Spring

Spring is a framework that is powerful and flexible that is focused on building web applications in java

Spring regularly introduces features based on the needs of the Dev community – an example is spring boot

At the core of spring there is spring core:-

1. IOC inversion of control
2. DI dependency injection
3. Beans
4. Context
5. Spring expression language
6. IOC Container

IOC is the principle which states that instead of the programmer taking charge of the flow of the program, the framework takes control of the program flow

Dependency injection is the pattern thru which IOC is achieved.

Dependency injection shifts responsibility of object creation from the application to the IOC container.

Beans – object in java manged by the spring ioc container

Context – this is a memory location where we add the objects we want spring to manage

IOC Container is responsible for managing the beans that are placed in the spring context. It assembles the dependencies between the objects.

Heres an example

A screen shot of a computer

Description automatically generated

Here, we are telling spring that we want these 2 beans to be created and placed in the context

A screen shot of a computer

Description automatically generated

We also want this above bean of the Compo class to be placed in the context – it has to be assembled with a bean of type Integer – this assembly is done by the IOC container

A screen shot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

In the above, ${xxx.yyy.xxx} is called spring expression language

NoUniqueBeanDefinition Exception

This exception occurs when we tell the context that we want a bean, but we are ambiguous – lets say there are 2 beans of type String , and we tell the context that we want a bean of type String, and spring is confused which bean it should return.

A black rectangular object with a black strip

Description automatically generated

A computer screen shot of a computer screen

Description automatically generated

Ways to resolve this :

1. Give beans a name or use the @Primary annotation

A black and white line

Description automatically generated with medium confidence

There is another way to create beans and that is the Component annotation

A screen shot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Lets say we have 2 beans of type Byte

And we try to do this in the component below

A screen shot of a computer

Description automatically generated

This wont work.

This is again nonuniquebeandefiniteion exception

A black rectangular object with a line

Description automatically generated

A black rectangular object with a white border

Description automatically generated

This will work – we use the @Qualifier to tell spring which bean we want exactly

Then we have stereotype annotations

@Component, @Service, @Reporisoty, @Controller, @RestController

These tell spring to create these beans automatically in the context

If you use @Bean, you can add many instances of the class to the context

If you use @Component, you can add only one instance of the class to the context

We can use the @PostConstruct annotation to configure our @Component beans

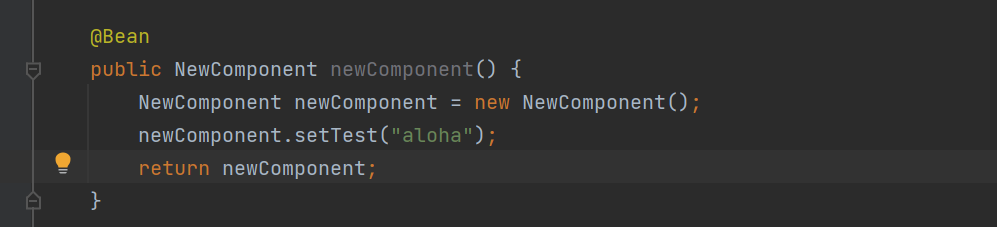
A screen shot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

Output on the console is what we have specified using the @PostConstruct annotation



Autowired annotation is best used as constructor injection

A screen shot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

In constructor injection for autowiring, you don’t need to use the @Autowired annotation.

Now we go on to bean scopes!

A screen shot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

The bean scope is prototype .. each time this bean is needed, a new instance will be created.

This scope is suitable for repository layer beans

Now we go on to eager and lazy

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

Overview of a web app

A screenshot of a web application

Description automatically generated

A diagram of a web apps

Description automatically generated

A diagram of a web application

Description automatically generated

A diagram of a web application

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Spring vs spring boot**

**Spring boot – easier dependency resolution**

When we use spring and we need for example hibernate

I need to find out all the dependencies I need and I also need to look out for version compatibility – we need to do all this ourselves

Then I will need to add it to my pom.xml

When we use spring boot and we create a web app using start.spring.io,

Then we just need to choose the dependencies we need, and spring boot will add starter dependencies, and will internally figure out things like version compatibility

**Spring boot – minimum configuration using autoconfiguration**

We need to add a lot of configurations in the .xml files

But in spring boot, we just need to add minimum configs in the application.properties files. Based on dependencies found in the classpath, spring boot auto configures the spring boot our application needs

**Spring boot – embedded server**

For spring, we need to download and install an external server like tomcat and then run the application on that external server

When we use spring boot, it has an embedded tomcat server

**Spring boot actuator – provides a pre defined list of endpoints for us to monitor our app**

**Spring boot starters**

**Autoconfiguration**

**Actuator**

**Embedded web server**

**@Springbootapplication**

* **Contains @EnableAutoConfiuguration**
* **Contains @ComponentScan**
* **Contains @Configuration**

**You can create beans in your main class**

**Example below**

**A screenshot of a computer

Description automatically generated**

A screenshot of a computer

Description automatically generated

Adding Lombok.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

A screen shot of a computer code

Description automatically generated

Validations

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

A screen shot of a computer

Description automatically generated

A computer screen shot of a black screen

Description automatically generated

Here we forgot the @Valid annotation

@PostMapping("/add")  
public void test3(@Valid @RequestBody PojoAnother pojoAnother) {  
 System.*out*.println("all good!");  
}

A screenshot of a computer

Description automatically generated

@PostMapping("/add")  
public void test3(@Valid @RequestBody PojoAnother pojoAnother, Errors errors) {  
  
 if (errors.hasErrors()) {  
 System.*out*.println("something is wrong");  
 return;  
 }  
  
 System.*out*.println("all good!");  
}

With the inclusion of Errors above, we can decide what to do in our code itself and a 400 bad request wont be thrown.

Spring security is something I need to study☹

@GetMapping("/purposelyError")  
public void doing() {  
 throw new RuntimeException();  
}

In the above, we purposely throw an error

This error will be caught by @RestControllerAdvice

@RestControllerAdvice  
public class AdviceExample {  
  
 @ExceptionHandler(Exception.class)  
 public String someStuffWentWrong() {  
 return "aiyo!";  
 }  
}

Now we go to spring boot and db operations.

In the pom, we need below

<dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
  
 <dependency>  
 <groupId>com.h2database</groupId>  
 <artifactId>h2</artifactId>  
 <scope>runtime</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 </dependency>  
</dependencies>

In application properties

spring.application.name=demo  
spring.h2.console.enabled=true  
spring.datasource.url=jdbc:h2:mem:testdb  
spring.datasource.driverClassName=org.h2.Driver  
spring.datasource.username=sa  
spring.datasource.password=password

then we need an entity – this will auto be created in the h2 db

package com.example.db.demo.entity;  
  
import jakarta.persistence.\*;  
import lombok.AllArgsConstructor;  
import lombok.Data;  
import lombok.NoArgsConstructor;  
import org.springframework.boot.autoconfigure.domain.EntityScan;  
  
@Entity  
@Data  
@AllArgsConstructor  
@NoArgsConstructor  
public class Student {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*AUTO*)  
 private long id;  
  
 @Column(name = "nishisherebro")  
 private String nishName;  
  
  
}

now, we use JPA

package com.example.db.demo.repositories;  
  
import com.example.db.demo.entity.Student;  
import org.springframework.data.jpa.repository.JpaRepository;  
import org.springframework.stereotype.Repository;  
  
@Repository  
public interface StudentRepo extends JpaRepository<Student,Long> {  
}

when the app startus up, we want to add some data to the student table

package com.example.db.demo.runner;  
  
import com.example.db.demo.entity.Student;  
import com.example.db.demo.repositories.StudentRepo;  
import org.springframework.boot.ApplicationArguments;  
import org.springframework.boot.ApplicationRunner;  
import org.springframework.stereotype.Component;  
  
@Component  
public class AppRunner implements ApplicationRunner {  
  
 private final StudentRepo studentRepo;  
  
 public AppRunner(StudentRepo studentRepo) {  
 this.studentRepo = studentRepo;  
 }  
  
  
 @Override  
 public void run(ApplicationArguments args) throws Exception {  
 Student student = new Student();  
 student.setNishName("brodafone!");  
 this.studentRepo.save(student);  
 }  
}

so now the question what is JPA

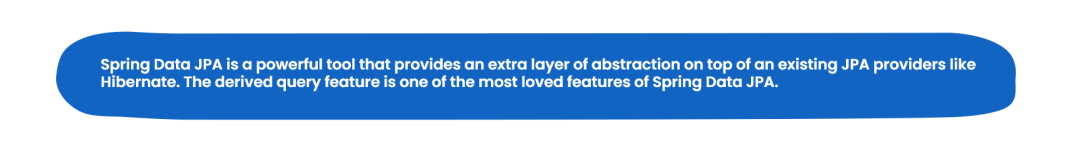
In the jpa repo class, we can query for data by using derived query methods

For example

package com.example.db.demo.repositories;  
  
import com.example.db.demo.entity.Student;  
import org.springframework.data.jpa.repository.JpaRepository;  
import org.springframework.stereotype.Repository;  
  
@Repository  
public interface StudentRepo extends JpaRepository<Student,Long> {  
  
 Student findByNishName(String name);  
}

package com.example.db.demo.runner;  
  
import com.example.db.demo.entity.Student;  
import com.example.db.demo.repositories.StudentRepo;  
import org.springframework.boot.ApplicationArguments;  
import org.springframework.boot.ApplicationRunner;  
import org.springframework.stereotype.Component;  
  
@Component  
public class AppRunner implements ApplicationRunner {  
  
 private final StudentRepo studentRepo;  
  
 public AppRunner(StudentRepo studentRepo) {  
 this.studentRepo = studentRepo;  
 }  
  
  
 @Override  
 public void run(ApplicationArguments args) throws Exception {  
 Student student = new Student();  
 student.setNishName("babumushai bandoonbaaz!");  
 this.studentRepo.save(student);  
  
 Student student2 = new Student();  
 student2.setNishName("ratlam");  
 this.studentRepo.save(student2);  
  
 Student student3 = new Student();  
 student3.setNishName("bruv");  
 this.studentRepo.save(student3);  
  
 Thread.*sleep*(3000);  
  
 Student byNishName = this.studentRepo.findByNishName("bruv");  
 System.*out*.println(byNishName.getId() + " this is the id! ");  
 }  
}

from the above u can see that I did not need to implement the findByNishName method



Continue with auditing support –

JPA stands for java persistence API and it is a specification.

Since it is a specification, under the hood, it needs an implementation. In spring boot, the out of the box implementation is hibernate.

Spring boot offers a jpa starter called spring data jpa starter that makes it even easier to work with jpa

We need to add in this dependency and provide a few configs in the app.properties for the database that we want to connect to, and we’re able to talk to the database.

For spring data jpa, we need an entity and a repository. The entity is a pojo class that maps to the corresponding table in the db and the repository extends a class called JpaRepository that comes with out of the box methods like save, findAll() and also has a special feature called derived Query methods that allows us to retrieve data via the use of the column names such as findByName if the column is called name

U can do auditing in jpa

A screen shot of a computer

Description automatically generated

Tells spring that if u want auditor refs then the app can refer to the bean called

auditAwareBro

A screenshot of a computer

Description automatically generated

This is the listener above

Our class must now extend this listener

package com.example.db.demo.entity;  
  
import jakarta.persistence.\*;  
import lombok.AllArgsConstructor;  
import lombok.Data;  
import lombok.NoArgsConstructor;  
import org.springframework.boot.autoconfigure.domain.EntityScan;  
  
@Entity  
@Data  
@AllArgsConstructor  
@NoArgsConstructor  
public class Student extends BaseEntity{  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*AUTO*)  
 private long id;  
  
 @Column(name = "nishisherebro")  
 private String nishName;  
  
  
}

So now, we want to implement a functionality where a class has many students

1 class – many students.

1 student – 1 class.

We create a class dto cos we don’t want to return the entity

package com.example.db.demo.awrapper;  
  
import com.example.db.demo.entity.ClassNew;  
import com.example.db.demo.entity.StudentNew;  
import com.example.db.demo.entity.StudentNewNew;  
import lombok.AllArgsConstructor;  
import lombok.Data;  
import lombok.NoArgsConstructor;  
  
import java.util.List;  
  
@Data  
@AllArgsConstructor  
@NoArgsConstructor  
public class ClassDto {  
  
 private String className;  
  
 private List<StudentNewNew> students;  
  
  
}

Then we create a student rest controller

package com.example.db.demo.controllers;  
  
import com.example.db.demo.awrapper.ClassDto;  
import com.example.db.demo.entity.ClassNew;  
import com.example.db.demo.entity.StudentNew;  
import com.example.db.demo.entity.StudentNewNew;  
import com.example.db.demo.repositories.NewClassRepo;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
import java.util.ArrayList;  
import java.util.List;  
import java.util.Optional;  
  
@RestController  
@RequestMapping("/test")  
public class StudentController {  
  
 @Autowired  
 private NewClassRepo newClassRepo;  
  
 @GetMapping("/getClass")  
 public ClassDto getClassName() {  
  
 Optional<ClassNew> byId = newClassRepo.findById(1L);  
  
 ClassNew classNew = byId.get();  
  
 ClassDto classDto = new ClassDto();  
 classDto.setClassName(classNew.getClassName());  
  
 List<StudentNew> students = classNew.getStudents();  
 System.*out*.println("printing student size");  
 System.*out*.println(students.size());  
  
 List<StudentNewNew> list = new ArrayList<>();  
  
 for (StudentNew s : students) {  
 StudentNewNew a = new StudentNewNew();  
 a.setName(s.getName());  
 a.setId(s.getId());  
 list.add(a);  
 }  
  
 classDto.setStudents(list);  
  
 return classDto;  
 }  
}

Then we create a class which has one to many

package com.example.db.demo.entity;  
  
import jakarta.persistence.\*;  
import lombok.AllArgsConstructor;  
import lombok.Data;  
import lombok.NoArgsConstructor;  
  
import java.util.List;  
  
@Entity  
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
public class ClassNew {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*AUTO*)  
 private long classId;  
  
 @Column(name="className")  
 private String className;  
  
 @OneToMany(mappedBy="classNew", cascade = CascadeType.*ALL*, fetch = FetchType.*LAZY*)  
 private List<StudentNew> students;  
  
  
}

then we create a student that has many to one

package com.example.db.demo.entity;  
  
import jakarta.persistence.\*;  
import lombok.AllArgsConstructor;  
import lombok.Data;  
import lombok.NoArgsConstructor;  
  
@Entity  
@Data  
@AllArgsConstructor  
@NoArgsConstructor  
public class StudentNew {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*AUTO*)  
 private long id;  
  
 @Column(name = "name")  
 private String name;  
  
 @ManyToOne  
 @JoinColumn(name = "class")  
 private ClassNew classNew;  
}

Now we create a runner that will insert data into the db

package com.example.db.demo.runner;  
  
import com.example.db.demo.entity.ClassNew;  
import com.example.db.demo.entity.StudentNew;  
import com.example.db.demo.repositories.NewClassRepo;  
import com.example.db.demo.repositories.NewStudentRepo;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.ApplicationArguments;  
import org.springframework.boot.ApplicationRunner;  
import org.springframework.stereotype.Component;  
  
@Component  
public class RunnerNewClass implements ApplicationRunner {  
  
 @Autowired  
 private NewClassRepo newClassRepo;  
  
 @Autowired  
 private NewStudentRepo newStudentRepo;  
  
 @Override  
 public void run(ApplicationArguments args) throws Exception {  
  
 ClassNew classNew = new ClassNew();  
 classNew.setClassName("maths");  
 ClassNew save = newClassRepo.save(classNew);  
  
 StudentNew studentNew = new StudentNew();  
 studentNew.setName("nish");  
 studentNew.setClassNew(save);  
  
 StudentNew studentNew2 = new StudentNew();  
 studentNew2.setName("bhatt");  
 studentNew2.setClassNew(save);  
  
 StudentNew studentNew3 = new StudentNew();  
 studentNew3.setName("blingz");  
 studentNew3.setClassNew(save);  
  
 newStudentRepo.save(studentNew);  
 newStudentRepo.save(studentNew2);  
 newStudentRepo.save(studentNew3);  
  
  
 }  
}

first we save the class and we get back the saved object

then we create some students and we add this saved class object to the students

then we save these students to the db

now, the class knows about these students because of the @OneToMany annotation on the class.