Selected Spring Topics

Petr Křemen, Martin Ledvinka

KBSS

Winter Term 2024

Contents

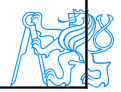
Data JPA

Spring Data Repositories

- Abstraction intended to significantly reduce boilerplate code of data access
- `Repository<T, ID>` interface defines basic interaction for an entity type `T` with identifier of type `ID`
- `CrudRepository<T, ID>` defines basic create, retrieve, update, delete methods
- `PagingAndSortingRepository<T, ID>` defines methods for paging and sorting
- `JpaRepository<T, ID>` for JPA
- `@EnableJpaRepositories` to enable JPA-based Spring data repositories
- Spring is able to generate implementation of such interfaces

JpaRepository

```
interface JpaRepository<T, ID> extends
  PagingAndSortingRepository<T, ID> {
 <S extends T> S save(S entity);

  Optional<T> findById(ID primaryKey);

  List<T> findAll();

  List<T> findAll(Sort sort);

  List<T> findAll(Pageable pageable);

  long count();

  void delete(T entity);

  boolean existsById(ID primaryKey);
  // ...
}
```



Custom Queries

- JPA named queries
- Using `@Query` to declare query directly on the interface method

Example

```
@Query("SELECT p FROM Person p WHERE p.name =
:name")
Person retrieveByName(@Param("name") String name);
```

- By default using positional parameters
- Use `@Param` to annotate method parameters for named parameters
- `native = true` to mark query as native SQL
- Can use SpEL expressions in the query

Generated Queries

- Spring is able to generated queries based on method names mentioning entity attributes
- Language is restricted, but sufficiently expressive

Example

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

- `find | read | query | count | get` (type of query)
- `distinct`
- `lastname` , `firstname` (fields of the entity)
- possibility to express *not*, string operators (e.g.*startingWith*), comparison (e.g. *greaterThan*), ordering,

Paging, Sorting, Filtering

Paging and Sorting

- input: `Pageable` for page parameter specification, possibly `Sort` for sorting specification
- output:
  - `List` as return value containing data page
  - `Page` with total result count,
  - `Slice` with next page pointer
- Compatible with Spring data repositories

Example

```
// ...
Page<Product> findAll(Pageable pageable);
List<Product> findAll(Sort sort);
//...

Pageable sortedByPriceDesc = PageRequest.of(0, 10,
  Sort.by("price").descending());
Page<Product> result =
  productRepository.findAll(sortedByPriceDesc);
```

Paging in REST Services

- Page and page size as regular request query parameters
- Page result
  - Application listener to enrich response with HATEOAS paging headers
  - Links to next, previous, first, last page
- It is also possible to return links to other pages in response body

Example Response Headers

```
Content-Type: application/json;charset=UTF-8
Date: Sun, 17 Nov 2019 13:45:50 GMT
Last-Modified: Sun, 17 Nov 2019 13:44:45 GMT
Link: <http://localhost/eshop/rest/products?page=1&size=100>;
    rel="next", <http://localhost/eshop/rest/products?page=19&
    size=100>; rel="last"
```

Filtering Parameters in REST Services

- Can use `@RequestParam` to declare parameters for a REST endpoint
- But what if there are many parameters, often optional?
- Query parameters can be loaded from a map
- Combination of both possible

Example

```
@GetMapping(produces = {MediaType.APPLICATION_JSON_VALUE})
public ResponseEntity getProducts(
  @RequestParam(name = "page", required = false) Integer page,
  @RequestParam(name = "size", required = false) Integer size,
  @RequestParam MultiValueMap<String, String> params);
```

Specifications

Data Repositories + Criteria API

```
List<T> findAll(Specification<T> spec);
```

Specification:

```
public interface Specification<T> {
      Predicate toPredicate(Root<T> root,
        CriteriaQuery<?> query, CriteriaBuilder
        builder);
}
```

Specifications

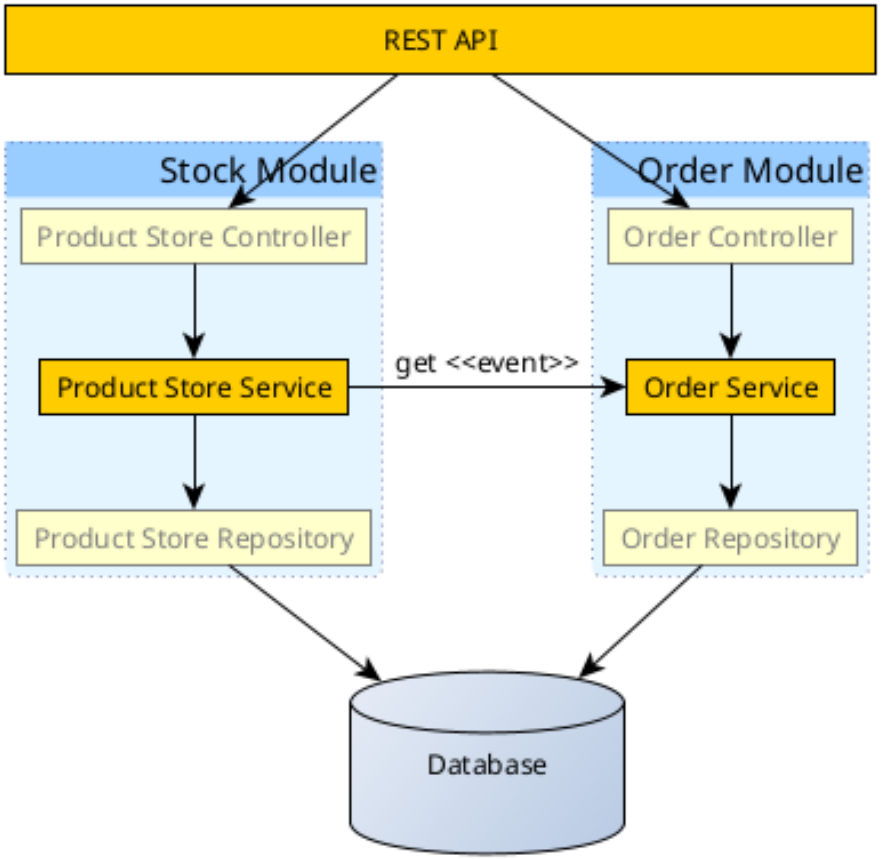
Example

```
public class ProductSpecs {
  public static Specification<Product> isValidProduct() {
       return (root, query, builder) ->
        builder.lessThan(root.get(Product_.createdAt),LocalDate.now());
  }

  public static Specification<Customer>
    hasSalesOfMoreThan(MonetaryAmount value) {
       return (root, query, builder) -> {
             ...
       }
  }
}
```

```
public interface ProductRepository extends
  CrudRepository<Product, Long>,
  JpaSpecificationExecutor<Product> {
       List<Product> findAll(Specification<Product> spec);
}
```

Spring Modulith

Modularizing your codebase via Spring Modulith



Spring Modulith Features

- package = module with public API.
- no cyclic dependencies between modules
- only API classes in the package root (e.g. Service), the rest in subpackages (even if public!)
- modules communicate via Application Events
- automated check of module dependencies and dependency diagram generation via tests.

References

- https://docs.spring.io/spring-data/jpa/docs/current/reference/html
- https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html
- https://spring.io/projects/spring-modulith