Advanced Java

Agenda

- REST services
  - DTOs
  - Content Negotiation
  - CORS
  - React client
- Testing
  - Unit Testing
  - Integration Testing
- Spring REST client
- Spring Data Introduction
  - ORM

Spring REST

DTO objects

- DTO = Data Transfer Objects
- Entity Pojos
  - POJO classes used to send/receive data from database -- Jdbc RowMapper, Jpa/Hibernate, ...
  - These POJOs deal with DAO layer.

```
class MovieEntity {
    private int id;
    private String title;
    private Date release;
    // ...
}
```

- DTO Pojos
    - POJO classes used to send/receive data from client (REST client).
    - These POJOs deal with Front-end layer.

    ```
    class MovieDTO {
        private int id;
        private String title;
        private String release;
        // ...
    }
    ```

    - Using Entity POJO objects for DTO is bad practice; because changes in database will directly reflect to client and vice-versa.
    - Ideally these layers should be isolated -- using Service layer. Here, we can convert DTO POJO to Entity POJO and vice-versa.

    ```
    @Service
    class MovieService {
        @Autowired
        private MovieDao dao;
        public MovieDTO getMovie(int id) {
            MovieEntity entity = dao.findById(id);
            MovieDTO dto = new MovieDTO(entity.getId(), entity.getTitle(), entity.getRelease().toString());
            return dto;
        }
        // ...
    }
    ```

    - Such conversions are automated using third party libraries e.g. ModelMapper, MapStruct, JMapper, Dozer, ...

Content Negotiation

- Server is capable of producing/consuming different data formats e.g. JSON, XML, ...
- Client can get any format in which it is interested.
- Spring REST application can support multiple formats by multiple message converters. Example:
  - MappingJackson2HttpMessageConverter -- JSON format
  - MappingJackson2XmlHttpMessageConverter -- XML format
- Spring Boot will auto-create these message converters if respective dependencies are added in project. Spring will also auto add these converters into the list of converters used RequestResponseBodyMethodProcessor.
  - Maven jackson-databind -- auto create -- MappingJackson2HttpMessageConverter
  - Maven jackson-dataformat-xml -- auto create -- MappingJackson2XmlHttpMessageConverter

  ```xml
  <dependency>
      <groupId>com.fasterxml.jackson.dataformat</groupId>
      <artifactId>jackson-dataformat-xml</artifactId>
  </dependency>
  ```

- Client application/Postman can now request REST API with request header "Accept"="application/json" to get the Json response and "Accept"="text/xml" to get XML response. The default is "Json".
- It is possible that server may produce only XML or only Json result.

CORS

- Cross-Origin Resource Sharing (CORS) is an HTTP-header based mechanism that allows a server to indicate any origins (domain, scheme, or port) other than its own from which a browser should permit loading resources.
- CORS also relies on a mechanism by which browsers make a "preflight" request to the server hosting the cross-origin resource, in order to check that the server will permit the actual request. In that preflight, the browser sends headers that indicate the HTTP method and headers that will be used in the actual request.
- The server should send response header showing which websites are allowed to access the resource. The "*" indiate that all web-sites can access rge resource.

  ```
  Access-Control-Allow-Origin: *
  ```

- Spring Boot enable this using @CrossOrigin on @Controller/@RestController.

Calling REST API from React

- Spring REST server -- port=8080
  - REST API: /movies
  - CORS: @CrossOrigin on all @RestController
- React server -- port=3000

```
url = "http://localhost:8080/movies"
axios.get(url)
    .then((resp) => {
        //console.log(resp.data)
        setUsers(resp.data)
    })
```

Jackson Library

- Jackson is a high-performance JSON processor used for Java. It is the most popular library used for serializing Java objects or Map to JSON and vice-versa. It is completely written in Java.
- Convert Java object into JSON

```
ObjectMapper mapper = new ObjectMapper();
Category c = new Category(...);
String json = mapper.WriteValueAsString(c);
System.out.println(json);
```

- Convert JSON to Java Object

```
ObjectMapper mapper = new ObjectMapper();
String json = "{\"id\": 2, \"title\": \"Movie\", \"description\": \"Movie related blogs\"}";
Category c = mapper.readValue(json, Category.class);
System.out.println(c.toString());
```

Spring REST client

- RestTemplate is used to create applications that consume RESTful Web Services. The application can be console application, web application or another REST service.
- RestTemplate can be created directly OR using RestTemplateBuilder.

```
@Bean
public RestTemplate restTemplate() {
    return new RestTemplate();
}
```

```
@Bean
public RestTemplate restTemplate(RestTemplateBuilder builder) {
    return builder.build();
}
```

- Creating RestTemplate using builder is a good practice. It can be used to customize restTemplate with custom interceptors. It also enables application metrics and work with distributed tracing
- RestTemplate sends GET, POST, PUT or DELETE request to the REST resource/api.
- Important RestTemplate methods

```
T getForObject(String url, Class<T> responseType, Object... uriVariables);

ResponseEntity<T> getForEntity(String url, Class<T> responseType, Object... uriVariables);

T postForObject(String url, Object request, Class<T> responseType, Object... uriVariables);

ResponseEntity<T> postForEntity(String url, Object request, Class<T> responseType, Object... uriVariables);

void put(String url, Object request, Object... uriVariables);

void delete(String url, Object... uriVariables);

ResponseEntity<T> exchange(URI url, HttpMethod method, HttpEntity<?> requestEntity, Class<T> responseType);

ResponseEntity<T> exchange(String url, HttpMethod method, HttpEntity<?> requestEntity,  ParameterizedTypeReference<T>
responseType, Object... uriVariables);

ResponseEntity<T> exchange(RequestEntity<?> entity, Class<T> responseType);
```

- UriComponentsBuilder can be used to build Uri as per following steps.
  - Create a UriComponentsBuilder with one of the static factory methods (such as fromPath(String) or fromUri(URI))
  - Set the various URI components through the respective methods (scheme(String), userInfo(String), host(String), port(int), path(String), pathSegment(String...), queryParam(String, Object...), and fragment(String).
  - Build the UriComponents instance with the build() method.

```
String url = UriComponentsBuilder.fromHttpUrl(BOOKS_URL).path("/{id}").build().toUriString();
```

- RequestEntity<> can be used to build request object in readable manner (method chaining).

```
RequestEntity<MyRequest> request = RequestEntity
    .post("https://example.com/{foo}", "bar")
    .accept(MediaType.APPLICATION_JSON)
    .body(reqBodyObject);
ResponseEntity<MyResponse> response = template.exchange(request, MyResponse.class);
```

Spring Data

- Spring Data provides unified access to databases -- RDBMS or NoSQL.
- Spring Data JPA provides repository support for the Java Persistence API (JPA) -- RDBMS.

Spring Data Architecture

- Refer slides

Spring Data JPA - Advantages

- Eases development of applications that need to access JPA data sources.
- Lot of boilerplate/repeated code is eliminated.
- Consistent configuration and unified data access.

Spring Data Jpa - version

- JPA is a specification for ORM tools.
- Spring Data JPA - 3.2.5
  - Hibernate 6.4.4 (ORM - default)
  - EclipseLink 3.0.4 (ORM)
  - Querydsl 5.0.0 (Annotation processing)

Object relational mapping (ORM)

- Converting Java objects into RDBMS rows and vice-versa is done manually in JDBC code.
- This can be automated using Object Relational Mapping.

  - Java Class is mapped to RDBMS Table, while it's field are mapped to Column.
  - It also map table relations into entities associations/inheritance and auto-generates SQL queries.
- Most used ORM are Hibernate, EclipseLink, iBatis, Torque, ...

ORM (JPA)

- All ORM annotations are in javax.persistence package.
- ORM using annotations
  - @Entity: Mark POJO as hibernate entity.
  - @Table: Map entity class with the RDBMS table.
  - @Column: Map entity class field with the RDBMS table column.
  - @Id: Mark primary key field of the entity class.
  - @Temporal
    - @Temporal(TemporalType.DATE) -- java.util.Date, the java.util.Calendar, or LocalDate
    - @Temporal(TemporalType.TIME) -- java.util.Date, or LocalTime
    - @Temporal(TemporalType.TIMESTAMP) -- java.util.Date, the java.util.Calendar, or LocalDateTime
  - @Lob
    - Mapped with byte[]
  - @Transient: Do not map entity class field to the RDBMS table column.
- @Column can be used on field level or on getter methods.

Data JPA interfaces

Repository -- Marker interface

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

CrudRepository

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {
    long count();
    //Returns the number of entities available.
    void delete(T entity);
    //Deletes a given entity.
    void deleteAll();
    //Deletes all entities managed by the repository.
    void deleteAll(Iterable<? extends T> entities);
    //Deletes the given entities.
    void deleteById(ID id);
    //Deletes the entity with the given id.
    boolean existsById(ID id);
    //Returns whether an entity with the given id exists.
    Iterable<T> findAll();
    //Returns all instances of the type.
    Iterable<T> findAllById(Iterable<ID> ids);
    //Returns all instances of the type with the given IDs.
    Optional<T> findById(ID id);
    //Retrieves an entity by its id.
    <S extends T> S save(S entity);
    //Saves a given entity.
    <S extends T> Iterable<S> saveAll(Iterable<S> entities);
    //Saves all given entities.
}
```

PagingAndSortingRepository

```
public interface PagingAndSortingRepository<T, ID> extends Repository<T, ID> {
    Iterable<T> findAll(Sort sort);
    Page<T> findAll(Pageable pageable);
}
```

```
// Example: Fetch second page (index=1) of size 20
Page<User> users = dao.findAll(PageRequest.of(1, 20));
```

JpaRepository

- Inherited from PagingAndSortingRepository, CrudRepository, QueryByExampleExecutor.

```
public interface JpaRepository<T, ID> extends CrudRepository<T, ID>, ... {
    void deleteAllInBatch();
    // Deletes all entities in a batch call.
    void deleteInBatch(Iterable<T> entities);
    // Deletes the given entities in a batch which means it will create a single Query.
    <S extends T> List<S> findAll(Example<S> example);
    <S extends T> List<S> findAll(Example<S> example, Sort sort);
    List<T> findAllById(Iterable<ID> ids);
    void flush();
    // Flushes all pending changes to the database.
    T getOne(ID id);
    // Returns a reference to the entity with the given identifier.
}
```

MongoRepository

- Inherited from PagingAndSortingRepository, CrudRepository, QueryByExampleExecutor.

```
public interface MongoRepository<T, ID> extends CrudRepository<T, ID>, ... {
    List<T> findAll();
    List<T> findAll(Sort sort);
    <S extends T> S insert(S entity);
    // ...
}
```

JpaRepository Example

- step 1: Create new Spring Initialzr project with dependencies Spring Data JPA, MySQL.
- step 2: In application.properties configure Database and JPA settings.

```
spring.datasource.url=jdbc:mysql://localhost:3306/dmcdb
spring.datasource.username=dmc
spring.datasource.password=dmc

spring.jpa.show-sql=true
```

- step 3: Add entity class Category.

```
@Entity
@Table(name = "categories")
public class Category {
    @Id
    private int id;
    @Column
    private String title;
    @Column(name="description")
    private String desc;
    // ...
}
```

- step 4: Create repository interface .

```
interface CategoryDao extends JpaRepository<Category, Integer> {
    // ...
```

```
  }
```

- step 5: In main class, @Autowired CategoryDao categoryDao and test its functionlities.
- step 6: Add various query methods (findBy) in CategoryDao and test them **one by one** in main class.
- step 7: In Category entity class add `@GeneratedValue(strategy = GenerationType.IDENTITY)` to auto-generate ids. Test save() method in main class.