Spring-data

Mapping et plus si affinité (Neo4J, Solr, …)

Plan

- Rappels: Spring + Boot
- Spring data JDBC
- Spring data JPA
- Spring data LDAP
- Spring data MongoDB
- Spring data Redis
- Spring Data REST

Rappels

- Spring Boot facilite la création d'applications autonomes à base de Spring que l’on peut aisément "exécuter".

- Via des starters de la plate-forme Spring et l’auto-configuration, il y a très peu de configuration Spring pour démarrer et

  - Créer des applications Spring autonomes
  - Intégrer directement Tomcat, Jetty ou Undertow (pas besoin de déployer des fichiers WAR)
  - Fournir des dépendances avisées pour simplifier la configuration de construction
  - Configurer automatiquement les bibliothèques Spring et tierces autant que possible
  - Fournir des fonctionnalités prêtes à la production telles que les mesures, les vérifications de l'état et la configuration externalisée
  - Absolument aucune génération de code et aucune exigence pour la configuration XML

Rappels sur l’auto-configuration

- Cette fonctionnalité est la plus importante de Spring Boot. Elle permet de **configurer automatiquement** une application à partir des *jar* trouvés dans le Classpath.

- En d'autres termes, si on a importé des dépendances, Spring Boot ira consulter cette liste puis produira la configuration nécessaire pour que tout fonctionne correctement.

`@EnableAutoConfiguration`

- Avec cette annotation, Spring Boot ira scanner la liste des dépendances, trouvant par exemple `Hibernate`.
- Ayant constaté qu’il n’y a aucun autre datasource, il créera la configuration nécessaire et l'ajoutera à `ApplicationContext`.

Rappels sur les starters

- Les starters viennent compléter l'auto-configuration et font gagner énormément de temps.
  - Un starter va apporter à un projet un **ensemble de dépendances**, communément utilisées pour un type de projet donné.

- L'autre avantage est la **gestion des versions**.
  - Plus besoin de chercher quelles versions sont compatibles puis de les ajouter une à une dans le *pom.xml* !
  - Il suffit d'ajouter une simple dépendance au starter choisi. Cette dépendance va alors ajouter, à son tour, les éléments dont elle dépend, avec les bonnes versions.

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

Spring data JDBC

- L'idée derrière Spring Data JDBC est de fournir un accès aux BDs relationnelles sans se soumettre aux complexités de JPA.

- JPA offre des fonctionnalités telles que le chargement différé, la mise en cache et le suivi des données modifiées.

- Si elles sont utiles, elles peuvent en réalité rendre la réflexion sur JPA et son comportement plus ardu que nécessaire.
  - Le chargement paresseux peut déclencher des déclarations coûteuses lorsque l'on ne s'y attend pas ou échouer avec une exception.
  - La mise en cache peut gêner lorsque l'on veut réellement comparer 2 versions d'une entité et qu'elle est dite « dirty », il est difficile de trouver un point unique où toutes les opérations de persistance passent.

Spring data JDBC

- Spring Data JDBC vise un modèle beaucoup plus simple.
  - Il n’y aura pas de mise en cache, de suivi incorrect ou de chargement paresseux.
  - Au lieu de cela, les instructions SQL sont émises uniquement lorsqu’on appelle une méthode du référentiel. L'objet renvoyé à la suite de cette méthode est entièrement chargé avant le retour de la méthode.
  - Il n'y a pas de "session" et pas de proxy pour les entités. Tout cela devrait rendre Spring Data JDBC plus facile à exploiter.

- Bien entendu, cette approche plus simple engendre des contraintes sur des données partagées par exemple.

Spring data JDBC exemple

- Une entité

```
class Customer {
    @Id
    Long id;
    String firstName;
    LocalDate dob;
}
```

- Il faut déclarer un référentiel.
  Le moyen le plus simple de le faire est d'étendre `CrudRepository`:

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

- Pas besoin de getters ni de setters.
- La seule exigence est que l'entité ait une propriété annotée avec `Id` (`@org.springframework.data.annotation.Id`, et non celle `javax.persistence`).

Spring data JDBC exemple

- Enfin, on doit configurer `ApplicationContext` pour permettre la création de référentiels:
- (1) `EnableJdbcRepositories` permet la création de référentiels. Comme cela nécessite la présence de quelques beans dans la configuration.
- (2) L'extension de `JdbcConfiguration` ajoute des beans par défaut à `ApplicationContext`.
  On peut redéfinir ses méthodes pour personnaliser les comportements de Spring Data JDBC.
  Pour l'instant, on utilise les implémentations par défaut.
- (3) La partie importante est `NamedParameterJdbcOperations`, utilisée en interne pour soumettre des instructions SQL à la BD.
- (4) Le gestionnaire de transaction n'est pas nécessaire. Mais il procure assistance pour les transactions qui couvrent plus d'une déclaration,
- (5) Spring Data JDBC n'utilise pas directement `DataSource`, mais comme `TransactionManager` et `NamedParameterJdbcOperations` en ont besoin, son enregistrement en tant que bean est un moyen de garantir que les 2 utilisent la même instance.

```
@Configuration
@EnableJdbcRepositories (1)
public class CustomerConfig extends JdbcConfiguration { (2)

    @Bean
    NamedParameterJdbcOperations operations() { (3)
        return new NamedParameterJdbcTemplate(dataSource());
    }

    @Bean
    PlatformTransactionManager transactionManager() { (4)
        return new DataSourceTransactionManager(dataSource());
        }

    @Bean
    DataSource dataSource(){ (5)
        return new EmbeddedDatabaseBuilder()
                .generateUniqueName(true)
                .setType(EmbeddedDatabaseType.HSQL)
                .addScript("create-customer-schema.sql")
                .build();
    }
}
```

Spring data JDBC exemple

```
@RunWith(SpringRunner.class)
@Transactional
@ContextConfiguration(classes = CustomerConfig.class)
public class CustomerRepositoryTest {

    @Autowired CustomerRepository customerRepo;

    @Test
    public void createSimpleCustomer() {

        Customer customer = new Customer();
        customer.dob = LocalDate.of(1904, 5, 14);
        customer.firstName = "Albert";

        Customer saved = customerRepo.save(customer);
        assertThat(saved.id).isNotNull();
        saved.firstName = "John Deuf";
        customerRepo.save(saved);
        Optional<Customer> reloaded = customerRepo.findById(saved.id);
        assertThat(reloaded).isNotEmpty();
        assertThat(reloaded.get().firstName).isEqualTo("John Deuf");
    }
}
```

Spring data JDBC

- Annotation `@Query`
- On ne peut pas aller très loin avec les méthodes de base CRUD du `CrudRepository`.
  Il est utile d'ajouter ses propres requêtes: la fonctionnalité répandue dans laquelle Spring Data dérive la requête à utiliser à partir d'un nom de méthode (cf plus loin).
- En attendant, on utilise une annotation `@Query` pour spécifier une requête sur une méthode de référentiel:

```
@Query("select id, first_name, dob from customer
where upper(first_name) like '%' || upper(:name) || '%' ")
List<Customer> findByName(@Param("name") String name);
```

- L'annotation `@Param` n'est pas requise lors d'une compilation avec l'indicateur `-parameters`.

- Si vous souhaitez exécuter une instruction update ou delete, vous pouvez ajouter une annotation `@Modifying` à la méthode.

Spring data JDBC exemple

```
@Test
public void findByName() {

    Customer customer = new Customer();
    customer.dob = LocalDate.of(1964, 11, 11);
    customer.firstName = "Leopold";

    Customer saved = customerRepo.save(customer);
    assertThat(saved.id).isNotNull();

    customer.id= null; (1)
    customer.firstName = "Leon";
    customerRepo.save(customer);

    customer.id= null;
    customer.firstName = "Sarah";
    customerRepo.save(customer);
    assertThat(customerRepo.findByName("leo")).hasSize(2); (2)
}
```

(1) Étant donné que la connexion entre un objet Java et sa ligne correspondante correspond uniquement à son ID et à son type, définir l'ID sur null et le sauvegarder à nouveau crée une autre ligne dans la base de données.

(2) On fait une recherche (comme) insensible à la casse et trouvons donc « Leopold" et « Leon" mais pas « Sarah".

Spring data JPA

- La création de référentiels utilisant l'API Java Persistence est un processus fastidieux qui prend beaucoup de temps. On procède en 2 étapes:
  1. Créez une classe de référentiel de base abstraite qui fournit des opérations CRUD pour les entités.
  2. Créez la classe de référentiel concret qui étend la classe de référentiel de base abstraite.

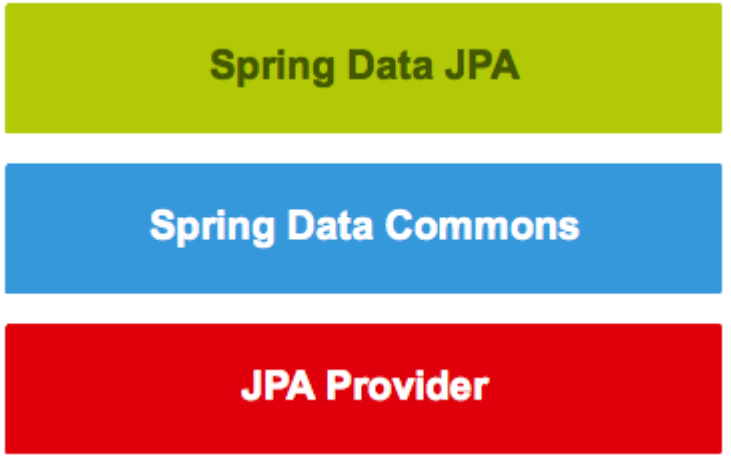
- Le problème de cette approche est qu'il reste à écrire le code qui crée et interroge avec nos requêtes la BD.
  Pour aggraver les choses, on doit le faire chaque fois que l'on veut créer une nouvelle requête en BD.

Spring data JPA

- On peut créer des référentiels JPA sans écrire de code ou presque !
- Spring Data JPA n'est pas un fournisseur JPA.
  - C'est une bibliothèque / framework qui ajoute une couche supplémentaire d'abstraction au sommet de JPA.
  - Si on décide d'utiliser Spring Data JPA, la couche de référentiel de notre application contient 3 couches en plus,

1. Spring Data JPA prend en charge la création de référentiels JPA en étendant les interfaces du référentiel Spring Data.
2. Spring Data Commons fournit l'infrastructure partagée par les projets Spring Data spécifiques au cache de données.
3. Le fournisseur JPA implémente l'API Java Persistence.

Cela ajoute une couche supplémentaire à notre couche de référentiel, mais en même temps, cela libère de l'écriture de tout code standard.



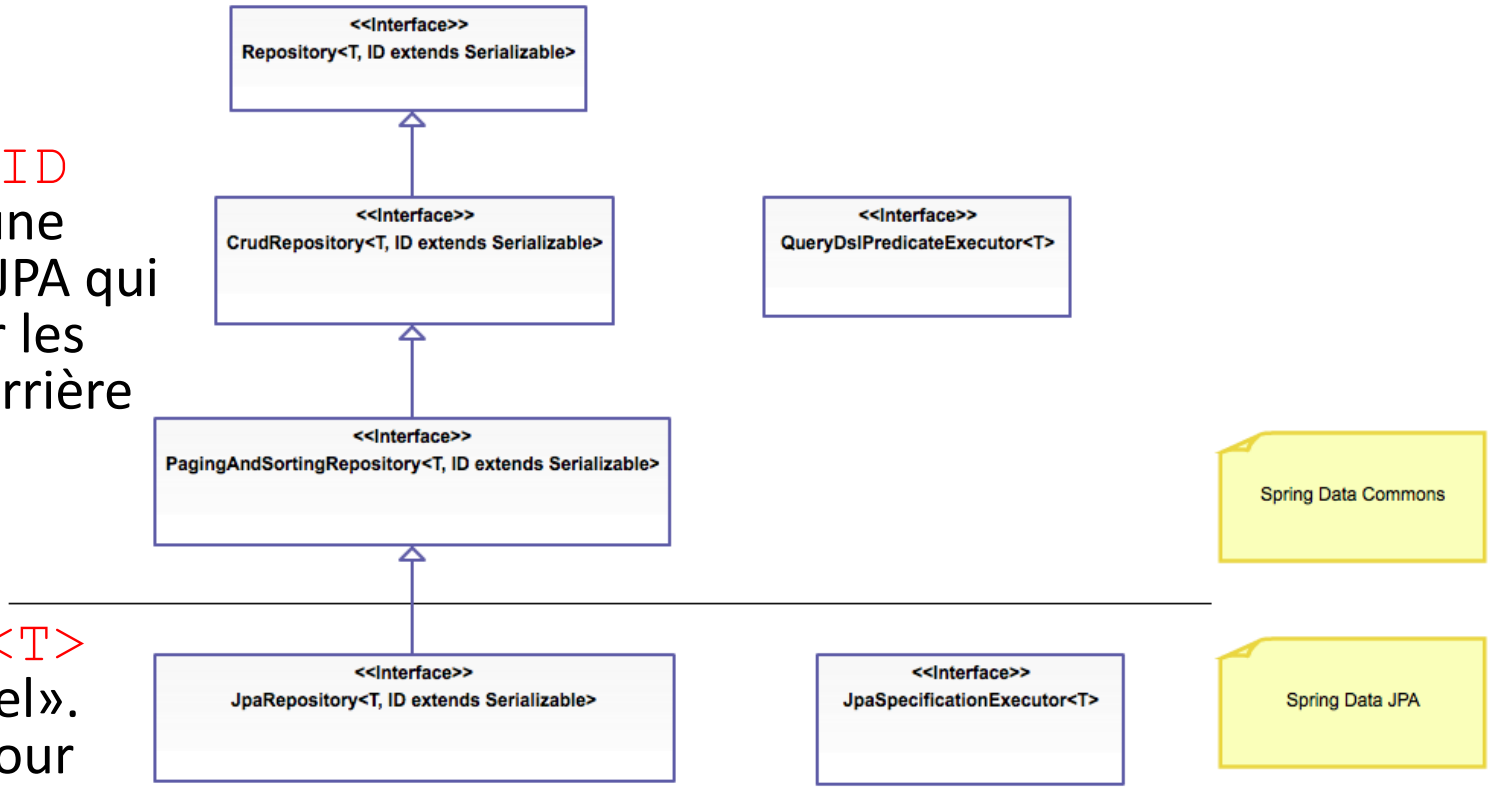
Spring Data Repositories

- La puissance de Spring Data JPA réside dans l'abstraction de référentiel fournie par le projet Spring Data Commons et étendue par les sous-projets spécifiques au cache de données.

- On peut utiliser Spring Data JPA sans faire attention à l'implémentation réelle de l'abstraction du référentiel, on doit connaître les interfaces du référentiel Spring Data. Ces interfaces de Spring Data Commons sont :
  - L'interface `Repository<T, ID extends Serializable>` est une interface ou marqueur ayant 2 objectifs:
  1. Il capture le type de l'entité gérée et le type de l'identifiant de l'entité.
  2. Il aide le conteneur Spring à découvrir les interfaces «concrètes» du référentiel lors de l'analyse du chemin d'accès aux classes.
  - L'interface `CrudRepository<T, ID extends Serializable>` fournit des opérations CRUD pour l'entité gérée.
  - L'interface `PagingAndSortingRepository <T, ID extends Serializable>` déclare les méthodes utilisées pour trier et paginer les entités extraites de la BD.
  - L'interface `QueryDslPredicateExecutor<T>` n'est pas une «interface de référentiel». elle déclare les méthodes utilisées pour extraire des entités de la BD à l'aide d'objets `QueryDslPredicate`.

Spring Data Repositories

- Les interfaces de Spring Data JPA
  - L'interface `JpaRepository<T, ID extends Serializable>` est une interface de référentiel spécifique à JPA qui combine les méthodes déclarées par les interfaces de référentiel commun derrière une interface unique.
  - L'interface `JpaSpecificationExecutor<T>` n'est pas une «interface de référentiel». Elle déclare les méthodes utilisées pour extraire des entités de la BD à l'aide d'objets `Specification<T>` utilisant l'API de critères JPA.



Configuration de la couche de persistance d'une application Spring

- Créer la classe de configuration pour la couche de persistance d'une application Spring, prend 6 étapes:

1. Créez le fichier de propriétés contenant les propriétés utilisées par notre classe de configuration de contexte d'application.
2. Configurez le bean source de données.
3. Configurez le bean de fabrique du gestionnaire d'entités.
4. Configurez le bean du gestionnaire de transactions.
5. Activer la gestion des transactions par annotation.
6. Configurez Spring Data JPA.

```
@Configuration
class PersistenceContext {


  //Configure the required beans here
}
```

Configuration de la couche de persistance d'une application Spring

- Création du fichier de propriétés
- pour utiliser une configuration légèrement différente selon les environnements, une bonne pratique consiste à déplacer la configuration dans un fichier de propriétés.

- Le fichier `application.properties` contient la configuration utilisée pour l'application. il contient:

- Configurez la connexion à la base de données de l'application: le nom de la classe de pilote JDBC, l'URL JDBC, le nom d'utilisateur de l'utilisateur de la BD et le mot de passe de l'utilisateur de la BD.
- Configurez `Hibernate` en procédant comme suit:
  - Configurez le dialecte de base de données utilisé.
  - Assurez-vous que `Hibernate` crée la BD au démarrage de l'application et la supprime à la fermeture de notre application.
  - Configurez la stratégie de dénomination utilisée lorsque `Hibernate` crée de nouveaux objets de base de données et éléments de schéma.
  - Configurez `Hibernate` pour NE PAS écrire les instructions SQL appelées sur la console.
  - Assurez-vous que si `Hibernate` écrit les instructions SQL sur la console, il utilisera `prettyprint`.

```
#Database Configuration
db.driver=org.h2.Driver
db.url=jdbc:h2:mem:datajpa
db.username=root
db.passwordroot
```

```
#Hibernate Configuration
hibernate.dialect=org.hibernate.dialect.H2Dialect
hibernate.hbm2ddl.auto=create-drop
hibernate.ejb.naming_strategy=org.hibernate.cfg.ImprovedNamingStrategy
hibernate.show_sql=false
hibernate.format_sql=true
```

Configuration du bean source de données

- 3 étapes:

1. Assurez-vous que la méthode `close()` de l'objet `DataSource` créé est appelée lorsque le contexte de l'application est fermé.
2. Configurez la connexion à la BD.
   On doit définir le nom de la classe de pilote JDBC, l'URL JDBC, le nom d'utilisateur de l'utilisateur de la base de données et le mot de passe de l'utilisateur de la BD.
3. Créez un nouvel objet `HikariDataSource` et renvoyez l'objet créé.

```
@Configuration
public class PersistenceContext {

    @Bean(destroyMethod = "close")
    DataSource dataSource(Environment env) {
        HikariConfig dataSourceConfig = new HikariConfig();
        dataSourceConfig.setDriverClassName(env.getRequiredProperty("db.driver"));
        dataSourceConfig.setJdbcUrl(env.getRequiredProperty("db.url"));
        dataSourceConfig.setUsername(env.getRequiredProperty("db.username"));
        dataSourceConfig.setPassword(env.getRequiredProperty("db.password"));

        return new HikariDataSource(dataSourceConfig);
    }

    //Add the other beans here

}
```

Configuration du bean Factory Entity Manager

- 5 étapes:

1. Créer un nouvel objet `LocalContainerEntityManagerFactoryBean`.
   On doit créer cet objet car il crée le JPA `EntityManagerFactory`.
2. Configurer la source de données utilisée.
3. Configurer l'implémentation spécifique à `Hibernate` de l'interface `JpaVendorAdapter`.
   Il initialisera la configuration avec les paramètres par défaut compatibles avec `Hibernate`.
4. Configurer les packages qui sont analysés pour les classes d'entité.
5. Configurer les propriétés JPA utilisées pour fournir une configuration supplémentaire au fournisseur JPA utilisé.

```
@Configuration
public class PersistenceContext {

    @Bean
    LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource dataSource,
                                                                 Environment env) {
        LocalContainerEntityManagerFactoryBean entityManagerFactoryBean
                                            = new LocalContainerEntityManagerFactoryBean();
        entityManagerFactoryBean.setDataSource(dataSource);
        entityManagerFactoryBean.setJpaVendorAdapter(new HibernateJpaVendorAdapter());
        entityManagerFactoryBean.setPackagesToScan("fr.paris.springdata.jpa.todo");

        Properties jpaProperties = new Properties();
        jpaProperties.put("hibernate.dialect", env.getRequiredProperty("hibernate.dialect"));
        jpaProperties.put("hibernate.hbm2ddl.auto",
                env.getRequiredProperty("hibernate.hbm2ddl.auto")
        );
        jpaProperties.put("hibernate.ejb.naming_strategy",
                env.getRequiredProperty("hibernate.ejb.naming_strategy")
        );
        jpaProperties.put("hibernate.show_sql",
                env.getRequiredProperty("hibernate.show_sql")
        );
        jpaProperties.put("hibernate.format_sql",
                env.getRequiredProperty("hibernate.format_sql")
        );
        entityManagerFactoryBean.setJpaProperties(jpaProperties);

        return entityManagerFactoryBean;
    }
    //Add the other beans here
}
```

Configuration du bean du gestionnaire de transactions

- Il faut créer un bean de gestionnaire de transactions qui intègre le fournisseur JPA au mécanisme de transaction Spring.
  On le fait en utilisant la classe `JpaTransactionManager` en tant que gestionnaire de transactions de notre application.

- On configure le bean du gestionnaire de transactions en 2 étapes:

1. Créez un nouvel objet `JpaTransactionManager`.
2. Configurez la fabrique de gestionnaires d'entités dont les transactions sont gérées par l'objet `JpaTransactionManager` créé.

```
@Configuration
class PersistenceContext {


    @Bean
    JpaTransactionManager transactionManager(
                            EntityManagerFactory entityManagerFactory) {
        JpaTransactionManager transactionManager =
                                                new JpaTransactionManager();
        transactionManager.setEntityManagerFactory(entityManagerFactory);
        return transactionManager;
    }


    //Add the other beans here
}
```

Activation de la gestion des transactions par annotation

- On active la gestion des transactions basée sur des annotations en annotant la classe `PersistenceContext` avec l'annotation `@EnableTransactionManagement`.

- La partie pertinente de la classe `PersistenceContext` est:

```
@Configuration
@EnableTransactionManagement
class PersistenceContext {

    //The beans are configured here
}
```

Configuration de Spring Data JPA

- On configure Spring Data JPA en 2 étapes:

1. Activez Spring Data JPA en annotant la classe `PersistenceContext` avec l'annotation `@EnableJpaRepositories`.

2. Configurez les packages de base analysés lorsque Spring Data JPA crée des implémentations pour les interfaces de référentiel.

```
@Configuration
@EnableJpaRepositories(basePackages = {
        "fr.paris.springdata.jpa.todo"
})
@EnableTransactionManagement
class PersistenceContext {


    //The beans are configured here
}
```

Création d'un repository

1. Créer une classe d'entité contenant les informations d'une instance.

2. La partie pertinente de la classe `Todo` :

- Pour créer le 1er référentiel Spring Data JPA qui fournit les opérations CRUD pour des objets `Todo` 2 méthodes possibles:
  - Créez une interface qui étend l'interface `CrudRepository`.
  - Créez une interface qui étend l'interface du `Pepository` et ajoutez les méthodes requises à l'interface créée.

```
import org.hibernate.annotations.Type;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.PrePersist;
import javax.persistence.Table;
import javax.persistence.Version;
import java.time.ZonedDateTime;

@Entity
@Table(name = "todos")
final class Todo {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(name = "creation_time", nullable = false)
    @Type(type = "org.jadira.usertype.dateandtime.threeten.PersistentZonedDateTime")
    private ZonedDateTime creationTime;
    @Column(name = "description", length = 500)
    private String description;
    @Column(name = "modification_time")
    @Type(type = "org.jadira.usertype.dateandtime.threeten.PersistentZonedDateTime")
    private ZonedDateTime modificationTime;

    @Column(name = "title", nullable = false, length = 100)
    private String title;
    @Version
    private long version;

    //The constructor, builder, and other methods are omitted
}
```

Extension de `CrudRepository` Interface

- Pour créer un référentiel en étendant l'interface `CrudRepository`, il faut fournir 2 paramètres de type:
  - Le type d'entité géré par le référentiel: `Todo`
  - Le type du champ `id` de l'entité: `Long`,

```
import org.springframework.data.repository.CrudRepository;


interface TodoRepository extends CrudRepository<Todo, Long> {

}
```

L'interface `CrudRepository` déclare des méthodes, telles que:

- `void delete(T entity)` méthode qui efface l'entité dont l'id est donné en paramètre.
- `Iterable<T> findAll()` méthode qui retourne toutes les entités qui dont sauvegardées dans la BD.
- `T findOne(Long id)` méthode qui retourne l'entité dont l'id est donné en paramètre. ISi aucune entité n'est trouvée alors elle retourne null.
- `T save(T entity)` méthode qui sauvegarde l'entité donné en paramètre et retourne une entité gérée.

Extension de `Repository` Interface

- Pour créer un référentiel en étendant l'interface du `Repository`, il y a 2 étapes:

1- Fournir 2 paramètres de type:

- Le type de l'entité gérée (`Todo`).
- Le type du champ id de l'entité (`Long`).

2- Ajouter les méthodes requises à l'interface du référentiel:

- La méthode `void delete(Todo delete)` supprime l'objet `Todo` donné en tant que paramètre de méthode.
- La méthode `List<Todo> findAll()` renvoie tous les objets `Todo` trouvés dans la BD.
- La méthode optionnelle `<Todo> findOne(Long id)` recherche l'entrée à faire dont l'identifiant est donné en tant que paramètre de méthode. Si aucune entrée de tâche n'est trouvée, cette méthode renvoie un fichier facultatif vide.
- La méthode `Todo save(Todo persisted)` enregistre l'objet `Todo` donné en tant que paramètre de méthode et renvoie l'objet persistant.

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

import java.util.List;

import java.util.Optional;


interface TodoRepository extends Repository<Todo, Long> {

    void delete(Todo deleted);

    List<Todo> findAll();

    Optional<Todo> findOne(Long id);

    Todo save(Todo persisted);

}
```

Bonne pratique

- 2 règles sont:
  - Pour exposer toutes les méthodes du référentiel déclarées par l'interface `CrudRepository` ET pour ne pas renvoyer d'objets `Optional` (Guava / Java 8), les interfaces de référentiel doivent étendre l'interface `CrudRepository`.
  - Pour ne pas exposer toutes les méthodes de référentiel déclarées par l'interface `CrudRepository` OU pour renvoyer des objets `Optional` (Guava / Java 8), les interfaces de référentiel doivent étendre l'interface `Repository`.

- Privilégier la 2ème méthode pour 2 raisons:
  - Lorsqu'on crée une interface, on ne doit pas y ajouter de méthodes inutiles.
  - On doit garder l'interface aussi petite que possible car les petites interfaces sont plus faciles à utiliser et aident à créer des composants qui n'ont qu'un seul travail.

Bonne pratique

- Pour créer nos référentiels en étendant l'interface `Repository` et en ajoutant les méthodes requises aux interfaces de référentiel créées, on doit ajouter les «mêmes» méthodes à chaque interface ? Faux

- Pour éviter cela en suivant ces étapes:
  - Créez une interface de base qui étend l'interface `Repository` et ajoutez les méthodes courantes à cette interface.
  - Créez l'interface secondaire qui étend notre interface de base.

- 3 étapes

1. Créer l'interface `BaseRepository` qui étend l'interface `Repository`. Cette interface a 2 paramètres de type:
   - T décrit le type de l'entité gérée.
   - ID décrit le type du champ id de l'entité.
2. Annoter l'interface créée avec l'annotation `@NoRepositoryBean`. Cela garantit que Spring Data JPA n'essaye pas de créer une implémentation pour notre interface de référentiel de base.
3. Ajouter les méthodes communes à l'interface créée.

```
import
org.springframework.data.repository.NoRepositoryBean;
import org.springframework.data.repository.Repository;
import java.util.List;
import java.util.Optional;

@NoRepositoryBean
interface BaseRepository<T, ID extends Serializable>
extends Repository<T, ID> {

    void delete(T deleted);
    List<T> findAll();
    Optional<T> findOne(ID id);
    T save(T persisted);
}
```

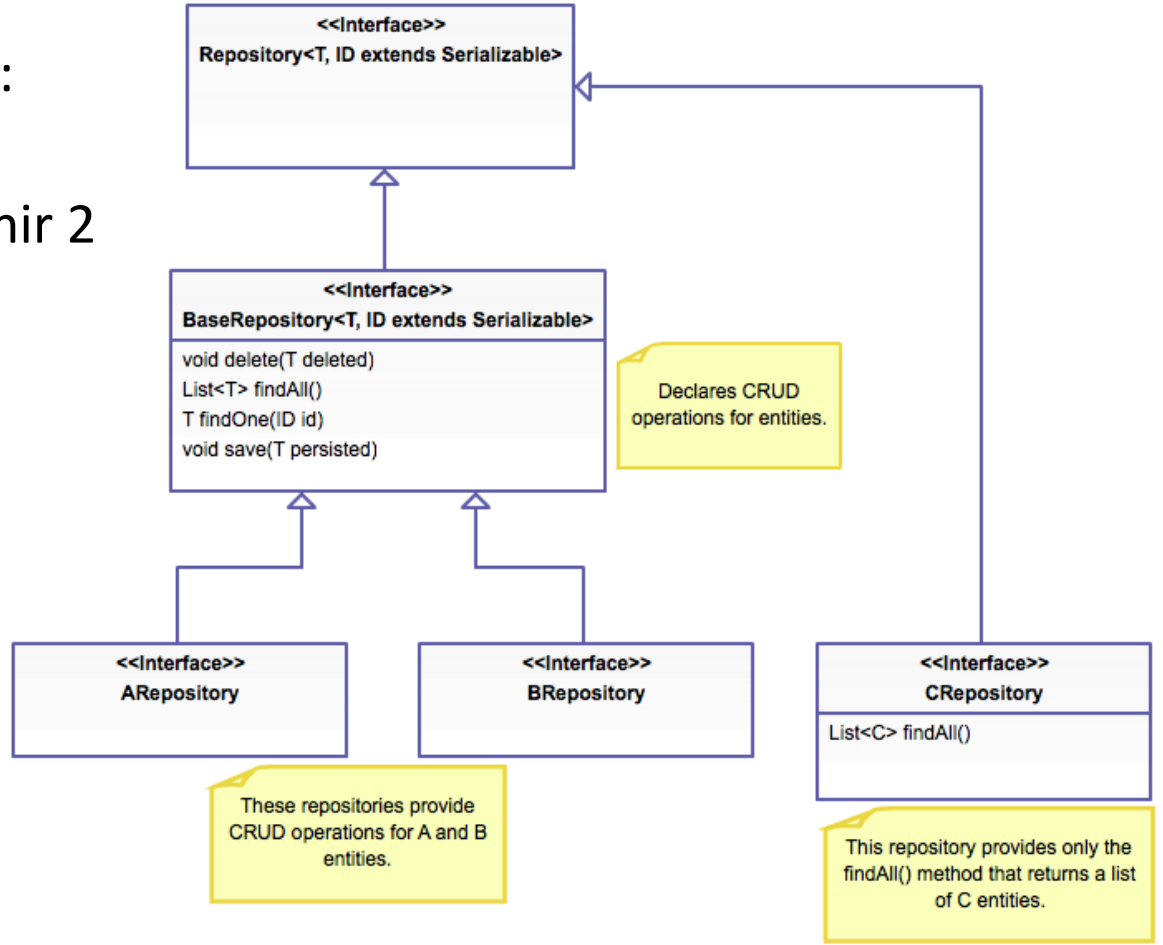
Bonne pratique

- Interface qui étend l'interface de base en 2 étapes:

1. Créez l'interface `TodoRepository`.
2. Étendre l'interface `BaseRepository` et fournir 2 paramètres de type:
   - Le type de l'entité gérée est `Todo`.
   - Le type du champ id de l'entité est `Long`.

```
interface TodoRepository extends
BaseRepository<Todo, Long> {


}
```



Une hiérarchie de référentiels qui permet de:
Créez des référentiels qui fournissent des opérations CRUD pour les entités sans déclarer les «mêmes» méthodes dans chaque interface de référentiel.

Méthode Query

- Une méthode pour rechercher des informations dans une BD consiste à utiliser des méthodes Query.
- Questions :
  - Quelles sont les méthodes Query ?
  - Quel type de valeurs de retour peut on utiliser?
  - Comment passer des paramètres à des méthodes Query?

- Les méthodes Query recherchent des informations dans la BD et sont déclarées sur l'interface `Repository`.
- Par exemple, si on crée une requête en BD qui trouve l'objet `Todo` ayant un identifiant spécifique, on peut créer la méthode `Query` en ajoutant la méthode `findById()` à l'interface `TodoRepository`.
  Après cela, l'interface `Repository` devient:

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


interface TodoRepository extends Repository<Todo,
Long> {


    //This is a query method.
    Todo findById(Long id);
}
```

Valeurs issues de requêtes Query

- si on fait une requête qui ne doit renvoyer qu'un seul résultat, on peut renvoyer les types suivants:
  - Type de base: la méthode Query retournera le type de base trouvé ou null.
  - Entité: la méthode Query retournera un objet entité ou null.
  - Guava / Java 8 `Optional<T>`: la méthode Query renverra un `Optional` contenant l'objet trouvé ou un `Optional` vide.

```
import java.util.Optional;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.Repository;
import org.springframework.data.repository.query.Param;

interface TodoRepository extends Repository<Todo, Long> {

    @Query("SELECT t.title FROM Todo t where t.id = :id")
    String findTitleById(@Param("id") Long id);

    @Query("SELECT t.title FROM Todo t where t.id = :id")
    Optional<String> findTitleById(@Param("id") Long id);

    Todo findById(Long id);

    Optional<Todo> findById(Long id);
}
```

Valeurs issues de requêtes Query

- si on fait une requête qui peut renvoyer plusieurs résultats, on peut renvoyer les types suivants:
  - `List<T>`: la méthode Query renverra une liste contenant les résultats de la requête ou une liste vide.
  - `Stream<T>`: la méthode Query renverra un flux pouvant être utilisé pour accéder aux résultats de la requête ou à un flux vide.

```
import java.util.stream.Stream;
import org.springframework.data.repository.Repository;

interface TodoRepository extends Repository<Todo, Long> {

    List<Todo> findByTitle(String title);

    Stream<Todo> findByTitle(String title);
}
```

Valeurs issues de requêtes Query

- Pour une méthode de requête exécutée de manière asynchrone, il faut l'annoter avec l'annotation `@Async` et renvoyer un objet `Future <T>`.

```
import java.util.concurrent.Future;
import java.util.stream.Stream;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.Repository;
import org.springframework.data.repository.query.Param;
import org.springframework.scheduling.annotation.Async;


interface TodoRepository extends Repository<Todo, Long> {

    @Async
    @Query("SELECT t.title FROM Todo t where t.id = :id")
    Future<String> findTitleById(@Param("id") Long id);

    @Async
    @Query("SELECT t.title FROM Todo t where t.id = :id")
    Future<Optional<String>> findTitleById(@Param("id") Long id);

    @Async
    Future<Todo> findById(Long id);

    @Async
    Future<Optional<Todo>> findById(Long id);

    @Async
    Future<List<Todo>> findByTitle(String title);

    @Async
    Future<Stream<Todo>> findByTitle(String title);
}
```

Passer des paramètres à une méthode Query

```
import java.util.Optional
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.Repository;

public interface TodoRepository extends Repository<Todo, Long> {

    public Optional<Todo> findByTitleAndDescription(String title, String description);

    @Query("SELECT t FROM Todo t where t.title = ?1 AND t.description = ?2")
    public Optional<Todo> findByTitleAndDescription(String title, String description);

    @Query(value = "SELECT * FROM todos t where t.title = ?0 AND t.description = ?1", nativeQuery=true)
    public Optional<Todo> findByTitleAndDescription(String title, String description);
}
```

- Pour passer des paramètres à des requêtes en BD on place des paramètres aux méthodes Query.
- Spring Data JPA prend en charge la liaison de paramètres basée sur la position et les paramètres nommés.
  - La liaison de paramètre basée sur la position signifie que l'ordre des paramètres de la méthode décide quels emplacements sont remplacés par ceux-ci.

- L'utilisation de la liaison de paramètres basée sur la position est **une source d'erreur**, car on ne peut pas modifier l'ordre des paramètres de la méthode. On résout ce problème en utilisant des paramètres nommés.

Passer des paramètres à une méthode Query

- On utilise des paramètres nommés en annotant les paramètres des méthodes avec l'annotation `@Param`.

```
import java.util.Optional
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.Repository;
import org.springframework.data.repository.query.Param;


public interface TodoRepository extends Repository<Todo, Long> {

    @Query("SELECT t FROM Todo t where t.title = :title AND t.description = :description")
    public Optional<Todo> findByTitleAndDescription(@Param("title") String title,
                                                    @Param("description") String description);

    @Query(value = "SELECT * FROM todos t where t.title = :title AND t.description = :description",
        nativeQuery=true
    )
    public Optional<Todo> findByTitleAndDescription(@Param("title") String title,
                                                    @Param("description") String description);
}
```

1- Création de requêtes en BD à partir de noms de méthodes

- La génération de requête à partir du nom de méthode est une stratégie de génération de requête dans laquelle la requête invoquée est dérivée du nom de la méthode de requête.

- On créer des méthodes Query qui utilisent cette stratégie en 6 règles:

1. Le nom de la méthode Query doit commencer par l'un des préfixes suivants: `find…By`, `read…By`, `query…By`, `count…By`, et `get…By`.
2. Si on veut limiter le nombre de résultats de requête renvoyés, on peut ajouter le mot clé `First` ou le mot clé `Top` avant le premier mot `By`.

   Si on veut obtenir plus d'un résultat, on doit ajouter la valeur numérique facultative aux mots-clés `First` et `Top`.
   Par exemple, `findTopBy`, `findTop1By`, `findFirstBy` et `findFirst1By` retournent tous la première entité qui correspond aux critères de recherche spécifiés.
3. Si on veut sélectionner des résultats uniques, on doit ajouter le mot clé `Distinct` avant le premier `By`.
   Par exemple, `findTitleDistinctBy` ou `findDistinctTitleBy` signifie que l'on sélectionne tous les titres uniques trouvés dans la BD.
4. On doit ajouter les critères de recherche de la méthode Query après le premier `By`.
   On peut spécifier les critères de recherche en combinant les expressions de propriété avec les mots-clés pris en charge.
5. Si la méthode Query spécifie `x` conditions de recherche, on doit y ajouter des paramètres de méthode.
   Donc, le nombre de paramètres doit être égal au nombre de conditions de recherche.
   De plus, les paramètres de la méthode doivent être donnés dans le même ordre que les conditions de recherche.
6. On doit définir le type de retour de la méthode Query

1- Création de requêtes en BD à partir de noms de méthodes

- Si on crée une méthode Query qui renvoie l'entrée `Todo` dont l'`id` est donné en tant que paramètre de méthode, on ajoute l'une des méthodes de requête suivantes à l'interface de référentiel:

```
import org.springframework.data.repository.Repository;
import java.util.Optional;

public interface TodoRepository extends Repository<Todo, Long> {

    /**
     * Retourne l'instance Todo don't l'id en parameter est le
     * critere de recherche. Si aucun todo n'est trouvé, alors
     * elle retourne null.
     */
    public Todo findById(Long id);

    /**
     * Retourne un Optional qui contient l'instance Todo trouvé
     * en utilisant son id comme critère de rechercher. Si
     * rien n'est trouvé, alors elle retourne un Optional vide.
     */
    public Optional<Todo> findById(Long id);
}
```

1- Création de requêtes en BD à partir de noms de méthodes

- Si on créer une méthode Query qui renvoie une liste de `Todo` dont l'id est donné en tant que paramètre de méthode,
on ajoute l'une des méthodes de requête suivantes à l'interface de référentiel:

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

import java.util.List;


public interface TodoRepository extends Repository<Todo, Long> {


    /**
     * Retourne l'instance Todo don't la propriété title ou description
     * est passé en paramètre. Si aucun todo n'est trouvé, alors elle
     * retourne une liste vide.
     */

    public List<Todo> findByTitleOrDescription(String title, String
description);

}
```

1- Création de requêtes en BD à partir de noms de méthodes

- Si on crée une méthode Query qui retourne le nombre de `Todo` dont le titre est donné en tant que paramètre de méthode,
on ajoute la méthode de requête suivante à notre interface de référentiel:

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

public interface TodoRepository extends Repository<Todo, Long> {

    /**
     * Retourne un nombre d'instances de Todo dont le title
     * est passé en paramètre.
     */
    public long countByTitle(String title);
}
```

1- Création de requêtes en BD à partir de noms de méthodes

- Si on renvoie les `Todo` distinctes dont le titre est donné en tant que paramètre de méthode,
on doit ajouter la méthode Query suivante à notre interface de référentiel:

```
import org.springframework.data.repository.Repository;
import java.util.List;

public interface TodoRepository extends Repository<Todo, Long> {

    /**
     * Retourne une liste d'instances uniques de Todo dont le
     * title est passé en paramètre. Si aucun Todo n'est
     * trouvé alors elle retourne une liste vide.
     */
    public List<Todo> findDistinctByTitle(String title);
}
```

1- Création de requêtes en BD à partir de noms de méthodes

- Si on renvoie les 3 premières instances de Todo dont le titre est donné en tant que paramètre de méthode, on ajoute l'une des méthodes de requête suivantes à notre interface de référentiel:

```
import org.springframework.data.repository.Repository;
import java.util.List;

public interface TodoRepository extends Repository<Todo, Long> {

    /**
     * Retourne une liste des 3 premières instances de Todo dont le
     * title est passé en paramètre. Si aucun Todo n'est trouvé alors
     * elle retourne une liste vide.
     */
    public List<Todo> findFirst3ByTitleOrderByTitleAsc(String title);

    /**
     * Retourne une liste des 3 premières instances de Todo dont le
     * title est passé en paramètre. Si aucun Todo n'est trouvé alors
     * elle retourne une liste vide.
     */
    public List<Todo> findTop3ByTitleOrderByTitleAsc(String title);
}
```

Exemple

- Pour implémenter une fonction Query on procède comme suit:

1. Créer une méthode Query dont le nom commence par le préfixe `findBy`.
2. Contrôler que la méthode renvoie les entrées `Todo` dont la description contient le terme recherché.
   On le fait en ajoutant l'expression de propriété: `Description` et le mot clé: `Contains` au nom de la méthode.
3. Configurer la méthode pour renvoyer les informations d'un `Todo` si la condition de recherche `previous` ou `next` est vraie.
   On le fait en ajoutant le mot-clé: `Or` au nom de la méthode.
4. Contrôler que la méthode renvoie les entrées `Todo` dont le titre contient le terme de recherché.
   On le faire en ajoutant l'expression de propriété: `Title` et le mot clé: `Contains` au nom de la méthode.
5. Contrôler que la recherche est insensible à la casse.
   On le faire en ajoutant le mot-clé: `AllIgnoreCase` au nom de la méthode.
6. Ajoutez 2 paramètres de méthode à la méthode de requête:
   - Spring Data JPA utilise le paramètre `descriptionPart` lorsqu'il veut que la description du `Todo` à renvoyer contient le terme recherché.
   - Spring Data JPA utilise le paramètre `titlePart` lorsqu'il veut que le titre du Todo à renvoyer renvoyée contient le terme de recherché.
7. Définir le type de l'objet renvoyé sur `List<Todo>`

```
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.Repository;
import java.util.List;

public interface TodoRepository extends Repository<Todo, Long> {
    List<Todo> findByDescriptionContainsOrTitleContainsAllIgnoreCase(String descriptionPart,
                                                                     String titlePart);
}
```

Application de la génération de requêtes ?

- Cette stratégie de génération de requêtes présente **les avantages suivants**:
  - La création de requêtes simples est rapide.
  - Le nom de la méthode Query décrit la ou les valeurs sélectionnées et la ou les conditions de recherche utilisées.

- Cette stratégie de génération de requêtes présente **les faiblesses suivantes**:
  - Les fonctionnalités de l'analyseur de nom de méthode déterminent le type de requêtes que l'on crée.
    Si l'analyseur de nom de méthode ne prend pas en charge le mot clé requis, on ne peut pas utiliser cette stratégie.
  - Les noms des méthodes Query complexes sont longs et laids.
  - Il n'y a pas de support pour les requêtes dynamiques.

2- Utilisation de l’annotation `@Query`

- On configurer la requête en BD en annotant avec `@Query`.

  Il prend en charge les requêtes `JPQL` et `SQL`.

  La requête spécifiée à l'aide de l'annotation `@Query` précède toutes les autres stratégies de génération de requête.

- Si on crée une méthode appelée `findbyId()` et l’annote avec `@Query`, Spring Data JPA ne trouvera pas l’entité dont la propriété `id` est égale au paramètre de méthode indiqué.

  Il appelle la requête configurée à l'aide de l'annotation `@Query`.

```
import org.springframework.data.repository.Repository;
import java.util.Optional;


interface TodoRepository extends Repository<Todo, Long> {


    @Query("SELECT t FROM Todo t WHERE t.title = 'title'")
    public List<Todo> findById();
}
```

Bien que la méthode `findById()` respecte la convention de dénomination utilisée pour créer des requêtes en BD à partir du nom de méthode, la méthode `findById()` renvoie les `Todo` dont le titre est `'title'`, car c'est la requête spécifiée. en utilisant l'annotation `@Query`.

2- @Query pour du JPQL

- On crée une requête JPQL avec l'annotation `@Query` en 2 étapes:

1. Ajouter une méthode de requête à l'interface du référentiel.
2. Annoter la méthode de requête avec l'annotation `@Query` et spécifier la requête invoquée en la définissant comme valeur de l'annotation `@Query`.

```
import org.springframework.data.repository.Repository;
import java.util.Optional;

interface TodoRepository extends Repository<Todo, Long> {

    @Query("SELECT t FROM Todo t WHERE t.title = 'title'")
    public List<Todo> findByTitle();
}
```

2- @Query pourdu SQL

- On crée une requête SQL avec l'annotation `@Query` en 3 étapes:

1. Ajouter une méthode de requête à l'interface de référentiel.
2. Annoter la méthode de requête avec l'annotation `@Query`
   et spécifier la requête invoquée en la définissant comme valeur de l'attribut value de l'annotation `@Query`.
3. Définir la valeur de l'attribut `nativeQuery` de l'annotation `@Query` sur true.

```
import org.springframework.data.repository.Repository;
import java.util.Optional;


interface TodoRepository extends Repository<Todo, Long>
{


    @Query(value = "SELECT * FROM todos t WHERE t.title
= 'title'", nativeQuery=true)
    public List<Todo> findByTitle();
}
```

Exemple @Query JPQL

- On veut implémenter une fonction de recherche simple avec 2 exigences:

1. Il doit renvoyer les `Todo` dont le titre ou la description contient le terme recherché indiqué.
2. La recherche doit être insensible à la casse.

- Cela se réalise en 4 étapes:

1. Créer une méthode qui renvoie une liste d'objets `Todo`.
2. Annoter la méthode avec `@Query`.
3. Créer la requête JPQL qui utilise des paramètres nommés et retourne les `Todo` dont le titre ou la description contient le terme de recherche donné. Définir la requête créée comme valeur de l'annotation `@Query`.
4. Ajouter un paramètre à la méthode de requête et configurer le nom du paramètre nommé en annotant celui-ci avec l'annotation `@Param`.

```
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.Repository;
import org.springframework.data.repository.query.Param;
import java.util.List;

public interface TodoRepository extends Repository<Todo, Long> {

    @Query("SELECT t FROM Todo t WHERE " +
           "LOWER(t.title) LIKE LOWER(CONCAT('%',:searchTerm, '%')) OR " +
           "LOWER(t.description) LIKE LOWER(CONCAT('%',:searchTerm, '%'))")
    List<Todo> findBySearchTerm(@Param("searchTerm") String searchTerm);
}
```

Exemple @Query SQL

- Cela se réalise aussi en 4 étapes:

1. Créer une méthode de requête qui renvoie une liste d'objets `Todo`.
2. Annoter la méthode avec l'annotation `@Query`.
3. Créer la requête SQL appelée en procédant en 2 étapes:
   1. Créer la requête SQL qui utilise des paramètres nommés et retourne les `Todo` dont le titre ou la description contient le terme de recherche donné.

      Définir la requête créée comme valeur de l'attribut value `@Query`.
   2. Définir la valeur de l'attribut `nativeQuery` de l'annotation `@Query` sur true.
4. Ajouter un paramètre à la méthode de requête et configurer le nom du paramètre nommé en annotant celui-ci avec `@Param`.

```
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.Repository;
import org.springframework.data.repository.query.Param;
import java.util.List;


public interface TodoRepository extends Repository<Todo, Long> {


    @Query(value = "SELECT * FROM todos t WHERE " +
         "LOWER(t.title) LIKE LOWER(CONCAT('%',:searchTerm, '%')) OR " +
         "LOWER(t.description) LIKE LOWER(CONCAT('%',:searchTerm, '%'))",
         nativeQuery = true
    )
    List<Todo> findBySearchTermNative(
                                    @Param("searchTerm") String searchTerm);
}
```

Comment choisir avec les méthodes générées

- L'annotation `@Query` présente les avantages suivants:
  - Il supporte à la fois `JPQL` et `SQL`.
  - La requête invoquée se trouve au dessus de la méthode de requêtage.
    - Donc, il est facile de savoir ce que fait la méthode de requête.
  - Il n'y a pas de convention de dénomination pour les noms de méthodes d'interrogation.
  - Si on doit déterminer quelle requête en BD est appelée par une méthode de requêtage, on peut trouver la requête invoquée au-dessus de la méthode de requêtage.

- L'annotation `@Query` présente les inconvénients suivants:
  - Il n'y a pas de support pour les requêtes dynamiques.
  - Si on utilise des requêtes `SQL`, on ne peut pas modifier la base de données utilisée sans vérifier que nos requêtes SQL fonctionnent comme prévu.
  - il est clair que les méthodes de requêtage qui utilisent cette stratégie ne sont pas aussi faciles à écrire ou à lire que les méthodes de requêtage qui utilisent la génération de requête à partir de la stratégie de nom de méthode.

3- Création de requêtes en BD avec des requêtes nommées

- On spécifie des requêtes nommées avec Spring Data JPA via un fichier de propriétés, des annotations ou un fichier `orm.xml`.

- Règles à suivre pour spécifier les noms des requêtes nommées:

1. Si on utilise la stratégie de dénomination par défaut de Spring Data JPA, on spécifie le nom de la requête nommée en utilisant la syntaxe:
   `[nom de la classe d'entité].[Nom de la méthode de requêtage invoquée]`

2. Si on utilise une autre syntaxe, on configure le nom de la requête nommée lorsqu'on crée la méthode de requêtage qui l'appelle.
   On le fait via l'attribut `name` de l'annotation `@Query`.

3- Création de requêtes en BD avec des requêtes nommées par fichier de properties

- Utiliser un fichier de propriétés
  - Pour déclarer des requêtes nommées, on les ajoute dans le fichier `jpa-named-queries.properties` qui se trouve dans le dossier `META-INF` du CLASSPATH.

- On déclare une nouvelle requête nommée en 2 étapes:
  1. Définir le nom de la requête nommée comme nom de la propriété.
  2. Définir la requête invoquée comme valeur de la propriété.

  Le fichier de propriétés utilise le format suivant:
  `name=query`

- On crée une requête nommée dont le nom est '`Todo.findByTitleIs`'. Toutes les `Todo` dont le titre est '`title`' sont renvoyées.

- Si on utilise JPQL, on ajoute une ligne dans le fichier de propriétés:

```
Todo.findByTitleIs=SELECT t FROM Todo t
WHERE t.title = 'title'
```

- Si on utilise SQL, on ajoute une ligne dans le fichier de propriétés:

```
Todo.findByTitleIs=SELECT * FROM todos t
WHERE t.title = 'title'
```

3- Création de requêtes en BD avec des requêtes nommées par annotation

- On déclare des requêtes nommées en annotant les entités avec les annotations suivantes:
  - Si on crée une requête JPQL, on annote l'entité avec l'annotation `@NamedQuery`.
  - Si on crée une requête SQL, on annote l'entité avec l'annotation `@NamedNativeQuery`.

- Si on crée plusieurs requêtes nommées, on enveloppe les requêtes dans l'annotation `@NamedQueries` ou `@NamedNativeQueries`.

```
import javax.persistence.Entity;
import javax.persistence.NamedQuery;
import javax.persistence.Table;

@Entity
@NamedQuery(name = "Todo.findByTitleIs",
        query = "SELECT t FROM Todo t WHERE t.title = 'title'"
)
@Table(name = "todos")
public final class Todo {

}
```

- On crée une requête nommée dont le nom est '`Todo.findByTitleIs`'. Toutes les entrées dont le titre est 'title' sont renvoyées.

- Pour créer une requête JPQL, on a 3 étapes:

1. Annoter l'entité avec l'annotation `@NamedQuery`.
2. Définir le nom de la requête nommée (`Todo.findByTitleIs`) comme valeur de l'attribut name de l'annotation `@NamedQuery`.
3. Définir la requête JPQL (`SELECT t FROM à WHERE t.title = 'title'`) comme valeur de l'attribut de requête de l'annotation `@NamedQuery`.

3- Création de requêtes en BD avec des requêtes nommées par annotation

- Si on crée une requête SQL, on a 4 étapes:

1. Annoter l'entité avec l'annotation `@NamedNativeQuery`.
2. Définir le nom de la requête nommée (`Todo.findByTitleIs`) comme valeur de l'attribut `name` de l'annotation `@NamedNativeQuery`.
3. Définir la requête SQL (`SELECT * FROM todos WHERE t.title = 'title'`) comme valeur de l'attribut name de l'annotation `@NamedNativeQuery`.
4. Définir la classe d'entités renvoyée (`Todo.class`) comme valeur de l'attribut `resultClass` de l'annotation `@NamedNativeQuery`.

```
import javax.persistence.Entity;
import javax.persistence.NamedNativeQuery;
import javax.persistence.Table;


@Entity
@NamedNativeQuery(name = "Todo.findByTitleIs”,
        query="SELECT * FROM todos t WHERE t.title = 'title'",
        resultClass = Todo.class
)
@Table(name = "todos")
final class Todo {


}
```

3- Création de requêtes en BD avec des requêtes nommées par fichier `orm.xml`

- On peut ajouter les requêtes au fichier `orm.xml` dans le dossier `META-INF` à la racine du jar ou dans le `CLASSPATH`. 2 solutions possibles en XML:

- Si on crée une requête JPQL, on utilise la balise `<named-query>`.
- Si on crée une requête SQL, on utilise la balise `<named-navive-query>`.

- On crée une requête nommée dont le nom est '`Todo.findByTitleIs`'. Tous les `Todos` dont le titre est '`title`' sont renvoyées.
- Si on crée une requête JPQL, on a 3 étapes:

1. Ajouter une balise `<named-query>` dans le fichier `orm.xml`.
2. Définir le nom de la requête nommée (`Todo.findByTitleIs`) comme valeur de l'attribut `name` de balise `<named-query>`.
3. Ajouter une balise `<query>` fils de `<named-query>` nommée et définir la requête JPQL invoquée (`SELECT t FROM todos WHERE t.title = 'title'`) en tant que valeur de la balise `<query>`.

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings
        xmlns="http://java.sun.com/xml/ns/persistence/orm"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
                                    http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"
        version="2.0">
    <named-query name="Todo.findByTitleIs">
        <query>SELECT t FROM Todo t WHERE t.title = 'title'</query>
    </named-query>
</entity-mappings>
```

3- Création de requêtes en BD avec des requêtes nommées par fichier `orm.xml`

- Si on crée une requête SQL, on a 4 étapes:

1. Ajouter une balise `<named-native-query>` dans le fichier `orm.xml`.
2. Définir le nom de la requête nommée (`Todo.findByTitleIs`) en tant que valeur de l'attribut `name` de la balise `<named-native-query>`.
3. Définir le type de l'objet retourné (`fr.spring.data.jpa.todo.Todo`) comme valeur de l'attribut `result-class` de balise `<named-native-query>`.
4. Ajouter une balise `<query>` fils de la balise `<named-native-query>` et définissez la requête SQL invoquée (`SELECT * FROM todos WHERE t.title = 'title'`) en tant que valeur de la balise `<query>`.

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings
   xmlns="http://java.sun.com/xml/ns/persistence/orm"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"
         version="2.0">

    <named-native-query name="Todo.findByTitleIs"
      result-class="fr.spring.data.jpa.todo.Todo">
      <query>SELECT * FROM todos t WHERE t.title = 'title'
      </query>
    </named-native-query>
</entity-mappings>
```

Exemple d’utilisation

- On crée une méthode de requêtage qui appelle la requête nommée dont le nom est: `Todo.findByTitleIs`.
  Cette requête nommée retournant les `Todo`s dont le titre est `"title"`, elle n’a aucun paramètre.

- Si la requête nommée invoquée est une requête `JPQL`, on ajoute la méthode de requêtage suivante dans l'interface de référentiel:

```
import
org.springframework.data.repository.Repository;
import java.util.List;


interface TodoRepository extends Repository<Todo,
Long> {
    public List<Todo> findByTitleIs();
}
```

- Si la requête nommée invoquée est une requête SQL, on ajoute la méthode de requête suivante dans l'interface de référentiel:

```
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.Repository;
import java.util.List;


interface TodoRepository extends Repository<Todo,
Long> {

    @Query(nativeQuery = true)
    public List<Todo> findByTitleIs();
}
```

Autre exemple d'utilisation

- On crée la méthode de requêtage qui appelle la requête nommée dont le nom est: `Todo.findByDesc`. Cette requête nommée avec un paramètre nommé appelé `description`.

- Si la requête invoquée est une requête JPQL, on ajoute la méthode de requêtage suivante dans l'interface de référentiel:

```
import org.springframework.data.repository.Repository;
import org.springframework.data.repository.query.Param;
import java.util.List;


interface TodoRepository extends Repository<Todo, Long>
{


    public List<Todo> findByDesc(@Param("description")
String description);
}
```

- Si la requête invoquée est une requête SQL, on ajoute la méthode de requête suivante dans l'interface de référentiel:

```
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.Repository;
import org.springframework.data.repository.query.Param;
import java.util.List;


interface TodoRepository extends Repository<Todo, Long> {


    @Query(nativeQuery=true)
    public List<Todo> findByDesc(@Param("description")
String description);
}
```

Spring data LDAP

- Les serveurs d'annuaire LDAP sont des sacs de données hiérarchiques optimisés en lecture.
Ils sont utilisés pour stocker les informations relatives à l'utilisateur requises pour l'authentification et l'autorisation des utilisateurs.

- On utilise les API Spring LDAP pour s'authentifier et rechercher des utilisateurs, ainsi que pour créer et modifier des utilisateurs dans le serveur d'annuaire.

- Le projet Spring LDAP offre la possibilité d'associer les entrées LDAP aux objets Java à l'aide du mapping ODM (Object-Directory Mapping).

- Définir l'entité qui sera utilisée pour mapper les annuaires LDAP

```
@Entry(
  base = "ou=users",
  objectClasses = { "person", "inetOrgPerson", "top" })
public class User {
    @Id
    private Name id;

    private @Attribute(name = "cn") String username;
    private @Attribute(name = "sn") String password;

    // standard getters/setters
}
```

Spring data LDAP

- `@Entry` est similaire à `@Entity` (de JPA / ORM), qui est utilisé pour spécifier quelle entité est mappée à la racine du répertoire des entrées LDAP.

- Une annotation `@Id` doit être déclarée dans une classe `Entry` dans un champ de type `javax.naming.Name` qui représente l'entité `DN`.
  L'annotation `@Attribute` est utilisée pour associer les attributs d'objet aux champs d'entité.

- Spring Data Repository est une abstraction qui fournit une implémentation prête à l'emploi des couches d'accès aux données pour divers sources de persistance.

- Spring Framework fournit en interne l'implémentation des opérations CRUD pour une classe donnée dans le référentiel de données.

- Spring Data LDAP fournit une abstraction similaire qui fournit l'implémentation automatique des interfaces de référentiel, y compris l'opération CRUD de base pour les annuaires LDAP.

Spring data LDAP

- Spring Data Framework peut créer une requête personnalisée basée sur un nom de méthode.

- Définir l'interface de référentiel qui sera utilisée pour gérer l'entrée `User`:

```
@Repository
public interface UserRepository extends
LdapRepository<User> {
    User findByUsername(String username);
    User findByUsernameAndPassword(String username,
                                   String password);
    List<User> findByUsernameLikeIgnoreCase(
                                  String username);
}
```

- On déclare l'interface en étendant `LdapRepository` pour `l'Entry User`. Spring Data Framework fournira automatiquement l'implémentation de base de la méthode CRUD telle que `find()`, `findAll()`, `save()`, `delete()`, etc.

- On déclare des méthodes personnalisées. Spring Data Framework fournira l'implémentation en analysant le nom de la méthode avec une stratégie du générateur de requêtes.

Exemple

- On configure Spring Data LDAP à l'aide de classes `@Configuration` basées sur Java ou d'un espace de noms XML.

```
@Configuration
@EnableLdapRepositories(basePackages =
"fr.spring.ldap.**")
public class AppConfig {
}
```

- `@EnableLdapRepositories` indique à Spring d'analyser le package donné à la recherche des interfaces marquées `@Repository`.

- Définir une classe de service qui utilisera `UserRepository` pour fonctionner sur les annuaires LDAP:

```
@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;

    // business methods
}
```

Exemple suite

- On implémente une simple logique pour authentifier un utilisateur existant:

```
public Boolean authenticate(String u, String p) {
    return userRepository.findByUsernameAndPassword(u, p) != null;
}
```

- Ensuite, on crée un nouvel utilisateur et on stocke le hachage du mot de passe:

```
public void create(String username, String password) {
    User newUser = new User(username,digestSHA(password));
    newUser.setId(LdapUtils.emptyLdapName());
    userRepository.save(newUser);
}
```

- On modifie un utilisateur ou une entrée existante avec la méthode:

```
public void modify(String u, String p) {
    User user = userRepository.findByUsername(u);
    user.setPassword(p);
    userRepository.save(user);
}
```

- On recherche des utilisateurs existants en utilisant une méthode personnalisée:

```
public List<String> search(String u) {
    List<User> userList = userRepository
      .findByUsernameLikeIgnoreCase(u);

    if (userList == null) {
        return Collections.emptyList();
    }

    return userList.stream()
      .map(User::getUsername)
      .collect(Collectors.toList());
}
```

Test final

- Pour finir le test d'authentification pour John Deuf:

```
@Test
public void givenLdapClient_whenCorrectCredentials_thenSuccessfulLogin()
{
    Boolean isValid = userService.authenticate("Deuf", "john");

    assertEquals(true, isValid);
}
```

Spring data MongoDB

- `MongoTemplate` suit le modèle de modèle standard de Spring et fournit une API de base prête à l'emploi pour le moteur MongoDB.

- Le référentiel suit l'approche centrée sur les données Spring et est livré avec des opération une API, basées sur les modèles d'accès de Spring Data.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-mongodb</artifactId>
    <version>2.1.1.RELEASE</version>
</dependency>
```

- Il faut configurer une BD locale MongoDB.
- La 1ère chose à faire est une classe qui représente des données à conserver

```
@Getter
@Setter
@ToString(exclude = {"id", "dateOfBirth"})
public class Person {

  @Id private String id;

  // on suppose que les mêmes noms de colonne
  private String firstName;
  private String secondName;
  private LocalDateTime dateOfBirth;
  private String profession;
  private int salary;

  public Person(
      final String firstName,
      final String secondName,
      final LocalDateTime dateOfBirth,
      final String profession,
      final int salary) {
    this.firstName = firstName;
    this.secondName = secondName;
    this.dateOfBirth = dateOfBirth;
    this.profession = profession;
    this.salary = salary;
  }

  // générer les getters, setters et toString
}
```

Spring data MongoDB

- La seule annotation réelle de Spring Data `@Id` qui représente l'identificateur unique de l'objet associé sur `_id` et qui est généré lorsqu'il est conservé dans la base de données.
  - L'annotation peut être omise si le champ est nommé `id` ou `_id`.
  - Donc, dans l'exemple, l'annotation n'est pas nécessaire.

- Si l'annotation ou une propriété correctement nommée n'est pas incluse dans le programme persistant, un champ `_id` sera créé lors de son enregistrement tant que MongoDB nécessite que le champ soit rempli.

- Les autres propriétés sont laissées sans annotation
  - lors de la persistance ou de l'enregistrement dans la BD, elles feront association vers des champs qui partagent le même nom dans la BD.

- Créer un référentiel qui effectue toutes les opérations de BD à effectuer avec l'objet `Person`.

```
public interface PersonRepository extends
MongoRepository<Person, String> {

  List<Person> findBySalary(final int salary);
}
```

- L'interface étend `MongoRepository`, qui fournit de nombreuses méthodes CRUDS et quelques extras.
- Sur cette interface, une définition réelle a été ajoutée, et d'autres si nécessaire.
- Par conséquent, `findBySalary` trouvera les objets `Person` stockés dans la BD par salaire.
  `GetBySalary` pourrait également être utilisé.
  Pour exécuter ces requêtes, Spring Data utilise `MongoTemplate`.

Ex. MongoDB

- L'application principale possède l'annotation `@SpringBootApplication`.
- La classe a implémenté `CommandLineRunner` comme moyen pratique d'afficher rapidement le résultat de certaines des méthodes de référentiel.

- L'annotation `@EnableMongoRepositories` est requise, car `PersonRepository` ne se trouve pas dans le même package ou sous-package que la classe `@SpringBootApplication`.
- L'annotation est donc nécessaire pour spécifier que le référentiel doit être injecté dans l'application.
- Si le référentiel a été déplacé dans le même package que l'application, la propriété `basePackageClasses` peut être supprimée de l'annotation `@EnableMongoRepositories` ou même être supprimée.

```
@SpringBootApplication
// necessaire car le repository n'est pas dans le meme package
@EnableMongoRepositories(basePackageClasses = PersonRepository.class)
public class Application implements CommandLineRunner {

  @Autowired private PersonRepository personRepository;

  public static void main(final String args[]) {
    SpringApplication.run(Application.class, args);
  }

  @Override
  public void run(String... strings) throws Exception {
    personRepository.deleteAll();
    final Person john = new Person("John", "Doe", LocalDateTime.now(),
"Winner", 100);
    final Person harry = new Person("Harry", "Cover", LocalDateTime.now(),
"Loser", 50);
    personRepository.save(john);
    personRepository.save(harry);
    System.out.println("Find all");
    personRepository.findAll().forEach(System.out::println);
    System.out.println("Find by findBySalary");
    personRepository.findBySalary(100).forEach(System.out::println);

    System.out.println("Making John a loser");
    john.setProfession("Loser");
    personRepository.save(john);
    System.out.println("Find all");
    personRepository.findAll().forEach(System.out::println);
  }
}
```

Ex. Sortie

- la sortie suivante est générée

```
Find all

Person(firstName=John, secondName=Doe, profession=Winner, salary=100)

Person(firstName=Harry, secondName=Cover, profession=Loser, salary=50)

Find by getBySalary

Person(firstName=John, secondName=Doe, profession=Winner, salary=100)

Making John a loser

Find all

Person(firstName=John, secondName=Doe, profession=Loser, salary=100)

Person(firstName=Harry, secondName=Cover, profession=Loser, salary=50)
```

- Ce qui laisse les données suivantes persistées dans la base de données.

```
{
 "_id" : ObjectId("592071e667c5852be43d20e8"),
 "_class" : "fr.spring.data.mongodb.entities.Person",
 "firstName" : "John",
 "secondName" : "Doe",
 "dateOfBirth" : ISODate("2018-05-20T14:42:14.629+01:00"),
 "profession" : "Loser",
 "salary" : 100
}

{
 "_id" : ObjectId("592071e667c5852be43d20e9"),
 "_class" : "fr.spring.data.mongodb.entities.Person",
 "firstName" : "Harry",
 "secondName" : "Cover",
 "dateOfBirth" : ISODate("2018-05-20T14:42:14.632+01:00"),
 "profession" : "Loser",
 "salary" : 50
}
```

Spring data Redis

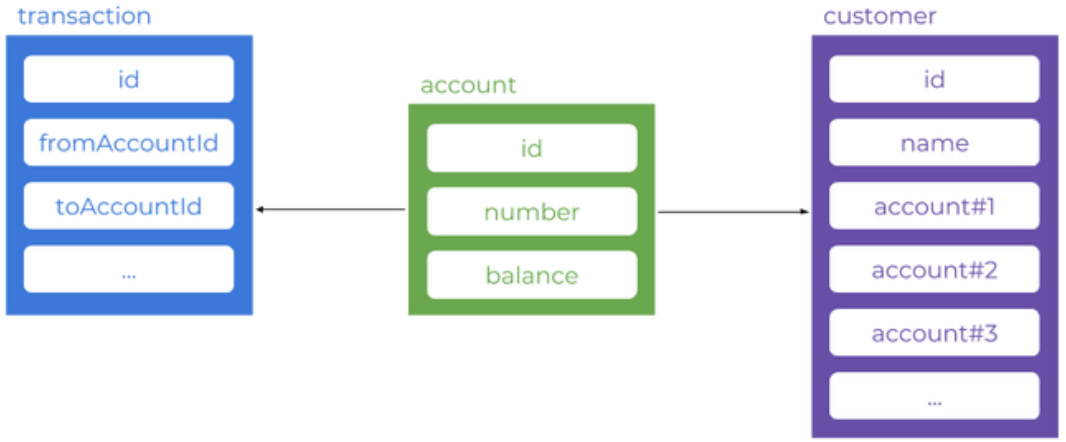
- Redis est un sac de couples de données en mémoire avec une durabilité optionnelle, utilisé comme BD.
- Actuellement, c'est l'outil le plus populaire dans la catégorie des BD de clés / valeur
- Redis est généralement utilisé pour mettre en cache des données stockées dans une BD relationnelle.
- Il existe des limitations lorsqu'on intègre Redis via les référentiels Spring Data Redis.
  Il faut au moins la version 2.8.0 de Redis Server et ne fonctionnent pas avec les transactions.
- Donc, on désactive la prise en charge des transactions pour `RedisTemplate`, qui est exploitée par les référentiels Redis.

Spring data Redis

- En supposant qu'on ait le système, composé de 3 classes : `Customer`, `Account` et `Transaction`,

- La transaction est toujours liée à 2 comptes: l'expéditeur (`fromAccountId`) et le destinataire (`toAccountId`).
  Chaque `Customer` peut avoir plusieurs `Accounts`.

- `Customer` et `Account` sont stockés dans la même structure unique.
  Tous les `Accounts` du `Customer` sont stockés sous forme de liste dans l'objet `Customer`.



Spring data Redis

- Pour activer les référentiels Redis pour une application Spring Boot.

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

- 2 drivers possibles:
  - Lettuce
  - Jedis.

```
@Configuration
@EnableRedisRepositories
public class SampleSpringRedisConfiguration {
    @Bean
    public LettuceConnectionFactory redisConnectionFactory() {
        return new LettuceConnectionFactory();
    }
    @Bean
    public RedisTemplate<?, ?> redisTemplate() {
        RedisTemplate<byte[], byte[]> template = new RedisTemplate<>();
        template.setConnectionFactory(redisConnectionFactory());
        return template;
    }
}
```

Spring data Redis

- Chaque entité du domaine, au moins, doit être annotée avec `@RedisHash` et contenir la propriété annotée avec `@Id`.

- Ces 2 éléments sont responsables de la création de la clé utilisée pour conserver le hachage.
  Outre les propriétés d'identifiant annotées avec `@Id`, on peut utiliser des index secondaires.

```
@RedisHash("customer")
public class Customer {
    @Id private Long id;
    @Indexed private String externalId;
    private String name;
    private List<Account> accounts = new ArrayList<>();
    public Customer(Long id, String externalId, String name) {
        this.id = id;
        this.externalId = externalId;
        this.name = name;
    }
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getExternalId() {
        return externalId;
    }
    public void setExternalId(String externalId) {
        this.externalId = externalId;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public List<Account> getAccounts() {
        return accounts;
    }
    public void setAccounts(List<Account> accounts) {
        this.accounts = accounts;
    }
    public void addAccount(Account account) {
        this.accounts.add(account);
    }
}
```

Spring data Redis

- `Account` n'a pas son propre hash. Il est contenu par le client sous forme de liste d'objets.

- L'identifiant de la propriété est indexé sur Redis afin d'accélérer la recherche en fonction de la propriété.

```
public class Account {
    @Indexed private Long id;
    private String number;
    private int balance;
    public Account(Long id, String number, int balance) {
        this.id = id;
        this.number = number;
        this.balance = balance;
    }
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getNumber() {
        return number;
    }
    public void setNumber(String number) {
        this.number = number;
    }
    public int getBalance() {
        return balance;
    }
    public void setBalance(int balance) {
        this.balance = balance;
    }
}
```

Spring data Redis

- `Transaction` utilise uniquement les identifiants de compte, pas les objets entiers.

- Les méthodes les plus courantes telles que `save`, `delete` ou `findById` sont générées par Spring Data.
  Il ne reste qu'à créer nos méthodes de recherche personnalisées si nécessaire.

```
@RedisHash("transaction")
public class Transaction {
    @Id
    private Long id;
    private int amount;
    private Date date;
    @Indexed
    private Long fromAccountId;
    @Indexed
    private Long toAccountId;
    public Transaction(Long id, int amount, Date date, Long fromAccountId,
Long toAccountId) {
        this.id = id;
        this.amount = amount;
        this.date = date;
        this.fromAccountId = fromAccountId;
        this.toAccountId = toAccountId;
    }
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public int getAmount() {
        return amount;
    }
    public void setAmount(int amount) {
        this.amount = amount;
    }
    public Date getDate() {
        return date;
    }
    public void setDate(Date date) {
        this.date = date;
    }
    public Long getFromAccountId() {
        return fromAccountId;
    }
    public void setFromAccountId(Long fromAccountId) {
        this.fromAccountId = fromAccountId;
    }
    public Long getToAccountId() {
        return toAccountId;
    }
    public void setToAccountId(Long toAccountId) {
        this.toAccountId = toAccountId;
    }
}
```

Spring data Redis - repository

```
public interface CustomerRepository extends
CrudRepository {
    Customer findByExternalId(String externalId);
    List findByAccountsId(Long id);
}
```

- la méthode `findByExternalId` sont plutôt évidentes
- la méthode `findByAccountsId` peut ne pas l'être.
  La `transaction` ne contient que les identifiants de `Account`; il n'a pas de relation directe avec le `Customer`.

```
public interface TransactionRepository extends
CrudRepository {
    List findByFromAccountId(Long fromAccountId);
    List findByToAccountId(Long toAccountId);
}
```

- Et si on a besoin d'apprendre les détails sur les `Customers` en plus d'une transaction donnée, on peut trouver un `Customer` par l'un de ses `Accounts`.

Test

- Pour tester la fonctionnalité des référentiels Redis à l'aide de Spring Boot Test on utilise `@DataRedisTest`.
- Ce test suppose que vous avez une instance en cours d'exécution du serveur Redis sur l'adresse déjà configurée 192.168.X.Y.

```
@RunWith(SpringRunner.class)
@DataRedisTest
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class RedisCustomerRepositoryTest {

    @Autowired
    CustomerRepository repository;

    @Test
    public void testAdd() {
        Customer customer = new Customer(1L, "80010121098", "John Smith");
        customer.addAccount(new Account(1L, "1234567890", 2000));
        customer.addAccount(new Account(2L, "1234567891", 4000));
        customer.addAccount(new Account(3L, "1234567892", 6000));
        customer = repository.save(customer);
        Assert.assertNotNull(customer);
    }
    @Test
    public void testFindByAccounts() {
        List<Customer> customers = repository.findByAccountsId(3L);
        Assert.assertEquals(1, customers.size());
        Customer customer = customers.get(0);
        Assert.assertNotNull(customer);
        Assert.assertEquals(1, customer.getId().longValue());
    }
    @Test
    public void testFindByExternal() {
        Customer customer = repository.findByExternalId("80010121098");
        Assert.assertNotNull(customer);
        Assert.assertEquals(1, customer.getId().longValue());
    }
}
```

Spring Data REST

- Spring Data REST permet de créer un backend pour manipuler les données très facilement.
- Cela supprime une grande partie du travail manuel généralement associé à ce type de tâches et simplifie la mise en œuvre des fonctionnalités de base de CRUD pour les applications Web.
- Les dépendances Maven suivantes sont requises pour une application simple:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId
    <artifactId>spring-boot-starter-data-rest
    </artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```

Spring Data REST

- Ecrire une classe du domaine pour représenter un utilisateur accessible depuis un site web:

```
@Entity
public class WebsiteUser {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    private String name;
    private String email;

    // standard getters and setters
}
```

- Chaque utilisateur a un nom et un email, ainsi qu'un identifiant généré automatiquement.
- Ecrire un simple référentiel:

```
@RepositoryRestResource(collectionResourceRel = "users",
                                path = "users")

public interface UserRepository extends
PagingAndSortingRepository<WebsiteUser, Long> {

    List<WebsiteUser> findByName(@Param("name") String name);
}
```

- C'est une interface qui permet d'effectuer des opérations avec les objets `WebsiteUser`. On peut définir une requête personnalisée qui fournira une liste d'utilisateurs basée sur un nom donné.

- L'annotation `@RepositoryRestResource` est facultative et permet de personnaliser l'endpoint REST. Si on l'omet, Spring créerait automatiquement un endpoint dans «`/websiteUsers`» au lieu de «`/users`».

Spring Data REST

- La classe principale Spring Boot standard pour initialiser l'application:

```
@SpringBootApplication
public class SpringDataRestApplication {
    public static void main(String[] args) {
        SpringApplication.run(
          SpringDataRestApplication.class,
          args);
    }
}
```

- Il existe également un endpoint standard «`/profile`», qui fournit des métadonnées d'application.
  Il est important de noter que la réponse est structurée de manière à respecter les contraintes du style d'architecture REST.
- Donc, il fournit une interface uniforme et des messages auto-descriptifs. Cela signifie que chaque message contient suffisamment d'informations pour décrire comment traiter le message.

- Si on exécute l'application et que on accéde à `http://localhost:8080/` dans un navigateur, on reçoit le JSON suivant:

```
{
  "_links" : {
    "users" : {
      "href" :
"http://localhost:8080/users{?page,size,sort}",
      "templated" : true
    },
    "profile" : {
      "href" : "http://localhost:8080/profile"
    }
  }
}
```

- Comme on le constate, un endpoint "/user" est disponible et il dispose déjà des options «`?page`», «`?size`» et «`?sort`».

Spring Data REST

- Il n'y a pas encore d'utilisateurs dans l'application, donc `http://localhost:8080/users` n'affichera qu'une liste vide d'utilisateurs.
- Ajouter un utilisateur avec RESTClient.

- Les en-têtes de réponse:

```
HTTP/1.1 201 Created
Server: Apache-Coyote/1.1
Location: http://localhost:8080/users/1
Content-Type: application/hal+json;charset=UTF-8
Transfer-Encoding: chunked
```

```
$ curl -i -X POST -H "Content-Type:application/json" -d '{  "name" : "Test", \
"email" : "test@test.com" }' http://localhost:8080/users
{
  "name" : "test",
  "email" : "test@test.com",
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/users/1"
    },
    "websiteUser" : {
      "href" : "http://localhost:8080/users/1"
    }
  }
}
```

Spring Data REST

- On accède à cet utilisateur à l'adresse http://localhost:8080/users/1

```
{
  "name" : "test",
  "email" : "test@test.com",
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/users/1"
    },
    "websiteUser" : {
      "href" : "http://localhost:8080/users/1"
    }
  }
}
```

- Avec `curl` ou tout autre client REST pour émettre des demandes `PUT`, `PATCH` et `DELETE`.
- On note que Spring Data REST suit automatiquement les principes de HATEOAS.
- HATEOAS est l'une des contraintes du style d'architecture REST et signifie que l'hypertexte doit être utilisé pour se frayer un chemin dans l'API.
- On accède à la requête personnalisée décrite précédemment pour trouver tous les utilisateurs portant le nom «test».

http://localhost:8080/users/search/findByName?name=test

```
{
  "_embedded" : {
    "users" : [ {
      "name" : "test",
      "email" : "test@test.com",
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/users/1"
        },
        "websiteUser" : {
          "href" : "http://localhost:8080/users/1"
        }
      }
    } ]
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/users/search/findByName?name=test"
    }
  }
}
```