Spring Boot CrudRepository Sample

Hi, in this sample we'll be looking up how to create and use `SpringBoot CrudRepository`.

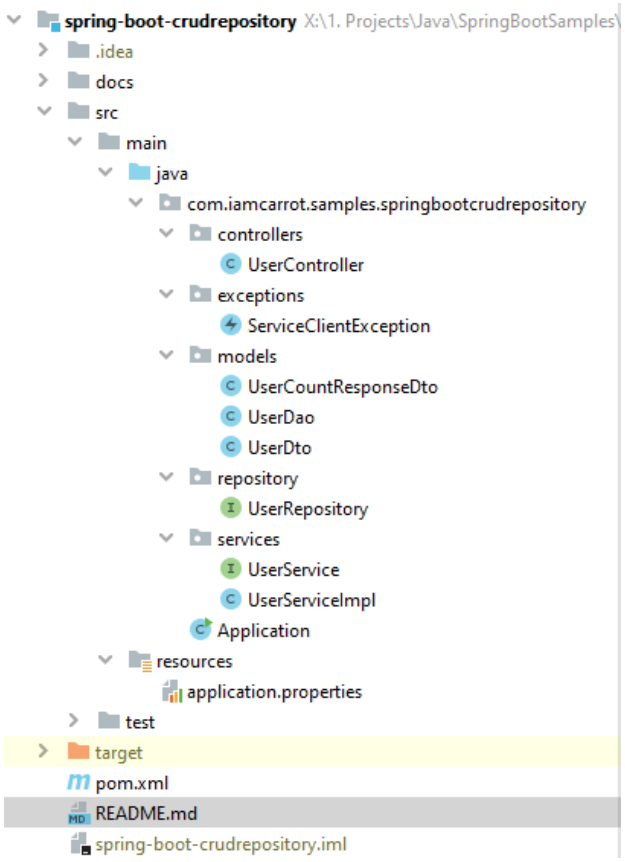
SpringBoot CrudRepository provides sophisticated CRUD functionality for the type of entity you want to be managed. This interface acts primarily as a marker interface to capture the types to work with and to help you to discover interfaces that extend this one. It takes the the domain class to manage as well as the id type of the domain class as type arguments.

Generally, to use up a `CrudRepository` you'll need a `database` and SpringFramework supports wide number of them including the most common ones like SQL, `MySQL`, `MongoDB`, `H2 database` and more. Let's get on with it then.

For this sample you'll need:

- Java 8+
- Maven 3+
- Spring Boot (the version of your choice we'll be using v2.2.2.RELEASE but the concepts are valid for 2.x SpringBoot)
- A Database of your choice (We'll be using MongoDB)
- IDE of your choice (we'll be using Intellij Idea)

Project Structure:



Let's get started on the sample.

Getting Started

Get a File > New Project and add necessary dependencies, you can go to Spring Starter IO and add the below dependencies:

- spring-boot-starter-web
- spring-boot-starter-data-mongodb

After that your `pom.xml` would have the below:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.2.2.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>

<dependencies>

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

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

</dependencies>
```

Spring basically is a huge umbrella of frameworks and the way it supports multiple databases is leveraging starter projects and their respective auto-configurations. You can read all about them here and here The `starter-parent` is how spring handles dependency management and solves a few of many problems with transitive dependencies read more.

The Repository Class

Create a new interface called `UserRepository`

```
@Repository
public interface UserRepository extends CrudRepository<UserDao, String> {

}
```

The `UserRepository` would extend the `CrudRepository<T, Y>` interface. Here T is the class of the entity that needs to be created, read, updated, deleted in/from a database and Y the type of the id field of the entity.

The `@Repository` annotation marks the interface as a `Spring Stereotype` and spring would provide and implementation for this interface at runtime (reducing boiler-plate database code) read more.

The `CrudRepository` has basic create, read, update, delete operation methods:

- `<S extends T> S save(S entity);` Saves a given record to the DB
- `<S extends T> Iterable<S> saveAll(Iterable<S> entities);` Saves multiple records to the DB.
- `Optional<T> findById(ID id);` Finds a record that matches the Id
- `boolean existsById(ID id);` Checks if a record exists with the provided Id
- `Iterable<T> findAll();` Fetches all records from DB
- `Iterable<T> findAllById(Iterable<ID> ids);` fetches all records that match the provided Ids
- `long count();` returns the number of records in the DB.
- `void deleteById(ID id);` Deletes the entity with the given id.
- `void delete(T entity);` Deletes a given entity.
- `void deleteAll(Iterable<? extends T> entities);` Deletes all the records provided.
- `void deleteAll();` deletes all records from the DB

Most of the functions are self-explanatory and we'll look at a few of them in this sample but let's first talk about the `save` and `saveAll` method.

The Save method

The reason why `save` gets a whole section is because something that got to me as well when I was in the initial stages with `SpringBoot`.

The `save` and `saveAll` basically works as an `upsert` (update or insert). If the object you pass in the save method has an `id` (annotated with the `@Id` in MongoDB UserDao) value, spring would understand that there seems to be a record in the DB with the same id and will try to find it to perform the update, in case it doesn't find it Spring would create a new record and set the id field (in DB) to the value that was present in the id field of the object. In case the `id` field is null or empty, spring would treat it as a new record and perform the insert.

> *Careful though with the `id` field, if let's say that you don't have any id field in your database class (DAO) even if you read the record from the DB and call the `save()` method, spring would consider it as a new record and a Duplicate record would be created. If you have any unique constraints in your database collection they would fail and throw a DuplicateKeyException*

The User Entity Class

The UserDao is the class that represents the user record stored in the database.

```
@Document(collection = "users")
public class UserDao {

    @Id
    private String userId;

    @Indexed(unique = true)
    @Field("username")
    private String username;

    @Field("firstName")
    private String firstName;

    @Field("lastName")
    private String lastName;

    public UserDao() {
    }

    public UserDao(UserDto context) {
        this.userId = context.getUserId();
        this.username = context.getUsername();
        this.firstName = context.getFirstName();
        this.lastName = context.getLastName();
    }

    public UserDao updatedFrom(UserDto context) {

        if (context.getUsername() != null && !context.getUsername().trim().isEmpty(
))
            this.username = context.getUsername();

        if (context.getUsername() != null && !context.getUsername().trim().isEmpty(
))
            this.firstName = context.getFirstName();

        if (context.getUsername() != null && !context.getUsername().trim().isEmpty(
))
            this.lastName = context.getLastName();

        return this;
    }

    public String getUserId() {
        return userId;
    }

    public String getUsername() {
        return username;
    }
```

```
    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }
}
```

The class contains a few fields like userId, username, first name, last name. The userId is the field that represents the `id` field of the database and hence is annotated as `@Id` (that's how it's done for MongoDB).

The `@Document` annotation is another Spring Stereotype that marks the class as a database class and spring would use that class for any data that's read or written to the users collection (where users is the name of the collection)

The `@Field` annotation defines the name of the field in the database and maps it to the property. If no field is annotated with `@Field` then the name of the variable is taken as that of the field name. This is primarily useful when the database record has different names as posed to class fields. e.g. the id field, in the class the id field is `userId` but when it comes to MongoDB the id field name is `_id` and that's what the `@Id` field does. Alternatively, you can use the `@Field("_id")` instead of the `@Id` and it'll work the same.

The `@Indexed(unique=true)` field is what sets the username field to be unique and no two records can have the same username. The reason why it's not put over the userId field is because it's the `id` field and each DB has its id field unique out of the box

> *Caution for a class to be able be de-serialized from DB it's important to have a default constructor (NoArgsConstructor) so that spring can create a new object of the class and set relevant data. The java rules do apply meaning when you do not define any constructor then each class has an auto implemented default constructor from Object base class.*

The `UserDto` is just another class that represents the `Data Transfer Object` returned from the service to the controller. You can look up the file in the GitRepository.

The downside of having the UserDto with almost the same fields is the code repetition and multiple object creation but if those doesn't seem to be a huge trade-off the core advantage this approach gives you is that you have a layer of adapter in the middle that separates your database fields to that of your API response so that when anything in your DB changes, your API users/clients do not get affected.

Bringing the UserDao and CrudRepository together

How does Spring know what repository to link to which db collection? It's all with the help of Generics. Spring first resolves `CrudRepository<UserDao, String>` and pulls out `UserDao` and looks for a class with that type which is annotated with `@Document`, when it finds it, it resolves the annotation for the name of the collection from the `@Document(collection=users)` and now it has linked UserRepository to CrudRepository to users collection in the DB.

The Service

We've wired up our repository and entity class now let's look at the service that the controller would call to interact.

Below is an implementation of the `UserService`. The code is trimmed to only the relevant ones. For a complete sample you should look up the GitRepository. You can refer to the above section about the `CrudRepository` methods to know what method does what

```
@Service
public class UserServiceImpl implements UserService {

    private final UserRepository userRepository;

    public UserServiceImpl(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @Override
    public String createUser(UserDto context) throws ServiceClientException {

        try {

            UserDao savedUser = userRepository.save(new UserDao(context));

            return savedUser.getUserId();

        } catch (DuplicateKeyException e) {
            // handle in case of duplicate username
            return null;
        }
    }

    @Override
    public UserCountResponseDto userCount() {

        long userCount = userRepository.count();

        return new UserCountResponseDto(userCount);
    }

    @Override
    public Collection<UserDto> getUsers() {

        Iterable<UserDao> users = userRepository.findAll();

        if (!users.iterator().hasNext())
            return null;

        Collection<UserDto> returner = new ArrayList<>();
        users.forEach(x -> returner.add(new UserDto(x)));

        return returner;
    }
```

```
    @Override
    public UserDto getUsers(String userId) throws ServiceClientException {

        Optional<UserDao> userOptional = userRepository.findById(userId);
        UserDao user = userOptional.orElse(null);
        return new UserDto(user);
    }

    @Override
    public Collection<UserDto> getUsersByPet(String petName) {

        Collection<UserDao> users = userRepository.findByPets(petName);

        return users.stream().map(UserDto::new).collect(Collectors.toList());
    }

    @Override
    public boolean updateUser(String userId, UserDto context) throws ServiceClientE
xception {

        try {

            UserDao user = userRepository.findById(userId).orElse(null);
            userRepository.save(user.updatedFrom(context));
            return true;

        } catch (DuplicateKeyException e) {
            // handle in case of duplicate username

        } catch (Exception e) {
        }
        return false;
    }

    @Override
    public boolean deleteUser(String userId) throws ServiceClientException {

        userRepository.deleteById(userId);
        return false;
    }
}
```

Notice the delete method, it directly calls the `deleteById()` and this would work because the `userId` is the `id` field of the database but what if the `userId` is not the `id` field of the database? Another way would be read the user first and then delete that record by passing the instance.

```
@Override
public boolean deleteUser(String userId) throws ServiceClientException {

    UserDao existingUser = userRepository.findById(userId).orElse(null);
    userRepository.delete(existingUser)
    return false;
}
```

Conclusion

We looked at the `SpringBoot CrudRepository`, it's methods and a sample service that interacts with the repository. A lot of code (like Exception Handling, null checks and the controller) were removed from the samples above to bring more focus on actual methods of the CrudRepository. The whole (un-cut) version of the code is available on github here.

> *Remember the CrudRepository is just a basic out of the box functionality and while it's a great way to get started you may would want to define your own methods in the UserRepository that would be more bound to your use cases. This is out of scope for this blog post. The deleteById(), findById() and save() they all work on the `id` field of the database and if you want your own id fields then you can create them and write queries and methods in the UserRepository interface and spring would wire up an implementation for you at runtime.*