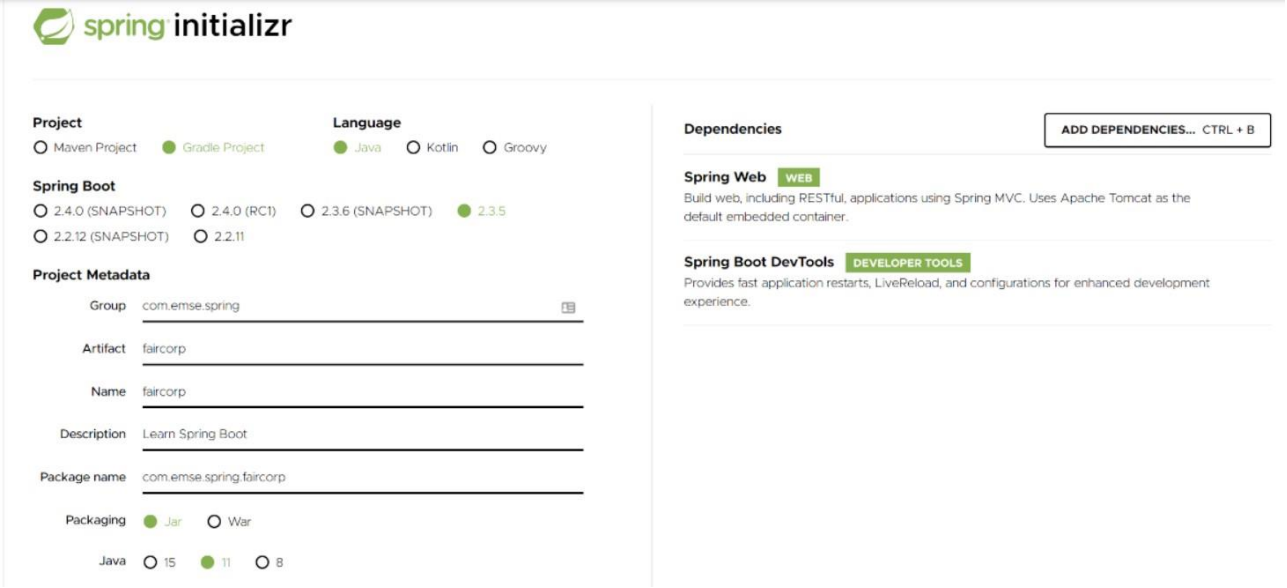
What’s Spring ?

- Abstraction of the complexity
- Dealing with ideas rather than events.
- Provides utility classes
- Spring provides useful classes
- Helps to focus on the business code and not on the technical plumbing
- Plumbing is a term used to describe the technology and connections between systems in a cloud computing model.
- The term is an analogy to the plumbing of water systems. Just as water is carried from reservoirs to homes and usage centers,
- the data from data centers is delivered to the end stations via its plumbing.
- So, Spring gives the opportunity to focus on business logic and not on the technical plumbing. So, let's say one class can use the method of another class with simple annotations, helping us not to worry about how that particular annotation is working.

- Spring is an entire ecosystem.
- The simplest definition of an ecosystem is that it is a community or group of living organisms that live in and interact with each other in a specific environment.
- So let's say that Spring let's the classes live in the framework and interacct with each other with easy.
- Spring Core provides a framework to simplify project technical aspects
- The above point is for me to accept, because I have struggled to get the entire scenario in my head.

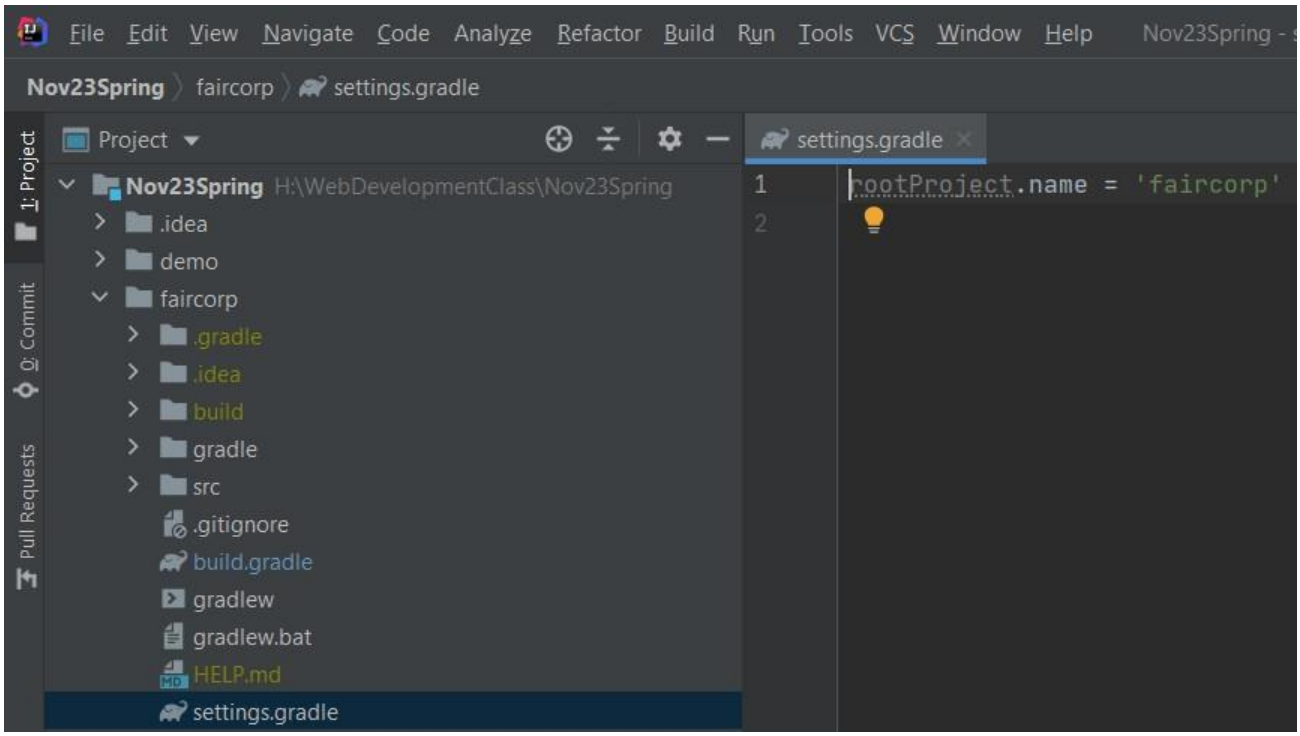
Generate

- To start a new project, you can (you should) use the official generator https://start.spring.io/



- the details of the theory can be found in here https://dev-mind.fr/training/spring/spring-intro.html . Just the key points are mentioned here
- So, basically it all starts from downloading those files from the website, unzip it in a folder. Consider those set of files as your project. Yeah! The project name can be given in the site itself and we can move on.
- Moving on to do the gradle settings.

- Gradle is basically our build tool. The way we do pip install someLibrary in python and then import thatLibrary in the python project, exactly the same work is handled by gradle and maven. Maven handles it in .xml file and gradle in build.gradle file
- settings.gradle



- File : > build.gradle contains informations used by Gradle to build app

- The above Adds the Spring Boot plugin to be able to manage your app with Gradle.
- A virtual environment is a Python environment such that the Python interpreter, libraries and scripts installed into it are isolated from those installed in other virtual environments,
- and (by default) any libraries installed in a “system” Python, i.e., one which is installed as part of your operating system.
- Similarly, we have Spring dependency management plugin to do the exact same thing

```
id 'io.spring.dependency-management' version '1.0.10.RELEASE
```

- the above Adds Spring dependency management plugin to use the compatible dependencies with the Spring Boot version
- Now check the below,

```
id 'java'
```

- Adds the Java plugin to help Gradle to manage our app lifecyle
- Probably this is necessary because with the growing popularity of Kotlin, this is can also be used. Let's check the internet. Apparently, it seems that Kotlin can be used for springBoot:

```
group = 'com.emse.spring'
version =  '0.0.1-SNAPSHOT'
sourceCompatibility =  '14'
```

- the above shows Project id and versions

```
repositories {
        mavenCentral()
}
```

- The above would Tell Gradle where it will find all libraries
- We would later find out that MavenCentral is not the only place, others are there as well.

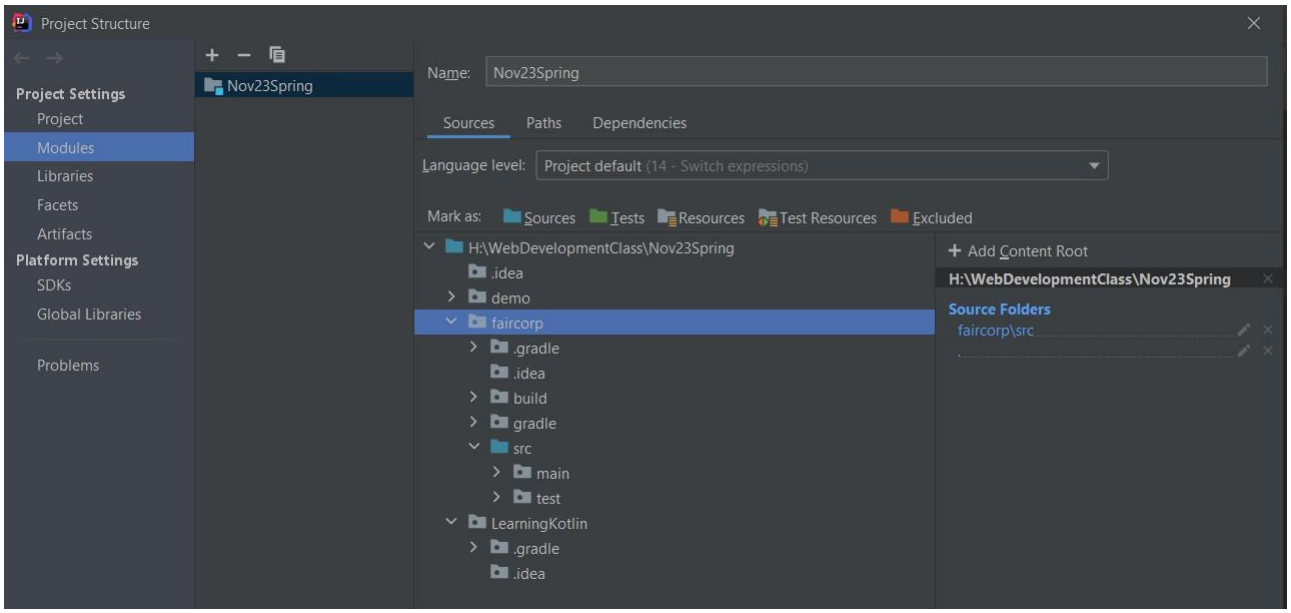
```
dependencies {
        implementation 'org.springframework.boot:spring-boot-starter-web'
        developmentOnly 'org.springframework.boot:spring-boot-devtools'
        testImplementation('org.springframework.boot:spring-boot-starter-
test') {
                exclude group: 'org.junit.vintage', module:  'junit-vintage-
engine'
        }
```

- spring-boot-starter-test : Starter for testing Spring Boot applications with libraries including JUnit, Hamcrest and Mockito
- here we are just choosing JUNit

- we have learnt before that in a good application, all files are tested and verified by unit tests
- spring-boot-devtools : devtools is a set of tools that can make the application development experience a little more pleasant (live reload or automatic restart).
- Okay, let's hope that spring-boot-devtools gives what it promises and we'd get an experience similar to LiveServerByRitwicKDey for expressJS
- spring-boot-starter-web : Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container
- about the above statement, I understand that it comes with an integrated TOMCAT and since this TOMCAT is intgrated so we'll not be able to do activities like LiveServerByRitwicKDey for expressJS.
- Now, again, the above question arrises that what is the use of developmentOnly 'org.springframework.boot:spring-boot-devtools' then.
- Moving on spring-boot-starter-web : Starter for building web, including RESTful. This is understandable.
- what is the meaning of applications using Spring MVC is my question. Let's find out. Came across this link, which seems to be useful https://www.javatpoint.com/spring-mvc-tutorial
- question now is that what is the difference between SpringMVC and SpringBoot. From the above link, it seems that SpringMVC and SpringBoot is absolutely same(based on my current knowledge status)
- Basically, I have got some idea by exploring the web, but more ideas would follow later.
- Now, for right now, the below class is my entry point to spring. So the document states that "This is your app entry point". So based on my understanding, we are making a spring boot app which can be accessed by this class.
- In order to mark it as the entry point to our SpringBootApp, we choose to annotate this by using @SpringBootApplication. So, here's the class as below

```
package com.emse.spring.faircorp;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MyWebApplication{

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

}
```

- Now, we'd attempt to create a test file of our FaircorpApplication. So we would create the test file for our entryClass which is named here MyWebApplication. So this is the className that I have chosen as the entry point to our Faircorp application. We would now try to test this entry point class.
- A problem came up. This problem came up maybe because I tried to initiate the work from the basics again.
- The problem was that I was not able to create a class with the create class option from the package.
- https://stackoverflow.com/questions/38254612/intelli-j-cannot-create-class-file . The last comment from the link was an easy read. I tried to explore something and this idea popped up.
- from the top right corner, I chose the ProjectStructure and got the box as below

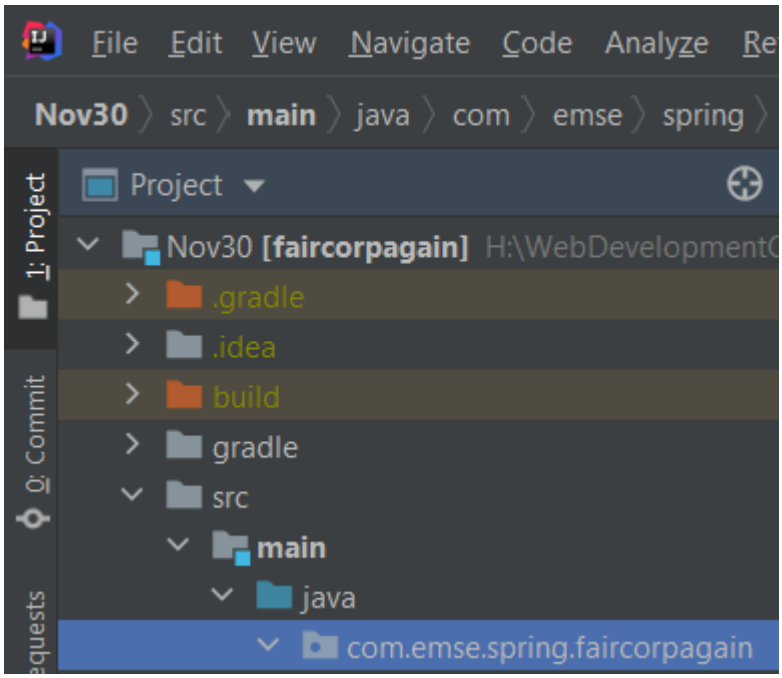


- from the dialogue box, I chose the option modules, went to sources option and made the src of my project as the source folder. I believe the problem is fixed now. I will try to check now.
- I dont know, if the problem is resolved or not. Let me try to check more.
- since, I couldn't understand what I was doing, so decided to create a new project. So this time, I didn't go to the website but instead decided to do it from the intellij. Giving this some time to rebuild. Since the entire idea of this project is to copy and paste, so let's hope that this is not gonna be a problem.
- so, as of right now, the below is my build.gradle situation. To be noted, the sourceCompatibility was 11, I made it 14 and did a gradlew build in the terminal. let's find out what's the problem in store for me, next.

```
plugins {
    id 'org.springframework.boot' version '2.4.0'
    id 'io.spring.dependency-management' version '1.0.10.RELEASE'
    id 'java'
}

group = 'com.emse.spring'
version =  '0.0.1-SNAPSHOT'
sourceCompatibility =  '14'

repositories {
    mavenCentral()
}

dependencies {
```

```
    implementation 'org.springframework.boot:spring-boot-starter-web'
    developmentOnly 'org.springframework.boot:spring-boot-devtools'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

test {
    useJUnitPlatform()
}
```

- at-least right now, I have the right project structure with the right package name which has been separated. It also gives me the option to create class files. This particular thing needs to be checked after the application is closed. It would be checked later, with the sole purpose of making it a little more complicated later.
- anyway, here's the look



- Now, again, we" try to create that entryPoint for the app named faircorpagain, based on my understanding. Let's find out.
- So, while creating the entryPoint, we imported a couple of class. The first option came in for simply copying the lines like below

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

- we imported this library(the term library was just used based on my python concept)

```
import org.springframework.boot.SpringApplication;
```

- again we imported below to use @SpringBootApplication

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

- the final code is as below

```
package com.emse.spring.faircorpagain;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

}
```

- again, we tried to create a testing class named FaircorpApplicationTests
- in this we use the annotation @RunWith(SpringRunner.class) to use a Runner to use when we want to test a Spring class..

predefined classes, not to be changed

- FaircorpApplication.java was already there in the package. So the entryPoint to the springBoot Application comes by default. We dont have to re-create that.
- Similarly FaircorpagainApplicationTests.java comes by default. So I feel that there needs not be any over-analysing with these two.
- Finally, we create the springBoot Application skeleton from either the website or from the intellijPremium, by choosing the right settings. After the project is build, we dont try to change anything. It comes with the right setup of gradle, entryPoint of the app, and the testingClass. Again, the main purpose of SpringBoot is not to concentrate on the plumbing but to take

care of the business logic and uptil now we have not even started with the business logic.

Summarisation of what have been understood by now.

- we create the springBoot Application skeleton from either the website or from the intellijPremium, by choosing the right settings. After the project is build, we dont try to change anything. It comes with the right setup of gradle, entryPoint of the app, and the testingClass. Again, the main purpose of SpringBoot is not to concentrate on the plumbing but to take care of the business logic and uptil now we have not even started with the business logic.
- when we run the testingClass in the beginning, it is supposed to go on smooth because we just tested something that has been given to us by default.
- the below command seems to get stopped at 75 or 80% and if we go to localhost:8080, the webpage is gonna come up which doesnt have anything as of yet.

```
gradlew --continuous bootRun
```

- Again, the core technical reasons could be read on https://dev-mind.fr/training/spring/spring-intro.html . this is the first page of springBoot and I have not yet progressed. MovingOn now.
- Now, based on the documentation, we are instructed to run the startingPoint which comes as default. Once attempted, we get a message saying that TOMCAT is already running on port 8080.
- Now, this TOMCAT is an integrated computer(in my terms) within springBoot which has many ports like a regular computer but it accepts&serves one request at any given point of time; because comeon 1. it's integrated as of right now. I dont know how to run an individual tomcat which is not intgrated within springBoot. So, it is like an integrated graphicsCard which comes with most of the motherboards. This integrated system doesn't have the capability to update the webpage just by getting the code updated. This kind of codeUpdateWebpageUpdate reference is being made based on the LiveServerByRitwickDey for expressJS. Anyway, even if we open a differrent port for this intgrated TOMCAT by saying the below, it wont work. We would just get an error message saying "PORT 8081 is already is use"

```
server.port=8081
```

- the above, if being written in the file named application.properties would give an error saying that port8081 is already in use.
- so basically, we have to free up port8080 of this integrated TOMCAT to proceed.
- Now, according to the document, we did a test and later tried to start the integrated TOMCAT by saying gradlew --continuous bootRun
- now, when we tried to start the entryPoint by just running it we get 8080AlreadyInUse.
- so that is why ctrl+C was used to stop the TOMCAT and later when we tried to initialise the entryPointClass, no error came up in regards to portInUse.
- Anyway, if we wanna see something on that server then the following HTML is copied in the resources/static/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
Hello World
</body>
</html>
```

- Again, more theory is available in the document, but we could come back to that theory later. It seems that we can finally move to the next document.

https://dev-mind.fr/training/spring/spring-ioc.html

dependency injection

- Dependency injection is a programming technique that makes a class independent of its dependencies.

- That enables you to replace dependencies without changing the class that uses them.
- It also reduces the risk that you have to change a class just because one of its dependencies changed.
- we got this wonderful explanation from https://stackify.com/dependency-injection/
- When we want to create an object we write for example

```
package com.emse.spring.faircorpagain;

public class NameService {
    public String getName()  {
        return "Guillaume";
    }
}
```

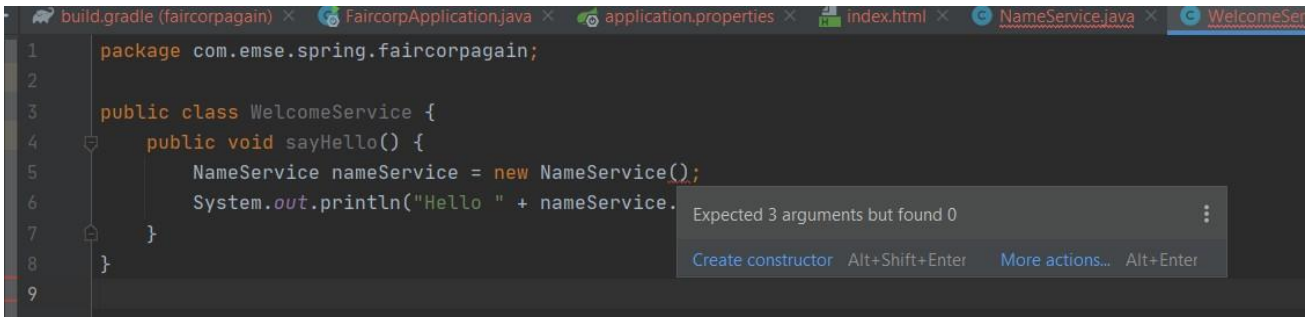
- And to use this object elsewhere we have to create a new instance with a new instruction

```
package com.emse.spring.faircorpagain;

public class WelcomeService {
    public void sayHello() {
        NameService nameService = new NameService();
        System.out.println("Hello " +  nameService.getName());
    }
}
```

- We have a strong coupling between these classes WelcomeService and NameService. If we want to change NameService we have a good chance of having to update WelcomeService
- Now, let's check the updated NameService class which needs to be checked in the document. So, in here we have given whatever is in the document. It's just that the setter method for the private objects contains parameters. To call this class and the emthod getName, we just used a constructor so that it can be used in other class.

```
package com.emse.spring.faircorpagain;

public class NameService {

    private UserService userService;
    private SettingService settingService;
    private UserRepository userRepository;

    public NameService(UserService userService, SettingService
settingService, UserRepository userRepository) {
        this.userService = userService;
        this.settingService = settingService;
        this.userRepository = userRepository;
    }

    public NameService() {

    }


    public String getName() {
        return "Guillaume";
    }
}
```

- "if NameService need to use others objects, you have to update the WelcomeService classc constructor", this is being said in the document
- below is a snap of WelcomeService situation now.



- it says that "Expected 3 arguments but found 0" <-- this is the exact error message
- after much investigation, the following is the code for NameService. Just an additional note, we have created classes named UserService,SettingService,UserRepository in order to use them in our current class namely NameService

```
package  com.emse.spring.faircorpagain;


public class WelcomeService {
    public void sayHello() {
        NameService nameService = new NameService();
        System.out.println("Hello " +  nameService.getName());
    }
}
```

Inversion of Control

- just above we saw that by changing the class named NameService, we have disturbed the className WelcomeService; which was just using the method named getName of NameService class.
- so we can well realise that for simple usage of method from other class, due to tight-coupling, can cause so much of hassle.
- to break the hassle, we have a looseCoupling concept and this concept can be implemented by Invesion of Control.

- I am ignoring all the above explanations as of right now, because in order to understand those and breaking it up like an artist, it would take me a very long time, it seems right now. I dont have time. So, I am just trying to do the exercise that is expected to be done
- so we have an interface below,

```
package com.emse.spring.faircorp.hello;

public interface GreetingService {
    void greet(String name);
}
```

1. Now, I would try to implement this interface
2. I am also supposed to mark the class, which would implement the above interface, with **@Service** annotation. I am still not so sure, why am I using this @Service annotation, hoping to find it out soon.

```
package com.emse.spring.faircorp.hello;
```

```
import org.springframework.stereotype.Service;

@Service
public class ConsoleGreetingService implements GreetingService{
    @Override
    public void greet(String name) {
        System.out.println("Hello,  "+name+"!");
    }
}
```

3. Now, is the time to test the work.

```
package com.emse.spring.faircorp.hello;

import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.boot.test.system.CapturedOutput;
import org.springframework.boot.test.system.OutputCaptureExtension;

@ExtendWith(OutputCaptureExtension.class) // (1)
class GreetingServiceTest {

    @Test
    public void testGreeting(CapturedOutput output) {
        GreetingService greetingService = new ConsoleGreetingService();  //
(2)
        greetingService.greet("Spring");
        Assertions.assertThat(output.getAll()).contains("Hello, Spring!");
    }
}
```

4. to do the testing, we do the below first.

```
gradlew test
```

5. After we get the message that the **build successful**, we run the the testinging class which is **GreetingServiceTest** in here and hit the play button. A video, is available in the documentation as well and we get the desired output.
6. Just a note, I had to take a keen look at the documentation and with the firm believe in myself was able to find out that the testing class was also under **hello** package.

Inject your bean

Your second Job is to create a new interface **UserService** in package `com.emse.spring.faircorp.hello`

```
package com.emse.spring.faircorp.hello;

public interface UserService {
  void greetAll();
}
```

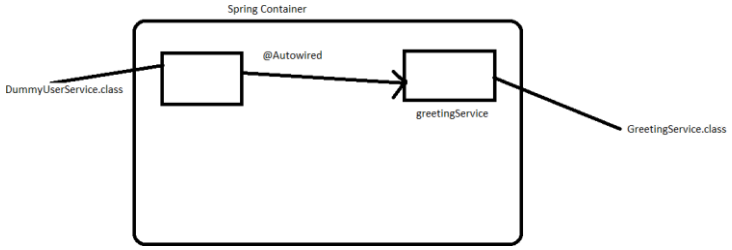
You can now

1. create an implementation of this interface called **DummyUserService**
2. Mark it as a service.
3. Inject service **GreetingService** (use interface and not implementation)
4. Write `greetAll` method. You have to create a List of String with 2 elements ("Elodie" and "Charles") and for each one you have to call `greet` method of the **GreetingService**

As for the first service, we're going to check this new service with a unit test

1. create an implementation of this interface called DummyUserService
2. Mark it as a service.
3. Inject service GreetingService (use interface and not implementation)

```
package com.emse.spring.faircorp.hello;

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

@Service
public class DummyUserService implements UserService{
    @Override
    public void greetAll() {

    }
    @Autowired
    GreetingService greetingService;

}
```



4. Write greetAll method. You have to create a List of String with 2 elements ("Elodie" and "Charles") and for each one you have to call greet method of the GreetingService
5. Testing with the below,

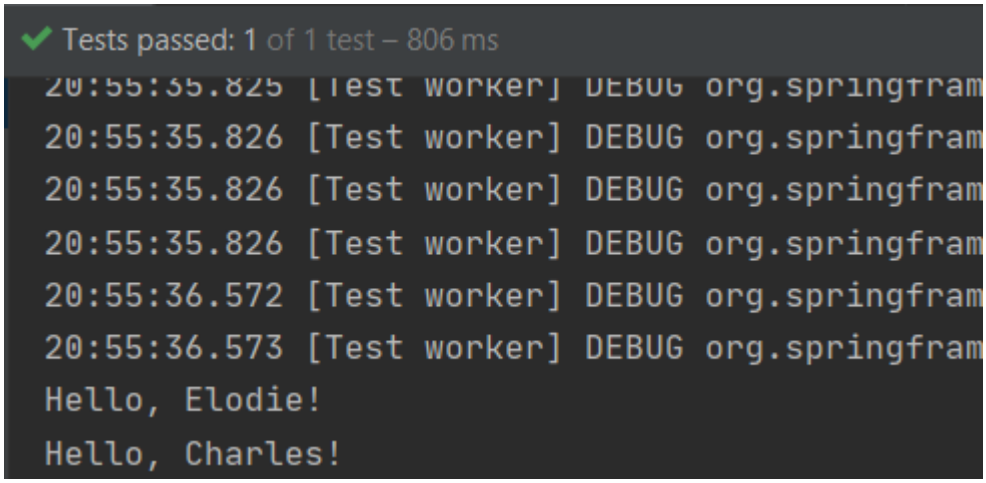
```
package com.emse.spring.faircorp.hello;

import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.system.CapturedOutput;
import org.springframework.boot.test.system.OutputCaptureExtension;
import   org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.test.context.junit.jupiter.SpringExtension;

@ExtendWith(OutputCaptureExtension.class)
@ExtendWith(SpringExtension.class) // (1)
class DummyUserServiceTest {

    @Configuration // (2)
    @ComponentScan("com.emse.spring.faircorp.hello")
    public static class  DummyUserServiceTestConfig{}

    @Autowired // (3)
    public DummyUserService dummyUserService;

    @Test
    public void testGreetingAll(CapturedOutput output) {
        dummyUserService.greetAll();
        Assertions.assertThat(output).contains("Hello, Elodie!", "Hello,
Charles!");
```

```
    }
}
```

- Now, again, I did,

```
gradlew test
```

- build successful and then the play button
- output below



- Now, Instructions in regards to how the testing part has been written.

- (1) We use `SpringExtension` to link our test to Spring. With this annotation a Spring Context will be loaded when this test will run
- (2) We have to configure how the context is loaded. In our case we added `@ComponentScan("com.emse.spring.faircorp.hello")` to help Spring to found our classes. In our app this scan is made by SpringBoot, but in our test SpringBoot is not loaded
- (3) As our test has is own Spring Context we can inject inside the bean to test#

You can verify that your implementation is working properly by running `./gradlew test` command.

- Instructions again

Inject your bean in configuration bean

Now, a new class `FaircorpApplicationConfig` in **com.emse.spring.faircorp** package next **FaircorpApplication** class. We want to create a new bean of type `CommandLineRunner`.

CommandLineRunner instances are found by Spring Boot in the Spring context and are executed during the application startup phase.

```
// (1)
public class FaircorpApplicationConfig {

  // (2)
  public CommandLineRunner greetingCommandLine() { // (3)
      return new CommandLineRunner() {
        @Override
        public void run(String... args) throws Exception {
            // (4)
        }
      };
  }
}
```

- (1) First, annotate this class to mark it as a configuration bean
- (2) Add annotation to say that this method return a new Bean Spring
- (3) Then, tell Spring that here we need here a **GreetingService** component, by declaring it as a method argument
- (4) Finally, call here some service method to output the "Hello, Spring!" message at startup; since we're getting **GreetingService**, no need to instantiate one manually

- Complete code

```
package com.emse.spring.faircorp;

import com.emse.spring.faircorp.hello.GreetingService;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

// (1)
@Configuration
public class FaircorpApplicationConfig {

    // (2)
    @Bean
    public CommandLineRunner greetingCommandLine(GreetingService
greetingService) { // (3)
        return new CommandLineRunner() {
            @Override
            public void run(String... args) throws Exception {
                greetingService.greet("Spring");
                // (4)
            }
        };
    }
}
```

- output

Other cases

- Now, we're going to test a few cases to understand how a Spring Application reacts to some situations. For each case, try the suggested modifications, restart your application and see what happens.
- Of course, after each case, revert those changes, to get "back to normal". (You can use Git for that)

1. What happens if you comment the @Component / @Service annotation on your ConsoleGreetingService?

- did that, just commented @Service
- Reading the error

```
Error starting ApplicationContext. To display the conditions report re-run
your application with 'debug' enabled.
2020-12-31 00:00:08.422 ERROR 16940 --- [  restartedMain]
o.s.b.d.LoggingFailureAnalysisReporter    :

***************************
APPLICATION FAILED TO START
***************************

Description:

Field greetingService in com.emse.spring.faircorp.hello.DummyUserService
required a bean of type 'com.emse.spring.faircorp.hello.GreetingService'
that could not be found.

The injection point has the following annotations:
        -
@org.springframework.beans.factory.annotation.Autowired(required=true)


Action:

Consider defining a bean of type
'com.emse.spring.faircorp.hello.GreetingService' in your configuration.


Process finished with exit code 0
```

- we found that **GreetingServiceTest** uses the class **ConsoleGreetingService**'s object. Since, @Service has been commented out and so this is not happening anymore as in the GreetingServiceTest is not able to use the object anymore. So, testing is failing.

2. Now, try adding **AnotherConsoleGreetingService** (which says "Bonjour" instead of "Hello"), **marked as a component as well.** The highlighted statement makes me feel that **@Service is equivalent to @Component**

- @Component
  - This tells that, I authorise this class to be used as bean

```
package   com.emse.spring.faircorp.hello;

import  org.springframework.stereotype.Service;

@Service
public class AnotherConsoleGreetingService implements GreetingService{
    @Override
```

```
    public void greet(String name) {
        System.out.println("Bonjour,  "+name+"!");
    }
}
```

2. Contd.. Try again this time after adding a @Primary annotation on ConsoleGreetingService.

3. Finally, try the following - what happens and why?

```
@Service
public class CycleService {

  private final ConsoleGreetingService consoleGreetingService;

  @Autowired
  public CycleService(ConsoleGreetingService consoleGreetingService) {
    this.consoleGreetingService = consoleGreetingService;
  }
}
```

- the below code has **1 related problem**.. This problem is coming because of the failed testing by **GreetingServiceTest**

```
GreetingService greetingService = new ConsoleGreetingService();
```

- an argument is required there..

```
package com.emse.spring.faircorp.hello;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Primary;
import org.springframework.stereotype.Service;

//@Service
@Primary
public class ConsoleGreetingService implements GreetingService{

    private final CycleService cycleService;

    @Autowired
    public ConsoleGreetingService(CycleService cycleService) {
        this.cycleService = cycleService;
    }

    @Override
```

```
    public void greet(String name) {
        System.out.println("Hello,  "+name+"!");
    }
}
```

- we do have an output but the test is failing, got to fix that

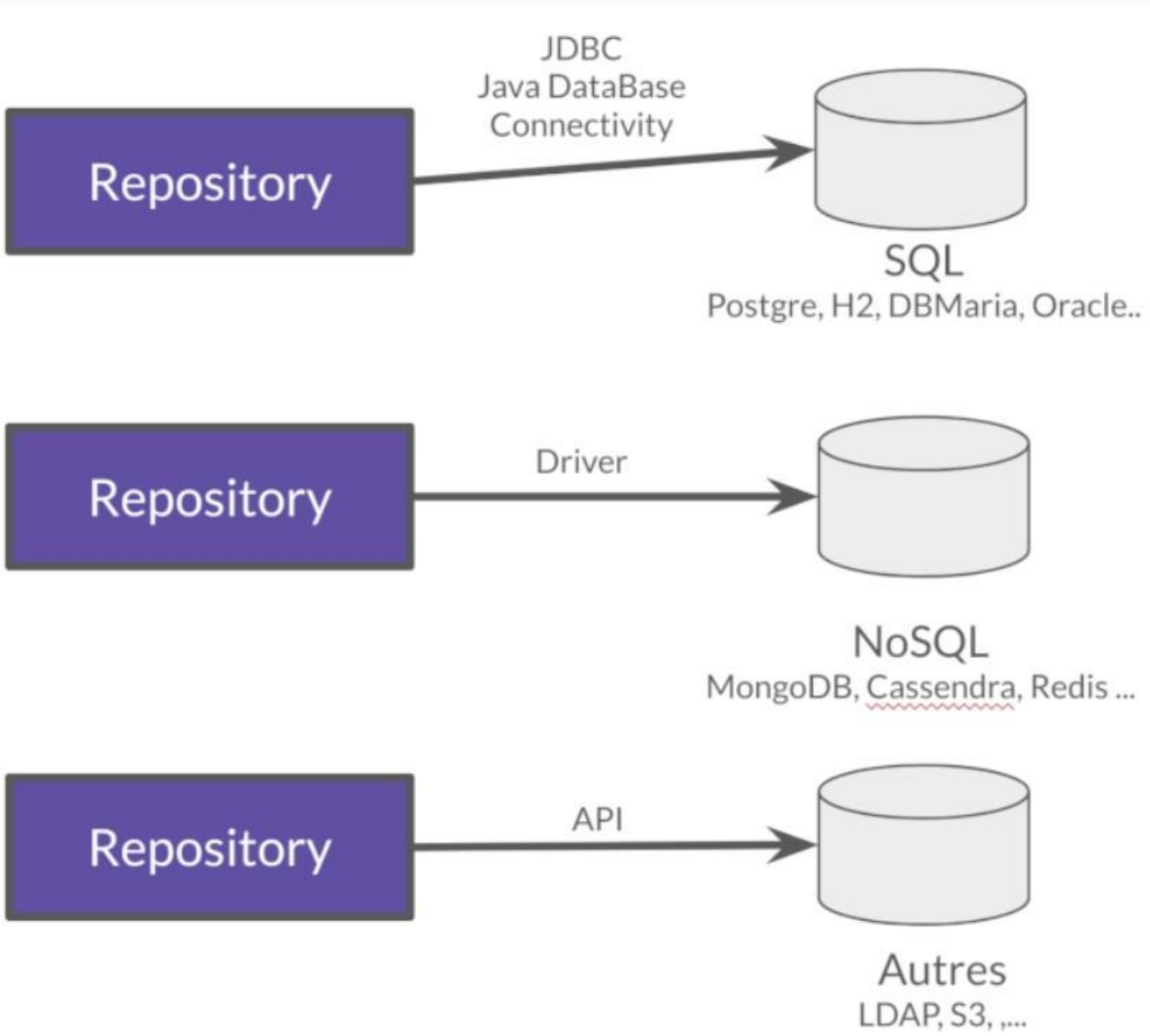
```
Bonjour, Spring!
```

- Anyway, if we look closely into the code, we would find that we are injecting into cycleService, consoleGreetingService and vice-versa. Such kind of closed loop was not supposed to happen. Anyway, this part was also not supposed to be tested. As of right now, am not disturbing the current code in intellij. I would later save and then modify the code, if required.The current situation of the code gives no error to tests or anything further. Moving to Database

---

Spring in practice : database and JPA

- How do we use a database in our Spring project with Spring Data JPA and a H2 Database?
- Hibernate will be the JPA implementation
- Database: An application needs access to data, write data, update these data …
- Today we can access a multitude of data sources … and Spring will help us
- Spring is connecting to databases or dataSaving places



How can we persist data in Java?

- Persistence is "the continuance of an effect after its cause is removed". In the context of storing data in a computer system, this means that the data survives after the process with which it was created has ended. In other words, for a data store to be considered persistent, it must write to non-volatile storage.

1. a low level standard : **JDBC** (Java Database Connectivity)
2. an API, **JPA** (Java Persistence API) and frameworks as Hibernate
3. an API, **JTA** (Java Transaction API) to manage transactions

JDBC

- common API used by all relational databases

- each DBMS editor (DataBase Management System) provides its driver (a jar added to your project)
- each DBMS accept SQL code to execute request and execute order on the database

H2

- In our tests we will use a database written in Java, the H2 database
- Open source, JDBC driver
- embedded database
- in memory database (perfect for tests)
- Browser based Console application
- Small footprint **In information technology, a footprint is the amount of space a particular unit of hardware or software occupies. Marketing brochures frequently state that a new hardware control unit or desktop display has a "smaller footprint," meaning that it occupies less space in the closet or on your desk.**

Lab : Database and SQL

- Go in your FaircorpApplication. We need to add new Spring Boot starters

```
implementation 'org.springframework.boot:spring-boot-starter-data-jpa' //
libs to use JPA in your project
implementation 'com.h2database:h2' // libs to use a H2 database
```

- You can use the H2 console. To verify that everything is fine, open this URL in your browser: http://localhost:8080/console
- The above database url worked with

```
gradlew --continuous bootRun
```

and not with the running of the entry class.

- Now, we are on the database page and we can create several SQL orders
- SQL order to create a table (Id is generated by the database and with use option auto_increment)

```
CREATE TABLE ROOM(ID BIGINT auto_increment PRIMARY KEY, NAME VARCHAR(255)
NOT NULL);
```

- The above creates and with Telusko, we were creating this from the Spring Boot.
- SQL order to insert data in this table (We use a negative id because we don't use the ID generator in manual inserts)

```
INSERT INTO ROOM(ID, NAME) VALUES(-10, 'Room1');
```

- SQL order to select this data

```
SELECT * FROM ROOM;
```

Java and JDBC

- To understand the value of Spring and JPA, it is important to see the code that would have to be done if we wanted to directly use the JDBC API which is a low level API requiring a lot of code.
- This entire thing is not part of the task and hence skipping all of this and moving to JPA.

JPA

- The Java Persistence API (JPA) is a Java application programming interface specification that describes the management of relational data in applications using Java Platform, Standard Edition and Java Platform, Enterprise Edition
- Hibernate ORM is the JPA implementation that we're going to use in this lab.
- We're going to use Spring Data JPA to store and retrieve data in our relational database.
- With Persistence API/Framework, the approach is to :
    - work object (Java entities) and not with database table
    - add annotations to map entity properties to table columns
    - generate common database request (Create, Update, Delete, Read)

    - fill the SQL imperfections: inheritance, relationships, customs types, validation

- Spring provides several sub projects to make database interactions easy
- Do not confuse Spring Data with Spring Data JPA. We can read on in the offical doc that "Spring Data's mission is to provide a familiar and consistent, Spring-based programming model for data access while still retaining the special traits of the underlying data store. It makes it easy to use data access technologies, relational and non-relational databases, map-reduce frameworks, and cloud-based data services. This is an umbrella project which contains many subprojects that are specific to a given database [...] Spring Data JPA is part of Spring Data, lets implement JPA based repositories. It makes it easier to build Spring-powered applications that use data access technologies."

JPA Entity

- Let's take the example of a Java class named Sensor and see how to use JPA to bind it to the SP_SENSOR table of our database.

```
package com.emse.spring.faircorp;
import javax.persistence.*;


@Entity // (1):indicates that this class is an entity managed by Hibernate
@Table(name = "SP_SENSOR") // (2):you can customize the table name (optional
if table name = entity name)

public class Sensor {
    @Id // (3):you have always an id annotated with @javax.persistence.Id
(auto generated in this example). This ID is immutable(cannot be changed)
(as the primary key in the database)
    @GeneratedValue
    private Long id;

    @Column(nullable=false, length=255)  // (4):by default, each property is
mapped to a column. You can customize the nullability or the column  name.
    private String name;

    private String  description;

    @Column(name = "power") // (4).
    private Integer defaultPowerInWatt;

    @Transient // (5):If a property should not be persisted/continued, use
```

```
    private Integer notImportant;

    @Enumerated(EnumType.STRING) // (6):Java enum persisted as a String
(choose always EnumType.STRING)

    private PowerSource powerSource; //Enum of powerSource have been  created

    public Sensor() { // (7).:an entity must have an empty
constructor(public or protected).An empty constructor is needed to create a
new instance via reflection (using Class<T>.newInstance()) by Hibernate
which has to instantiate your Entity dynamically. If you don't provide any
additional constructors with arguments for the class, you don't need to
provide an empty constructor because you get one per default. Java always
gives you a default invisible empty constructor. If an argument constructor
is provided in your class, then jvm will not add the no-argument
constructor.


    }

    public Sensor(String name) { // (8). you can add (and you should) a
constructor to build an object with all required properties
        this.name = name;
    }

    public Long getId() { // (9).you have to define a getter and a setter
for each property
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public Integer getDefaultPowerInWatt() {
```

```
        return defaultPowerInWatt;
    }

    public void setDefaultPowerInWatt(Integer defaultPowerInWatt) {
        this.defaultPowerInWatt = defaultPowerInWatt;
    }

    public Integer getNotImportant() {
        return notImportant;
    }

    public void setNotImportant(Integer notImportant) {
        this.notImportant = notImportant;
    }

    public PowerSource getPowerSource() {
        return powerSource;
    }

    public void setPowerSource(PowerSource powerSource) {
        this.powerSource = powerSource;
    }
}
```

- I am skipping everything intermediate and moving to

Lab 4 : JPA

- It’s time for you to start to build your real application which is able to manage sensors in a building. The management of the sanitary conditions (COVID-19 pandemic, pollution), user comfort, and energy efficiency, require concurrent management of window openings in the École des Mines buildings.
- It is necessary to ventilate as much as possible to limit the spread of the cirus and air pollution in general, but as winter approaches it will become important to heat the buildings to ensure user comfort. Windows should be open during and after classes, closed at night or in case of heavy rain or severe cold. Thus the management of the health crisis becomes concurrent with the quest for energy efficiency in the building.
- We will now create an application which will able to manage the building windows.
    - the building has an outside temperature, and rooms

- each room has zero or more heaters, has zero or more windows, a name, a floor, a current temperature, a target temperature.
- each heater has a name, an on or off status, possibly a power.
- each window has a name, an a status open or closed

Entity creation

- Create a Java enum called HeaterStatus in package com.emse.spring.faircorp.model. This enum has 2 values : ON and OFF

```
package com.emse.spring.faircorp.model;

public enum Status {
    ON, OFF
}
```

- Create another Java enum called WindowStatus in package com.emse.spring.faircorp.model. This enum has 2 values : OPEN and CLOSED

```
package com.emse.spring.faircorp.model;

public enum WindowStatus {
    OPEN,CLOSED
}
```

- Create an Entity called Window in package com.emse.spring.faircorp.model (an entity is a class)

```
package com.emse.spring.faircorp.model;

import javax.persistence.*;

@Entity
@Table(name = "RWINDOW")
public class Window {
        @Id
        @GeneratedValue
        private Long id;

        @Column(nullable=false, length=255)
        private String name;
```

```
        @Column(nullable=false)
        @Enumerated(EnumType.STRING)
        private WindowStatus windowStatus;

//        @Column(nullable=false)
        @ManyToOne
        private Room room;

        public Window() {
        }

        public Window(Room room, String name, WindowStatus status) {
            this.room  =room;
            this.windowStatus = status;
            this.name = name;
        }

        public Window(String name, WindowStatus status) {
            this.windowStatus = status;
            this.name = name;
        }

        public Long getId() {
            return this.id;
        }

        public void setId(Long id) {
            this.id = id;
        }

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }

        public WindowStatus getWindowStatus() {
            return windowStatus;
        }

        public void setWindowStatus(WindowStatus windowStatus) {
            this.windowStatus = windowStatus;
        }
        public Room getRoom() {
            return room;
        }

        public void setRoom(Room room) {
            this.room = room;
```

```
        }
}
```

Used the good annotations to

(1) Mark this class as a JPA entity

(2) You must give a different name for your table. H2 can't call a table Window because it is a reserved word. So call it RWINDOW

(3) Declare this field as the table ID. This ID must to be auto generated

(4) This field must be not nullable

(5) WindowStatus is also not nullable, and this field is an enumeration (you have to use @Enumerated). You have to add these informations

Instruction contd..

- Write now the Heater entity with
  - an auto generated id
  - a non nullable String name
  - a nullable Long power
  - a non nullable room
  - a non nullable status (ON, OFF). This field is an enumeration (you have to use @Enumerated).
  - create a constructor with non nullable fields and a default constructor

```
package com.emse.spring.faircorp.model;

import javax.persistence.*;

@Entity
@Table(name = "HEATER")
public class Heater {

    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable=false, length=255)
    private String name;

    @Column(nullable=true)
    private  Long power;
```

```
    //    @Column(nullable=false)
    @ManyToOne
    private Room room;

    @Column(nullable=false)
    @Enumerated(EnumType.STRING)
    private HeaterStatus heaterStatus;

    public Heater(){
    }

    public Heater(Long id, String name,  HeaterStatus heaterStatus) {
        this.id = id;
        this.name = name;
        this.heaterStatus=  heaterStatus;
    }

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public Long getPower() {
        return power;
    }

    public Room getRoom() {
        return room;
    }
//    public Room getHeaterStatus() {
//        return heaterStatus;
//    }

    public void setId(Long id) {
        this.id = id;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setPower(Long power) {
        this.power = power;
    }

    public void setRoom(Room room) {
        this.room = room;
```

```
    }

    public void setHeaterStatus(HeaterStatus heaterStatus) {
        this.heaterStatus = heaterStatus;
    }
}
```

- the enum class

```
package com.emse.spring.faircorp.model;

public enum HeaterStatus {
    ON, OFF
}
```

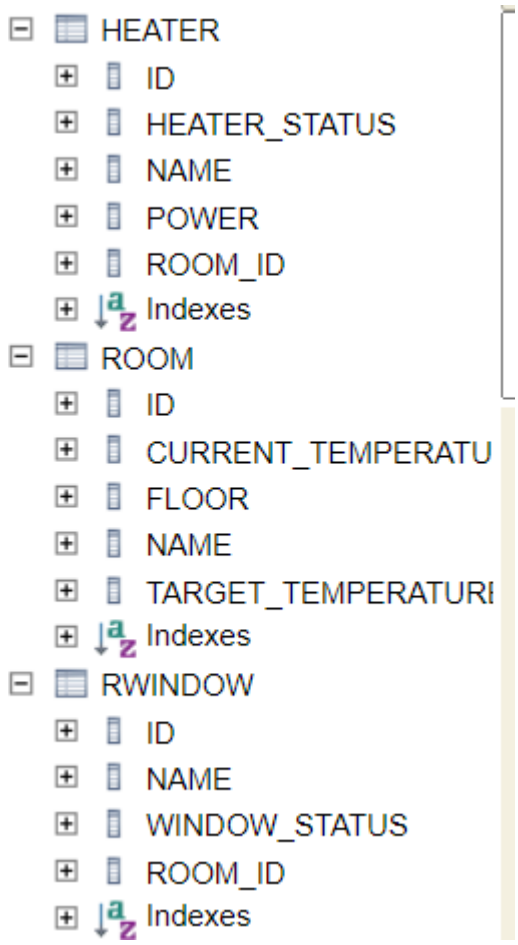
Instructions contd..

- the Room entity
  - an auto generated id
  - a non nullable floor (Integer)
  - a non nullable String name
  - a current temperature (Double)
  - a target temperature (Double)
  - a list of heaters. You have to define a bidirectional association between Room and Heater
  - a list of windows. You have to define a bidirectional association between Room and Window : update the Window entity constructor to always send the room when a room is created, ie add an argument Room in the Window constructor
  - create a constructor with non nullable fields and a default constructor

You can start your application. If you updated your configuration (see this **chapter**) you should see in your app logs

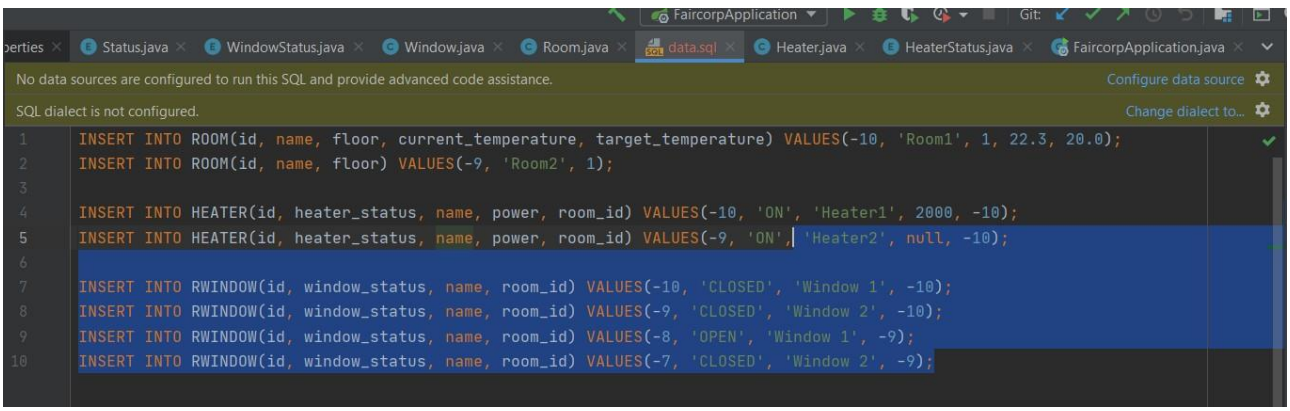
```
Hibernate: drop table if exists heater CASCADE
Hibernate: drop table if exists room CASCADE
Hibernate: drop table if exists room_windows CASCADE
Hibernate: drop table if exists rwindow CASCADE
Hibernate: drop sequence if exists hibernate_sequence
Hibernate: create sequence hibernate_sequence start with 1 increment by 1
Hibernate: create table heater (id bigint not null, heater_status varchar(255) not null, name varchar(255) not null, power bigint, room_id bigint not null, primary key (id))
Hibernate: create table room (id bigint not null, current_temperature double, floor integer not null, name varchar(255) not null, target_temperature double, primary key (id))
Hibernate: create table room_windows (room_id bigint not null, windows_id bigint not null, primary key (room_id, windows_id))
Hibernate: create table rwindow (id bigint not null, name varchar(255) not null, window_status varchar(255) not null, room_id bigint not null, primary key (id))
Hibernate: alter table room_windows add constraint UK_ojyqj1kyod1v2m4eoltwj7sfa unique (windows_id)
Hibernate: alter table heater add constraint FKba10hbsubifukw0monybjx3a0 foreign key (room_id) references room
Hibernate: alter table room_windows add constraint FKbpyirwvjc67pkk61hnh5qpn6y foreign key (windows_id) references rwindow
Hibernate: alter table room_windows add constraint FKaihtbjcqqmheg9s19aala347k foreign key (room_id) references room
Hibernate: alter table rwindow add constraint FK621arbu4dtuis4nf2o3xtapfn foreign key (room_id) references room
```

- exact h2 console status



Populate data

- We’re going to populate our database and insert data in tables.
- You can execute the script below in your H2 console, but data will be - deleted on the next app reload. Fortunately Spring Boot offers a mechanism to populate a database at startup.
- Create a file data.sql in src/main/resources next to application.properties
- Create a file data.sql in src/main/resources next to application.properties
- data.sql



Dao creation

- Simple DAO
- Write now 3 Spring data DAO WindowDao, HeaterDao and RoomDao in package com.emse.spring.faircorp.dao (interface that extends JpaRepository with the good types for entity and its id)
- You’re going to write your own DAO methods (for specific requests), you have to create custom interfaces and implementations with your custom methods.
- WindowDao.java

```
package com.emse.spring.faircorp.dao;

import com.emse.spring.faircorp.model.Window;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Modifying;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

public interface WindowDao  extends JpaRepository<Window, Long> {

    @Modifying
    @Query(value = "delete from Rwindow w where w.room.id = :roomId",
nativeQuery = true )
    void deleteRwindowByRoom(@Param("roomId") Long roomId);

}
```

- RoomDao

```
package com.emse.spring.faircorp.dao;

import com.emse.spring.faircorp.model.Room;
import org.springframework.data.jpa.repository.JpaRepository;

public interface RoomDao extends JpaRepository<Room, Long> {
}
```

- HeaterDao

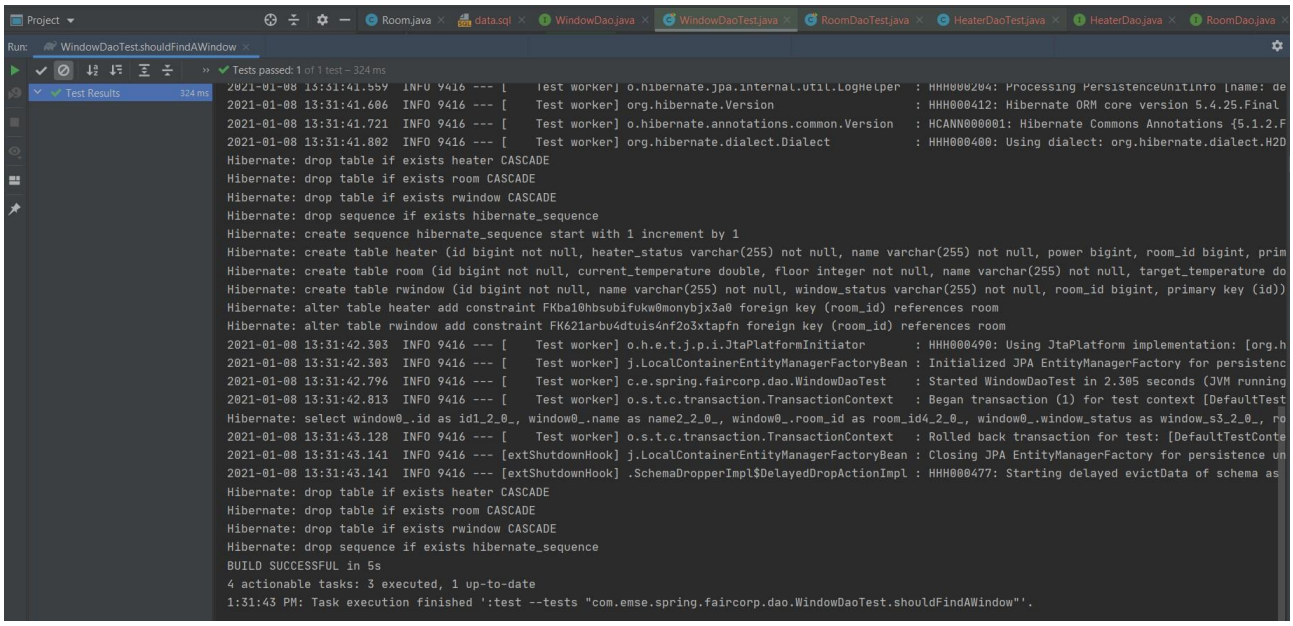
```
package com.emse.spring.faircorp.dao;

import com.emse.spring.faircorp.model.Heater;
import org.springframework.data.jpa.repository.JpaRepository;
```

```
import org.springframework.data.jpa.repository.Modifying;
import org.springframework.data.jpa.repository.Query;

public interface HeaterDao  extends JpaRepository<Heater, Long> {

    @Modifying
    @Query("delete from Heater r where r.room.id = ?1")
    void deleteHeaterByRoom(Long roomId);
}
```

- To check WindowDao, create a class WindowDaoTest in src/test/java/com.emse.spring.faircorp.dao

```
import  com.emse.spring.faircorp.model.Window;
import com.emse.spring.faircorp.model.WindowStatus;
import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import  org.springframework.test.context.junit.jupiter.SpringExtension;

@ExtendWith(SpringExtension.class)
@DataJpaTest
class WindowDaoTest {
    @Autowired
    private WindowDao windowDao;

    @Test
    public void shouldFindAWindow() {
        Window window = windowDao.getOne(-10L);
        Assertions.assertThat(window.getName()).isEqualTo("Window 1");

Assertions.assertThat(window.getWindowStatus()).isEqualTo(WindowStatus.CLOSED)

    }
}
```

- in order to check how to do the testing, we can refer to do this video https://www.youtube.com/watch?v=a_245NeyCrM
- WindowDaoTest results



- RoomDaoTest.java

```
package  com.emse.spring.faircorp.dao;
import  com.emse.spring.faircorp.model.Room;
import  com.emse.spring.faircorp.model.Window;
import com.emse.spring.faircorp.model.WindowStatus;
import org.assertj.core.api.Assertions;
import org.assertj.core.groups.Tuple;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import  org.springframework.test.context.junit.jupiter.SpringExtension;

import java.util.List;

@ExtendWith(SpringExtension.class)
@DataJpaTest
public class RoomDaoTest {
    @Autowired
    private RoomDao roomDao;

    @Test
    public void shouldFindOneRoom() {
        Room room =  roomDao.getOne(-10L);
        Assertions.assertThat(room.getName()).isEqualTo("Room1");
//
Assertions.assertThat(window.getWindowStatus()).isEqualTo(WindowStatus.CLOSED)

    }

}
```

- rightClick on test/java/faircorp/dao and do "run tests on com.emse..."
- result 2 test passed, seems that **HeaterDaoTest** doesn't happen
- now did change to "toto" , in accordance to video and indeed the test fails and hence reverted back
- Execute your test. This test shoyld be green. You can write similar tests to test **RoomDao** and **HeaterDao**
- putting across the HeaterDaoTest.java code

```
package com.emse.spring.faircorp.dao;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import  org.springframework.test.context.junit.jupiter.SpringExtension;

import java.util.List;

@ExtendWith(SpringExtension.class)
@DataJpaTest
public class HeaterDaoTest {
}
```

Custom DAO

- Create your own interface WindowDaoCustom in package com.emse.spring.faircorp.dao

```
package com.emse.spring.faircorp.dao;

import com.emse.spring.faircorp.model.Window;

import java.util.List;

public interface WindowDaoCustom  {
    List<Window> findRoomOpenWindows(Long id);
}
```

- Refactor your WindowDao interface : it must extend JpaRepository and WindowDaoCustom
- so the updated code with a simple update, , **WindowDaoCustom**

```
package com.emse.spring.faircorp.dao;

import com.emse.spring.faircorp.model.Window;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Modifying;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

public interface WindowDao  extends JpaRepository<Window, Long>,
WindowDaoCustom {

    @Modifying
    @Query(value = "delete from Rwindow w where w.room.id = :roomId",
nativeQuery = true )
    void deleteRwindowByRoom(@Param("roomId") Long roomId);

}
```

- Create your own implementation of WindowDaoCustom with your custom methods and inject the EntityManager (JPA)

```
package com.emse.spring.faircorp.dao;

import  com.emse.spring.faircorp.model.Window;
import  com.emse.spring.faircorp.model.WindowStatus;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import java.util.List;

public class WindowDaoCustomImpl implements WindowDaoCustom{
    @PersistenceContext
    private EntityManager em;
    @Override
    public List<Window> findRoomOpenWindows(Long id) {
        String jpql = "select w from Window w where w.room.id = :id and
w.windowStatus= :status";
        return em.createQuery(jpql, Window.class)
                .setParameter("id", id)
                .setParameter("status", WindowStatus.OPEN)
                .getResultList();
    }
}
```

- You have to test your DAO. When Spring context is loaded, the database is populated with the file data.sql and we can test these values. For that update WindowDaoTest test and add these methods

- the WindowDaoTest to be updated by including the below code

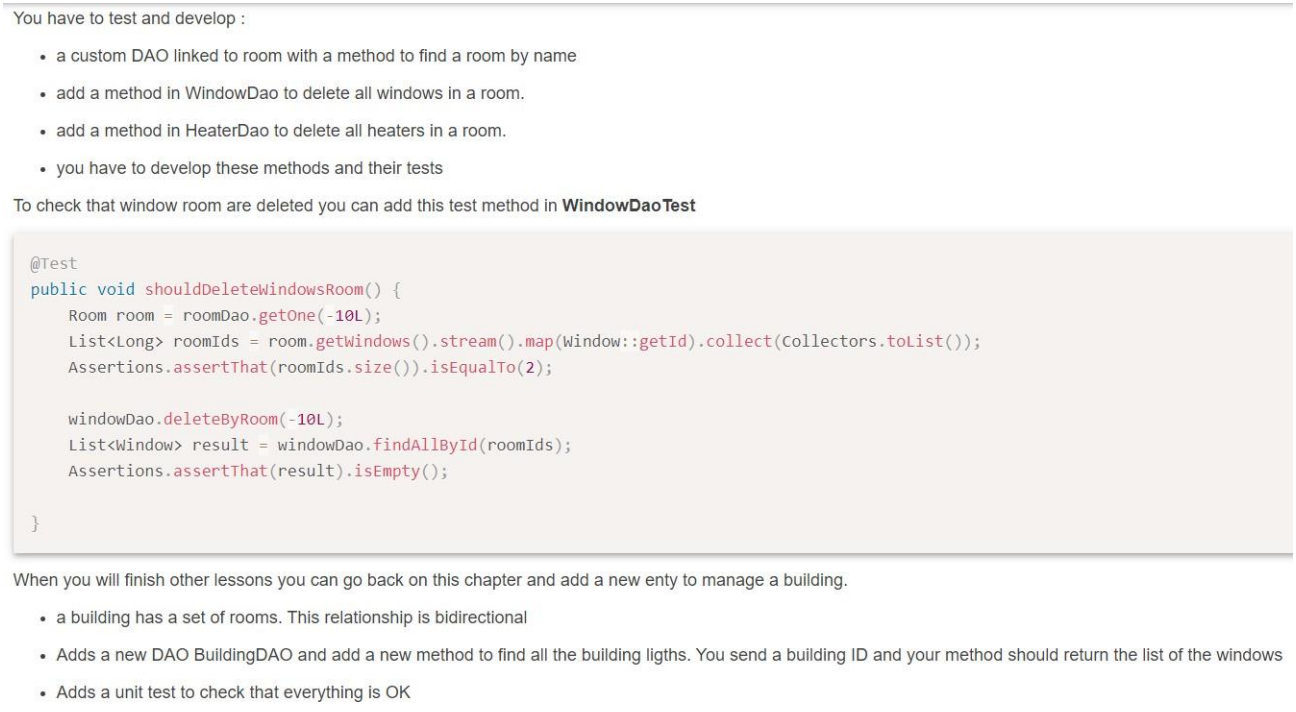
```
@Test
    public void shouldFindRoomOpenWindows() {
        List<Window> result = windowDao.findRoomOpenWindows(-9L);
        Assertions.assertThat(result)
                .hasSize(1)
                .extracting("id", "windowStatus")
                .containsExactly(Tuple.tuple(-8L, WindowStatus.OPEN));
    }

    @Test
    public void shouldNotFindRoomOpenWindows()  {
        List<Window> result = windowDao.findRoomOpenWindows(-10L);
        Assertions.assertThat(result).isEmpty();
    }
```

- the updated code for WindowDaoTest

```
package com.emse.spring.faircorp.dao;


import com.emse.spring.faircorp.dao.WindowDao;
import  com.emse.spring.faircorp.model.Window;
import com.emse.spring.faircorp.model.WindowStatus;
import org.assertj.core.api.Assertions;
import org.assertj.core.groups.Tuple;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import  org.springframework.test.context.junit.jupiter.SpringExtension;

import java.util.List;

@ExtendWith(SpringExtension.class)
@DataJpaTest
class WindowDaoTest {
    @Autowired
    private WindowDao windowDao;

    @Test
    public void shouldFindAWindow() {
        Window window = windowDao.getOne(-10L);
        Assertions.assertThat(window.getName()).isEqualTo("Window 1");

Assertions.assertThat(window.getWindowStatus()).isEqualTo(WindowStatus.CLOSED)

    }
```

```
    @Test
    public void shouldFindRoomOpenWindows() {
        List<Window> result = windowDao.findRoomOpenWindows(-9L);
        Assertions.assertThat(result)
                .hasSize(1)
                .extracting("id", "windowStatus")
                .containsExactly(Tuple.tuple(-8L,  WindowStatus.OPEN));
    }

    @Test
    public void shouldNotFindRoomOpenWindows()  {
        List<Window> result = windowDao.findRoomOpenWindows(-10L);
        Assertions.assertThat(result).isEmpty();
    }

}
```

- last part



You have to test and develop :

- a custom DAO linked to room with a method to find a room by name
- add a method in WindowDao to delete all windows in a room.
- add a method in HeaterDao to delete all heaters in a room.
- you have to develop these methods and their tests

To check that window room are deleted you can add this test method in **WindowDaoTest**

```
@Test
public void shouldDeleteWindowsRoom() {
    Room room = roomDao.getOne(-10L);
    List<Long> roomIds = room.getWindows().stream().map(Window::getId).collect(Collectors.toList());
    Assertions.assertThat(roomIds.size()).isEqualTo(2);

    windowDao.deleteByRoom(-10L);
    List<Window> result = windowDao.findAllById(roomIds);
    Assertions.assertThat(result).isEmpty();

}
```

When you will finish other lessons you can go back on this chapter and add a new enty to manage a building.

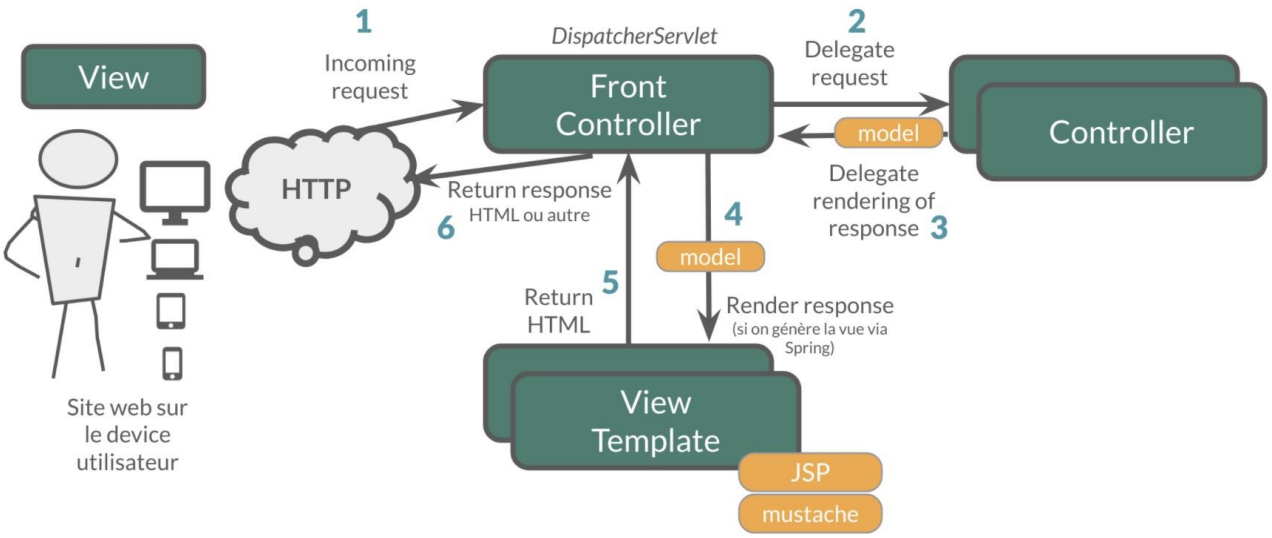
- a building has a set of rooms. This relationship is bidirectional
- Adds a new DAO BuildingDAO and add a new method to find all the building ligths. You send a building ID and your method should return the list of the windows
- Adds a unit test to check that everything is OK

Spring in practice : REST service

- Spring MVC is the Web Framework built in Spring;
- It helps you write web applications and takes care of a lot of boilerplate code, so you just have to focus on your application features.

- With Spring Web (Spring MVC) you can write screens with a template solution which are used to generate HTML. But we don't use this solution in this course. We will see how to write REST services. However if you are interested you can read official documentation.

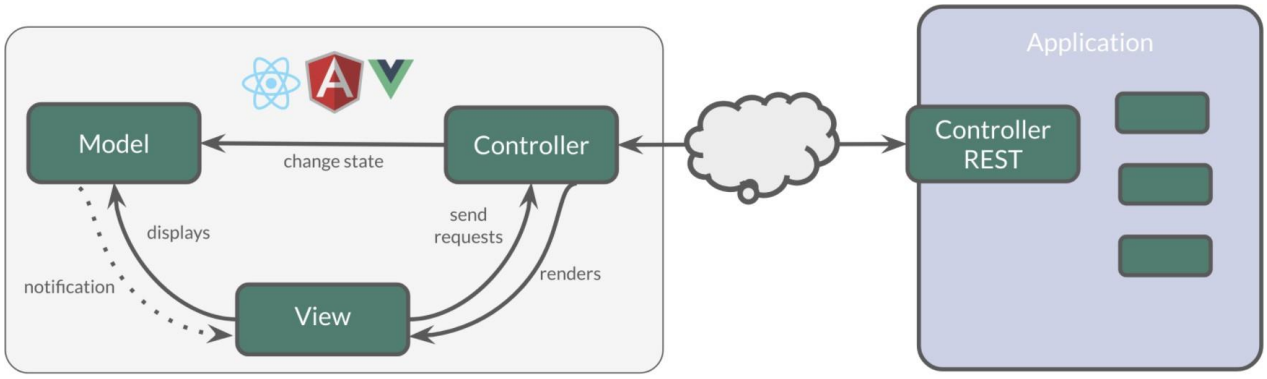
- diagramatic visualisation of the entire



- With Spring Web you can expose REST services to another app (web api, JS app, android app...). This is the purpose of this lesson. You will learn how to develop endpoints on a backend application. **These REST endpoints will be used later by a JS app or an Android app.**

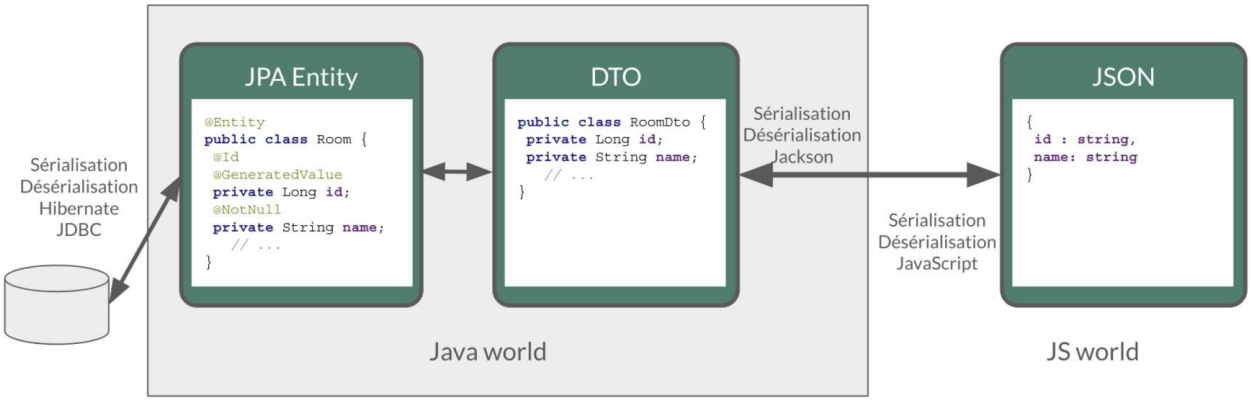
- The below diagram shows, how the backendSpringBoot's RestController exists at one end and this RestController is connected to **ControllerOfJSFrameworkPROBABLY** with the help of cloud

- backend and front end connected through cloud



Data Transfer Object

- A DTO is an object that carries data between processes. Data need to be serializable to go across the HTTP connection
- In order to exchange data, we would need Json data. So a website sends json data on GET requests. Then we POST json. SO json is important. We used interface to extend JpaRepository which in turn would use CrudRepository along with JpaRepository features.We did this in the previoud i.e. database section. Moreover JpaRepository returns list. Such listJavaObjects are transformed to json by **Jackson** which is available in the gradle for Spring Boot.
- The below diagram shows the backend which includes database on the left. And also, frontend on the right. Frontend and Backend would exchange data with json and hence jackson would be used and this jackson has been mentioned in the diagram



- Serialization is the process of translating data structures or object into a format that can be transmitted.
- In our situation, the format that can be transmitted is json and to achieve that format we would use jackson library which comes with springBoot. Just recall that, if we would have chosen xml then we would have had to implement manually into gradle, a 3rd party library.
- Our attempt here, is to do serialisation. The input to the serialisation are the private objects and these private objects are being accessed with getters&setters
- now, created a **dto** package and inside that, tried to copy and paste the code for **WindowDto** that's available in the documentation.
- Tried to import the relavant packages. Later, found that **getRoom** method is asking to create a getter&setter in the model.Window, so did that appropriately
- putting the additional code of the **model.Window**

```
public Room getRoom() {
        return room;

    }
    public void setRoom(Room room) {
        this.room = room;
    }
```

- Refactored code of model.Window is as below,

```
package com.emse.spring.faircorp.model;
import javax.persistence.*;

@Entity// (1)
@Table(name = "RWINDOW")// (2)
public class Window {
    @Id// (3)
    @GeneratedValue
    private Long id;

    @Column(nullable=false, length=255)// (4)
    private String name;

    @Column(nullable=false)
    @Enumerated(EnumType.STRING)// (5)
    private WindowStatus windowStatus;

    @ManyToOne
    private Room  room;

    public Window() {
    }

    public Window(Room room,String name, WindowStatus status) {
        this.room  =room;
        this.windowStatus = status;
        this.name = name;
    }
    public Window(String name, WindowStatus status) {
        this.windowStatus = status;
        this.name = name;
    }

    public Long getId() {
        return this.id;
    }

    public void setId(Long id) {
        this.id = id;
```

```
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public WindowStatus getWindowStatus() {
        return windowStatus;
    }

    public void setWindowStatus(WindowStatus windowStatus) {
        this.windowStatus = windowStatus;
    }

    public Room getRoom() {
        return room;

    }
    public void setRoom(Room room) {
        this.room = room;
    }

}
```

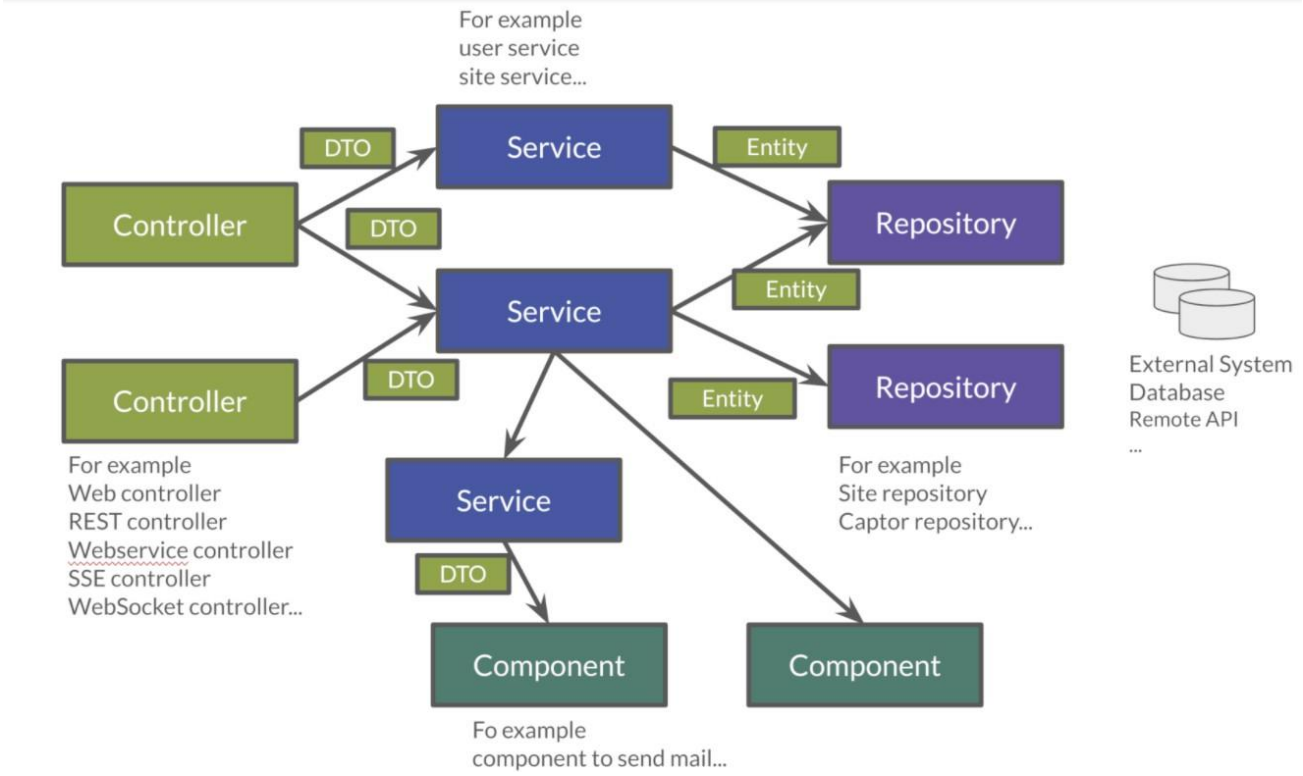
- now the code that is modified based on the requirement but originally it has been copied fromm the documentation
- WindowDto

```
package com.emse.spring.faircorp.dto;

import  com.emse.spring.faircorp.model.Window;
import  com.emse.spring.faircorp.model.WindowStatus;

public class WindowDto {
    private Long id;
    private String name;
    private WindowStatus windowStatus;
    private String roomName;
    private Long  roomId;

    public WindowDto()  {
    }

    public WindowDto(Window window) {
        this.id = window.getId();
```

```
        this.name = window.getName();
        this.windowStatus = window.getWindowStatus();
        this.roomName = window.getRoom().getName();
        this.roomId = window.getRoom().getId();
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public WindowStatus getWindowStatus() {
        return windowStatus;
    }

    public void setWindowStatus(WindowStatus windowStatus) {
        this.windowStatus = windowStatus;
    }

    public String getRoomName() {
        return roomName;
    }

    public void setRoomName(String roomName) {
        this.roomName = roomName;
    }

    public Long getRoomId() {
        return roomId;
    }

    public void setRoomId(Long roomId) {
        this.roomId = roomId;
    }
}
```

- DTO will be used to transfer and to receive data in our REST controllers (entry point in our Java webapp).

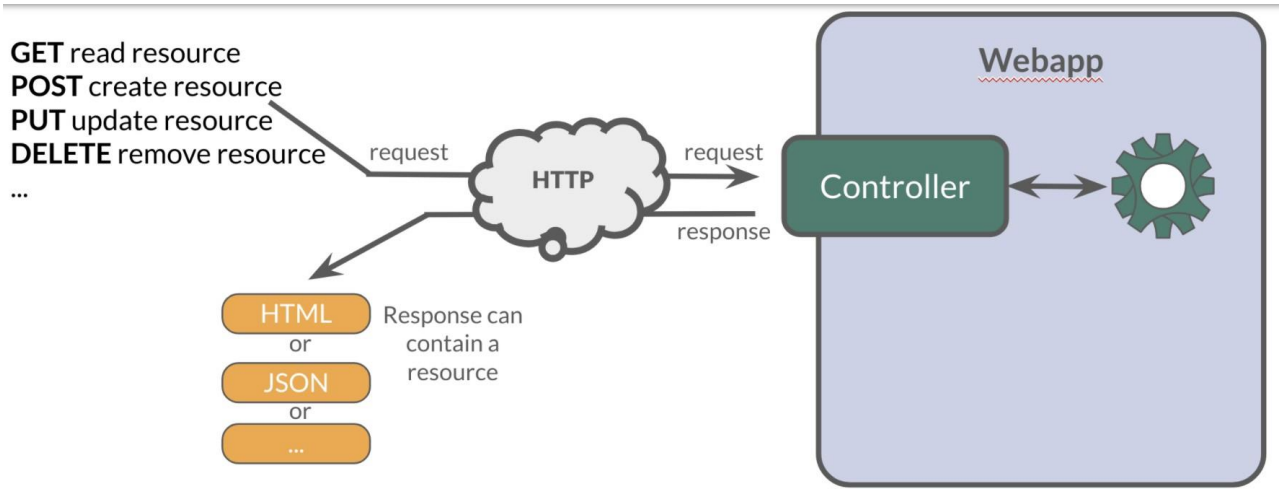
- Very often we find a constructor with the entity allowing to build a new instance. But beware, a DTO must always have an empty constructor. Libraries used to serialize or deserialize an object use the Java reflection API. In our case we will have a constructor allowing to build a WindowDto from Window entity.



HTTP

- The Hypertext Transfer Protocol (HTTP) is an application protocol used for data communication on the World Wide Web.
- HTTP defines methods (sometimes referred to as verbs) to indicate the desired action to be performed on the identified resource
- A resource can be an image, a video, an HTML page, a JSON document.
- To receive a response you have to send a request with a verb in a client or an application as Curl, Wget…. or with a website
- in the below diagram, webApp is the SpringBootApp. Recall the difference between app and project from the Django.

- client sends request using the http verb which is received by the controller of the web app and this controller sends resource like html/json/etc to the client
- client HTTPverbs request Controller



- Let's attempt tho remember the below diagram forever
- status codes

Each HTTP response has a status identified by a code. This code is sent by the server, by your app

- 1XX : Wait... request in progress
- 2XX : Here ! I send you a resource
- 3XX : Go away !
- 4XX : You made a mistake
- 5XX : I made a mistake

REST

- HTTP requests are handled by the methods of a REST service. In Spring's approach a REST service is a controller. It is able to respond to HTTP requests
- Please note, that Http requests are implemented with REST and no-REST.

- GET: read resource
- POST: creates new record or executing a query
- PUT: edit a resource (sometimes we use only a post request)
- DELETE: delete a record

- Controllers are the link between the web http clients (browsers, mobiles) and your application. They should be lightweight and call other components in your application to perform actual work (DAO for example).
- These components are easily identified bythe **@RestController** annotation.
- Example of addressable resources
  - Retrieve a window list : GET /api/windows
  - Retrieve a particular window : GET /api/windows/{window_id}
  - Create or update a window : POST /api/windows
  - Update a window and update its status : PUT /api/windows/{window_id}/switch
  - Delete a window : DELETE /api/windows/{window_id}

This WindowController handles GET requests for /api/windows by returning a list of WindowDto.

- A complete example to manage windows
- We create a package named **controller** first and then put this code based on the document

```
package com.emse.spring.faircorp.controller;

import com.emse.spring.faircorp.dao.RoomDao;
import com.emse.spring.faircorp.dao.WindowDao;
import com.emse.spring.faircorp.dto.WindowDto;
import com.emse.spring.faircorp.model.Room;
import  com.emse.spring.faircorp.model.Window;
import com.emse.spring.faircorp.model.WindowStatus;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.stream.Collectors;

@RestController // (1)
```

```
@RequestMapping("/api/windows") // (2)
@Transactional // (3)
public class WindowController {
    private final WindowDao windowDao;
    private final RoomDao  roomDao;

    public WindowController(WindowDao windowDao, RoomDao roomDao) { // (4)
        this.windowDao = windowDao;
        this.roomDao = roomDao;
    }

    @GetMapping // (5)
    public List<WindowDto> findAll() {
        return
windowDao.findAll().stream().map(WindowDto::new).collect(Collectors.toList());
  // (6)
    }

    @GetMapping(path = "/{id}")
    public WindowDto findById(@PathVariable Long id) {
        return windowDao.findById(id).map(WindowDto::new).orElse(null); //
(7)
    }

    @PutMapping(path = "/{id}/switch")
    public WindowDto switchStatus(@PathVariable Long id) {
        Window window =
windowDao.findById(id).orElseThrow(IllegalArgumentException::new);
        window.setWindowStatus(window.getWindowStatus() == WindowStatus.OPEN
? WindowStatus.CLOSED: WindowStatus.OPEN);
        return new WindowDto(window);
    }

    @PostMapping // (8)
    public WindowDto create(@RequestBody WindowDto dto) {
        // WindowDto must always contain the window room
        Room room = roomDao.getOne(dto.getRoomId());
        Window window = null;
        // On creation id is not defined
        if (dto.getId() == null) {
            window = windowDao.save(new Window(room, dto.getName(),
dto.getWindowStatus()));
        }
        else {
            window = windowDao.getOne(dto.getId());  // (9)
            window.setWindowStatus(dto.getWindowStatus());
        }
        return new WindowDto(window);
    }

    @DeleteMapping(path = "/{id}")
```

```
    public void delete(@PathVariable Long id) {
        windowDao.deleteById(id);
    }
}
```

- below are the annotations based on the REST

- (1) **RestController** is a Spring stereotype to mark a class as a rest service
- (2) **@RequestMapping** is used to define a global URL prefix used to manage a resource (in our example all requests that start with `/api/windows` will be handle by this controller)
- (3) **@Transactional** is used to delegate a transaction opening to Spring. Spring will initiate a transaction for each entry point of this controller. This is important because with Hibernate you cannot execute a query outside of a transaction.
- (4) DAOs used by this controller are injected via constructor
- (5) **@GetMapping** indicates that the following method will respond to a GET request. This method will return a window list. We transform our entities `Window` in `WindowDto`
- (6) (7) We use **Java Stream API** to manipulate our data
- (8) **@PostMapping** indicates that the following method will respond to a POST request (for saving).
- (9) For an update you don't need to call the DAO save method. Method `getOne` load the persisted data and all changes on this object (attached to a persistent context) will be updated when the transaction will be commited.

Lab : Create your rest services

- A basic example
- This is the time to create your first REST controller with Spring.
- Create a new class HelloController in package com.emse.spring.faircorp.api

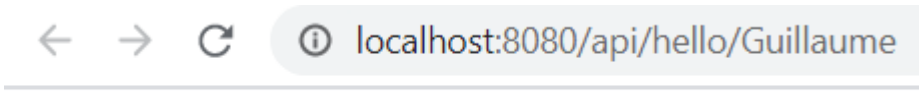
```
package com.emse.spring.faircorp.api;

import org.springframework.transaction.annotation.Transactional;
import  org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import  org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api/hello")
@Transactional
public class HelloController {
    @GetMapping("/{name}")
    public MessageDto welcome(@PathVariable String name) {
        return new MessageDto("Hello " + name);
    }


    class MessageDto {
        String message;

        public MessageDto(String message) {
            this.message = message;
        }

        public String getMessage() {
```

```
            return message;
        }
    }
}
```

- Test your service
- Browser for GET requests
- If your REST service expose an handler for a GET HTTP request, this handler can be tested in a browser.
- Launch your app and open the URL http://localhost:8080/api/hello/Guillaume in your browser
- When you type an URL in the adress bar, your browser send a GET HTTP request. You should see a response as this one

```
{"message":"Hello Guillaume"}
```

- **And yes, we got the output after initialising the FaircorpApplication**




```
{"message":"Hello Guillaume"}
```

Swagger for all requests

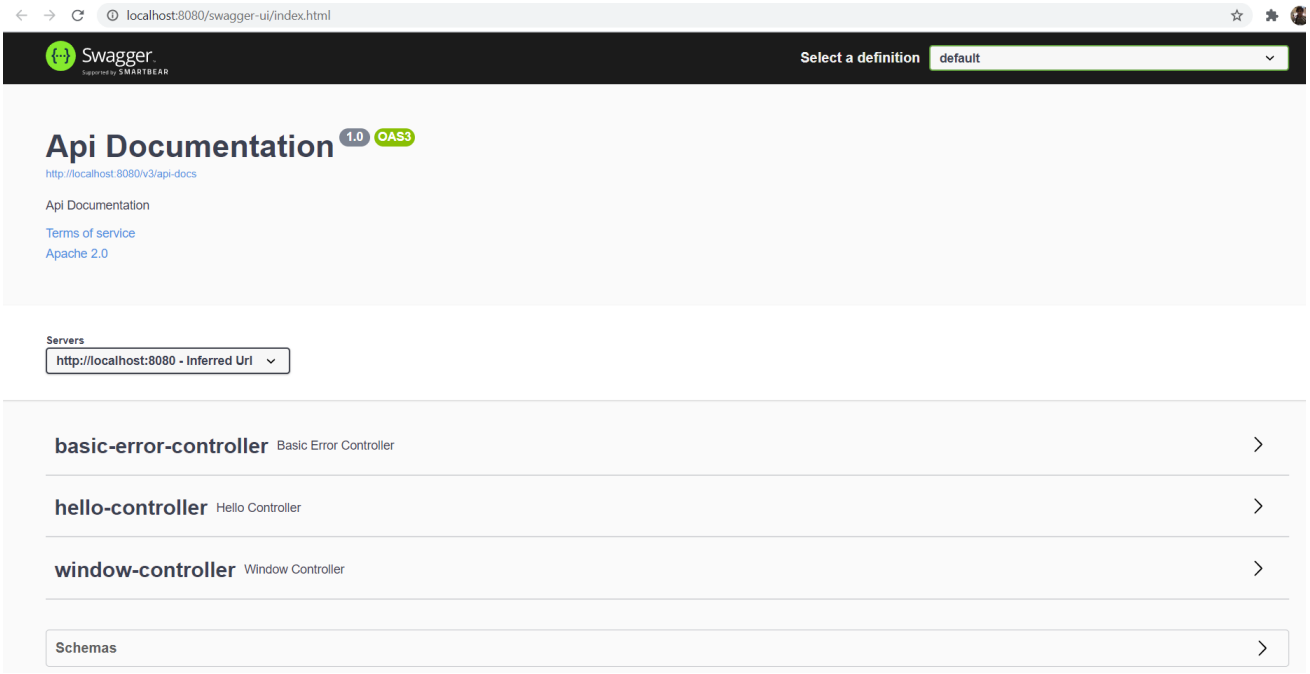
- With a browser you are limited to GET requests.
- If you want to test PUT, POST or DELETE HTTP requests, you need another tool. We will use swagger.
- The advantage of swagger is that it is very well integrated into the Spring world. Update your build.gradle file and add these dependencies
- BTW, we could have also used **Postman** based on my knowledge

```
implementation 'io.springfox:springfox-boot-starter:3.0.0'
```

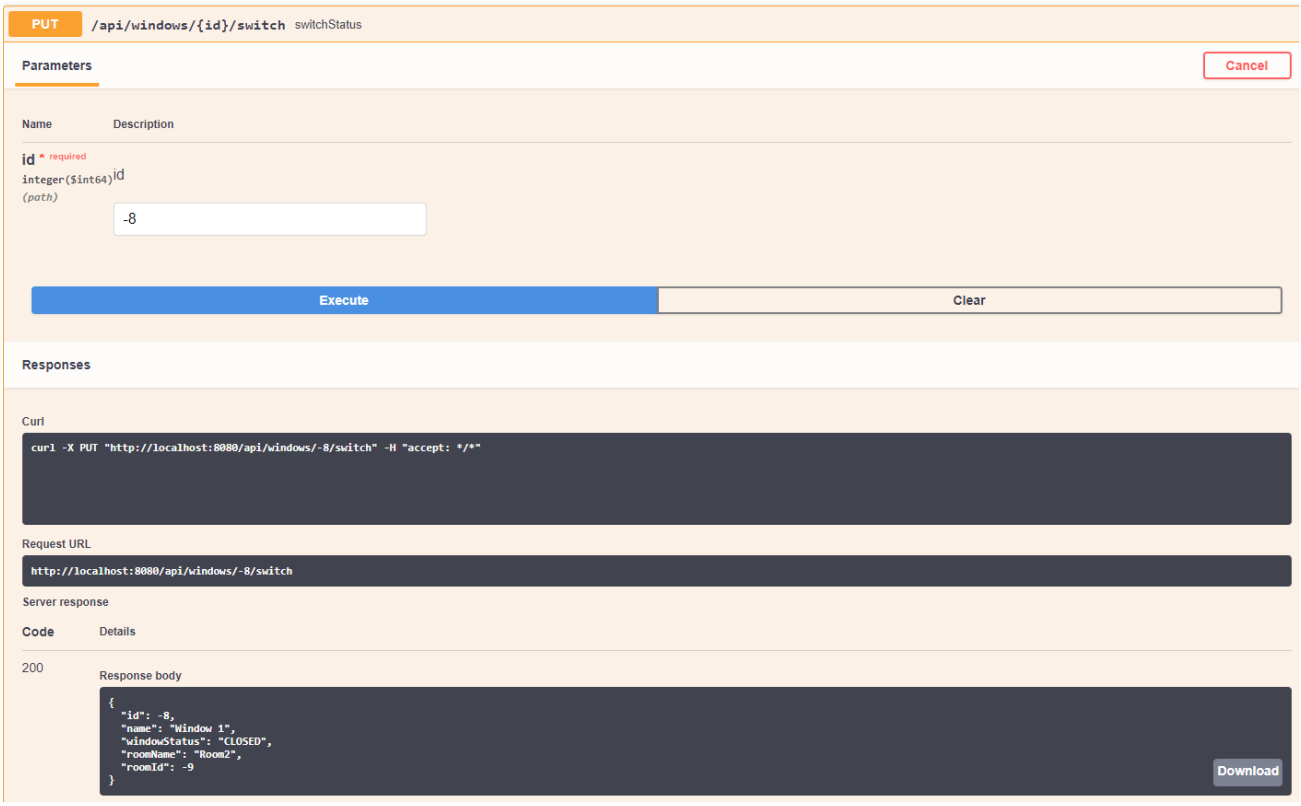
- after putting in the above in build.gradle, I instigated

```
gradlew --continuous bootRun
```

- The thing is stuck at80%, let's see
- And now you can relaunch your app and open swagger interface http://localhost:8080/swagger-ui/index.html
- the above link takes us to the below



- All your endpoints are available. You can click on one of them to test it like the this video, https://www.youtube.com/watch?v=f6FUpLs0H_4
- followed the video and did for **-8** because **8** was giving error. Also, did hit **try out** in the **PUT88



- Add WindowController
- Read the previous examples and create
  - a DTO WindowDto and the REST service WindowController
  - a rest service which is able to
    - Retrieve a window list via a GET
    - Retrieve a particular window via a GET
    - Create or update a window via a POST
    - Update a window and switch its status via a PUT
    - Delete a window via a DELETE
- Use swagger to test your API
  - create a new window
  - list all the window
  - find the window with id -8
  - switch its status
  - deletes this window

More Rest service

You can now create BuildingDto, RoomDtoo, HeaterDto and write services which follow this service

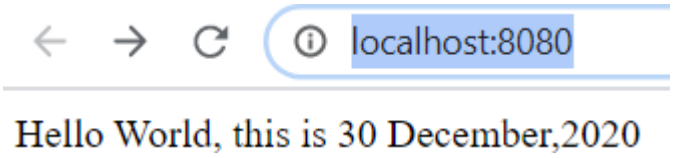
```
/api/heaters (GET) send heaters list
/api/heaters (POST) add a heater
/api/heaters/{heater_id} (GET) read a heater
/api/heaters/{heater_id} (DELETE) delete a heater
```

```
/api/rooms (GET) send room list
/api/rooms (POST) add a room
/api/rooms/{room_id} (GET) read a room
/api/rooms/{room_id} (DELETE) delete a room and all its windows and its heaters
/api/rooms/{room_id}/switchWindow switch the room windows (OPEN to CLOSED or inverse)
/api/rooms/{room_id}/switchHeaters switch the room heaters (ON to OFF or inverse)
```

```
/api/buildings (GET) send building list
/api/buildings (POST) add a building
/api/buildings/{building_id} (GET) read a building
/api/buildings/{building_id} (DELETE) delete a building and all its rooms and all its windows and heaters
```

CORS

- Today browsers forbid a website to access to resources served by another website defined on a different domain. If you want to call your API on http://localhost:8080 from a webapp you should have this error
- on http://localhost:8080, got below



- on http://localhost:8080/api/rooms, got below error

- CORS usage is required because of below

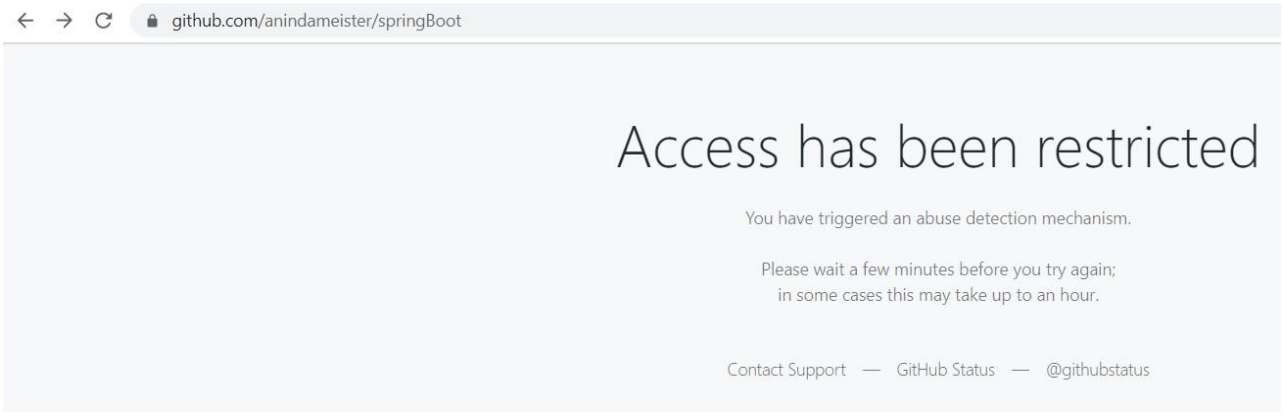
- I just recalled that, I dont have anything in regards to RoomController, just like the above WindowController. I believe, DevMind is asking us to create that RoomController first without the CORS annotation=>visualise the error=>then use Cors annotation and check again
- so created RoomController.java under controller package

```
package com.emse.spring.faircorp.controller;

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

@RestController
public class RoomController {
}
```

- restarting the app and would check the links, http://localhost:8080 and http://localhost:8080/api/rooms
- BTW, I have not restarted the app with the entryClassOfSpring because of my recalling of the JSP's jasper thingy. Similarly, I have implemented additionally, swagger, like the way I tried for jasper.
- now, the links contains the same exact output with no changes, gonna use the Cors annotation now for the RoomController
- No change, even with the Cors annotation, the final program is as follows

```
package com.emse.spring.faircorp.controller;

import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.RestController;

@CrossOrigin
@RestController
public class RoomController {
}
```

- Right now, I have github error, hahahahahaha



Write unit tests in Java

- In this course you will learn how to write a unit test, how to simulate the collaborator behaviors and how to check the results

Software testing

- checks if the actual results match the expected results
- but also
  - helps to identify errors by testing limits
  - helps to not reproduce errors
  - When a bug occurs, we create a new test case, we fix the bug.. And after each code update, we execute this tests to know if the bug won't occur anymore
- You can test your software manually
  - but you have to do that before each feature update
  - and more your application is rich more you need to do more tests
- The solution is to have automatic tests and code them

You have different types of tests

- Installation testing: A software is often a set of little apps (web app, spring boot app, datasource….). This kind of test helps to check if your installation procedure is correct and if the software can be used
- Security testing: Checks the security and if your data is confidential and not available from hackers

- Performance testing: to determine how a system or sub-system performs in terms of responsiveness and stability under a particular workload
- End to end testing: You test your app as a user. These tests are sometimes called functional tests
- Unit testing: We test every units of source code (each class, each methods…).
- Manual tests are cheaper on short-term but more expensive on long-term.Automated tests are expensive on short-term but cheaper on long-term.A human will tire when he has to execute the same tests continuously. He will be less conscience and less attentive. It’s not the case for a test program
- If you want to facilitate your tests you can apply 2 rules
    - use interface to define the contract to code
    - use dependency injection. This mechanism helps to use for example
        - a mock object, to simulate the object behavior in a test and
        - the real implementation in production code

🔗 Unit tests

- A unit test is a method that instantiates a small portion of your application (one method for example) and checks its behavior independently from other parts.
- Portion to test, can be viewed as an independent system. We talk about System Under Test (SUT)
- For example, if we want to test a service which follows this contract

```java
public interface FriendService {
    /**
     * Compute friend age from his birth year
     */
    int computeFriendAge(Friend friend);
}
```

Implementation to test is

```java
public class FriendServiceImpl implements FriendService {
    @Override
    public int computeFriendAge(Friend friend) {
        if(friend == null){
            throw new IllegalArgumentException("Friend is required");
        }
        return LocalDate.now().getYear() - friend.getBirthYear();
    }
}
```

- When you write a test you have to test all the cases. In our example you have to check when the user is nulland when a user is defined and has a birth year
- In an application this system SUT will interact with other components
- These other components are called collaborators.
- For example if we change our service

```java
public class FriendServiceImpl implements FriendService {

    private FriendRepository friendRepository;
    private IntegerComputer integerComputer;

    public FriendServiceImpl(FriendRepository friendRepository,
                             IntegerComputer integerComputer) {
        this.friendRepository = friendRepository;
        this.integerComputer = integerComputer;
    }


    // ...
}
```

- FriendRepository and IntegerComputer are collaborators
- When you want to write a test of your SUT, you need to simulate the collaborator behaviors.
- To simulate collabators, you have several possibilities
    - Use a fake object: You create an object only for your test (it's not a good solution)
    - Use a spy object: You create a spy from the the real implementation of one collaborator. You use a library for that and you can overrided the returned values
    - Use a mock object: A mock is created via a library from a specified contract (an interface). And you can pre preprogrammed these objects to return your wanted values during the test
- Black box
- When you want to write a test you have to consider this SUT (system under test) as a black box.
- The code to test is not important, it's the black box… you have to focus on inputs and outputs

- Your black box can have inputs (method parameters for example) In your test you will invoke the SUT and you test this one by sending inputs
- Your black box can return a result or update the system state (we have an output)
- In your test you will check the result and assert if this result is equals to the expected behavior
- When you write you can follow the AAA pattern : arrange /act /assert
- Another pattern is Given / When / Then
- We use Junit to write tests in Java

```java
package com.devmind.testinaction.service;

import com.devmind.testinaction.model.Friend;
import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class FriendServiceTest {

    private FriendService friendService;

    @BeforeEach // 1.
    public void init(){
        friendService = new FriendServiceImpl();
    }

    @Test // 2.
    public void computeFriendAge() {
        // Arrange
        Friend friend = new Friend("Guillaume", 1977);

        // Act
        int age = friendService.computeFriendAge(friend);

        // Assert
        Assertions.assertThat(age).isEqualTo(42); // 3.
    }

    @Test
    public void computeFriendAgeWithNullFriendShouldFail() { // 4.
        Assertions.assertThatThrownBy(() -> friendService.computeFriendAge(null))
                .isExactlyInstanceOf(IllegalArgumentException.class)
                .hasMessage("Friend is required");
    }
}
```

1. Method annotated with @BeforeEach is executed before each tests (a @AfterEach exists)
2. Method annotated with @Test is a unit test.
3. We use assertJ to write assertions
4. We expect an exception when friend is null. It's important to use an explicit test method name

Assertions

Assertions methods provided by Junit are not very readable. We prefer to use the AssertJ library

AssertJ provides a fluent API and with this API you always use the method `asserThat`

```
Assertions.assertThat(age).isEqualTo(41);
Assertions.assertThat(name).isEqualTo("Dev-Mind");
```

With assertJ you can test the exception thrown by a method, its type, its message

```
Assertions.assertThat(age).isEqualTo(41);
Assertions.assertThat(name).isEqualTo("Dev-Mind");
```

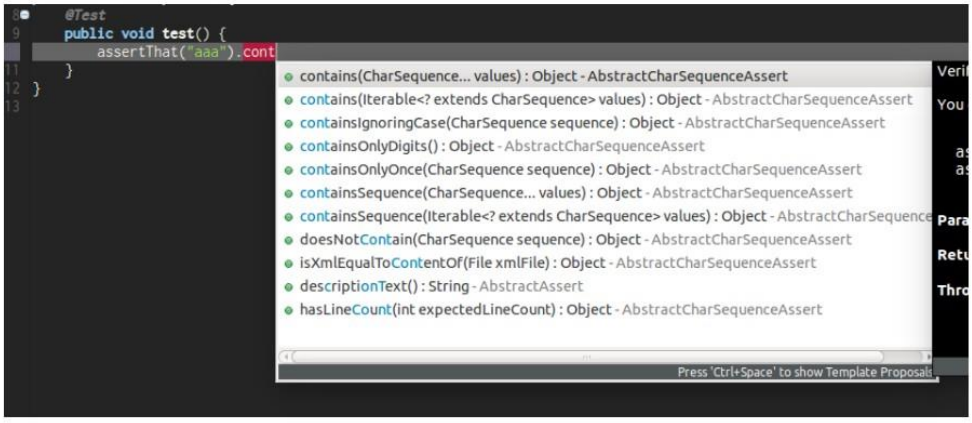
If you expected result is a list of friends

```
List<Friend> myFriends = Arrays.asList(
                new Friend("Elodie", 1999),
                new Friend("Charles", 2001));
```

you can check the content of this list

```
Assertions.assertThat(myFriends)
                  .hasSize(2)
                  .extracting(Friend::getName)
                  .containsExactlyInAnyOrder("Elodie", "Charles");

Assertions.assertThat(myFriends)
          .hasSize(2)
          .extracting(Friend::getName, Friend::getBirthYear)
          .containsExactlyInAnyOrder(
                  Tuple.tuple("Elodie", 1999),
                  Tuple.tuple("Charles", 2001));
```

AssertJ is IDE friendly and its fluent API can be discovered by completion



You can find more informations on the official website **http://joel-costigliola.github.io/assertj**

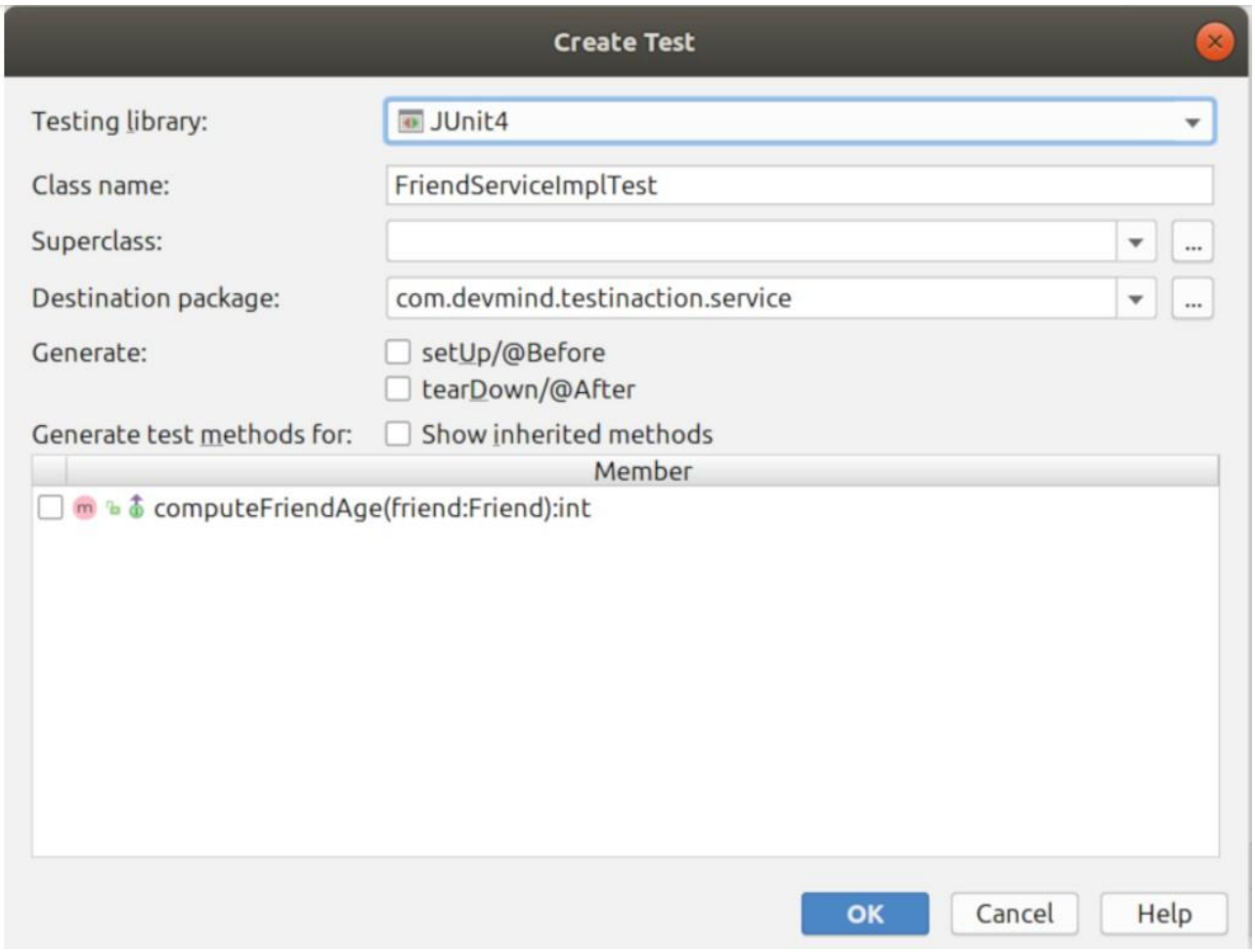
Lab 5 : Unit tests

- You need to clone a new project in your workspace. Use a terminal and launch

```
git clone https://github.com/Dev-Mind/unitTestInAction.git
```

- This project is a Gradle project. You can Open it in IntelliJ and configure it
- Go on FriendServiceImpl and generate a test class with

```
Ctrl + Shift + T
```



- Write the test of the method computeFriendAge

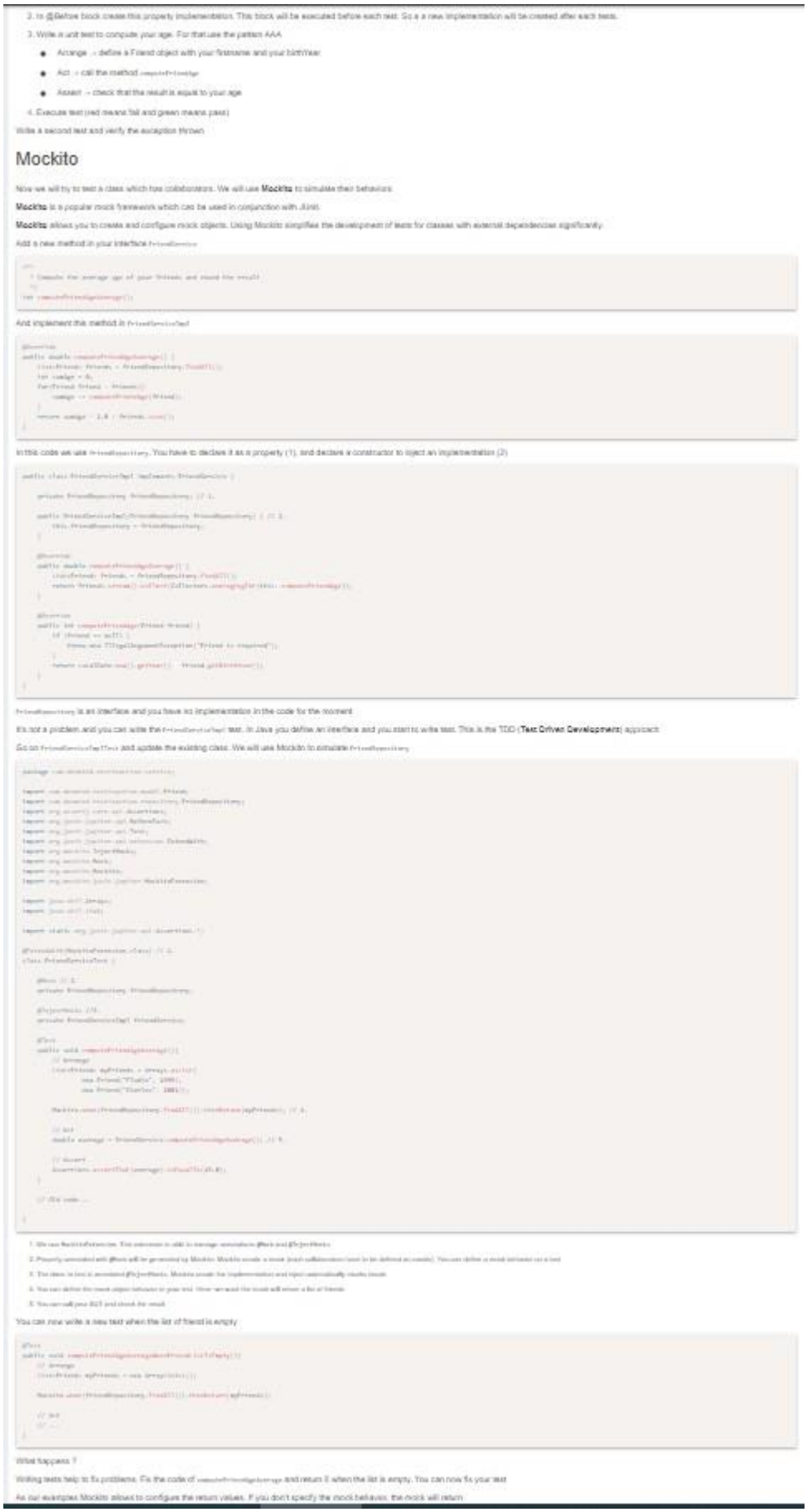
```
package com.devmind.testinaction.service;

import static org.junit.jupiter.api.Assertions.*;

class FriendServiceImplTest {
    void computeFriendAge(){

    }


}
```

- Declare a property of type FriendRepository

```
package com.devmind.testinaction.service;

import com.devmind.testinaction.repository.FriendRepository;

import static org.junit.jupiter.api.Assertions.*;
```

```
class FriendServiceImplTest {

    private FriendRepository friendRepostiory;

    void computeFriendAge(){

    }

}
```

- In @Before block create this property implementation. This block will be executed before each test. So a a new implementation will be created after each tests.



2. In @Before block create this property implementation. This block will be executed before each test. So a a new implementation will be created after each tests.
3. Write a unit test to compute your age. For that use the pattern AAA
   - Arrange → define a Friend object with your firstname and your birthYear
   - Act → call the method computeFriendAge
   - Assert → check that the result is equal to your age
4. Execute test (red means fail and green means pass)

Write a second test and verify the exception thrown

Mockito

Now we will try to test a class which has collaborators. We will use **Mockito** to simulate their behaviors.

Mockito is a popular mock framework which can be used in conjunction with JUnit.

Mockito allows you to create and configure mock objects. Using Mockito simplifies the development of tests for classes with external dependencies significantly.

Add a new method in your interface FriendService

```
/**
 * Compute the average age of your friends and round the result
 */
int computeFriendAgeAverage();
```

And implement this method in FriendServiceImpl

```
@Override
public double computeFriendAgeAverage() {
    List<Friend> friends = friendRepository.findAll();
    int sumAge = 0;
    for(Friend friend : friends){
        sumAge += computeFriendAge(friend);
    }
    return sumAge / 2.0 / friends.size();
}
```

In this code we use FriendRepository. You have to declare it as a property (1), and declare a constructor to inject an implementation (2)

```
public class FriendServiceImpl implements FriendService {

    private FriendRepository friendRepository; // 1.

    public FriendServiceImpl(FriendRepository friendRepository) { // 2.
        this.friendRepository = friendRepository;
    }

    @Override
    public double computeFriendAgeAverage() {
        List<Friend> friends = friendRepository.findAll();
        return friends.stream().collect(Collectors.averagingDouble(this::computeFriendAge));
    }

    @Override
    public int computeFriendAge(Friend friend) {
        if (friend == null) {
            throw new IllegalArgumentException("Friend is required");
        }
        return Calendar.now().getYear() - friend.getBirthYear();
    }
}
```

FriendRepository is an interface and you have no implementation in the code for the moment.

It's not a problem and you can write the FriendServiceImplTest test. In Java you define an interface and you start to write test. This is the TDD (**Test Driven Development**) approach.

Go on FriendServiceImplTest and update the existing class. We will use Mockito to simulate FriendRepository

```
package com.octest.instruction.service;

import com.octest.instruction.model.Friend;
import com.octest.instruction.repository.FriendRepository;
import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.junit.jupiter.MockitoExtension;

import java.util.Arrays;
import java.util.List;

import static org.junit.jupiter.api.Assertions.*;

@ExtendWith(MockitoExtension.class) // 1.
class FriendServiceTest {

    @Mock // 2.
    private FriendRepository friendRepository;

    @InjectMocks // 3.
    private FriendServiceImpl friendService;

    @Test
    public void computeFriendAgeAverage() {
        // Arrange
        List<Friend> myFriends = Arrays.asList(
                new Friend("Claire", 1990),
                new Friend("Clarisse", 1985));

        Mockito.when(friendRepository.findAll()).thenReturn(myFriends); // 4.

        // Act
        double average = friendService.computeFriendAgeAverage(); // 5.

        // Assert
        Assertions.assertThat(average).isEqualTo(29.0);
    }

    // Write code ...
}
```

1. We use MockitoExtension. This extension is able to manage annotations @Mock and @InjectMocks
2. Property annotated with @Mock will be generated by Mockito. Mockito create a mock (a fake collaborator) (we'll se to define it in a test). You can define a mock behavior on a test
3. The class to test is annotated @InjectMocks. Mockito create the implementation and inject automatically mocks inside
4. You can define the mock output behavior in your test. Here we want the mock will return a list of friends
5. You can call your SUT and check the result

You can now write a new test when the list of friend is empty

```
@Test
public void computeFriendAgeAverageWhenFriendListIsEmpty() {
    // Arrange
    List<Friend> myFriends = new ArrayList<>();

    Mockito.when(friendRepository.findAll()).thenReturn(myFriends);

    // Act
    // ...
}
```

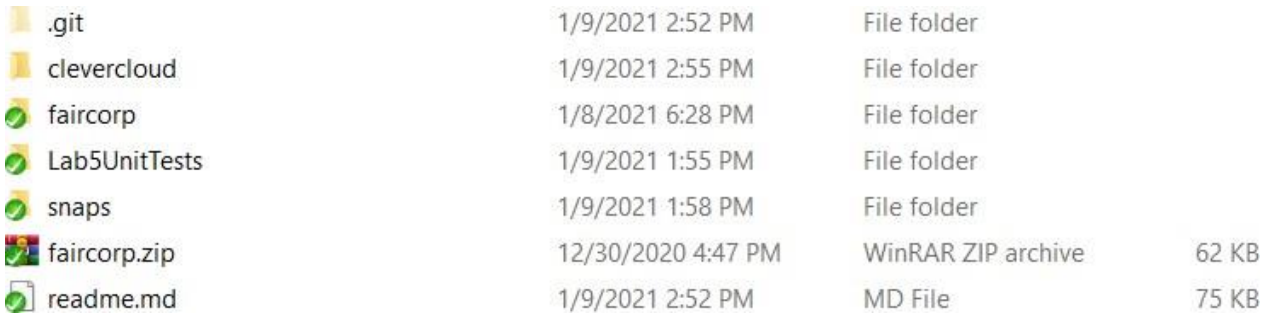
What happens ?

Writing tests help to fix problems. Fix the code of computeFriendAgeAverage and return 0 when the list is empty. You can now fix your test

As our examples Mockito allows to configure the return values. If you don't specify the mock behavior, the mock will return

Deploy your webapp on Clever Cloud

- Clever cloud is a platform designed by developers for developers.
- You Write Code. They Run It.
- When you write code, you push sources on Github. Clever Cloud is able to install your Github app
- Sign in on https://github.com/ with your account. If you haven't an account you have to create one
- When you are connected, you can create a new account on https://www.clever-cloud.com/en/
- The task of excepting the invitation from devMind and connecting my github account to clever clud has been done.
- Before this deployment you have to create a folder called "clevercloud" in your Spring Boot project
- did the above



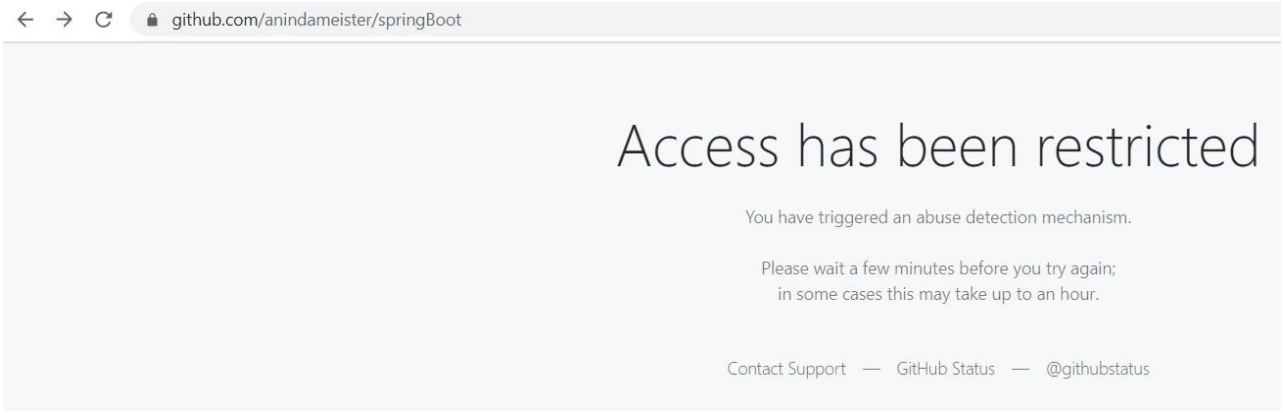
.git	1/9/2021 2:52 PM	File folder	
clevercloud	1/9/2021 2:55 PM	File folder	
faircorp	1/8/2021 6:28 PM	File folder	
Lab5UnitTests	1/9/2021 1:55 PM	File folder	
snaps	1/9/2021 1:58 PM	File folder	
faircorp.zip	12/30/2020 4:47 PM	WinRAR ZIP archive	62 KB
readme.md	1/9/2021 2:52 PM	MD File	75 KB

- Inside, add a file gradle.json. This file will contain

Inside, add a file `gradle.json`. This file will contain

```
{
  "build": {
    "type": "gradle",
    "goal": "assemble"
  },
  "deploy": {
    "jarName": "./build/libs/faircorp-0.0.1-SNAPSHOT.jar"
  }
}
```

- To know the name of your jar go on folder **./build/libs/.**
- found that part
- Push your last changes on your Github repository. For the moment, Clevercloud used only Github
- got the githubBehaviourMessage again



- got the above fixed
- attempted to deploy but failed

If you are on Windows you can have this error

```
2020-11-23T13:58:00+01:00 A gradlew script has been found. Let's use it.
2020-11-23T13:58:01+01:00 /home/bas/rubydeployer/scripts/build-java.sh: line 9: ./gradlew: Permission denied
2020-11-23T13:58:01+01:00 Build failed
```

- realised that it's an error because am using windows
- followed the below steps in gitbash and fixed the error

```
git ls-files  --stage
git update-index --chmod=+x gradlew
git ls-files --stage
git commit -m "made a file executable"
git push
```

- found a beautiful document on the way
- the above document gives some knowledge about windows as compared to linux
- after deployment was done, did go to the page to see the app and it was there. Need to check out the REST part later.

Kotlin for a Java developer (EN)

- In few words Kotlin is
    - Concise Drastically reduce the amount of boilerplate code
    - Interoperable Leverage existing libraries for the JVM, Android, and the browser. You can call Kotlin code in Java or Java code inKotlin
    - Safe Kotlin tries to help you reduce errors such as null pointer exceptions.
- Executable class
    - An executable Java class is a class which, when handed over to the JVM, starts its execution at a particular point in the class, the main method.
- For example

```
package yeahKotlin;

public class HelloWorldApplication {
    public static void main(String[] args) {
        String name = "Guillaume";
        System.out.println("Hello EMSE I am " + name);
    }
}
```

- In IntelliJ you can use the contextual menu (right click) to run this class and see the result in console

```
Hello EMSE I am Guillaume
```

- With Kotlin you can write to produce the same result

```
package yeahKotlin

fun main(args: Array<String>) {
    val name = "Guillaume"
    println("Hello EMSE I am $name")
}
```

-Simplest version

```
fun main() {
    val name = "Guillaume"
```

```
    println("Hello EMSE I am  $name")
}
```

- You can write functions not attached to a class (the compiler will do itfor you)
- The public visibility is the default in Kotlin and therefore no need to define it each time
- Semicolons are no longer necessary
- Kotlin does a lot of type inference (the compiler tries to guess which type you are using) and you don’t need to define the type if the compiler can infer it (example of the name or you don’t need to specify the type String)
- You can use String templates and directly access the content of a variable with $
- If you want to test Kotlin code in your browser you can use https://play.kotlinlang.org

Types

Kotlin use basic types. The most used are

- Integer numbers : Int (Integer in Java), Long
- Floating-point number : Double, Float
- String
- Boolean
- Arrays
- Collections : List, Set, Map…

Immutability

- Kotlin forces you to use immutability when you develop. An immutable object is an object whose state cannot be modified after it is created. It allows you to write safer and cleaner code.
- When you want to declare a variable you can use the keyword val. We did that in our first example

```
val name = "Guillaume"
```

- When the value is defined you can’t update it. With the code below, the compiler will fail with an Error "Val cannot be reassigned".

```
name = "Someone else"
```

- If you need to reassign the value you can use keyword var

```
var name = "Guillaume"
name = "Someone else"
```

- Collections (List, Set, Map...) are also immutable in Kotlin. The code below will fail because type List is immutable and method add does not exist
- Let's try to understand from a different example

```
fun main(){
    var list = listOf("Ajay","Vijay","Prakash")//read only, fix-size
    for(element in list){
        println(element)
    }
}
```

- output

```
Ajay
Vijay
Prakash
```

```
val rooms: List<Room> = listOf()
rooms.add(Room(1, "Room1"))
```

- Am not really able to comprehend the above example

```
fun main(){
    var rooms = listOf("Room-A","Room-B","Room-C")//read only, fix-size
    rooms.add("Room1")
}
```

- output

```
Unresolved reference: add
```

- When you want a mutable collection you have dedicated types

```
val rooms: MutableList<Room> = mutableListOf()
rooms.add(Room(1, "Room1"))
```

- my example1

```
fun main(){
    var rooms = mutableListOf("Room-A","Room-B","Room-C")//read only, fix-
size
    rooms.add("Room1")
    print(rooms)
}
```

- output

```
[Room-A, Room-B, Room-C, Room1]
```

- my example 2

```
fun main(){
    var rooms = mutableListOf("Room-A","Room-B","Room-C")//read only, fix-
size
    rooms.add(1,"Room1")
    print(rooms)
}
```

- output

```
[Room-A, Room1, Room-B, Room-C]
```

- Nullability
- One of the most common pitfalls in many programming languages, including Java, is that accessing a member of a null reference will result in a null reference exception. Kotlin's type system is aimed at eliminating the danger of null references from code.

```
var a: String = "abc" // Regular initialization means non-null by default
a = null // compilation error
```

- example

```
fun main(){
    var b: String = "abc"
        b = null
    print(b)
}
```

- the above gives big sized error
- In Kotlin, the type system distinguishes between references that can hold null (nullable references) and those that can not (non-null references). To allow nulls, we can declare a variable as nullable string, written String?:

```
var b: String? = "abc" // can be set null
b = null // ok
```

- example

```
fun main(){
    var b: String? = "abc" // can be set null
        b = null // ok
    print(b)
}
```

- output

```
null
```

- When you want declare a nullable value add ? to the type
- For more details read this article
- Functions
- Function declarations
- A function is define with the keyword fun. In Kotlin. Arguments args, returned type are always after For example

```
fun double(x: Int): Int {
    return 2 * x
```

```
}
```

- You can call this function

```
val result = double(2)
```

- above example has been executed and it was asking for a main method within that kotlinPlayGround

```
fun main(){
    fun double(x:Int):Int{
    return 2 * x
}
    val result = double(2)
    print(result)
}
```

- output

```
4
```

- Default arguments
- You can use default argument in Kotlin. For example:

```
fun double(x: Int = 4): Int {
    return 2 * x
}

double(2) // returns 4
double() // returns 8 (the default value is applied)
```

- let's look at this funny example

```
fun main(){
    fun double(x:Int=4):Int{
    return 2 * x
}
    val result1 = double(2)
    val result2 =double()

    print(result1)
```

```
    print(result2)
}
```

- output

```
48
```

- introucing the below

```
 print("****")
```

- output

```
4****8
```

- fixed by using

```
println(double(2));
    println(double());
```

- Named arguments
- When calling a function, you can name one or more of its arguments. This may be helpful when a function has a large number of arguments

```
fun foo(bar: Int = 0, baz: Int) : Int { /*...*/ }
val result = foo(baz = 4)
```

- we'll be going step bystep

```
fun main(){
    fun foo(bar: Int, baz: Int) : Int
    { val bay:Int;
        bay=bar+baz
    return bay}
    print(foo(2,4))

}
```

- output

```
6
```

- check this out

```
fun main(){
    fun foo(bar: Int= 0, baz: Int) : Int
    { val bay:Int;
        bay=bar+baz
    return bay}
    print(foo(baz = 4))

}
```

- output

```
4
```

https://dev-mind.fr/training/android/android-first-app.html

- chose Empty Activity and followed the instructions in the above url
- Moving on
- Manifest file

```
File : app > manifests > AndroidManifest.xml
```

- Manifest file is a kind of id card for your project.
- The manifest file describes essential information about your app to the Android build tools, the Android operating system, and Google Play.
- All activities must be defined inside and one of them will be defined as entry point for your app (with an intent filter).

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.faircorp">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

- Activity
- You can see 3 packages named **com.faircorp** in Android view.
  - The first one (not suffixed) contains all your Kotlin files used to write your app and our first activity
  - The second (suffixed with androidTest) contains test files executed to test your app on a device or on an emulator.
  - The last one (suffixed with test) contains unit test files used to control your code locally at each build
- Unfortunately we don't have enough time to see how to write these tests. But be aware that if you want to create a sustainable application, testing is the best way to limit regressions and make it easier to manage your application over time.

```
File : app > java > com.faircorp > MainActivity
```

- This is the main activity. It's the entry point for your app. When you build and run your app, the system launches an instance of this Activity and loads its layout.
- Each activity (as each components in Android) has a lifecyle and you can interact at each step

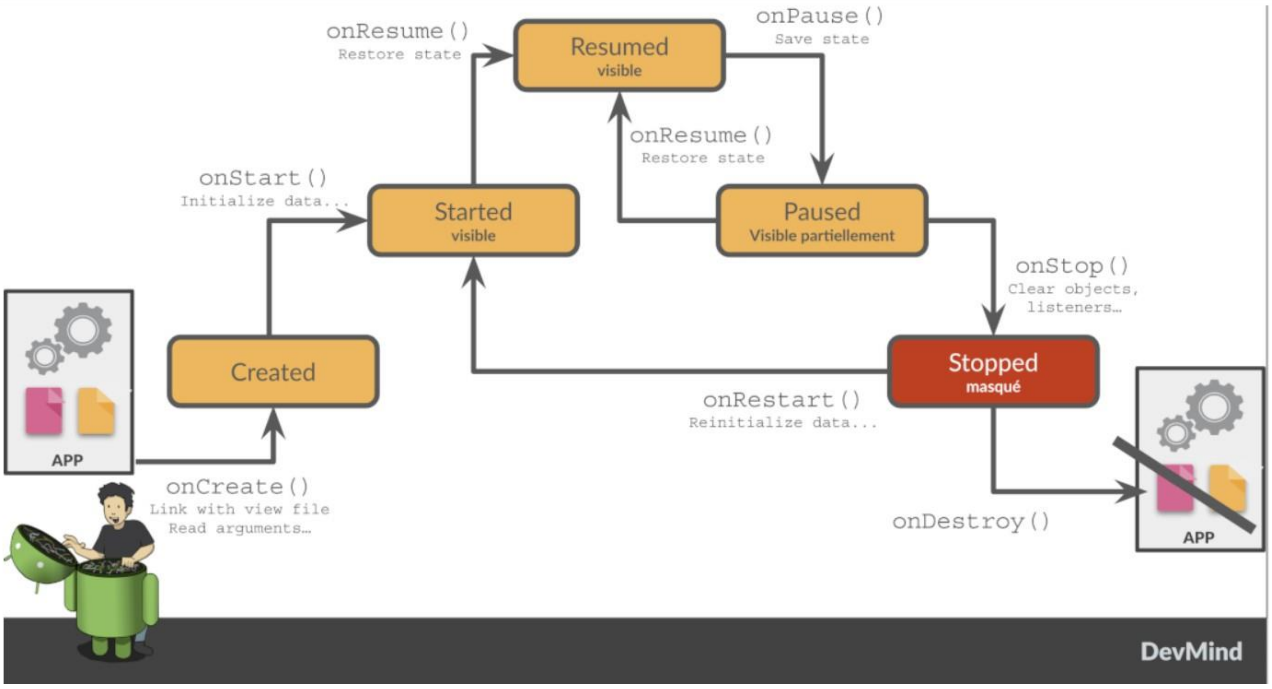
- For example in MainActivity, we declare the XML resource file where your view content is defined (R.layout.activity_main)
- R.layout.activity_main

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</android.support.constraint.ConstraintLayout>
```

- MainActivity.kt

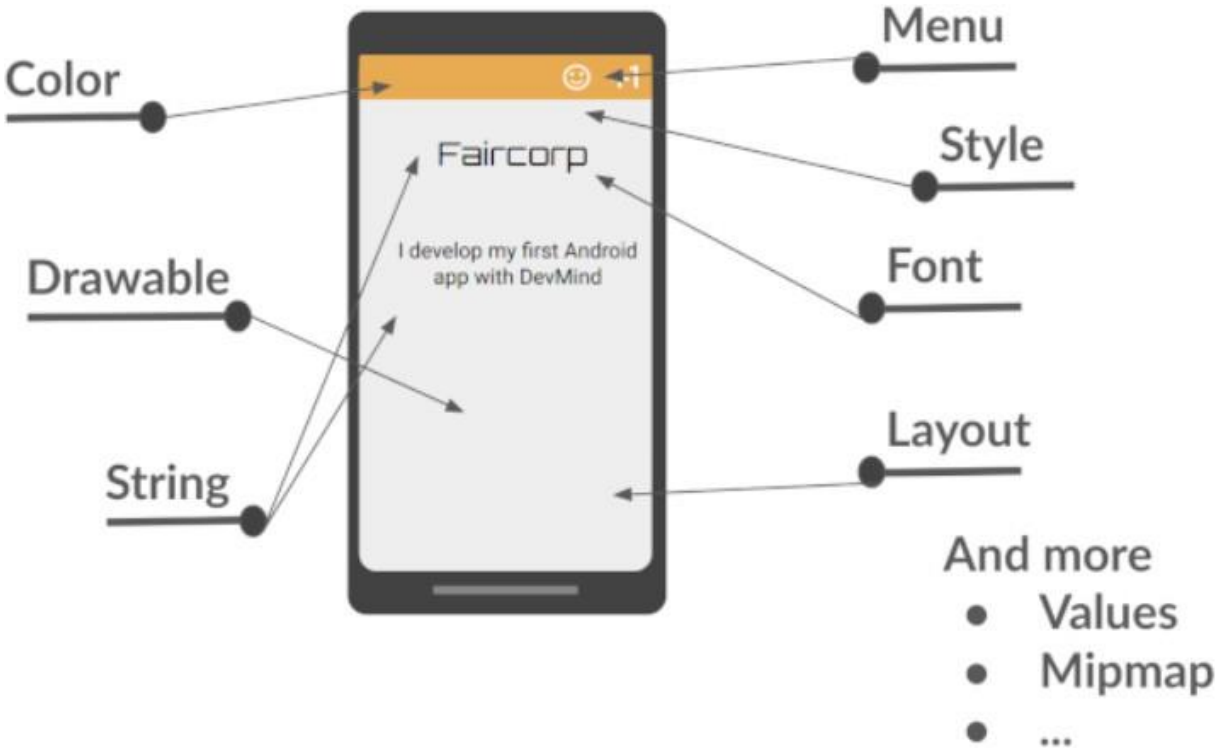
```
package com.faircorp

import android.support.v7.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

- NOTE : directory is named java to assure compatibility with old projects or libs written in Java but don’t be afraid we will use Kotlin :-)



Resource files

- Resources are the additional files and static content that your code uses, such as images, screen definitions, strings used in interfaces, styles, animation instructions, and more.



- You can provide alternative resources for specific device configurations, by grouping them in specially-named resource directories.
- **At runtime, Android uses the appropriate resource based on the current configuration.**
- **For example, you might want to provide a different UI layout depending on the screen size or different strings depending on user language.**

```
File : app > res > layout > activity_main.xml
```

- Here we use a constraint layout. It contains a TextView element with the text "Hello Aninda, Life is Beautiful!"

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello Aninda, Life is Beautiful!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</android.support.constraint.ConstraintLayout>
```

- We will see later how to update or create a new layout and include inside widgets

Gradle file

- File : Gradle Scripts > build.gradle
- There are two files with this name:
    - one for the project, Project: Faircorp, and
    - one for the app module, Module: app.

- Each module has its own build.gradle file, but this project currently has just one module.
- If you need to use external libraries you will add them in build.gradle (Project: Faircorp)
- If you want to configure android plugin (APi version, SDK version) you will update build.gradle (Module: app)
- Project: Faircorp

```
// Top-level build file where you can add configuration options common to
all sub-projects/modules.
buildscript {
    ext.kotlin_version = "1.4.20"
    repositories {
        google()
        jcenter()
    }
    dependencies {
        classpath "com.android.tools.build:gradle:4.0.0"
        classpath "org.jetbrains.kotlin:kotlin-gradle-
plugin:$kotlin_version"

        // NOTE: Do not place your application dependencies here;  they
belong
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
        google()
        jcenter()
    }
}

task clean(type: Delete) {
    delete  rootProject.buildDir
}
```

- Module: app

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
```

```
android {
    compileSdkVersion 30
    buildToolsVersion "30.0.1"

    defaultConfig {
        applicationId "com.faircorp"
        minSdkVersion 26
        targetSdkVersion 30
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner
"android.support.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-
optimize.txt'), 'proguard-rules.pro'
        }
    }
}

dependencies {
    implementation fileTree(dir: "libs",  include:  ["*.jar"])
    implementation "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
    implementation   'com.android.support:appcompat-v7:28.0.0'
    implementation 'com.android.support.constraint:constraint-layout:2.0.4'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.2'
    androidTestImplementation 'com.android.support.test.espresso:espresso-
core:3.0.2'

}
```

Launch your application

- In this part you will be able to launch your application on you phone or tablet. If you don’t have a device on Android operating system, you can use the emulator embedded in Android Studio.
- Configure a real Android device
- You need to set up your phone
    - Connect your device to your development machine with a USB cable. If you developed on Windows, you might need to install USB driver for your device.

        - done
    - You need to update your device to activate "Developer options"
        - done
    - Make sure to run **usb debugging** and another thing, for sure, while inserting usb into android phone, **choose access files and NOT charging only**
- Now you are ready to run your app
    - and it works perfectly on my phone

Analyse errors

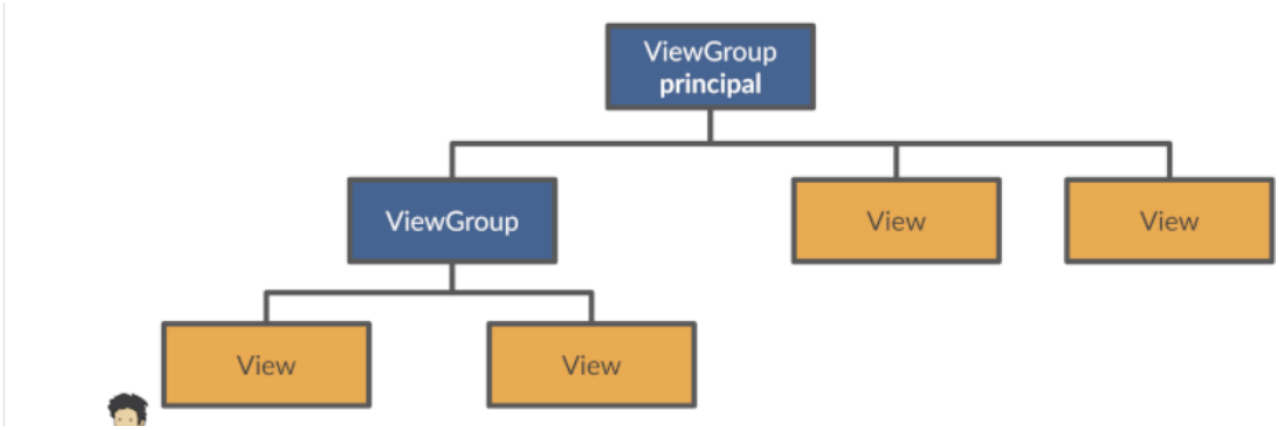
- To analyze errors you can open the run console on the bottom. This window contains messages send when app is launched with Gradle
- You also can open Logcat view to see logs send by your device or the emulated device

https://dev-mind.fr/training/android/android-update-ui.html

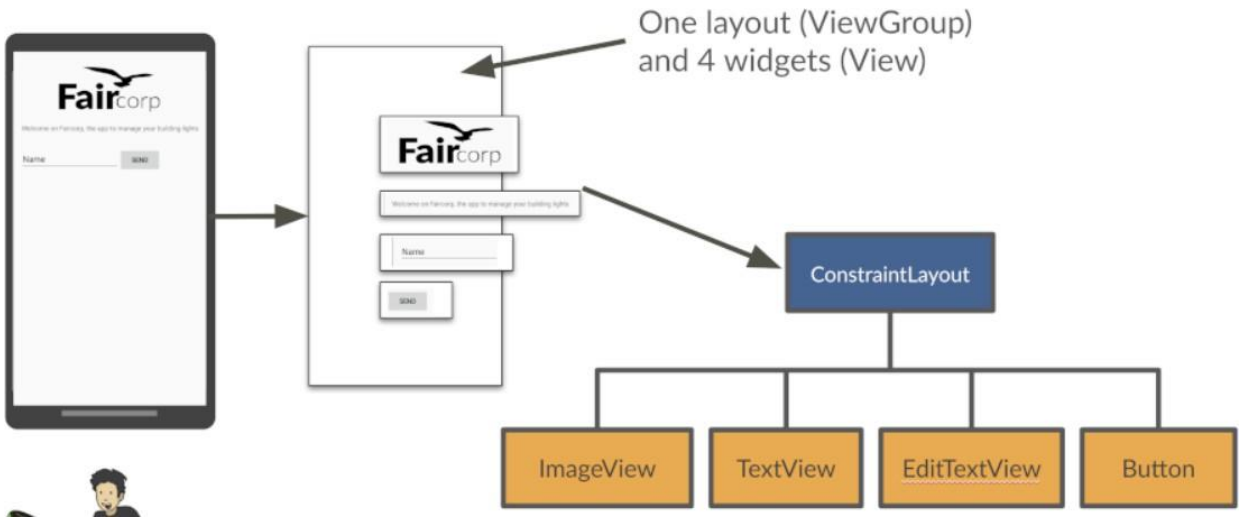
- Update UI and refactor welcome layout
- In this lesson, you will learn how to update a layout with Android Studio

User interface

- User interface for an Android app is built as a hierarchy of layouts and widgets.
    - The layouts are ViewGroup objects, containers that control how their child views are positioned on the screen.
    - Widgets are View objects, UI components such as buttons and text boxes.



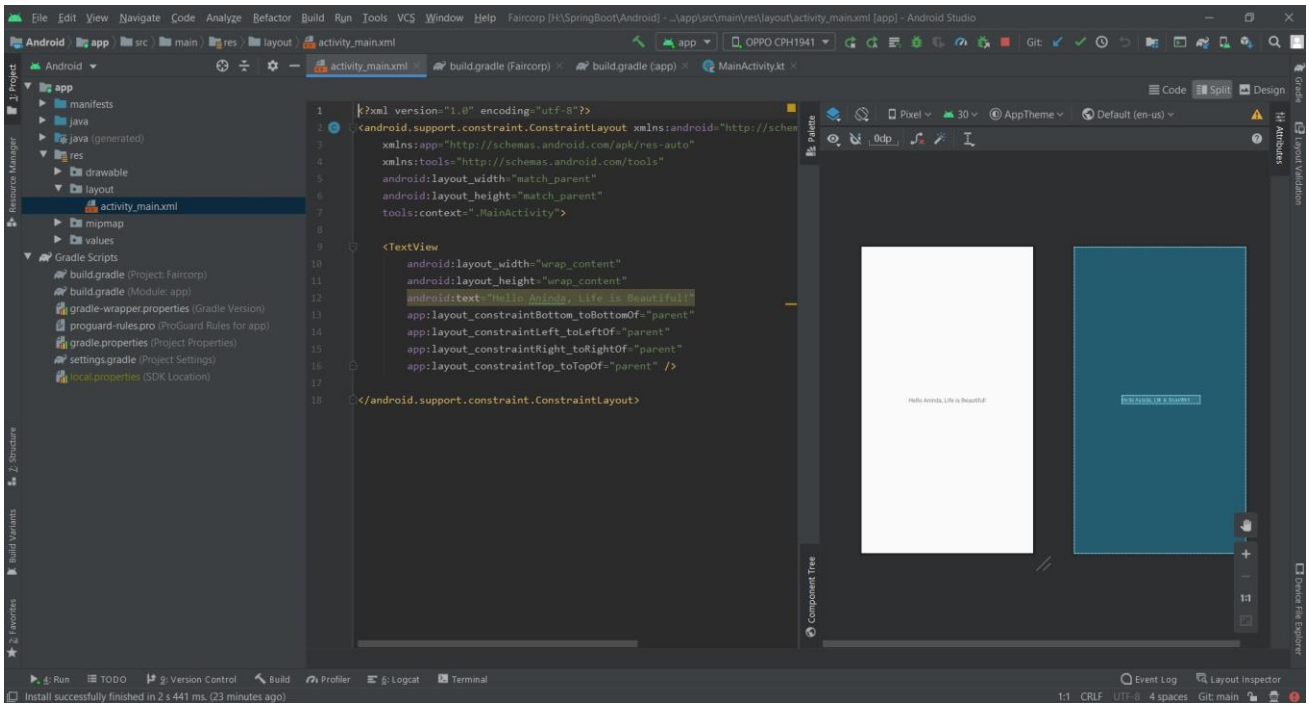
- In this new codelab you will update the hello world page to create a home page with a welcome message, an image, an edit text and a button.



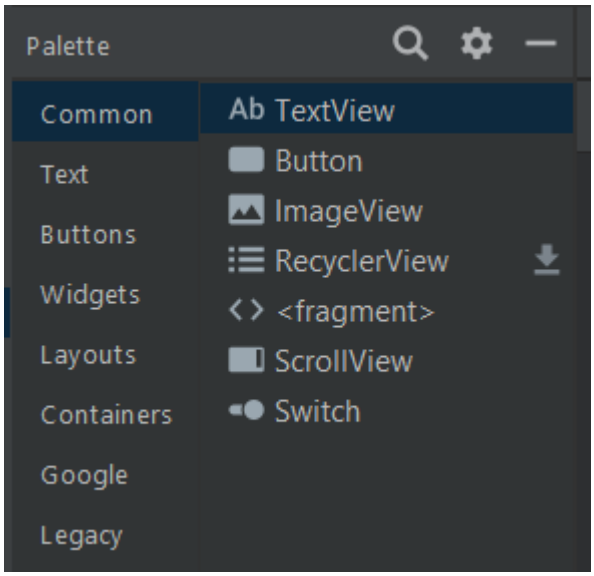
- Android provides an XML vocabulary for ViewGroup and View classes, and your UI is defined in XML files. Don't be afraid Android Studio provide a wysiwyg editor.

Android Studio Layout Editor

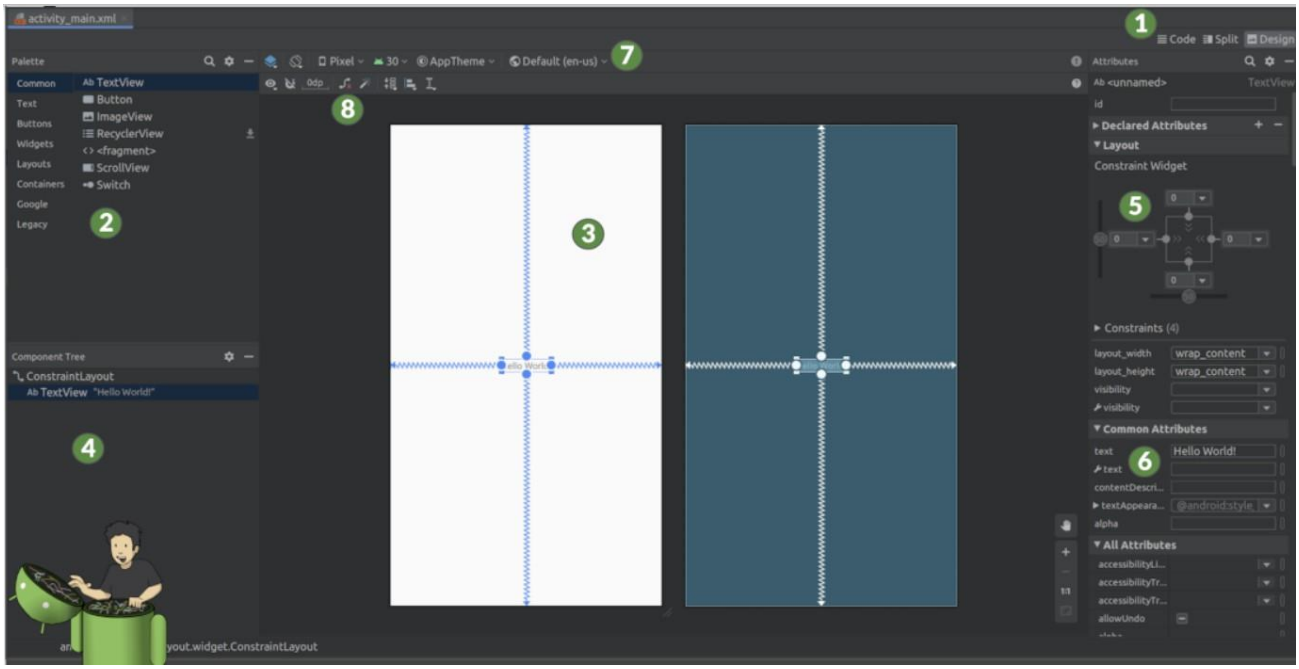
- In the Project window, open app > res > layout > activity_main.xml. Editor should be displayed



1. View mode: View your layout in either code mode (XML editor), Design mode (design view and Blueprint view), or Split mode icon (mix between code and design view)
2. Palette: Contains various views and view groups that you can drag into your layout.



3. Design editor: Edit your layout in Design view, Blueprint view, or both.
4. Component Tree: Shows the hierarchy of components in your layout. It is sometimes useful to select a given widget

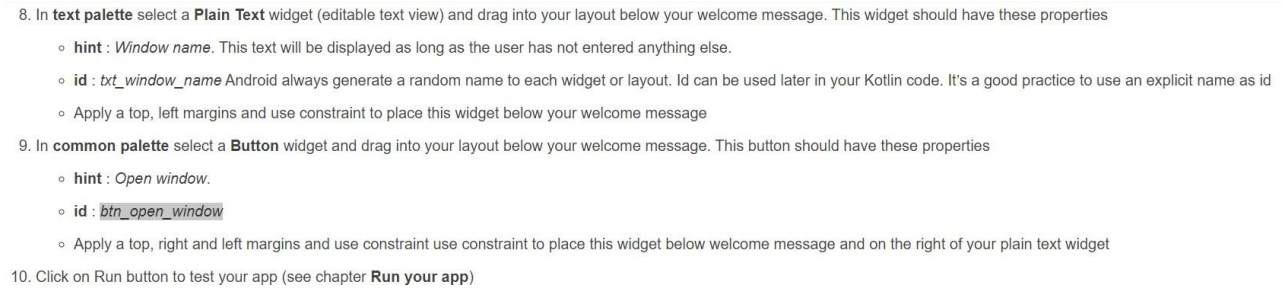


5. Constraint widget : Helps to place an item in relation to those around it
6. Attributes: Controls for the selected widget's attributes.
7. Layout Toolbar: Click these buttons to configure your layout appearance in the editor and change layout attributes as target phone, orientation, light, locale…
8. Widget Toolbar: Click these buttons to align your view. Button with red cross is useful to clear all widget constraints

Update home page

- For the moment our page contains only one readonly text field.
  - Select it and delete it with **Suppr key**
    - commented out the code
  - We will add an image. Copy this xml file ic_logo.xml in your directory _res > drawable. This file is a vector drawable image. **Directory drawable contains all your images. Several formats are available (png, jpg…) but the most optimized is a Vector drawable**
    - done that
  - In Common Palette on the left of the screen click on ImageView and drag into your layout. A window is opened to select an image. You will choose the imported image ic_logo.xml
    - done

- Click on OK button to import image in your layout
- We will use the blueprint view to add constraint to this image, to place it on the top of the screen and define a height. See video for more detail
- We will add a new read-only text below image to introduce our app. In common palette select a Textview widget and drag into your layout.
- In blueprint view you can add constraints to this textview
  - text : Welcome on faircorp\n the app to manage building windows
  - layout_width and layout_height : wrap_content
  - textSize: 18 sp
  - gravity : center
  - margin right and left 16dp, margin top 32dp
  - follow video to check out textAlign=Center..
  - Note: you can do anything and everything with the design. Just to note that designing needs a preBluePrintSettingInMindOrPaper



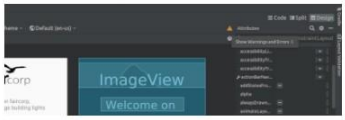
8. In **text palette** select a **Plain Text** widget (editable text view) and drag into your layout below your welcome message. This widget should have these properties
   - **hint** : *Window name*. This text will be displayed as long as the user has not entered anything else.
   - **id** : *txt_window_name* Android always generate a random name to each widget or layout. Id can be used later in your Kotlin code. It's a good practice to use an explicit name as id
   - Apply a top, left margins and use constraint to place this widget below your welcome message
9. In **common palette** select a **Button** widget and drag into your layout below your welcome message. This button should have these properties
   - **hint** : *Open window*.
   - **id** : *btn_open_window*
   - Apply a top, right and left margins and use constraint use constraint to place this widget below welcome message and on the right of your plain text widget
10. Click on Run button to test your app (see chapter **Run your app**)

- check the <a href=:"https://www.youtube.com/watch?v=l4CyARSHtDQ">video to remove confusion

Layout errors and strings

Layout errors and strings

When something is wrong, Android Studio add a warning or an error button on the right of the editor toolbar



1. Click on this button to see different problems (a window is opened on the bottom of your screen).
2. You can double click on an item to see the problem and have an explaination. Android studio display also a Fix button to help you to resolve problem
3. You added a Text Field and a text inside. As your application can be used by different people who speak different languages, you should always use text internalization mechanisms provided by Android.

Open the Project window and open file **app > res > values > strings.xml**. This is a string resources file, where you can specify all of your UI strings. It allows you to manage all of your UI strings in a single location, which makes them easier to find, update, and localize. For the moment you have only one text inside, your app name.

```
<resources>
    <string name="app_name">Faircorp</string>
</resources>
```

You can launch **Translations Editor**, to add or edit text for different languages. In this lab we will use only one language. You can update this file to have a text description for our logo, and the text content for our welcome message

```
<resources>
    <string name="app_name">Faircorp</string>
    <string name="app_logo_description">Faircorp logo</string>

    <string name="act_main_windowname_hint">Ligth name</string>
    <string name="act_main_welcome">Welcome on faircorp,\n the app to manage building windows</string>
    <string name="act_main_open_window">Open window</string>
</resources>
```

You can now update your layout and yours components to add a string reference for image description and welcome message. To make a reference to a String you have to use the prefix **@string/** followed by the string key

- Please watch the video in order to understand what exactly is being asked.

Launch action on button click

- An activity is always associated with a layout file. In Lab 2 we have updated our main activity layout with a logo, a welcome message and a button. In this lesson, you add some code in MainActivity to interact with this button.

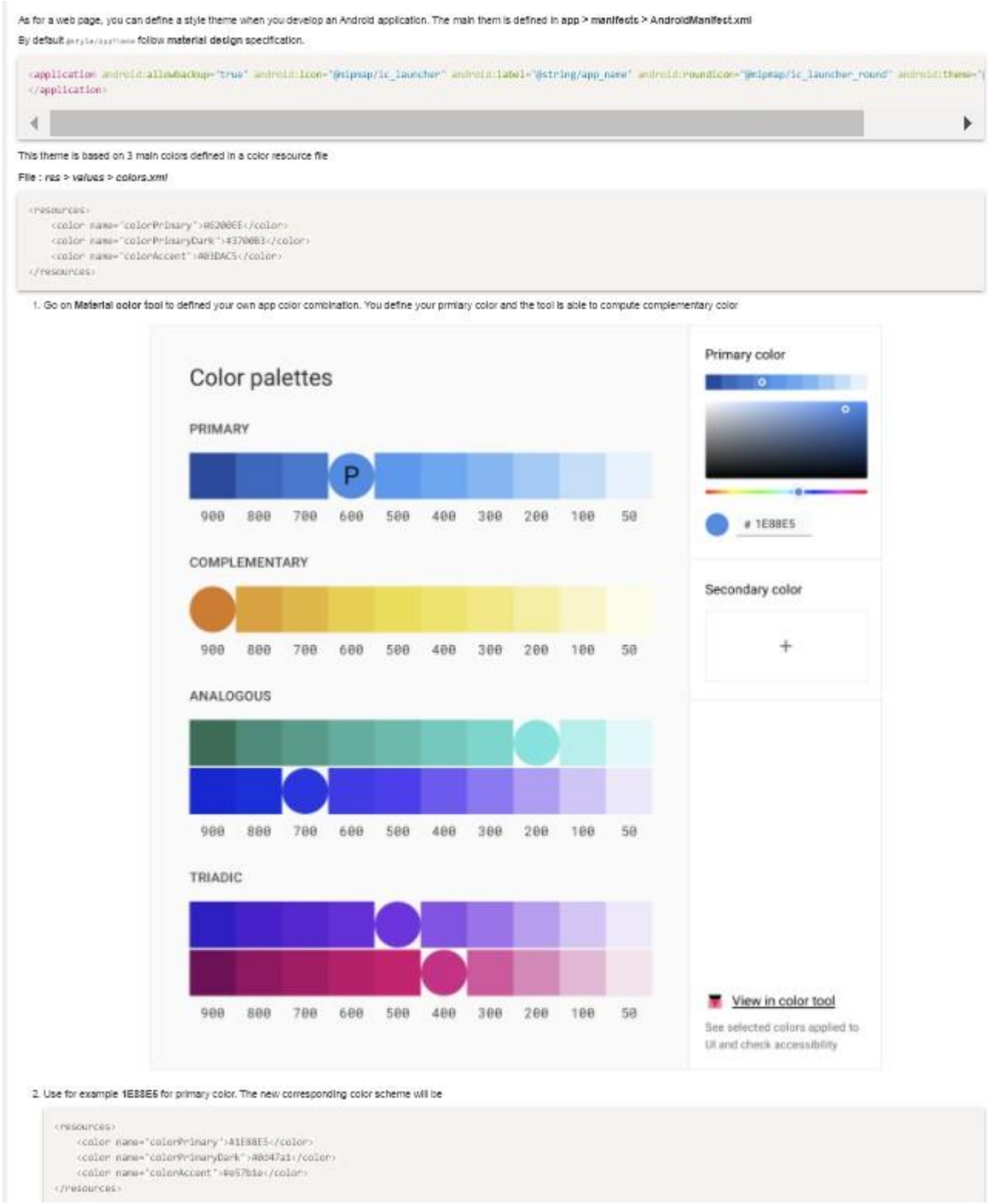
1. In the file app > java > com.faircorp > MainActivity, add the following openWindow() method stub:

```
package com.faircorp

import android.content.Intent
import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import android.view.View
import  android.widget.EditText
import android.widget.Toast



class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
```

```
    /** Called when the user taps the button */
    fun openWindow(view: View) {
        // Extract value filled in editext identified with  txt_window_name
id
        val windowName = findViewById<EditText>
(R.id.txt_window_name).text.toString()
        // Display a message
        Toast.makeText(this, "You choose $windowName",
Toast.LENGTH_LONG).show()

    }
}
```

2. Return to the activity_main.xml file and select the button in the Layout Editor. In Attributes window, locate onClick property and select **openWindow** from its drop-down list, because that's the function we are using
3. You can now relaunch your app,
    - In window name editext fill a name
    - Click on the button you a message should be displayed on the bottom of the screen with the light name filled

Update app color scheme [optional]



- the above worked by just updating the file, res/values/colors.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!--<resources>-->
<!--    <color name="colorPrimary">#6200EE</color>-->
<!--    <color name="colorPrimaryDark">#3700B3</color>-->
<!--    <color name="colorAccent">#03DAC5</color>-->
<!--</resources>-->
```

```
<resources>
    <color name="colorPrimary">#1E88E5</color>
    <color name="colorPrimaryDark">#0d47a1</color>
    <color name="colorAccent">#e57b1e</color>
</resources>
```

https://dev-mind.fr/training/android/android-add-activity.html

Add an activity

Create a new activity

1. In the **Project window**, right-click the app folder and select **New > Activity > Empty Activity**. You can also use menu **File > New > Activity > Empty Activity**
2. In the **Configure Activity window**, enter a name for our new activity : *WindowActivity*. Leave all other properties set to their defaults and click Finish.

   Android Studio automatically does three things:

   - Creates the **WindowActivity** file.
   - Creates the layout file **activity_window.xml**, which corresponds with the **WindowActivity** file.
   - Adds the required **<activity>** element in **AndroidManifest.xml**.
3. We will now update **activity_window.xml** to display a window name. Open this file
4. Add a new TextView with these properties
   - **text** : *Window name*
   - **margin top** 16dp
   - **margin left** 16dp
5. Add a new TextView below the first one with these properties
   - **text** : empty
   - **textAppearance** : *@style/TextAppearance.AppCompat.Large*
   - **margin top** 8dp
   - **margin left** 16dp
   - **id** txt_window_name

- This part seems incomplete without the video

Build an intent to open an activity

An **Intent** is an object that provides runtime binding between separate components, such as two activities. These activities can be in the same app or not. For example if you need to open a web page you won't develop a new browser. You will open this web page in installed browser as Firefox or Chrome.

The **Intent** represents an app's intent to do something. You can use intents for a wide variety of tasks, but in this lesson, your intent starts another activity in the same app.

When you create an Intent you define a context, a target and you can send zero, one or more informations to the target. An Intent can carry data types as key-value pairs called extras. In this lab you will open **WindowActivity** when a user will click on **MainActivity** button **Open Window**

Update method **openWindow** in **MainActivity** to

1. define an Intent
2. target **WindowActivity**
3. put the window name filled in **MainActivity** in the sent attributes (extra). Each extra is identified by a string. It's a good practice to define keys for intent extras with your app's package name as a prefix. This ensures that the keys are unique, in case your app interacts with other apps.

```
package com.faircorp

import android.content.Intent
import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import android.view.View
import android.widget.EditText
import android.widget.Toast


const val WINDOW_NAME_PARAM = "com.faircorp.windowname.attribute"


class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }

//    /** Called when the user taps the button */
//    fun openWindow(view: View) {
//        // Extract value filled in editext identified with txt_window_name
id
//        val windowName = findViewById<EditText>
(R.id.txt_window_name).text.toString()
//        // Display a message
//        Toast.makeText(this, "You choose $windowName",
Toast.LENGTH_LONG).show()
//
//    }

    /** Called when the user taps the button */
    fun openWindow(view: View) {
        val windowName = findViewById<EditText>
(R.id.txt_window_name).text.toString()

        // Do something in response to button
        val intent = Intent(this, WindowActivity::class.java).apply {
            putExtra(WINDOW_NAME_PARAM, windowName)
        }
        startActivity(intent)
    }
}
```

On the other side on **WindowActivity** you have to

1. read the name sent in intent
2. find Textview to update in Layout (this widget is identified by an id)
3. update this Textview with the name

```
package com.faircorp

import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import android.widget.TextView

class WindowActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_window)

        val param = intent.getStringExtra(WINDOW_NAME_PARAM)
        val windowName = findViewById<TextView>(R.id.txt_window_name)
        windowName.text = param
    }
}
```

Click **Apply Changes**

in the toolbar to run the app. Type a window name in the text field and click on the button to see the message in the second activity

- It worked in my phone. Am not using the emulator though my laptop has 8 gb ram because the laptop becomes really slow.

Manage back button to return on main activity

- When you are on WindowActivity we want to add a button to go back on MainActivity. To do that you need to update WindowActivity and add a line to activate option in action bar

```
supportActionBar?.setDisplayHomeAsUpEnabled(true)
```

- WindowActivity.kt

```
package com.faircorp

import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import android.widget.TextView

class WindowActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_window)

        supportActionBar?.setDisplayHomeAsUpEnabled(true)

        val param = intent.getStringExtra(WINDOW_NAME_PARAM)
        val windowName = findViewById<TextView>(R.id.txt_window_name)
        windowName.text = param
    }
}
```

- You also need to define your activity parent. This definition is made in AndroidManifest.xml with property parentActivityName

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.faircorp">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
<!--        <activity android:name=".WindowActivity"></activity>-->
        <activity android:name=".WindowActivity"
android:parentActivityName=".MainActivity"></activity>

        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Click **Apply Changes**

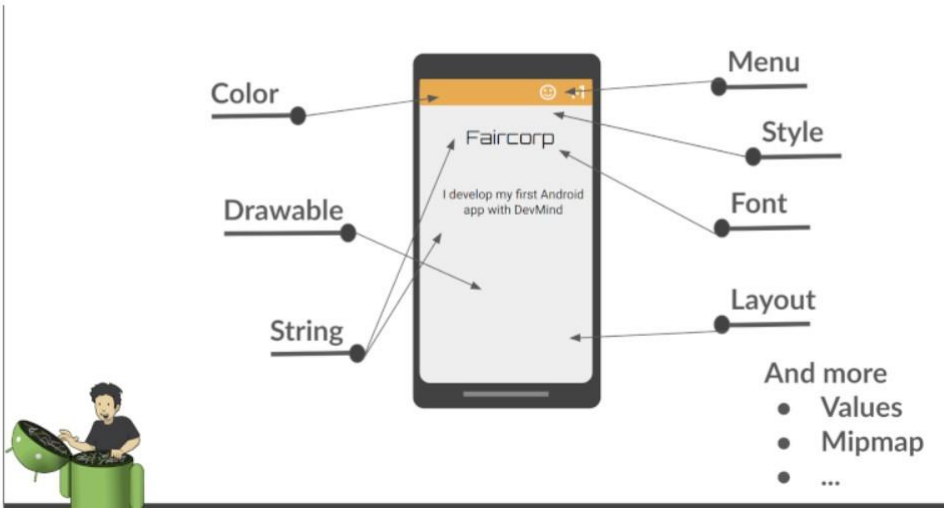
in the toolbar to run the app and test back button.

https://dev-mind.fr/training/android/android-add-menu.html

Add a menu in your app

Menus

Menus are a common user interface component in many types of applications. A **menu** is a resource as style, color, layout…



Remember, you can provide alternative resources for specific device configurations, by grouping them in specially-named resource directories. At runtime, Android uses the appropriate resource based on the current configuration.

It exists different types of menus. In this lesson we will implement a menu in app bar. It's where you should place actions that have a global impact on the app, such as "Compose email", "Settings"…

For all menu types, Android provides a standard XML format to define menu items. Instead of building a menu in your activity's code, you should define a menu and all its items in an XML file.

Using a menu resource is a good practice for a few reasons:

- It's easier to visualize the menu structure in XML.
- It separates the content for the menu from your application's behavioral code.
- It allows you to create alternative menu configurations for different platform versions, screen sizes, and other configurations by leveraging the app resources framework.

A menu will have different options. An option is a menu item

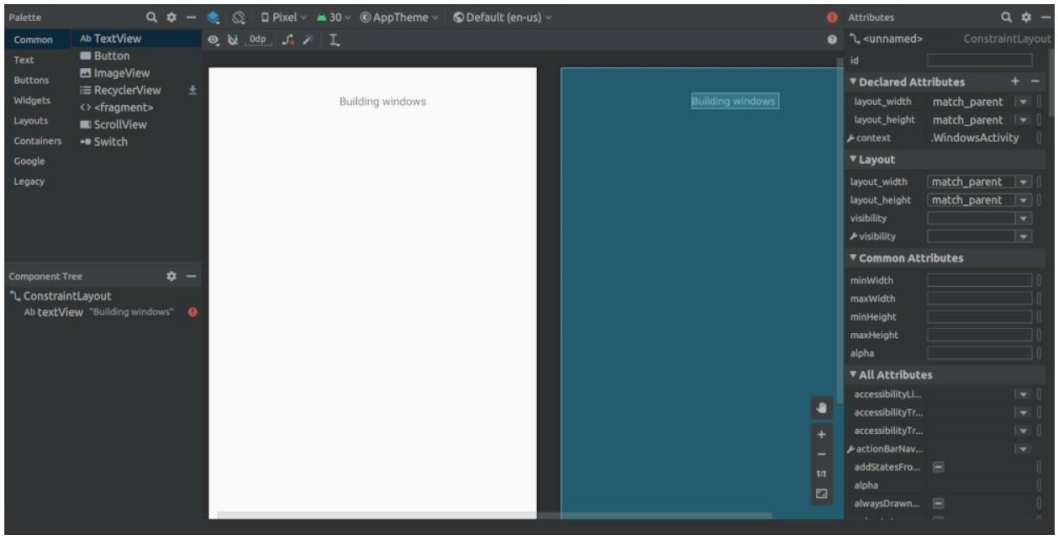
It's time to test by yourself. We want to add a menu in app bar with different options

- a link to open your corporate website (we will open an URL in favorite browser)
- a link to send you an email (we will open a window to write a new mail in favorite mail app)
- a link to open the activity used to list building windows

Create a new activity to list data

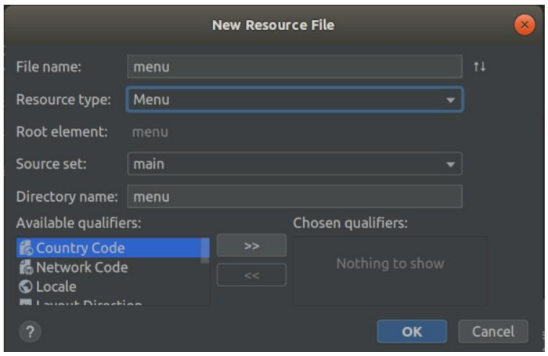
Create a new activity to list data

You need to create an empty activity with just a TextView with a label "Building windows". If you need some helps you can return on **last lesson**



Create a menu in app bar

1. Create an XML file inside your project's **res/menu/** directory. For that in the **Project window**, right-click the app folder and select **New > Android resource file**. You can also use menu **File > New > Android resource file**
2. In window "New resource File" choose
   - **File name** : menu
   - **Resource type** : menu
   - **Source set** : main
   - **Directory name** : menu



3. File **res/menu/menu.xml** is opened in menu editor. You can use **palette** to add elements. We will add 3 **Menu Items**. Item element supports several attributes to define an item's appearance and behavior. The main ones are :
   - **id** : an unique resource ID, which allows the application to recognize the item when the user wants to manipulate it in code.
   - **icon** : a reference to an optional drawable to use as the item's icon.
   - **title** : a reference to a string to use as the item's title.
   - **showAsAction** : specifies when and how this item should appear as an action item in the app bar. Possible values are
     - **ifRoom** : place this item in the app bar if there is room for it. If there is not room for all the items marked "ifRoom", last items are displayed in the overflow menu.
     - **withText**: also include the title text (defined by android:title) with the action item
     - **never** : place this item in the app bar's overflow menu.
   - you have more options. You can find them **here**
4. You can copy these string definitions in **res/values/string.xml**

```
<resources>
    <string name="app_name">Faircorp</string>
    <string name="app_logo_description">Faircorp logo</string>
```

```
    <string name="act_main_windowname_hint">Ligth name</string>
    <string name="act_main_welcome">Welcome on faircorp,\n the app to manage
building windows</string>
    <string name="act_main_open_window">Open window</string>

    <string name="menu_windows">Building windows</string>
    <string name="menu_website">Our website</string>
    <string name="menu_email">Send us an email</string>

</resources>
```

5. Add 3 menu entries with an id, a title and option showAsAction to the value never

- watch the video

6. We will attach this menu to activities MainActivity, WindowActivity and **ListDataActivity**. To prevent the add on each activity, we will create a parent activity and each activities will inherit from this parent activity. Select package com.faircorp, right-click and select New > Activity > Empty Activity. Set the name,BasicActivity

7. In this file you can copy this code

```
package com.faircorp

import android.content.Intent
import android.net.Uri
import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import android.support.v4.content.ContextCompat.startActivity
import android.view.Menu
import android.view.MenuInflater
import android.view.MenuItem

//class BasicActivity : AppCompatActivity() {
//    override fun onCreate(savedInstanceState: Bundle?) {
//        super.onCreate(savedInstanceState)
//        setContentView(R.layout.activity_basic2)
//    }
//}
open class BasicActivity : AppCompatActivity()
```

8. Update MainActivity, WindowActivity and ListDataActivity and replace AppCompatActivity by BasicActivity

- MainActivity

```
package com.faircorp

import android.content.Intent
import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import android.view.View
import android.widget.EditText
import android.widget.Toast


const val WINDOW_NAME_PARAM = "com.faircorp.windowname.attribute"

//
//class MainActivity : AppCompatActivity() {
//    override fun onCreate(savedInstanceState: Bundle?) {
//        super.onCreate(savedInstanceState)
//        setContentView(R.layout.activity_main)
//    }


//    /** Called when the user taps the button */
//    fun openWindow(view: View) {
//        // Extract value filled in editext identified with txt_window_name
id
//        val windowName = findViewById<EditText>
(R.id.txt_window_name).text.toString()
//        // Display a message
//        Toast.makeText(this, "You choose $windowName",
Toast.LENGTH_LONG).show()
//
//    }

//    /** Called when the user taps the button */
//    fun openWindow(view: View) {
//        val windowName = findViewById<EditText>
(R.id.txt_window_name).text.toString()
//
//        // Do something in response to button
//        val intent = Intent(this, WindowActivity::class.java).apply {
//            putExtra(WINDOW_NAME_PARAM, windowName)
//        }
//        startActivity(intent)
//    }
//}
open class MainActivity : BasicActivity()
```

- WindowActivity

```
package com.faircorp

import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import android.widget.TextView

//class WindowActivity : AppCompatActivity() {
//    override fun onCreate(savedInstanceState: Bundle?) {
//        super.onCreate(savedInstanceState)
//        setContentView(R.layout.activity_window)
//
//        supportActionBar?.setDisplayHomeAsUpEnabled(true)
//
//        val param = intent.getStringExtra(WINDOW_NAME_PARAM)
//        val windowName = findViewById<TextView>(R.id.txt_window_name)
//        windowName.text = param
//    }
//}

open class WindowActivity : BasicActivity()
```

- ListDataActivity

```
package com.faircorp

import android.support.v7.app.AppCompatActivity
import android.os.Bundle

//class ListDataActivity : AppCompatActivity() {
//    override fun onCreate(savedInstanceState: Bundle?) {
//        super.onCreate(savedInstanceState)
//        setContentView(R.layout.activity_list_data)
//    }
//}

open class ListDataActivity : BasicActivity()
```

9. We will now activate the menu. Override onCreateOptionsMenu() in BasicActivity. In this method, you can inflate your menu resource in the Menu provided in the callback

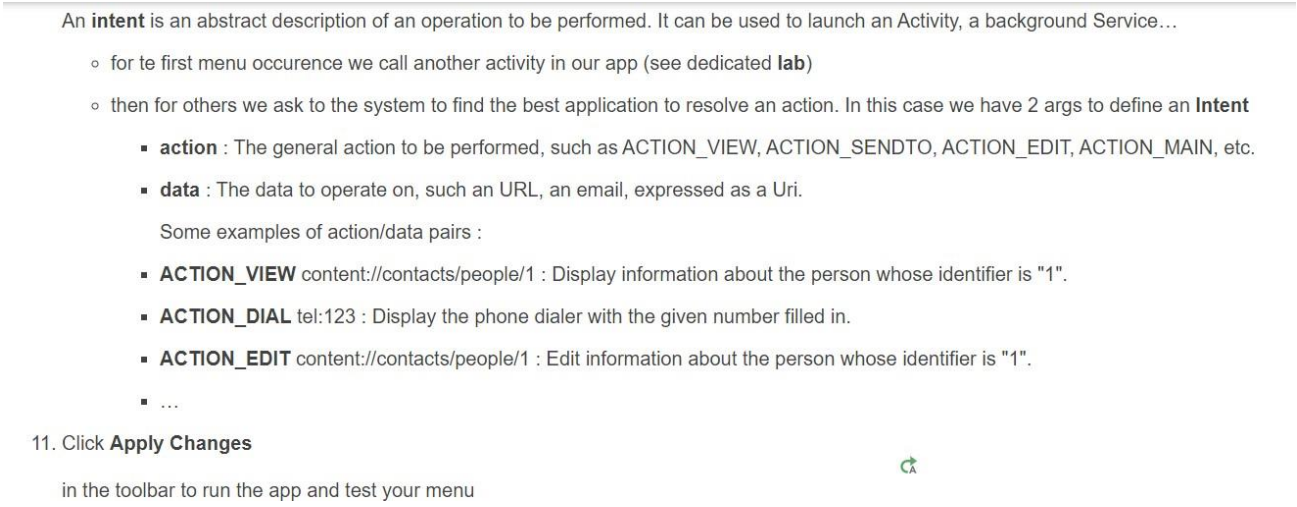
```
package com.faircorp

import android.content.Intent
import android.net.Uri
import android.support.v7.app.AppCompatActivity
import android.os.Bundle
```

```
import android.support.v4.content.ContextCompat.startActivity
import android.view.Menu
import android.view.MenuInflater
import android.view.MenuItem

//class BasicActivity : AppCompatActivity() {
//    override fun onCreate(savedInstanceState: Bundle?) {
//        super.onCreate(savedInstanceState)
//        setContentView(R.layout.activity_basic2)
//    }
//}
open class BasicActivity : AppCompatActivity() {
    override fun onCreateOptionsMenu(menu: Menu): Boolean {
        val inflater: MenuInflater = menuInflater
        inflater.inflate(R.menu.menu, menu)
        return true
    }

}
```

10. When the user selects an item from the options menu (including action items in the app bar), the system calls your activity's onOptionsItemSelected() method. This method passes the MenuItem selected. We will handle each possible values in BasicActivity class

```
package com.faircorp

import android.content.Intent
import android.net.Uri
import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import android.support.v4.content.ContextCompat.startActivity
import android.view.Menu
import android.view.MenuInflater
import android.view.MenuItem

//class BasicActivity : AppCompatActivity() {
//    override fun onCreate(savedInstanceState: Bundle?) {
//        super.onCreate(savedInstanceState)
//        setContentView(R.layout.activity_basic2)
//    }
//}
open class BasicActivity : AppCompatActivity() {
    override fun onCreateOptionsMenu(menu: Menu): Boolean {
        val inflater: MenuInflater = menuInflater
        inflater.inflate(R.menu.menu, menu)
        return true
    }
```

```
    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        when (item.itemId) {
            R.id.menu_windows -> startActivity(
                Intent(this,  ListDataActivity::class.java)
            )
            R.id.menu_website -> startActivity(
                Intent(Intent.ACTION_VIEW, Uri.parse("https://dev-mind.fr"))
            )
            R.id.menu_email -> startActivity(
                Intent(Intent.ACTION_SENDTO,
Uri.parse("mailto://guillaume@dev-mind.fr"))
            )

        }
        return super.onContextItemSelected(item)
    }
}
```



An **intent** is an abstract description of an operation to be performed. It can be used to launch an Activity, a background Service…

- for te first menu occurence we call another activity in our app (see dedicated **lab**)
- then for others we ask to the system to find the best application to resolve an action. In this case we have 2 args to define an **Intent**
  - **action** : The general action to be performed, such as ACTION_VIEW, ACTION_SENDTO, ACTION_EDIT, ACTION_MAIN, etc.
  - **data** : The data to operate on, such an URL, an email, expressed as a Uri.

  Some examples of action/data pairs :
  - **ACTION_VIEW** content://contacts/people/1 : Display information about the person whose identifier is "1".
  - **ACTION_DIAL** tel:123 : Display the phone dialer with the given number filled in.
  - **ACTION_EDIT** content://contacts/people/1 : Edit information about the person whose identifier is "1".
  - …

11. Click **Apply Changes**

in the toolbar to run the app and test your menu

https://dev-mind.fr/training/android/android-add-activity-list.html

Add a list activity in your app

- In this lesson, you will learn how to create a new View in your app to list data.
- Modelisation

Modelisation

We want to display room window information in our app.

A room is defined by several properties

- an id
- a name
- a current temperature (this property can be nullable if no data is available)
- a target temperature (this property can be nullable if no data is available)

A Window is defined by several properties

- an id
- a room
- a status (OPEN, CLOSED)

We are going to create classes to represent windows and rooms.

1. In the **Project window**, right-click the package *com.faircorp* and select **New > package**.
2. New package will be called **model**. Select this package, redo a right-click and select **New > Kotlin File/Class**.
3. Fill a name. For example **RoomDto** (dto = data transfer object) and create window properties. You can copy this code

```
package com.faircorp.model

data class RoomDto(val id: Long,
                   val name: String,
                   val currentTemperature: Double?,
                   val targetTemperature: Double?)
```

- Note: when a value is nullable you need to suffix type with ?. In our example currentTemperature can be null, so type is Double? and not Double

4. Redo same steps to create WindowDto

```
package com.faircorp.model

enum class Status { OPEN, CLOSED}

data class WindowDto(val id: Long, val name: String, val room: RoomDto, val
status: Status)
```

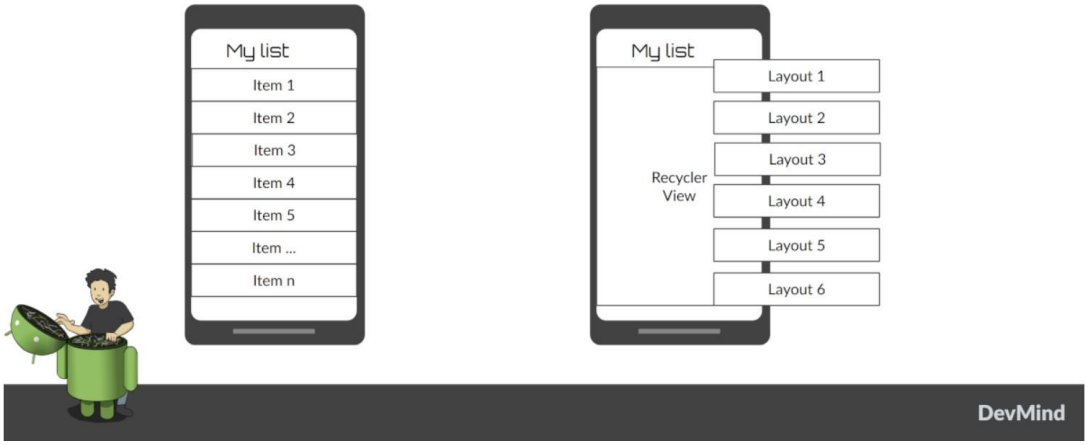
5. We will now create a service class to manage these windows. We will write 2 methods : one to find all building windows and a second to load only one window by its id. For the moment we will use fake data. In a next lesson we will learn how call a remote service to load real data.

```
package com.faircorp.model

class WindowService {
    companion object {
        // Fake rooms
        val ROOMS: List<RoomDto> = listOf(
            RoomDto(1, "Room EF 6.10", 18.2,  20.0),
            RoomDto(2, "Hall", 18.2, 18.0),
            RoomDto(3, "Room EF 7.10", 21.2, 20.0)
        )

        // Fake lights
        val WINDOWS: List<WindowDto> = listOf(
            WindowDto(1, "Entry Window", ROOMS[0], Status.CLOSED),
            WindowDto(2, "Back Window", ROOMS[0], Status.CLOSED),
            WindowDto(3, "Sliding door", ROOMS[1], Status.OPEN),
            WindowDto(4, "Window 1", ROOMS[2], Status.CLOSED),
            WindowDto(5, "Window 2", ROOMS[2], Status.CLOSED),
        )
    }

    fun findById(id: Long) = WINDOWS.firstOrNull { it.id == id}

    fun findAll() = WINDOWS.sortedBy { it.name }

}
```

RecyclerView

When you want to create a list view you will use a **RecyclerView** widget. This widget is able to manage a large data sets and scrool between elements.

The overall container for your user interface is a **RecyclerView** object that you add to your layout. The RecyclerView fills itself with views provided by a layout manager that you provide. The views in the list (used to dispaly items) are represented by view holder objects. Each view holder is in charge of displaying a single item with a view.



For example, if your list shows music collection, each view holder might represent a single album. The RecyclerView creates only as many view holders as are needed to display the on-screen portion of the dynamic content, plus a few extra. As the user scrolls through the list, the RecyclerView takes the off-screen views and rebinds them to the data which is scrolling onto the screen.

The view holder objects are managed by an adapter (create by extending **RecyclerView.Adapter**). This adapter creates view holders as needed. The adapter also binds the view holders to their data. It does this by assigning the view holder to a position.

Update window list activity

In **last session** you added an activity with just a TextView inside.

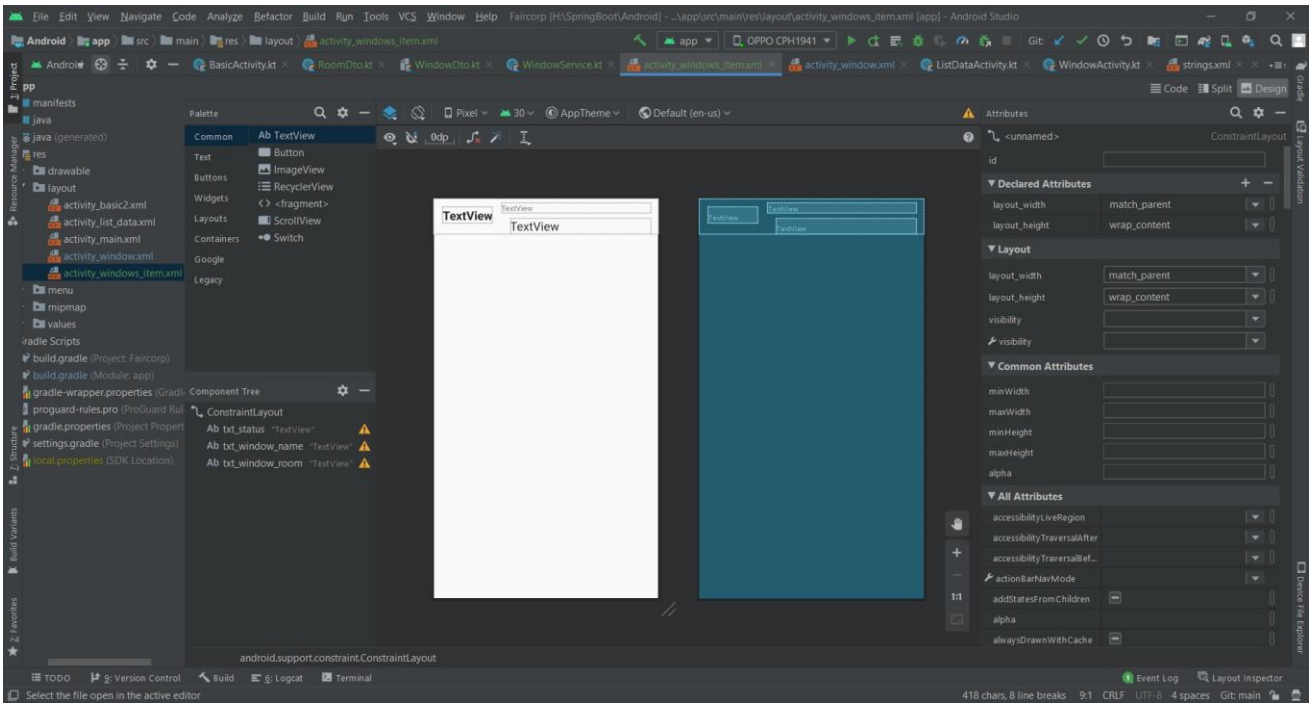
1. Open **res > layout > activity_windows.xml** and delete the TextView
2. In **common palette** select a **RecyclerView** widget and drag into your layout below your welcome message. A dialog should ask you if you want to add lib **androidx.recyclerview:recyclerview:+** in your app. Click on **OK** button to let Android Studio add this dependency in your Gradle config file (*build.gradle (Module app)*)
3. This **RecyclerView** widget should have these properties
    - **id** : *list_windows*
    - **margins** : *16dp* Apply a top, right and left margin
    - **layout_width** : widget should take all the width (0dp or match_parent)
    - **layout_height** : widget should take all the height (0dp or match_parent)

- follow the video please

Create a layout for a list item

1. Select res > layout right click and choose New > Layout resource file
2. Name your future layout activity_windows_item.xml
3. Add 3 Textviews

- A TextView on the left to display window status
  - **id** : *txt_status*
  - **marginStart** : *16dp*
  - **marginTop** : *16dp*
  - **marginBottom** : *16dp*
  - **textStyle** : *bold*
  - **textAppearance** : *@style/TextAppearance.AppCompat.Large*
  - **capitalize** : *characters*
- A TextView on the right to display window name
  - **id** : *txt_window_name*
  - **marginStart** : *16dp*
  - **marginTop** : *8dp*
  - **marginEnd** : *16dp*
  - **marginEnd** : *16dp*
  - **layout_width** : *0dp*
- A last TextView to display window room
  - **id** : *txt_window_room*
  - **marginStart** : *16dp*
  - **marginTop** : *8dp*
  - **marginBottom** : *8dp*
  - **marginEnd** : *16dp*
  - **layout_width** : *0dp*
  - **textAppearance** : *@style/TextAppearance.AppCompat.Small*
  - **capitalize** : *characters*



- just look at the struggle

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <!--    <TextView-->
    <!--        android:id="@+id/txt_status"-->
    <!--&lt;!&ndash;
app:layout_constraintStart_toStartOf="parent"&ndash;&gt;-->
    <!--&lt;!&ndash;
app:layout_constraintTop_toTopOf="parent"&ndash;&gt;-->
    <!--&lt;!&ndash;        android:layout_width="wrap_content"&ndash;&gt;--
>
    <!--&lt;!&ndash;        android:layout_height="wrap_content"&ndash;&gt;-
->
    <!--        android:layout_marginStart="16dp"-->
    <!--        android:layout_marginTop="16dp"-->
    <!--        android:layout_marginBottom="16dp"-->
    <!--        android:capitalize="characters"-->
    <!--
android:textAppearance="@style/TextAppearance.AppCompat.Large"-->
    <!--        android:textStyle="bold" />-->
    <!--    -->
    <!--    <TextView-->
    <!--        android:id="@+id/txt_window_name"-->
    <!--&lt;!&ndash;
```

```
app:layout_constraintStart_toStartOf="parent"&ndash;&gt;-->
    <!--&lt;!&ndash;
app:layout_constraintTop_toTopOf="parent"&ndash;&gt;-->
    <!--&lt;!&ndash;        android:layout_width="wrap_content"&ndash;&gt;--
>
    <!--&lt;!&ndash;        android:layout_height="wrap_content"&ndash;&gt;-
->
    <!--        android:layout_marginStart="16dp"-->
    <!--        android:layout_marginTop="8dp"-->
    <!--        android:layout_marginEnd="16dp"-->
    <!--        android:layout_width="0dp"-->
    <!--&lt;!&ndash;        android:capitalize="characters"&ndash;&gt;-->
    <!--&lt;!&ndash;
android:textAppearance="@style/TextAppearance.AppCompat.Large"&ndash;&gt;-->
    <!--&lt;!&ndash;        android:textStyle="bold" &ndash;&gt;-->
    <!--    />-->

    <!--    <TextView-->
    <!--        android:id="@+id/txt_window_room"-->
    <!--        android:layout_width="0dp"-->
    <!--&lt;!&ndash;        android:layout_height="wrap_content"&ndash;&gt;-
->
    <!--        android:layout_marginStart="16dp"-->
    <!--        android:layout_marginTop="8dp"-->
    <!--        android:layout_marginBottom="8dp"-->
    <!--        android:layout_marginEnd="16dp"-->
    <!--        android:capitalize="characters"-->
    <!--
android:textAppearance="@style/TextAppearance.AppCompat.Small"-->
    <!--&lt;!&ndash;
app:layout_constraintStart_toStartOf="parent"&ndash;&gt;-->
    <!--&lt;!&ndash;        app:layout_constraintTop_toTopOf="parent"
&ndash;&gt;-->
    <!--        android:capitalize="characters"-->
    <!--        />-->
    <!--    -->

    <TextView
        android:id="@+id/txt_status"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginTop="16dp"
        android:layout_marginEnd="16dp"
        android:capitalize="characters"
        android:text="TextView"
        android:textAppearance="@style/TextAppearance.AppCompat.Large"
        android:textStyle="bold"
        app:layout_constraintEnd_toStartOf="@+id/txt_window_room"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
```

```
    <TextView
        android:id="@+id/txt_window_name"

        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="16dp"
        android:text="TextView"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/txt_status"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/txt_window_room"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="16dp"


        android:capitalize="characters"
        android:text="TextView"
        android:textAppearance="@style/TextAppearance.AppCompat.Large"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.0"
        app:layout_constraintStart_toEndOf="@+id/txt_status"
        app:layout_constraintTop_toBottomOf="@+id/txt_window_name"  />
</android.support.constraint.ConstraintLayout>
```

Create an adapter class

- As we see in previous chapter, an adapter manages the view holder objects. The adapter also binds the view holders to their data. It does this by assigning the view holder to a position.

1. In the Project window, right-click the package com.faircorp.model and right-click and select New > Kotlin File/Class. We will create a new class called WindowsAdapterView
2. You can copy this code inside

```
package com.faircorp.model

import android.support.v7.widget.RecyclerView
import android.view.LayoutInflater
```

```
import android.view.View
import android.view.ViewGroup
import android.widget.TextView
import com.faircorp.R

class WindowAdapter : RecyclerView.Adapter<WindowAdapter.WindowViewHolder>()
{ // (1)

    inner class WindowViewHolder(view: View) :  RecyclerView.ViewHolder(view)
{ // (2)
        val  name:  TextView  =  view.findViewById(R.id.txt_window_name)
        val  room:  TextView  =  view.findViewById(R.id.txt_window_room)
        val status: TextView =  view.findViewById(R.id.txt_status)
    }

    private val items = mutableListOf<WindowDto>() // (3)

    fun update(windows:  List<WindowDto>) {  // (4)
        items.clear()
        items.addAll(windows)
        notifyDataSetChanged()
    }

    override fun getItemCount(): Int = items.size // (5)

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
WindowViewHolder { // (6)
        val view = LayoutInflater.from(parent.context)
            .inflate(R.layout.activity_windows_item, parent, false)
        return WindowViewHolder(view)
    }

    override fun onBindViewHolder(holder: WindowViewHolder, position: Int)  {
// (7)
        val window = items[position]
        holder.apply {
            name.text = window.name
            status.text = window.status.toString()
            room.text = window.room.name
        }
    }
}
```

- (1) an adapter must implement **RecyclerView.Adapter** wich manage a **RecyclerView.ViewHolder**
- (2) we create a **WindowViewHolder** which is able to hold fields defined in layout **activity_windows_item.xml**. When you scroll through the list view, system does not recreate these fields. It will update the values via method (7)
- (3) adapter has a mutable list to store elements to display
- (4) method used to update the list content. This method will be called when data will be ready
- (5) **RecyclerView.Adapter** abstract class asks you to implement a first method that returns the number of records
- (6) **RecyclerView.Adapter** abstract class asks you to implement a second method used to initialize a **ViewHolder**
  - we inflate **activity_windows_item.xml** layout
  - we send it to **ViewHolder** constructor
- (7) **RecyclerView.Adapter** abstract class asks you to implement a last method to define what to do when position in the list changes

Update activity ListDataActivity

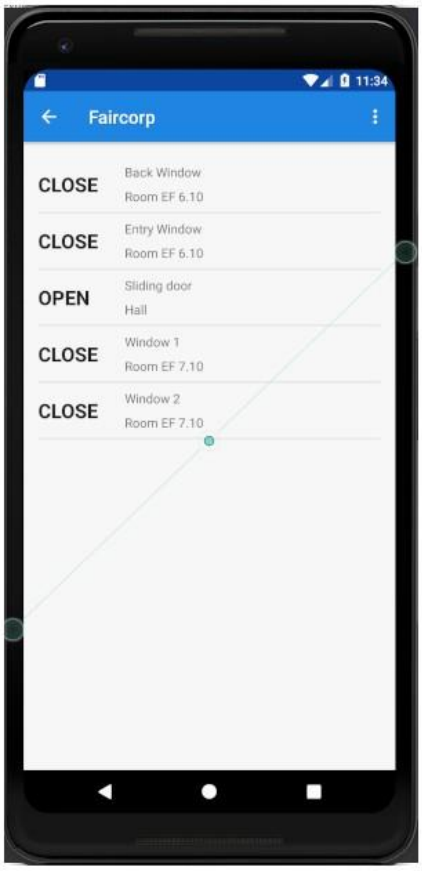
- We need to update ListDataActivity to initialize the recycler view

```
package com.faircorp

import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import android.support.v7.widget.DividerItemDecoration
import android.support.v7.widget.LinearLayoutManager
import android.support.v7.widget.RecyclerView
import com.faircorp.model.WindowAdapter
import  com.faircorp.model.WindowService

//class ListDataActivity : AppCompatActivity() {
//    override fun onCreate(savedInstanceState: Bundle?) {
//        super.onCreate(savedInstanceState)
//        setContentView(R.layout.activity_list_data)
//    }
//}

//open class ListDataActivity : BasicActivity()
class ListDataActivity : BasicActivity() {

    val windowService = WindowService() // (1)

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_list_data)

        val recyclerView = findViewById<RecyclerView>(R.id.list_windows) //
(2)
        val adapter = WindowAdapter() // (3)

        recyclerView.layoutManager = LinearLayoutManager(this)
        recyclerView.addItemDecoration(DividerItemDecoration(this,
DividerItemDecoration.VERTICAL))
        recyclerView.setHasFixedSize(true)
        recyclerView.adapter = adapter
```

```
        adapter.update(windowService.findAll()) // (4)
    }
}
```

- (1) we instantiate service created in **first chapter** of this lesson
- (2) we find the recycler view defined in layout by its id `list_windows`
- (3) adapter is created and recycler view properties are defined
- (4) on the last step we update adapter data

You can now open a list screen as this screenshot



- am not getting the output based on the snapshot
- Now, I used **ListDataActivity** instead of **WindowsActivity**
- hence got WindowsActivity and the corresponding activity_windows.xml

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".WindowsActivity">

    <TextView
```

```
        android:id="@+id/txt_building_windows"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Building Windows"
        tools:layout_editor_absoluteX="160dp"
        tools:layout_editor_absoluteY="223dp"
        tools:ignore="MissingConstraints" />
</android.support.constraint.ConstraintLayout>
```

- got the corresponding xml file for ListDataActivity, deleted, because apparently there was some troubles coming up while commenting
- there were red marks due to ListDataActivity, and hence got that deleted as well
- WindowsActivity code

```
package com.faircorp

import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import android.support.v7.widget.DividerItemDecoration
import android.support.v7.widget.LinearLayoutManager
import android.support.v7.widget.RecyclerView
import com.faircorp.model.WindowAdapter
import  com.faircorp.model.WindowService

//class ListDataActivity : AppCompatActivity() {
//    override fun onCreate(savedInstanceState: Bundle?) {
//        super.onCreate(savedInstanceState)
//        setContentView(R.layout.activity_list_data)
//    }
//}

//open class ListDataActivity : BasicActivity()
class WindowsActivity : BasicActivity() {

    val windowService = WindowService() // (1)

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_windows)

        val recyclerView = findViewById<RecyclerView>(R.id.list_windows) //
(2)
        val adapter = WindowAdapter() // (3)

        recyclerView.layoutManager = LinearLayoutManager(this)
        recyclerView.addItemDecoration(DividerItemDecoration(this,
DividerItemDecoration.VERTICAL))
```

```
        recyclerView.setHasFixedSize(true)
        recyclerView.adapter = adapter

        adapter.update(windowService.findAll()) // (4)
    }
}
```

- hoping to change WindowsActivity in the future of these slides
- moving on
- The entire system of Android has been upgraded to androidX, the new folder name is FairCorp2

Install Retrofit

To get this data into the app, your app needs to

- establish a network connection to remote server which exposes your REST service and
- communicate with that server, and then
- receive its response data and
- parse the data to be usable in your code.

We will use Retrofit to do the first three steps. For the last one we need a converter to deserialize HTTP body. Several converters are available. We will use **Moshi** library

1. Open **build.gradle (Module: Faircorp.app)**.
2. In the dependencies block, add 2 lines to load Retrofit and the Moshi converter (versions are available **here**)

```
implementation "com.squareup.retrofit2:retrofit:2.9.0"
implementation "com.squareup.retrofit2:converter-moshi:2.9.0"
```

3. You also need to update android plugin configuration to activate support for Java 8 language. Retrofit was written in JDK 8. Update android block and add

```
compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
}

kotlinOptions {
    jvmTarget = JavaVersion.VERSION_1_8.toString()
}
```

4. As you updated your gradle configuration, Android Studio display a message to synchronize your projet. Click on **Sync now**

- everything associated to this slide has been completed

Call a remote API in Android

- Followed the below instructions exactly

Install Retrofit

To get this data into the app, your app needs to

- establish a network connection to remote server which exposes your REST service and
- communicate with that server, and then
- receive its response data and
- parse the data to be usable in your code.

We will use Retrofit to do the first three steps. For the last one we need a converter to deserialize HTTP body. Several converters are available. We will use **Moshi** library

1. Open **build.gradle (Module: Faircorp.app)**.
2. In the dependencies block, add 2 lines to load Retrofit and the Moshi converter (versions are available **here**)

```
implementation "com.squareup.retrofit2:retrofit:2.9.0"
implementation "com.squareup.retrofit2:converter-moshi:2.9.0"
```

3. You also need to update android plugin configuration to activate support for Java 8 language. Retrofit was written in JDK 8. Update android block and add

```
compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
}

kotlinOptions {
    jvmTarget = JavaVersion.VERSION_1_8.toString()
}
```

4. As you updated your gradle configuration, Android Studio display a message to synchronize your projet. Click on **Sync now**

- Followed the exact instructions

Configure Retrofit

Now you are ready to write the code to call your API.

1. In package `com.faircorp.model` create a new interface called **WindowApiService**
2. In this interface we declare methods used to launch a remote call

```
interface WindowApiService {
    @GET("windows")
    fun findAll(): Call<List<WindowDto>>

    @GET("windows/{id}")
    fun findById(@Path("id") id: Long): Call<WindowDto>
}
```

   - Annotations (GET, POST, PUT, DELETE,...) on the interface methods and its parameters indicate how a request will be handled.
   - A request URL can be updated dynamically using replacement blocks and parameters on the method. A replacement block is an alphanumeric string surrounded by { and }.
     - define a parameter in path

```
@GET("windows/{id}")
fun findById(@Path("id") id: Long): Call<WindowDto>
```

     - define a parameter in query

```
@GET("windows")
fun findAll(@Query("sort") sort: String): Call<List<WindowDto>>
```

   - An object can be specified for POST or PUT HTTP requests @Body annotation. In this case, Retrofit will use converter defined in your conf to serialize body object in JSON

```
@PUT("windows/{id}")
fun updateWindow(@Path("id") id: Long, @Body window: WindowDto) Call<WindowDto>
```

     - you will find more information on **Retrofit** website

- ApiServices code

```
package com.faircorp.model

import retrofit2.Retrofit
import retrofit2.converter.moshi.MoshiConverterFactory

class ApiServices {
    val windowsApiService : WindowApiService by lazy {
```

```
        Retrofit.Builder()
            .addConverterFactory(MoshiConverterFactory.create())
            .baseUrl("https://app-420ce9e6-69f1-4afa-8c93-
613ae8951e47.cleverapps.io/api/")
            .build()
            .create(WindowApiService::class.java)
    }
}
```

The builder needs

- a converter factory to tell Retrofit what do with the data it gets back from the web service.
- an URL of the remote service (In this example I use an URL on my website but you can use your own API)

Use Retrofit

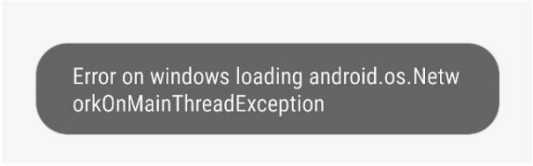
Call instances can be executed either synchronously or asynchronously (each instance can only be used once, but calling clone() will create a new instance that can be used). In our case we will use synchronous calls.

1. Open **com.faircorp.WindowsActivity**
2. Replace line `adapter.update(windowService.findAll())` with this code

```
runCatching { ApiServices().windowsApiService.findAll().execute() } // (1)
            .onSuccess { adapter.update(it.body() ?: emptyList()) }  // (2)
            .onFailure {
                Toast.makeText(this, "Error on windows loading $it", Toast.LENGTH_LONG).show()  // (3)
            }
```

   - (1) method **execute** run a synchronous call
   - (2) we use **runCatching** to manage successes and failures. On success we update adapter with the result contained in body property. If this response is null the list is empty
   - (3) on error we display a message in a **Toast notation**
3. Click **Apply Changes**

   in the toolbar to run the app. Try to open windows list.
4. Unfortunately you should have a toast notification with the following message :



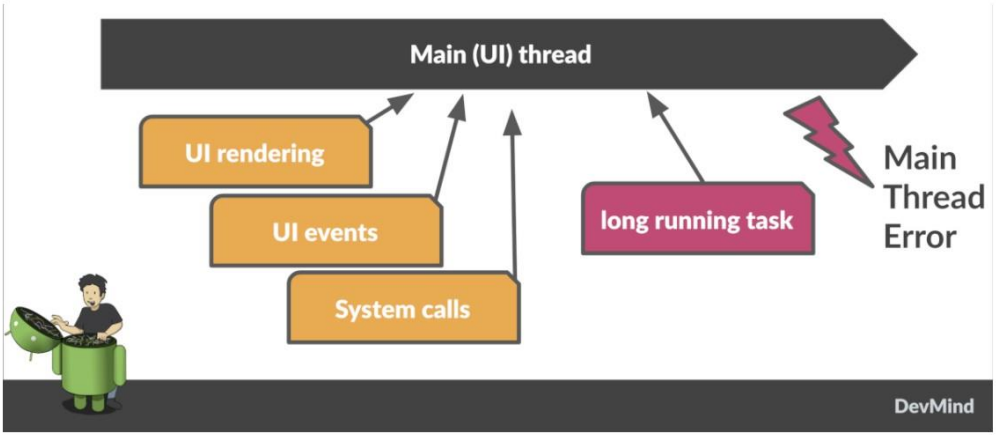
- followed the above instructions

Main thread

When the system launches your application, that application runs in a thread called **Main thread**. This main thread manages user interface operations (rendering, events ...), system calls...

Calling long-running operations from this main thread can lead to freezes and unresponsiveness.

Making a network request on the main thread causes it to wait, or block, until it receives a response. Since the thread is blocked, the OS isn't able to manage UI events, which causes your app to freeze and potentially leads to an Application Not Responding (ANR) dialog. To avoid these performance issues, Android throws a **MainThreadException** and kills your app if you try to use this main thread.



The solution is to run your network call, your long-running task in another thread, and when the result is available you can reattach the main thread to display the result. Only the main thread can update the interface.

If you develop in Java, Thread development can be difficult. With Kotlin you can use **coroutines**.

- followed the exact above instructions

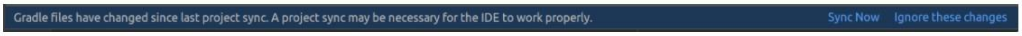
Coroutines

A **coroutine** is a concurrency design pattern that you can use on Android to simplify code that executes asynchronously. Coroutines help to manage long-running tasks that might otherwise block the main thread and cause your app to become unresponsive.

1. Open **build.gradle (Module: Faircorp.app)** to add the following dependency (in dependencies block)

```
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:$kotlin_version"
implementation 'androidx.lifecycle:lifecycle-runtime-ktx:2.2.0'
```

2. Android Studio display a message to synchronize your projet. Click on **Sync now**



In Kotlin, all coroutines run inside a **CoroutineScope**. A scope controls the lifetime of coroutines through its job. When you cancel the job of a scope, it cancels all coroutines started in that scope. On Android, you can use a scope to cancel all running coroutines when, for example, the user navigates away from an Activity or Fragment. Scopes also allow you to specify a default dispatcher. A dispatcher controls which thread runs a coroutine.

Each object in Android which has a **lifecycle** (Activity, Fragment…), has a **CoroutineScope**.

1. Open **com.faircorp.WindowsActivity**
2. Update code to call windowsApiService as follows

- followed the above instructions
- checkout my build.gradle file to get the right implementations

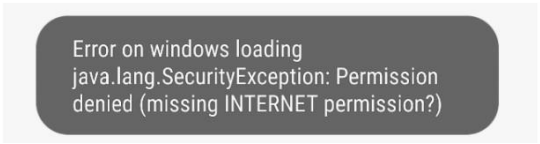
```
lifecycleScope.launch(context = Dispatchers.IO) { // (1)
    runCatching { ApiServices().windowsApiService.findAll().execute() } // (2)
        .onSuccess {
            withContext(context = Dispatchers.Main) { // (3)
                adapter.update(it.body() ?: emptyList())
            }
        }
        .onFailure {
            withContext(context = Dispatchers.Main) { // (3)
                Toast.makeText(
                    applicationContext,
                    "Error on windows loading $it",
                    Toast.LENGTH_LONG
                ).show()
            }
        }
}
```

- (1) method `lifecycleScope.launch` open a new directive. You must specify a context other than Dispatchers.Main (Main thread) for the code to be executed. `Dispatchers.IO` is dedicated to Input/Output tasks
- (2) you can call retrofit to read data
- (3) You cant' display something (result in list, error in toast notification) outside the main thread. `withContext` helps to reattach your code to another thread

3. Click **Apply Changes**

in the toolbar to run the app. Try to open windows list.

4. Unfortunately you should have another toast notification. You only have one more problem to solve before you can display the result in your app. The error message tells you that your app might be missing the INTERNET permission.



- followed the exact instructions above

Android permission

The purpose of a permission is to protect the privacy of an Android user. Android apps must request permission to access sensitive user data or features such as contacts, SMS, Internet... Depending on the feature, the system might grant the permission automatically or might prompt the user to approve the request.

By default, an app has no permission to perform any operations that would adversely impact other apps, the operating system, or the user.

To add a new permission to be able to call our remote API, open **app/manifests/AndroidManifest.xml**. Add this <uses-permission> tag (just before <application> tag)

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.example.snazzyapp">

    <uses-permission android:name="android.permission.INTERNET" />

    <application ...
         android:usesCleartextTraffic="true">
        ...
    </application>
</manifest>
```

You can now relaunch your app and you will be able to open the windoww list without error. For more informations about permissions you can read this **page**.

- followed the above and android works perfectly.