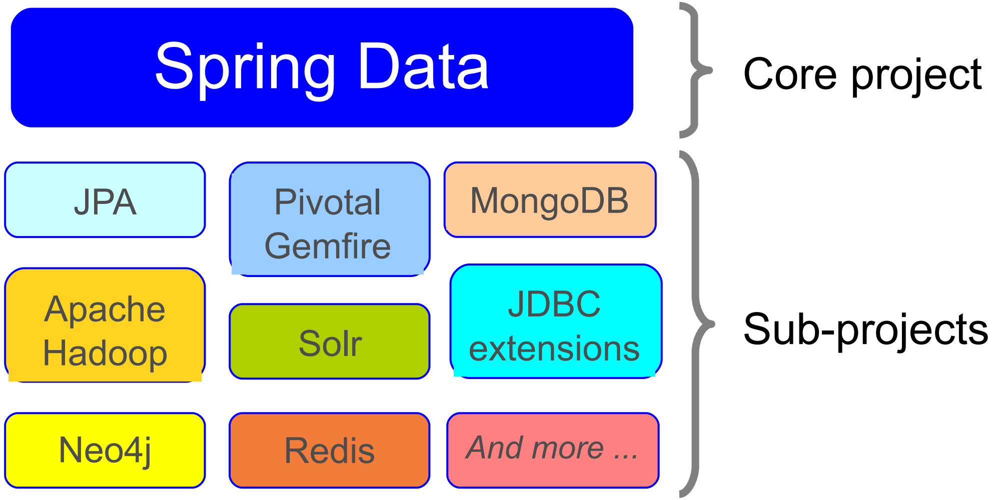
- And *lots* of defaults

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



Spring Data Philosophy

- Provide similar support for NoSQL databases that Spring does for RDBMS
  - Template classes to hide low-level, repetitive code
  - Common data-access exceptions

- But in addition, can implement repositories for you
  - We will show JPA
  - Works similarly for MongoDB, Gemfire, Neo4j ...

Instant Repositories

- 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 object as normal
  - Standard JPA

Domain Class

```
@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
}
```

Note: Key is a *Long*

Domain Objects: Other Data Stores

- Spring Data provides similar annotations to JPA
  - *@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 2: Define a Repository Interface

Must extend Repository<T, ID>

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

Marker interface – add any methods from *CrudRepository* and/or add custom finders

```
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)`

Defining a JPA Repository

- Auto-generated finders obey naming convention
  - find(First)By*<DataMember>**<Op>*
  - <Op> can be GreaterThan, NotEquals, Between, Like …

```
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();
}
```

id

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

Convention over Configuration

- **Note:** CustomerRepository is an *interface* (*not a class!*)

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
    Customer findOne(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

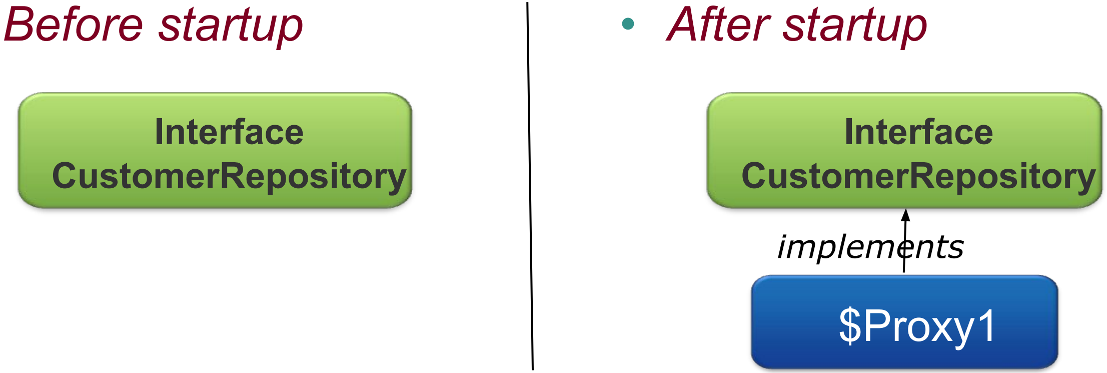
```
@Configuration
@EnableJpaRepositories(basePackages="com.acme.repository")
public class CustomerConfig { … }
```

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

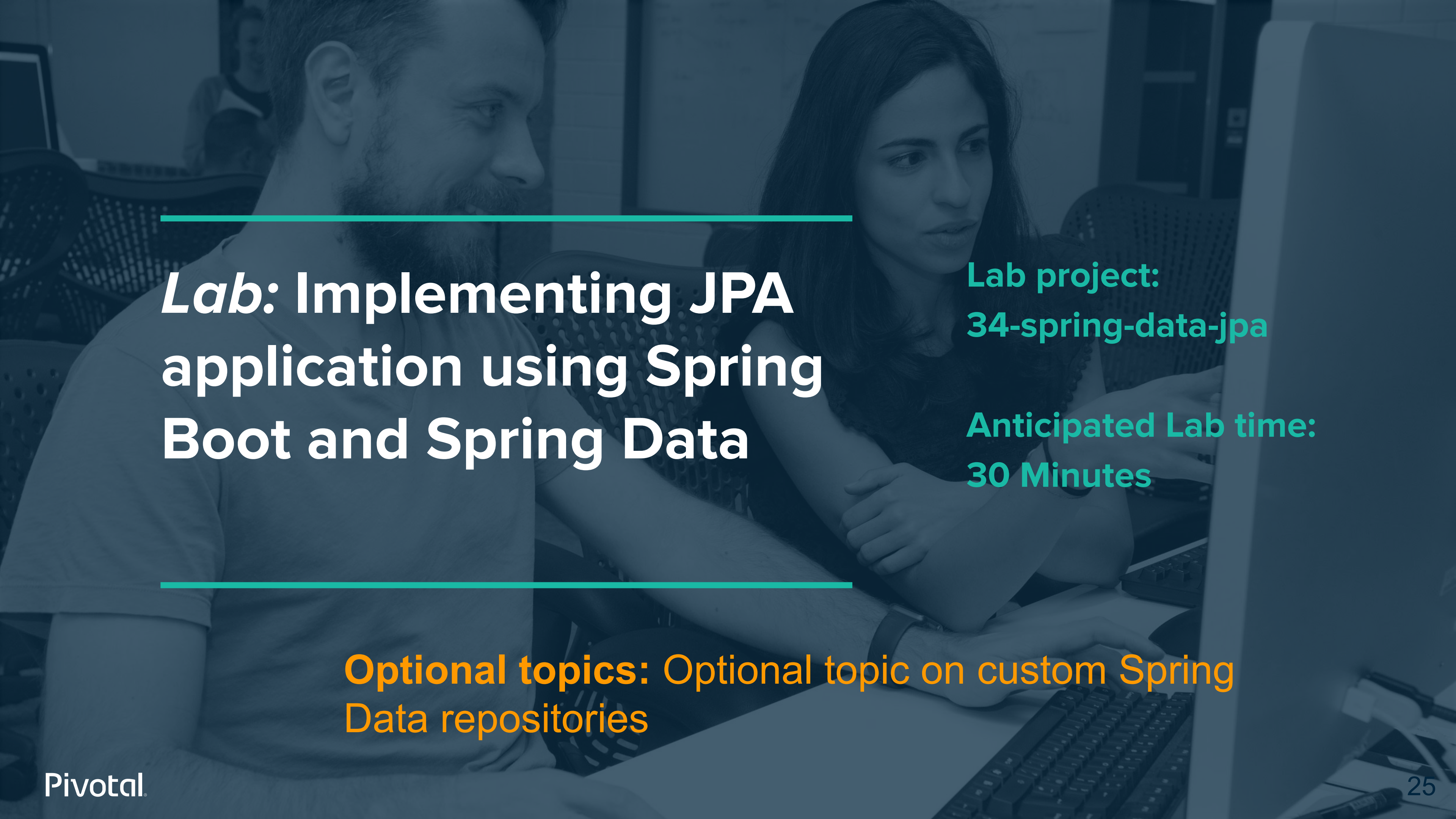
- Use Spring to inject `CustomerRepository` dependency

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

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

Summary

- Spring Boot significantly simplifies Spring setup
  - Will set up most of JPA for you

- Similarly, Spring Data simplifies Repositories
  - Just define an interface - you need no code!



Lab: Implementing JPA application using Spring Boot and Spring Data

Lab project:

34-spring-data-jpa

Anticipated Lab time:

30 Minutes

Optional topics: Optional topic on custom Spring Data repositories

Agenda

- Spring JPA using Spring Boot
- Spring Data – JPA
- Lab
- **Optional and Advanced Topics**
  - **Customized Spring Data Repositories**

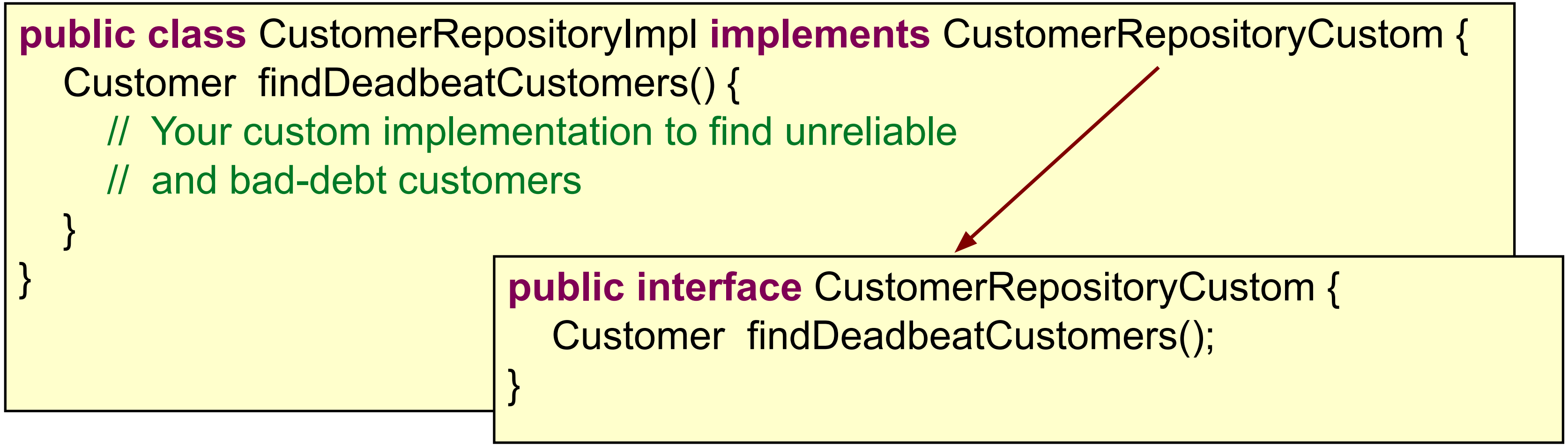
JPA Specific Interface

- Adds EntityManager specific options

```
public interface  JpaRepository<T, ID extends Serializable>
          extends PagingAndSortingRepository<T, ID> {

   <S extends T> S saveAndFlush(S entity);
   void flush();

   // Implemented as a single DELETE
   void deleteInBatch(Iterable<T> entities);
   void deleteAllInBatch();

   // Returns a lazy-loading proxy, using JPA's EntityManager.getReference()
   //  – equivalent to Hibernate's Session.load()
   T getOne(ID id);
}
```

Adding Custom Behavior (1)

- Not all use cases satisfied by automated methods
  - Enrich with custom repositories: *mix-ins*
- **Step 1**: Create normal interface and implementation



```
public class CustomerRepositoryImpl implements CustomerRepositoryCustom {
   Customer  findDeadbeatCustomers() {
      //  Your custom implementation to find unreliable
      //  and bad-debt customers
   }
}
```

```
public interface CustomerRepositoryCustom {
   Customer  findDeadbeatCustomers();
}
```

Adding Custom Behavior (2)

- **Step 2:** Combine with an automatic repository:

```
public interface CustomerRepository
        extends CrudRepository<Account, Long>, CustomerRepositoryCustom {
}
```

- Spring Data looks for implementation class or bean
  - Class or bean name = repository interface + “Impl”
    - This convention (*Impl*) is configurable
  - Either class: **`CustomerRepositoryImpl`**
    Or bean: **`CustomerRepositoryImpl`**
  - *Result:* **`CustomerRepository`** bean contains automatic and custom methods!

Using Optional

- Some methods can return null or Optional

```
public interface CustomerRepository extends Repository<Customer, Long> {
    // CRUD method using object type – returns null if not found
    Customer findOne(Long id);
    // Query method using object type – also returns null if not found
    Customer findFirstByEmail(String someEmail);
}
```

OR

```
public interface CustomerRepository extends Repository<Customer, Long> {
    // CRUD method using Optional
    Optional<Customer> findOne(Long id);
    // Query method using Optional
    Optional<Customer> findFirstByEmail(String someEmail);
}
```

Topics Covered

- Spring JPA using Spring Boot
- Spring Data – JPA
- Optional and Advanced Topics
  - Customized Spring Data Repositories