Spring Data JDBC Workshop

Jens Schauder, September 2019

What is Spring Data?

Spring Data offers APIs for persisting data to various data stores. Some examples are MongoDB, Neo4J, Elastic Search, Relational Databases using JPA or JDBC and many more.

The API is inspired by Domain Driven Design and comes in most cases with two levels:

1. The Repository abstraction. A `Repository` looks somewhat like a collection of all instances of a given type with methods to add, find and delete instances. Repository can be extended in various ways many of which just require defining additional methods to a `Repository` without actually implementing them. The implementation will be provided by Spring Data.
2. The Template abstraction. A template is a class that allows more direct access to special features of the underlying persistence technology.

Spring Data does not try to unify the access to all these stores in order to make them interchangeable. Such an endeavor is bound to fail since each an every persistence store has features that are impossible or very expensive to emulate in another. A unifying API would either need to provide such an emulation or not offer any of these features at all.

Instead Spring Data tries to provide a consistent API that makes it easy for users of one Spring Data variant to learn another while encouraging good application design at the same time.

First Spring Data JDBC example

Create a Spring Starter Project

With Spring Tools and IntelliJ you can do this directly in the IDE. This only describes the web based variant.

1. Goto https://start.spring.io
2. Select
   - latest Milestone version of Spring Boot (2.2.0.M5)
   - add HyperSQL Database
   - add Lombok
   - add JDBC API
   - generate the project
   - unpack
   - open in the IDE

Some Notes on the dependencies:

Lombok isn't really needed but it makes the code nice and concise. If you don't like Lombok you can create constructors, `equals` and `hashcode` implementations with your IDE just as well.

JDBC API is actually not what we need. But it's close and we'll change it in a moment.

Add Spring Data JDBC

Open the `pom.xml`. Find the following section:

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

And replace it with this one.

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

Build everything

From the commandline run

```
./mvnw clean install
```

Or run the `DemoApplication` to make sure everything works and can be downloaded.

Create an entity and a repository

Create an entity.

```
import org.springframework.data.annotation.Id;

@ToString
@EqualsAndHashCode
class Speaker {
    @Id
    Long id;
    String name;
}
```

Create a repository

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

interface SpeakerRepository extends CrudRepository<Speaker, Long>
{ }
```

And check that you can inject it, for example by changing the DemoApplication as follows:

```
@SpringBootApplication
public class DemoApplication implements CommandLineRunner {

    @Autowired
    SpeakerRepository speakers;

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


    @Override
    public void run(String... args) {
        if (speakers != null) {
            System.out.println("I'm running and have a speakers
repository");
        } else {
            System.out.println("drats, something is wrong");
        }
    }
}
```

We implement the `CommandLineRunner` interface in order to execute some code at start up. Add an `@Autowired` field of type `SpeakerRepository` and check that it is not `null`.

Actually Use the Repository.

Add the following to the `run` method in order to create a Speaker and persist it:

```
Speaker martin = new Speaker();
martin.name = "Martin Fowler";

Speaker saved = speakers.save(martin);

System.out.println(saved);
```

If you execute the changed application you'll get an exception like the following:

```
user lacks privilege or object not found: SPEAKER in statement
[INSERT INTO speaker (name) VALUES (?)]
```

This is because we didn’t create any tables yet!

Add the following `schema.sql` to the `source/main/resources` folder to fix this.

```
CREATE TABLE SPEAKER
(
  ID   BIGINT GENERATED BY DEFAULT AS IDENTITY (START WITH 1)
PRIMARY KEY,
  NAME VARCHAR(200)
);
```

Add some Logging

In order to better understand what is going on we should add some logging. Add the following line to your `application.properties` file.

```
logging.level.org.springframework.jdbc.core=TRACE
```

All execution of SQL statements happen through Springs `JdbcTemplate` so it’s logging facility may be used for gaining insights what statements get executed.

Move our Experiments to a Test.

Moving our experiments to a test makes it easier to … well … test stuff. Create a class like the following an run it in your IDE.

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.data.jdbc.DataJdbcTest
;
import org.springframework.test.context.junit4.SpringRunner;

import static org.assertj.core.api.Assertions.*;

@RunWith(SpringRunner.class)
@DataJdbcTest
public class SpeakerRepositoryTests {

    @Autowired
    SpeakerRepository speakers;

    @Test
    public void saveInstance() {

        Speaker martin = new Speaker();
        martin.name = "Martin Fowler";

        Speaker saved = speakers.save(martin);

        assertThat(saved.id).isNotNull();
    }
}
```

Make sure to remove the save operation in the `CommandLineRunner`. Otherwise it will run before each test and mess with your test data.

Wrap up of the first example

In order to use Spring Data JDBC in a Spring Boot application you need to do the following:

- add `spring-boot-starter-data-jdbc` to your dependencies.
- create entities.
- create repositories.
- inject the latter where needed.

- persist away.

Spring Data JDBC vs Spring Data JPA

Why are there two Spring Data modules for the same underlying technology: Relational databases?

JPA is pretty much the default technology when accessing RDBMSes on the JVM. This does not mean it is without problems.

- It considers the complete domain object graph a single graph. So you might load a few instances with a query and then navigate the whole database using objects. While this is possible it is very inefficient. Either you rely on lazy loading which means the required data is loaded piece by piece. Also a simple call of an getter might trigger an arbitrary amount of SQL statements. Or you rely on eager loading which leads to large amounts of data loaded even if not needed.
- Similar to lazy loading JPA uses write behind to dynamically determine when to write back to the database. This can lead to confusing scenarios where the user thinks data is stored in the database but it actually isn't.
- JPA employs an first level cache that guarantees that for a given class and id only a single instance will get loaded into a persistence context. In 99% of the cases this is rather nice. But in 1% of the cases this is really confusing. For example you can't easily load the current state of an object from the database if you already loaded it before.
- Finally JPA maintains a relationship between domain objects and the persistence context. This is to enable dirty checking. Again, often this is very useful. But it also can get very confusing when you actually don't want to persist changes to an entity. Or if you want to copy an entity.

This makes JPA hard to understand and hard to use correctly.

Spring Data JDBC tries to be a conceptually simpler alternative. We achieve that by not doing many things.

1. No Lazy Loading.
2. No write behind cache.
3. No cache at all. Of course you are free to add a cache on top of it.
4. No proxies.

Spring Data will save you data when you call `save` and only return once it is inserted or updated in the database. It will not save data you haven't invoked `save` for. All data will be fully loaded within the method call that loads the data.

Try the following examples. Also inspect the logs to see what kind of interaction is happening with the database.

```
Speaker martin = new Speaker();
martin.name = "Martin Fowler";
Speaker saved = speakers.save(martin);
assertThat(saved.id).isNotNull();


saved.id = null;

Speaker savedAgain = speakers.save(martin);

assertThat(savedAgain.id).isNotNull();

assertThat(speakers.findAll()).hasSize(2);
```

```
Speaker martin = new Speaker();
martin.name = "Martin Fowler";
Speaker saved = speakers.save(martin);
assertThat(saved.id).isNotNull();

Optional<Speaker> loaded = speakers.findById(saved.id);
Optional<Speaker> reloaded = speakers.findById(saved.id);

assertThat(loaded.get())
        .isEqualTo(reloaded.get())
        .isNotSameAs(reloaded.get());
```

While this makes things simpler it brings some extra challenges. We’ll see those later and how to tackle them.

Storing Relationships

Let's make `Speakers` more interesting. Each speaker should have a `Map` of `Website` references. Like so:

```
@EqualsAndHashCode
@ToString
@AllArgsConstructor
class Website {

    String link;
    String title;
}
```

Add the following line to the `Speaker` class.

```
Map<String, Website> websites = new HashMap<>();
```

Of course we need adapt our schema as well. Add the following to the `schema.sql` file.

```
CREATE TABLE WEBSITE
(
  SPEAKER     BIGINT,
  SPEAKER_KEY VARCHAR(20),
  LINK        VARCHAR(200),
  TITLE       VARCHAR(200),
  PRIMARY KEY (SPEAKER, SPEAKER_KEY)
);
```

Note that the table has two extra columns. One for a backreference to `SPEAKER` and one for the map key.

Create a test to play around with `Speaker` and `Website`.

For example.

```
Speaker martin = new Speaker();
martin.name = "Martin Fowler";
martin.websites.put("main", new
Website("https://martinfowler.com/", "Martin Fowler"));
martin.websites.put("wikipedia", new
Website("https://en.wikipedia.org/wiki/Martin_Fowler", "Martin
Fowler - Wikipedia"));

Speaker saved = speakers.save(martin);
assertThat(saved.id).isNotNull();

Speaker reloaded = speakers.findById(saved.id).get();

reloaded.websites.get("wikipedia").title = "Martin Fowler on
Wikipedia";

speakers.save(reloaded);

speakers.delete(reloaded);
```

Take a look at the SQL used. Do you have questions?

Aggregates in DDD and M:x Relationships Spring Data JDBC

In the last example you could observe that deleting a `Speaker` deleted all the `Website` instances. Currently even an update does that. In the update case there are improvements planned so that only those actually removed from a collection get deleted. But those that get removed from the collection still will get deleted and so are those referenced by a `Speaker` that gets deleted.

If we are only interested in a `Website` in the context of a `Speaker` this is ok. It is basically the `CascadeType.ALL` pluse `orphanRemoval = true` or in older versions `CascadeType.DELETE_ORPHAN` But what if we don't want this behavior?

For an example let's add a domain class `Talk` which has a list of `Speaker` instances. We don't want to delete a `Speaker` when a `Talk` is deleted or even just updated. That `Speaker` might be doing other talks after all.

At this point it is instructive to take a step back and take a look at what we are doing conceptually. A `Repository` is a concept from Domain Driven Design. It loads and saves "aggregates". "What is an aggregate?" you ask.

Let's hear Martin Fowler.

> A DDD aggregate is a cluster of domain objects that can be treated as a single unit. […] Aggregates are the basic element of transfer of data storage - you request to load or save whole aggregates.

Closely related to aggregates is the "aggregate root"

> An aggregate will have one of its component objects be the aggregate root. Any references from outside the aggregate should only go to the aggregate root. The root can thus ensure the integrity of the aggregate as a whole.

Spring Data JDBC saves, loads and deletes aggregates. This fits the approach of deleting everything. It also suggests that if we don't want to delete a `Speaker` when we delete a `Talk` that `Speaker` and `Talk` are different aggregates. This is actually true for all to many-to-many or many-to-one relationships. Because in all these cases the life cycle of the two sides of the relationship are independent of each other.

But all this still doesn't answer the question how to actually model these references between aggregates.

So lets ask David Masters

> It makes life much easier if you just keep a reference of the aggregate's ID rather than the actual aggregate itself.

Or Vaughn Vernon *Domain-Driven Design Distilled (Kindle Locations 1087-1088). Pearson Education. Kindle Edition.*

> Reference other Aggregates by identity only.

With this approach our original problem just disappears. Since in the domain model there is just an id, Spring Data JDBC doesn't even have the necessary knowledge to delete the referenced entity or aggregate.

So lets do that for the `Talk` class.

```
@ToString
@EqualsAndHashCode
class Talk {

    @Id
    Long id;
    String title;
    List<Long> speakerIds = new ArrayList<>();

    Talk(String title) {
        this.title = title;
    }

    void addSpeakers(Speaker... speakers) {

        for (Speaker speaker : speakers) {
            addSpeaker(speaker);
        }
    }

    private void addSpeaker(Speaker speaker) {

        assert speaker != null;
        assert speaker.id != null;

        speakerIds.add(speaker.id);
    }
}
```

```
interface TalkRepository extends CrudRepository<Talk, Long> { }
```

```
@RunWith(SpringRunner.class)
@DataJdbcTest
public class AggregateReferenceTests {

    @Autowired
    SpeakerRepository speakers;

    @Autowired
    TalkRepository talks;

    @Test
    public void speakerAndTalks() {

        Speaker steven = speaker("Steven Schwenke");
        Speaker tim = speaker("Tim Bourguignon");

        speakers.saveAll(asList(steven, tim));

        Talk teams = new Talk("Managing Distributed Teams");
        teams.addSpeakers(steven);

        Talk mentoring = new Talk("Mentoring Speed Dating");
        mentoring.addSpeakers(steven, tim);

        Talk unicorns = new Talk("Why Unicorn Developers don't
Grow on Trees?");
        unicorns.addSpeakers(tim);

        talks.saveAll(asList(teams, mentoring, unicorns));

        assertThat(speakers.count()).isEqualTo(2);
        assertThat(talks.count()).isEqualTo(3);

        talks.delete(teams);

        assertThat(speakers.count()).isEqualTo(2);
        assertThat(talks.count()).isEqualTo(2);


assertThat(talks.findById(mentoring.id).get().speakerIds).contains
ExactlyInAnyOrder(tim.id, steven.id);
```

```
    }

    private Speaker speaker(String name) {

        Speaker speaker = new Speaker();
        speaker.name = name;
        return speaker;
    }
}
```

```
CREATE TABLE TALK
(
  ID   BIGINT GENERATED BY DEFAULT AS IDENTITY (START WITH 1)
PRIMARY KEY,
  TITLE VARCHAR(200),
  SPEAKER_IDS BIGINT ARRAY
);
```

This approach uses a `BIGINT ARRAY` to store the references in the database. This has the benefit of getting loaded and saved in a single statement. But it also comes with drawbacks, like not all databases supporting it and those that do might not support foreign keys.

You can achieve the same with a traditional mapping table. For demoing that let’s introduce conferences which have a M:N relationship to talks:

```java
@ToString
@EqualsAndHashCode
class Conference {

    @Id
    Long id;
    String name;
    LocalDate startDate;

    Set<TalkReference> talks = new HashSet<>();

    Conference(String name, LocalDate startDate) {
        this.name = name;
        this.startDate = startDate;
    }

    void addTalks(Talk... talks) {

        for (Talk talk : talks) {
            this.addTalk(talk);
        }
    }

    private void addTalk(Talk talk) {

        assert talk != null;
        assert talk.id != null;

        talks.add(new TalkReference(talk.id));
    }
}
```

```java
@Value
@Table("CONFERENCE_TALK")
public class TalkReference {
    @Column("TALK")
    Long talkId;
}
```

```java
interface ConferenceRepository extends CrudRepository<Conference,
Long> { }
```

```
CREATE TABLE CONFERENCE
(
  ID   BIGINT GENERATED BY DEFAULT AS IDENTITY (START WITH 1),
  NAME VARCHAR(200),
  START_DATE DATE,
  CONSTRAINT PK_CONFERENCE PRIMARY KEY (ID)
);

CREATE TABLE CONFERENCE_TALK
(
  CONFERENCE BIGINT,
  TALK       BIGINT,
  PRIMARY KEY (CONFERENCE, TALK)
);

ALTER TABLE CONFERENCE_TALK
  ADD CONSTRAINT FK_CONFERENCE_TALK_CONFERENCE FOREIGN KEY
(CONFERENCE) REFERENCES CONFERENCE (ID);
ALTER TABLE CONFERENCE_TALK
  ADD CONSTRAINT FK_CONFERENCE_TALK_TALK FOREIGN KEY (TALK)
REFERENCES TALK (ID);
```

```
@Test
public void talksAndConferences() {

    Talk teams = new Talk("Managing Distributed Teams");
    Talk mentoring = new Talk("Mentoring Speed Dating");
    Talk unicorns = new Talk("Why Unicorn Developers don't Grow on
Trees?");

    talks.saveAll(asList(teams, mentoring, unicorns));

    Conference javaLand = new Conference("JavaLand",
LocalDate.parse("2020-03-17", DateTimeFormatter.ISO_DATE));
    javaLand.addTalks(teams, mentoring, unicorns);

    Conference javaForum = new Conference("Java Forum Nord",
LocalDate.parse("2019-09-24", DateTimeFormatter.ISO_DATE));
    javaForum.addTalks(teams, mentoring);

    conferences.saveAll(asList(javaLand, javaForum));
}
```

But Why?

Handling ids manually is cumbersome. Why would one want to do that?

Because it breaks the domain in modules. And this has all kinds of positiv effects.

- It is now really easy to understand what get loaded and saved in one go.
- The persistence question may now be answered for every aggregate separately. Since you are loading it from an repository anyway you might as well store it in a different database.
- You might rethink foreign keys between aggregates.
  - Using deferred constraints may help with integration tests, because you no longer need to create all referenced entities.
  - Dropping foreign keys completely between aggregates is a sensible way to eventual consistency.
- Eventually the boundaries between aggregates might turn into boundaries between microservices.

Congratulations

Understanding aggregates is the most important part of this workshop. Note that you may use this approach independent of Spring Data JDBC in order to achieve a modular domain model.

Events and Id Generation

So far ids where generated by id columns in the database. There are various reasons why this might be undesirable.

I princple there are two different scenarios:

1. You still want to create the id when the data gets stored in the database.
2. New entities already have an id that should be used.

For the first part let's use UUIDs for the primary key.

Let's create an entity and the matching table.

```
class UuidEntity {

    @Id
    UUID id;

    String name;
}
```

```
CREATE TABLE UUID_ENTITY
(
  ID   UUID PRIMARY KEY,
  NAME VARCHAR(200)
)
```

This doesn't work since we need to create Ids. We do that in a callback.

```
@Bean
BeforeConvertCallback<UuidEntity> uuidGenerator() {
    return new BeforeConvertCallback<UuidEntity>() {
        @Override
        public UuidEntity onBeforeConvert(UuidEntity aggregate) {
            aggregate.id = UUID.randomUUID();
            return aggregate;
        }
    };
}
```

Or written as a closure

```
@Bean
BeforeConvertCallback<UuidEntity> uuidGenerator() {
    return aggregate -> {
        aggregate.id = UUID.randomUUID();
        return aggregate;
    };
}
```

This is what is happening under the hood: Spring Data notices that the id is `null` and therefore determines that it is a new entity. The callback then sets the id and Spring Data continues to convert it and to store it with the generated id in the database.

This makes it obvious why we need a different strategy for the case where the id is set a priori. If the id is already set Spring Data considers it an existing entity by default and performs and update instead

Lets create another example how to handle this. There are two fundamentel different ways to achieve this. The first approach is to code the information if an entity is new or not into the entity by letting it implement `Persistable`.

```
@Table("UUID_ENTITY")
class UuidEntityPresetId implements Persistable {

    @Id
    UUID id;

    String name;

    @Transient
    boolean isNew;

    @PersistenceConstructor
    UuidEntityPresetId(UUID id, String name) {
        this.id = id;
        this.name = name;
        this.isNew = false;
    }

    UuidEntityPresetId(String name) {
        id = UUID.randomUUID();
        this.name = name;
        this.isNew = true;
    }

    @Override
    public Object getId() {
        return id;
    }

    @Override
    public boolean isNew() {
        return isNew;
    }
}
```

For this we use the field `isNew` which needs to get updated to `false` after the entity got saved for the first time. For this we need another callback

```
    @Bean
    AfterSaveCallback<UuidEntityPresetId> markAsSaved() {
        return aggregate -> {
            aggregate.isNew = false;
            return aggregate;
        };
    }
```

The flag shouldn’t get saved to the database which is why it is marked as `@Transient`.

For the other variant we first need to introduce custom methods

Custom Methods

Custom methods are methods in your repository for which you provide the implementation yourself. As an example we implement an `insert` method that does a directe insert without checking if the entity is new or not.

First we need an interface to hold the custom methods we want to add.

```
public interface CustomRepository {
    void insert(UuidEntityExternalyControlledId entity);
}
```

Our repository needs to extend the interface

```
public interface UuidEntityExternalyControlledIdRepository
        extends
        CrudRepository<UuidEntityExternalyControlledId, UUID>,
        CustomRepository {}
```

And we need to provide an implementation with the same name as the interface plus an added `Impl`. The implementation gets autowired just as a bean so we can inject stuff we need.

In this case we need `JdbcAggregateOperations` which offers an `insert` method which we can use.

```
public class CustomRepositoryImpl implements CustomRepository {

    @Autowired
    JdbcAggregateOperations operations;

    public void insert(UuidEntityExternalyControlledId entity){
        operations.insert(entity);
    };
}
```

@Query Annotation

So far we talked almost exclusively about CRUD operations. But an important part of working with a relational database is querying.

You can perform arbitrary queries using the `@Query` annotation. Let's take a look at a couple example based on talks, speakers and conferences used at the beginning.

You can query for scalar values.

```
@Query("select count(*) " +
    "from Conference_Talk ct " +
    "join Talk t " +
    "on ct.talk = t.id " +
    "where :speakerId in (unnest(t.speaker_ids))")
int countInstancesWith(Long speakerId);
```

You can query for aggregates.

```
@Query("Select * from conference where start_date > :threshold
order by start_date limit 1 offset 0")
Conference nextConferenceAfter(LocalDate threshold);
```

You can use various wrappers (`List`, `Optional` …)

```
    @Query("Select * from conference where start_date > :threshold
order by start_date")
    List<Conference> conferencesAfter(LocalDate threshold);
```

You can execute DML statements, e.g. updates. They just require an additional `@Modifying`.

```
@Modifying
@Query("update talk t " +
    "set title = " +
    "title || ' (with ' || (select s.name from speaker s where
s.id = t.speaker_ids[1])  || ')' ")
void addSpeakerToTitle();
```

For even more flexibility you can specify a `ResultSetExtractor` or `RowMapper` to be used. These are classes from *Spring JDBC*. Note that there is no *Data* in *Spring JDBC*. The `RowMapper` or `ResultSetExtractor` class get specified in the query annotation as using the `resultExtractorClass` or `rowMapperClass` attributes as in the following example.

```
    @Query(value = "select t.title, c.name " +
            "from talk t " +
            "join conference_talk ct on t.id = ct.talk " +
            "join conference c on c.id = ct.conference " +
            "order by t.title"
            , resultSetExtractorClass = TalkInfoExtractor.class
    )
    List<TalkInfo> talkInfos();
```

The example needs a `TalkInfo` class and the `TalkInfoExtractor` class.

```
class TalkInfo {

    private final String title;
    private final Set<String> conferenceTitles = new HashSet<>();


    TalkInfo(String title) {
        this.title = title;
    }

    void addConference(String name) {
        conferenceTitles.add(name);
    }
}
```

```
public class TalkInfoExtractor implements ResultSetExtractor {
    @Override
    public Object extractData(ResultSet resultSet) throws
SQLException, DataAccessException {

        String currentTalk = "";
        List<TalkInfo> result = new ArrayList<>();

        while (resultSet.next()) {
            String talkName = resultSet.getString(1);
            if (result.isEmpty() || !currentTalk.equals(talkName))
{
                result.add(new TalkInfo(talkName));
                currentTalk = talkName;
            }
            result.get(result.size()-
1).addConference(resultSet.getString(2));
        }

        return result;
    }
}
```

All statements are executed directly. If you don’t limit yourself to standard SQL the statements are not portable. There are plans to have queries externalized in property files and have separate statements per database but that is not there yet.

More Mapping

So far we used almost exclusively the default mapping behaviour of *Spring Data JDBC* but there are ways to customize the behaviour and also some kinds of mapping that we didn't mention yet.

General class requirements

Spring Data JDBC can use a default constructor for instantiating entities. It also accepts constructors which take some or all of the attributes as arguments. In this case the names of the constructor parameters must match the names of the attributes. If there is more than one constructor one needs to be marked as the one to use with `@PersistenceConstructor`.

Attributes that didn't get set by the constructor get set by * setter * "wither" * or direct field access

A "wither" is a setter for an immutable object. It uses the prefix `with` instead of `set` and since the attribute value can not be changed it instead returns a new instance with the attribute set to the argument value.

"Wither" can be easily produced by Lombok or written by hand. Kotlin data objects work as well.

Simple Types

Simple Types are those that can get stored in a single column. Examples are

- String
- primitive types and their boxed siblings
- BigInteger, BigDecimal
- Various Date related types
- Enum

Names

The column name for an attribute is the attributes name converted to *snake-case*. For example `dateOfBirth` becomes `date_of_birth`

You may customize the column name of an attribute using an `@Column` annotation.

```
@Column("dob")
```

NamingStrategy

If you want to change how columns and tables are named in general you can register your own `NamingStrategy`

```
@Bean
NamingStrategy namingStrategy() {
    return new NamingStrategy(){
        @Override
        public String getTableName(Class<?> type) {
            return "T_" + NamingStrategy.super.getTableName(type);
        }
    };
}
```

Collections & Arrays

We mentioned collections in the form of attributes of type `Map` shortly before we dug into aggregates.

You can use the following types of references to other entities

- `Set<ENTITY>`
- `List<ENTITY>`
- `Map<SIMPLE, ENTITY>`

Here `ENTITY` is the type of an entity while `SIMPLE` is a simple type.

The table for the referenced type needs an additional column named after the name of the referencing class. `List` and `Map` need an additional column to hold the index or key. This column again is named after the name of the referencing class with an additional `_KEY` suffix. The names of these additional columns may be customized with a `MappedCollection` annotation.

```
@MappedCollection(idColumn = "redner", keyColumn = "index")
Map<String, Website> websites = new HashMap<>();
```

Referenced entities in `List` or `Map` instances don’t need an id. The combination of back reference and key column form are used as a primary key instead.

A list or array of a simple type will get stored as an `ARRAY` if the database supports it.

Embeddable

If reference to an entity is annotated with `@Embedded` the attributes of the referenced type are included in the referencing table. The `onEmpty` attribute of the annotation determines how to deal with all `null` values. Should it result in a `null` embedded instance or an instance with all attributes set to `null`? The `prefix` attribute defines a prefix to be used for the columns of the embedded instance.

For example you might want to add this to the `Speaker` class.

```
@Embedded(onEmpty = Embedded.OnEmpty.USE_NULL)
ContactInfo contact;
```

```
class ContactInfo {
    String phone;
    String email;
}
```

Custom Conversions

You can register custom conversions if you want to control how values get written to the database or read from it. As an example we introduce a custom `Email` class.

```
@Value
class Email {
    String localPart;
    String domain;
}
```

We need to create two converters in order to write and read it to and from the database.

```
@WritingConverter
enum EmailToStringConverter implements Converter<Email, String> {

    INSTANCE;

    @Override
    public String convert(Email source) {
        if (source == null) {
            return null;
        }

        {
            return source.getLocalPart() + "@" +
source.getDomain();
        }
    }
}
```

```
@ReadingConverter
enum StringToEmailConverter implements Converter<String, Email> {

    INSTANCE;

    @Override
    public Email convert(String source) {
        if (source == null) {
            return null;
        }

        String[] split = source.split("@");

        if (split.length != 2) {
            throw new IllegalArgumentException("Can't parse %s
into a Email. Expecting exactly one @");
        }

        return new Email(split[0], split[1]);
    }
}
```

And we need to register these converters as `JdbcCustomConversions`. For this we let our configuration extend `AbstractJdbcConfiguration` and overwrite the `jdbcCuostomConversions()` bean method.

```
@SpringBootApplication
public class DemoApplication extends AbstractJdbcConfiguration {

    // .. other code that we might have here so far

    @Bean
    @Override
    public JdbcCustomConversions jdbcCustomConversions() {
        return new
JdbcCustomConversions(asList(EmailToStringConverter.INSTANCE,
StringToEmailConverter.INSTANCE));
    }

}
```

Now we can replace the email `String` in `ContactInfo` with a proper class

```
class ContactInfo {
    String phone;
    Email email;
}
```

Bidirectional Relationships

Bidirectional Relationships aren't directly supported.

For relations between aggregates appropriate query method should be added to the repositories.

Bidirectional relationships inside an aggregate shouldn't be necessary because any access should happen from the aggregate root. If it is still wanted it must be established using setters, constructors and event listeners.