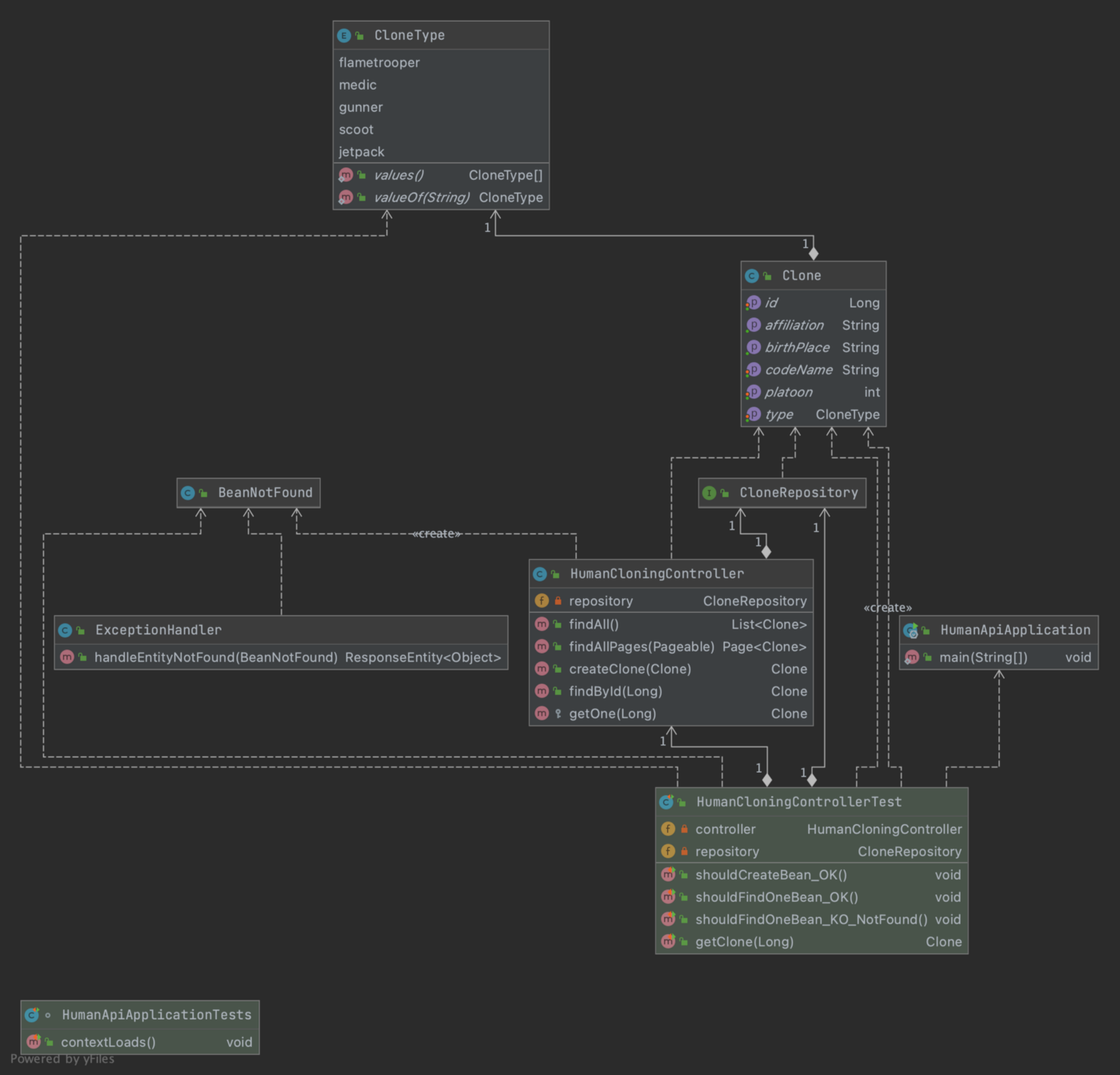
**What you will need**

To begin, you have to choices to initialize your project :  
- [Spring initializr](https://start.spring.io/)  
- from your IDE  
As we will use [Spring Boot](https://medium.com/javarevisited/10-advanced-spring-boot-courses-for-experienced-java-developers-5e57606816bd?source=collection_home---4------0-----------------------), you will need to add the following dependency to your project in order to it to work.

* **Spring Web** : Uses Apache Tomcat as the default embedded container.
* **Spring Data JPA**: Java Persistence API using [Spring Data](https://www.java67.com/2021/01/spring-data-jpa-interview-questions-answers-java.html) and [Hibernate](https://medium.com/javarevisited/top-5-hibernate-online-training-courses-for-beginners-and-advance-java-programmers-469460596b2b).
* **H2 Database**: Embedded in memory database.

**What il will look like**

[[](https://www.java67.com/2018/06/5-best-courses-to-learn-spring-boot-in.html)](https://www.java67.com/2018/06/5-best-courses-to-learn-spring-boot-in.html)

UML representation

**3, 2, 1 … Start**

**The entity**

First we need to declare the object we will use in the rest of this exemple, the definition of the Clone Troopers : *Clone.*

package com.erwan.human.domaine;

import com.erwan.human.reference.CloneType;

import com.sun.istack.NotNull;

import com.sun.istack.Nullable;

import lombok.Getter;

import lombok.Setter;

import javax.persistence.\*;

@Entity

@Getter

@Setter

public class Clone {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

@Column(name = "id")

private Long id;

private final String birthPlace = "Kamino";

@NotNull

private String codeName;

@NotNull

@Enumerated

private CloneType type;

@Nullable

private int platoon;

private final String affiliation = "Galactic Republic";

}

In order to persist our object in the [database](https://medium.com/hackernoon/top-5-sql-and-database-courses-to-learn-online-48424533ac61), it will need an Id, that will be auto-generated when we will save it.

## The Repository

To manipulate our objects in the database we need a repository

package com.erwan.human.dao;

import com.erwan.human.domaine.Clone;

import org.springframework.data.jpa.repository.JpaRepository;

public interface CloneRepository extends JpaRepository<Clone, Long>{

}

Our CloneRepository extends JpaRepository who extend PagingAndSortingRepository that permit to use pagination request, who extends CrudRepository (CRUD is an acronym to create, read, update, delete).

With ou repository, we can use method like save, findById, findAll (with or without pagination).

## The controller

Now that we have our object, and the repository to save and retrieve it, we need the entry point to our actions : create, find, delete …

package com.erwan.human.controller;

import com.erwan.human.dao.CloneRepository;

import com.erwan.human.domaine.Clone;

import com.erwan.human.exceptions.BeanNotFound;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

import org.springframework.data.domain.Sort;

import org.springframework.data.web.PageableDefault;

import org.springframework.data.web.SortDefault;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

import java.util.Optional;

@RestController

@RequestMapping("/kamino")

public class HumanCloningController {

@Autowired

private CloneRepository repository;

@GetMapping("/")

public List<Clone> findAll() {

return repository.findAll();

}

@GetMapping("/pages")

public Page<Clone> findAllPages(@PageableDefault(page = 0, size = 20)

@SortDefault.SortDefaults({

@SortDefault(sort = "id", direction = Sort.Direction.ASC)

}) Pageable pageable) {

return repository.findAll(pageable);

}

@PostMapping()

public Clone createClone(@RequestBody Clone clone){

return repository.save(clone);

}

@GetMapping("/{id}")

public Clone findById(@PathVariable("id") Long id) throws BeanNotFound {

return getOne(id);

}

@DeleteMapping("/{id}")

public void delete(@PathVariable("id") Long id) throws BeanNotFound {

Clone clone = getOne(id);

repository.delete(clone);

}

protected Clone getOne(Long id) throws BeanNotFound {

Optional<Clone> clone = repository.findById(id);

if(!clone.isPresent()){

throw new BeanNotFound("Can't find clone with id : " + id);

}

return clone.get();

}

}

As you can read, even with no line of code in ou repository class, we can freely use the method inside.

So let's see what we have here :

* [**@RestController**:](https://javarevisited.blogspot.com/2017/08/difference-between-restcontroller-and-controller-annotations-spring-mvc-rest.html#ixzz6OYNB9oii) is to indicate that our data will be returned in the response’s body and not in a template
* We have **@GetMapping** **@PostMapping** and **@DeleteMapping** that indicate the Http method that is linked to our method.
* We have a **BeanNotFound** throw by our methods (we will see later how we use it).

**Handling the exceptions**

What is a proper way to handle exceptions, a try/catch block in every method, or to intercept them in a specific way and then use them in a uniform way?

The second method is cleaner, as we can have all our exceptions in the same place, and treat them the same way.

package com.erwan.human.exceptions;

import org.springframework.core.Ordered;

import org.springframework.core.annotation.Order;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.ControllerAdvice;

import static org.springframework.http.HttpStatus.NOT\_FOUND;

@Order(Ordered.HIGHEST\_PRECEDENCE)

@ControllerAdvice

public class ExceptionHandler {

@org.springframework.web.bind.annotation.ExceptionHandler(BeanNotFound.class)

public ResponseEntity<Object> handleEntityNotFound(

BeanNotFound ex) {

return new ResponseEntity<>(ex.getMessage(), NOT\_FOUND);

}

}

As you can see, our **BeanNotFound** is intercepted by the **ExceptionHandler** and then we return our exception message as a [404 error](https://en.wikipedia.org/wiki/HTTP_404).

# Testing our API

To test our API we need to first launch our application.  
Then we can try to connect to the entry point that we created in our controller using [postman](https://medium.com/javarevisited/7-best-courses-to-learn-postman-tool-for-web-service-and-api-testing-f225c138fa5a?source=---------13------------------) or any other tool that permits you to make HTTP calls.

## Creating a clone

As described in the controller, we need to use the HTTP POST method a pass in it’s body the information of our clone.

HTTP POST <http://localhost:8080/kamino/>{  
 "codeName": "CT-7567 REX",  
 "type": "gunner",  
 "platoon": 501  
}

That will return us this response

{  
 "id": 1,  
 "birthPlace": "Kamino",  
 "codeName": "CT-7567 REX",  
 "type": "gunner",  
 "platoon": 501,  
 "affiliation": "Galactic Republic"  
}

The Id is auto-generated, so if we create another clone, the value will be 2, then 3, and will keep incrementing.

## Finding all the created clones

To find the created clones, we will call the GET “/” endpoint that will return us a list of all our created clones

HTTP GET http://localhost:8080/kamino/

that will gave us a response like this one :

[  
 {  
 "id": 1,  
 "birthPlace": "Kamino",  
 "codeName": "CT-7567 REX",  
 "type": "gunner",  
 "platoon": 501,  
 "affiliation": "Galactic Republic"  
 },  
 {  
 "id": 2,  
 "birthPlace": "Kamino",  
 "codeName": "CC-3636 WOLFFE",  
 "type": "gunner",  
 "platoon": 501,  
 "affiliation": "Galactic Republic"  
 }  
]

## Deleting a clone

To delete a clone in the database, we will call the delete endpoint

HTTP DELETE http://localhost:8080/kamino/1

passing 1 as a parameter, we will delete the clone that has the id : 1. So if we call again the find method we will only have one remaining clone.

{  
 "id": 2,  
 "birthPlace": "Kamino",  
 "codeName": "CC-3636 WOLFFE",  
 "type": "gunner",  
 "platoon": 501,  
 "affiliation": "Galactic Republic"  
}