5. Persistence Frameworks

DGP Solutions

Spring and JPA

- A connection to a database is represented by an EntityManager instance, which also provides functionality for performing operations on a database. An EntityManager has a **PersistenceContext** where "Managed" entity instances are cached
- The main role of an EntityManagerFactory instance is to support instantiation of EntityManager instances. An EntityManagerFactory is constructed for a specific database or persistence unit
- Spring provides classes for managing/injecting JPA persistence units and EntityManagers
- It is considered the better practice for integrating Spring and JPA rather than using Spring Template classes such as JpaTemplate, which will not be covered here.

- JPA is a specification, not implementation, it specifies a set of interfaces (and methods) for interacting with a variety of ORMs
- JPA abstracts the specifics of an ORM
- Hibernate can be used as an implementation of JPA (Provider)
- A Transactionally scoped EntityManager is injected into a class via @PersistenceContext

```
@PersistenceContext
private EntityManager em;
public Employee get (long id) {
    return em.find(Town.class, id);
}
```


Using Spring Data JPA

- The Java Persistence API (JPA) is the standard way of persisting Java objects into relational databases
- JPA consists of two parts: a mapping subsystem to map classes into relational tables as well as an EntityManager API to access the objects, define and execute queries, and more
- The Spring Data JPA module implements the Spring Data Commons repository abstraction to ease the repository implementations even more, making a manual implementation of a repository obsolete in most cases
- The central interface in Spring Data is the Repository abstraction

CrudRepository

- The Repository interface takes the domain class to manage as well as the id type of the domain class as its type arguments
- This interface acts primarily as a marker interface to capture the types to work with and to help you to discover interfaces that extend this one
- The CrudRepository provides sophisticated CRUD functionality for the entity class that is being managed. By simply extending this repository, we will enable proxy generation of the core CRUD methods by the spring-boot-starter-data-jpa jar

```
public interface TownRepository extends CrudRepository<Town, Long>{}
```


Dynamic Finders

- Queries can be created using naming conventions;
- Collection<Town> findByCityAndCounty(String x, String y) → Select e from Town e where e.name = x and e.county = y
- Returns a Collection of Town objects

Keyword	Sample	JPQL snippet
And	findByLastnameAndFirstname	… where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	… where x.lastname = ?1 or x.firstname = ?2
Between	findByStartDateBetween	… where x.startDate between 1? and ?2
LessThan	findByAgeLessThan	… where x.age < ?1
GreaterThan	findByAgeGreaterThan	… where x.age > ?1
IsNull	findByAgeIsNull	… where x.age is null
IsNotNull,NotNull	findByAge(Is)NotNull	… where x.age not null
Like	findByFirstnameLike	… where x.firstname like ?1
NotLike	findByFirstnameNotLike	… where x.firstname not like ?1
OrderBy	findByAgeOrderByLastnameDesc	… where x.age > ?1 order by x.lastname desc
Not	findByLastnameNot	… where x.lastname <> ?1

Defining Queries in your Repository

```
public interface TownRepository extends CrudRepository<Town, Long>{
    Collection<Town> findByCityAndCounty(String x, String y);

    @Query("SELECT e FROM Town e where e.city = :x AND e.county = :y AND e.phoneCode
    > :z")
    Collection<Town> get(String x, String y, int z);

    @Query("SELECT e FROM Town e where e.city = :x AND e.county = :y AND e.phoneCode
    > :z")
    Collection<Town> getWithParam(@Param("x") String county, @Param("y") String
    city, @Param("z") int phone);
}
```


PagingAndsortingRepository

- PagingAndSortingRepository is an extension of CrudRespository that provides methods for pagination and sorting.
- The method findAll is overloaded with or without a Pageable
- A Pageable object can contain properties used to specify at least, page size, current page number and perhaps sorting

```
public interface TownRepository extends PagingAndSortingRepository<Town,
Long>{
}
```

```
PageRequest pageable = PageRequest.of(0, 5, Sort.by("name") );
Iterable<Town> sortedtowns = repository.findAll(pageable);
sortedTowns.forEach(System.out::println);
```

@Query and @Param

```
public interface EmployeeRepository extends
PagingAndSortingRepository<Employee, Long>{
   Collection<Town> findByCityAndCounty(String x, String y);

   @Query("SELECT e FROM Town e where e.city = :x AND e.county = :y AND
   e.phoneCode > :z")
   Collection<Town> get(String x, String y, int z);

   @Query("SELECT e FROM Town e where e.city = :x AND e.county = :y AND
   e.phoneCode > :z")
   Collection<Town> getWithParam(@Param("x") String county, @Param("y")
   String city, @Param("z") int phone);

   Page<Town> findByCounty(String x, Pageable p);
}
```


JPA configuration

- If you are using auto-configuration, repositories will be searched from the package containing your main configuration. By default, JPA databases will be automatically created only if you use an embedded database (H2, HSQL, or Derby). You can explicitly configure JPA settings using spring.jpa.* properties
- Below, we have told Spring Boot NOT to generate a database schema from the Entity classes but instead let the jdbc actuator jar generate the schema and data sql scripts for an H2 database

```
spring.jpa.hibernate.ddl-auto=none
spring.jpa.generate-ddl=false
spring.jpa.properties.hibernate.hbm2ddl.auto=none
```

A new Dao

- Our repository is injected into a service class to take the place of a dao. The spring Container has generated a proxy implementation of the interface by this time

```
@RestController
@RequestMapping("/town")
public class TownController {
    @Autowired private TownRepository repo;

    @GetMapping(path="/{id}", produces=
    {MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE})
    public Town get(@PathVariable("id") long id) {
         return repo.findById(id)
              .orElse(repo.findById(1L).get());
    }
```



```
@GetMapping(path="/county/{county}")
public Collection<Town> getIt(@PathVariable("county")
String county) {
  PageRequest pageable = PageRequest.of(0, 5,
Sort.by("city") );
  return repo.findByCounty(county,
pageable).getContent();
}
```

Lab 5.1 JPA Repositories

Lab 5.1 PagingAndSortingRepository

- Objective: Use a JPA Repository as a data access object in a RestController
- In the lab setup directory for this lab, replace the class HealthPlan in your project with the new Class in the setup. This class has JPA annotations already placed in it. Copy the dependencies in the file maven.txt into your pom.xml inside the dependencies tag.
- Create the interface com.health.repository.HealthPlanRepository extends PagingAndSortingRepository<HealthPlan, Long>. Create abstract methods as in the notes.


```
List<HealthPlan> findByZipAndName(int zip, String name);
List<HealthPlan> findByNameAndCopayLessThan(String name, double copay);

@Query("SELECT plan FROM HealthPlan plan WHERE (plan.name = :name AND
plan.outOfPocketIndividual < :oop AND plan.deductableIndividual = :deductableInd)
")
List<HealthPlan> finderNameOutOfPocketDeductable(@Param("name") String name,
@Param("oop") int oop, @Param("deductableInd") int deductableInd);
```

Lab 5.1 – Using the Repository

- In the class HealthPlanController,
  - Delete the ,member variable for the existing dao. Replace it with an injected HealthPlanRepository called repo using @Autowired; Replace the delegation to the dao in each controller method to the repo instead
    - Method get(long id) → now returns repo.findById(id).get()
    - Method getEntity(long id) → now has repo.findById(id).get() in the body
    - Method getAll(Optional<String> optional) → now returns ;
      - ((Collection<HealthPlan>) repo.findAll()).stream().filter(p-> p.getName().equalsIgnoreCase(type)).collect(Collectors.toList());
    - Method add(HealthPlan plan → Replace method contents with code in lab setUp file post.txt

- Test your application as before

Spring Data Rest

- The goal of the Spring Data REST project is to provide a solid foundation on which to expose CRUD operations to your JPA Repository-managed entities using plain HTTP REST semantics
- With minimal code, one can create REST representations of JPA entities that follow the HATOAS principle. A REST client enters a REST application through a simple fixed URL. All future actions the client may take are discovered within representations of resources returned from the server i.e. XML, JSON
- In our case, a service returns a textural response that contains other URLs to other services


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

@RepositoryRestResource

- To expose our repository as a Restful endpoint, use the @RepositoryRestResource annotation

```
@RepositoryRestResource(collectionResourceRel="districts", path
="towns")
public interface TownRepository extends PagingAndSortingRepository<Town,
Long>{
```

- The @RepositoryRestResource enables you to define the root node in exported JSON responses i.e. “districts” and the path to the endpoint that the repostory implementation has been mapped to i.e. “/towns”

JSON Responses

- When you start the application up, the Repository has been exposed through a number of URLs. The basic http://localhost:8082/basic/staff URL to the mapped endpoint will list JSON representations of resources returned by the findAll of your repository. To the right, we see our “collection” exposed as JSON with HATOAS links


```
employees: [
{
firstName: "Fred",
lastName: "Flintstone",
dept: 200,
_links: {
self: {
href:
"http://localhost:8082/basic/staff/1"
}
}
},
{
firstName: "Eric",
lastName: "Colbert",
dept: 300,
_links: {
self: {
href:
"http://localhost:8082/basic/staff/2"
}
}
},
```


If you click on one of the hyperlinks (HATOAS links), you will activate the findOne(id) method of the repository to return JSON formatted output of an individual resource (employee)


```
{firstName: "Fred",
lastName: "Flintstone",
dept: 200,
_links:
{self:
{href: "http://localhost:8082/basic/staff/1"
}
}
}
```

Query Methods

- To execute queries, “search” is added to the URLTemplate, followed by the method name that will trigger the method in the Repository. For example, the URL, http://localhost:8082/basic/staff/search/findByDept?dept=100
- Will invoke our method in the repository

```
public Iterable<Employee> findByDept(@Param(“dept”)long zip);
```

- To return JSON (as in the notes). Note the search prefix and query string parameters in the URL that correspond by name to our @Param argument.

19920612 Copyright DGP Solutions 5-17

- If we want to hide a Repository method from being exposed as a service endpoint, we do the following:

```
@RestResource(exported=false) //hide
public String findSomething(String foo);
```

```
{
_embedded: {
employees: [
{
firstName: "Jane",
lastName: "Doe",
dept: 100,
_links: {
self: {
href: "http://localhost:8082/basic/staff/4"
}
}
```

```
    },
```


Creating Resources

- To invoke HTTP POST and persist an entity to the database, we can use Spring's RestTemplate

```
public String addREST(Employee empl) {
  String url = "http://localhost:8082/basic/staff";
  HttpHeaders requestHeaders = new HttpHeaders();
  requestHeaders.set("content-type", "application/json");
  HttpEntity<Employee> entity = new HttpEntity<Employee>(empl, requestHeaders);
  ResponseEntity<String> response = template.postForEntity(url, entity,
String.class);
  return response.toString();
}
```

- Above, an HTTP POST is made to our exposed repository
- The resulting response contains a link to the new resource and an HTTP status of 201 (new resource created)

<201 Created,{Server=[Apache-Coyote/1.1], X-Application-Context=[application:8082], Location=[http://localhost:8082/basic/staff/7], Content-Length=[0], Date=[Sun, 04 Sep 2016 23:01:45 GMT]}>

Projections

- Sometimes we do not want a service to return a representation of an entire Entity class but only a subset of its attributes. In such a case we can create a projection.

```
@Projection(name="town", types=Town.class)
public interface TownProjection {
	long getId();
	String getCity();
}
```

- Note the name of the projection and the Entities that the projection applies to and that we use an attribute wrapped around the getter.

- **Create projections in the same package as the repository**

19920612 Copyright DGP Solutions 5-19

- The Interface attributes MUST begin with get*

- You can use SPEL

@Value("#{target.getId()")
Long getSomething()

- In resultant json created will have akey of "something"

Use your Projection

http://localhost:8088/city/towns/1?projection=home


```
{
  "id": 1,
  "city": "Exeter",
  "_links": {
    "self": {
      "href": "http://localhost:8088/city/towns/1"
    },
    "town": {
      "href": "http://localhost:8
      "templated": true
    },
```

```
{
  "phoneCode": 1392,
  "district": "Heavitree",
  "city": "Exeter",
  "county": "Devon",
  "_links": {
    "self": {
      "href": "http://localhost:8088/city/towns/1"
```

Excerpts

- Excerpts are projections which apply as default views to resource collections. Adding the excerptProjection attribute to use our projection class “TownProjection” on our @RepositoryRestResource annotation, will cause resource collections to use the projection

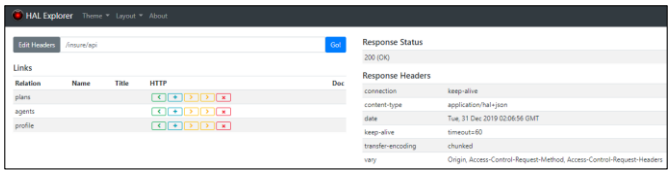
```
@RepositoryRestResource(collectionResourceRel="districts", path ="towns",
excerptProjection=TownProjection.class)
public interface TownRepository extends PagingAndSortingRepository<Town,
Long>{
```

A new Collection


```
{
  "_embedded": {
    "districts": [
      {
        "id": 1,
        "city": "Exeter",
        "_links": {
          "self": {
            "href": "http://localhost:8088/city/towns/1"
          },
          "town": {
            "href": "http://localhost:8088/city/towns/1{?projection}",
            "templated": true
          },
```

HAL Explorer

- The HAL Explorer, is a web application to provide a convenient front end to browse your HAL Json. You can point it at any Spring Data REST API and use its GUI to navigate. Navigate to the root url and it will redirect you to the HAL Explorer home page, http://localhost:8088/city/. You can change the base url for Spring Data Rest by adding a property in application.properties of; spring.data.rest.basePath=/api



Lab 5.2 - Spring Data Rest

Lab 5.2 Spring Data Rest

- Object: Expose our PagingAndSortingRepository as a service
- In application.properties add a base url for Spring Data Rest of; spring.data.rest.basePath=/api
- Annotate the Interface HealthPlanRepository with @RepositoryRestResource(collectionResourceRel="plans", path ="options“). This defines a wrapper tag for collections of Json and a path for our Repository of “/options”
- Create a class com.health.reporitory.Plan, annotate it with @Projection(name="plan", types=HealthPlan.class). Give it the attirbutes as in the notes.


```
long getId();
String getName();
@Value("#{target.outOfPocketFamily - target.outOfPocketIndividual}")
 int getDifferential();
```

Lab 5.2 Excerpt

- Amend the annotation @RepositoryRestResource for HealthPlanRepository to have the attribute excerptProjection=Plan.class
- Start the application and navigate to the url http://localhost:8082/insure/api . Your HAL Explorer will appear. All urls to the Repository will be prefixed with /insure/api/options regardless if we use the HAL Explorer or not.
- In the Explorer input box enter the url /insure/api/options → Click "Go". The formatted GUI output is on the left of the screen, the raw json output on the right
- Explore the Links and Embedded Links. Your Excerpt is being used to display each HealthPlan, use the "self" link for an Embedded Resource to trigger the HATEOAS link and navigate to the detailed output for the HealthPlan (Invoking another service)