YEAR 2011
Prepare: Md. Motaher Hossain
Email: md.motaherhossain04@gmail.com

Spring Notes

Java

—

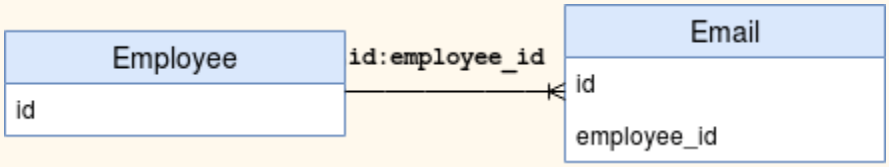
Notes

Database Mappings

One to many : for more:

What is owning side: https://stackoverflow.com/a/2749719/11411181

Example: https://www.baeldung.com/jpa-joincolumn-vs-mappedby



1. In a One-to-Many relationship, the owning side is the entity that has the reference (foreign key) to the other side, in the img, 'Email' is the owning side. Any CREATE, UPDATE, DELETE operations will be propagated from owning side to referencing side(Employee)
2. @JoinColumn annotation marks the foreign column name
3. 'mappedBy' attribute is used to define the referencing side (non-owning side) of the relationship. Meaning the other side is the Owning side.

What does it mean mappedBy ?

This means something like: "modifications on this side of the relation are already Mapped By the other side of the relation Email.employee, so no need to track it here separately in an extra table."

Are there any GOTCHAs, consequences?

Using mappedBy, If we only call employee.getEmails().add(email), the foreign key in Email will NOT be linked to the new email, because this is not the owning /tracked side of the relation!

To link the email to the employee, you need to explicitly call email.setEmployee(employee), because that is the owning side of the relation.

When using mappedBy, it is the responsibility of the developer to know what is the owning side, and update the correct side of the relation in order to trigger the persistence of the new relation in the database.

```
// referencing side
@Entity
public class Employee {
    @OneToMany(fetch = FetchType.LAZY, mappedBy = "employee")
    private List<Email> emails;
}

// owning side
@Entity
public class Email {
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "employee_id")
    private Employee employee;
}
```

Many to many : for more: https://stackoverflow.com/a/14111651/11411181
https://www.baeldung.com/jpa-many-to-many

1. Here, in the ‘baedlung’ there is mistake, all the documents points that the owning side exists in the many to many relation, the entity with ‘mappedBy’ is the referencing side, the other one is owning side.
2. Here , the ‘mappedBy’ contains the field name (there is no related column int the table) of the other side
3. The changes made in the owning side is only creates effects in the relationship
4. @JoinTable exists in the owning side, and contains necessary data to create the relation table.

```
// owning side
@Entity
class Student {
    @ManyToMany
    @JoinTable(
      name = "student_course",
      joinColumns = @JoinColumn(name = "student_id"),
      inverseJoinColumns = @JoinColumn(name = "course_id"))
    Set<Course> enrolledCourses;
}

// referencing side
@Entity
class Course {
    @ManyToMany(mappedBy = "likedCourses")
    Set<Student> likes;
}
```

Security user/pass

For more:
https://www.yawintutor.com/using-generated-security-password-spring-boot/

The log “Using generated security password: c581c5eb-8905-4461-9c70-ebb09e3a6950” appears at startup due to security feature is enabled in your spring boot application. The spring boot security dependency “spring-boot-starter-security” is added in the pom.xml file and default security configurations are not configured or customized in the application.
The default username is “user” and default password is generated password displayed in the console log at startup

To configure them as per necessity, add these in properties
spring.security.user.name = username
spring.security.user.password = password

Then the default pass won’t be printed in log

CommandLineRunner

It's an Functional Interface with a method named 'run' , Spring Boot will automatically call the run method of all beans implementing this interface after the application context has been loaded.

Security Config

If spring security added in maven, spring will create session, security filter, and a default user/password for login automatically.

To bypass that, or to point towards your own user class, create a class with @Configuration, @EnableWebSecurity,

Use auth0 maven dependency to process jwt generation process
Check the full implementation in 'springjwt' project

Actuator

if something goes wrong, we always need to analyze the logs and dig through the data flow of our application to check to see what's going on. So, the Spring Actuator provides easy access to those kinds of features. It provides many features, i.e. what beans are created, the mapping in the controller, the CPU usage, etc. Automatically gathering and auditing health and metrics can then be applied to your application.

For more: https://dzone.com/articles/top-10-spring-boot-interview-questions

Shutdown is an endpoint that allows the application to be gracefully shutdown. This feature is not enabled by default. You can enable this by using management.endpoint.shutdown.enabled=true in your application.properties file. But be careful about this if you are using this.

Disable web server in the Spring Boot application?

The major strong point in Spring is to provide flexibility to build your application loosely coupled. Spring provides features to disable the web server in a quick configuration. Yes, we can use the application.properties to configure the web application type, i.e. spring.main.web-application-type=none.

JDBC, JPA and everyone

JDBC: Java Database Connectivity is a low level standard for interaction with databases. It defines how a client may access a database.In JDBC you need to expose all dirty details needed for CRUD operations, such as table names, column names.

JPA: Java Persistence API provides a specification for persisting, reading, and managing data from your Java object to relational tables in the database.. JPA (which is using JDBC underneath) forms a bridge between object models (Java program) and relational models (database program). a programmer follows the 'JPA Provider' framework, which allows easy interaction with database instance

JPA Providers: JPA is an open source API, therefore various enterprise vendors such as Oracle, Redhat, Eclipse, etc. provide new products by adding the JPA persistence flavor in them. Some of these products include: Hibernate, Eclipselink, Toplink.

HIBERNATE: Hibernate is a Java-based ORM tool that provides a framework as a reference implementation of the Java Persistence API.

Spring Data JPA: Spring Data JPA is a part of the Spring Framework. It is not a JPA provider. It is a library/framework that adds an extra layer of abstraction on top of our JPA provider (like Hibernate, Eclipselink) to significantly reduce the amount of boilerplate code required to implement data access layers for various persistence stores.

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

This dependency includes JPA API, JPA Implementation, JDBC and other needed libraries. Since the default JPA implementation is Hibernate, this dependency is actually enough to bring it in as well.

IOC vs DI

In short, IoC is a much broader term that includes, but is not limited to, DI

The term Inversion of Control (IoC) originally meant any sort of programming style where an overall framework or run-time controlled the program flow

Before DI had a name, people started to refer to frameworks that manage Dependencies as Inversion of Control Containers, and soon, the meaning of IoC gradually drifted towards that particular meaning: Inversion of Control over Dependencies.

Inversion of Control (IoC) means that objects do not create other objects on which they rely to do their work. Instead, they get the objects that they need from an outside source (for example, an xml configuration file).

IoC is a design principle which recommends the inversion of different kinds of controls in object-oriented design to achieve loose coupling between application classes. In this case, control refers to any additional responsibilities a class has, other than its main responsibility, such as control over the flow of an application, or control over the dependent object creation and binding (Remember SRP - Single Responsibility Principle). If you want to do TDD (Test Driven Development), then you must use the IoC principle, without which TDD is not possible. Learn about IoC in detail in the next chapter.

Dependency Injection (DI) means that this is done without the object intervention, usually by a framework component that passes constructor parameters and set properties.

Dependency Injection (DI) is a design pattern which implements the IoC principle to invert the creation of dependent objects.

IoC is a generic term meaning that rather than having the application call the implementations provided by a library (also know as toolkit), a framework calls the implementations provided by the application.

DI is a form of IoC, where implementations are passed into an object through constructors/setters/service lookups, which the object will 'depend' on in order to behave correctly.

IoC without using DI, for example would be the Template pattern because the implementation can only be changed through sub-classing.

DI frameworks are designed to make use of DI and can define interfaces (or Annotations in Java) to make it easy to pass in the implementations.

IoC containers are DI frameworks that can work outside of the programming language. In some you can configure which implementations to use in metadata files (e.g. XML) which are less invasive. With some you can do IoC that would normally be impossible like inject an implementation at pointcuts.

The IoC container is a framework used to manage automatic dependency injection throughout the application, so that we as programmers do not need to put more time and effort into it

Bean Name

By default, configuration classes use a @Bean method's name as the name of the resulting bean.

```
@Bean public String beanA(){ return "bean A"; }

@Bean public String beanB(){ return "beanB"; }
```

```
@Autowired
public void setMyBean(String beanA){ this.myBean=beanA; }
```

Here, both beanA and beanB available, but setMyBean() will receive beanA.

Annotations

@SpringBootApplication

As per the Spring Boot doc, the @SpringBootApplication annotation is equivalent to using @Configuration, @EnableAutoConfiguration, and @ComponentScan with their default attributes.

@SpringBootApplication annotation provides a load of defaults (like the embedded servlet container), depending on the contents of your classpath and other things. It also turns on Spring MVC's @EnableWebMvc annotation, which activates web endpoints.

@ComponentScan

@ComponentScan enables Spring to scan for things like configurations, controllers, services, and other components we define.

Alternatively, Spring can also start scanning from the specified package, which we can define using basePackageClasses() or basePackages(). If no package is specified, then it considers the package of the class declaring the @ComponentScan annotation \

```
@Configuration

@ComponentScan(basePackages = {"com.baeldung.annotations.componentscanautoconfigure.healthcare",
```

"com.baeldung.annotations.componentscanautoconfigure.employee"},

basePackageClasses = Teacher.class)

@EnableAutoConfiguration

The @EnableAutoConfiguration annotation enables Spring Boot to auto-configure the application context. Therefore, it automatically creates and registers beans based on both the included jar files in the classpath and the beans defined by us.

For example, when we define the spring-boot-starter-web dependency in our classpath, Spring boot auto-configures Tomcat and Spring MVC. However, this auto-configuration has less precedence in case we define our own configurations.

We can use exclude to disable a list of classes that we do not want to be auto-configured:

@Configuration @EnableAutoConfiguration(excludeName = {"org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration"})

@EnableAutoConfiguration(exclude={JdbcTemplateAutoConfiguration.class})