**1.مقدمة في Spring Boot**

**سنتعرف في هذه المقدمة على مفاهيم أساسية وهي :**

* منظم مكتبات Package Manager ووظيفته تنظيم و تثبيت المكتبات الخارجية.
* مصطلح RestApi وهو أسلوب أساسي لنقل البيانات على الشبكة يمكن التعرف عليه أكثر من هنا.
* ملف Json Object وهو طريقة لعرض البيانات وإرسال البيانات
* ملف XML وهو طريقة لترتيب وعرض البيانات .
* الفرق بين Spring Framework و Spring Boot.
* متطلبات التشغيل.
* تثبيت المتطلبات.
* بناء مشروع (Hello World) في إطار العمل Spring.

يساعدك Spring Boot على إنشاء تطبيقات قائمة بذاتها جاهزة للتشغيل وبشكل سريع، بحيث يمكنك البدء بأقل قدر من التعقيد، وتحتاج معظم تطبيقات Spring Boot إلى القليل جدًا من التخصيص لتكون جاهزة للعمل.



**1.1 مميزات Spring Boot الأساسية هي:**

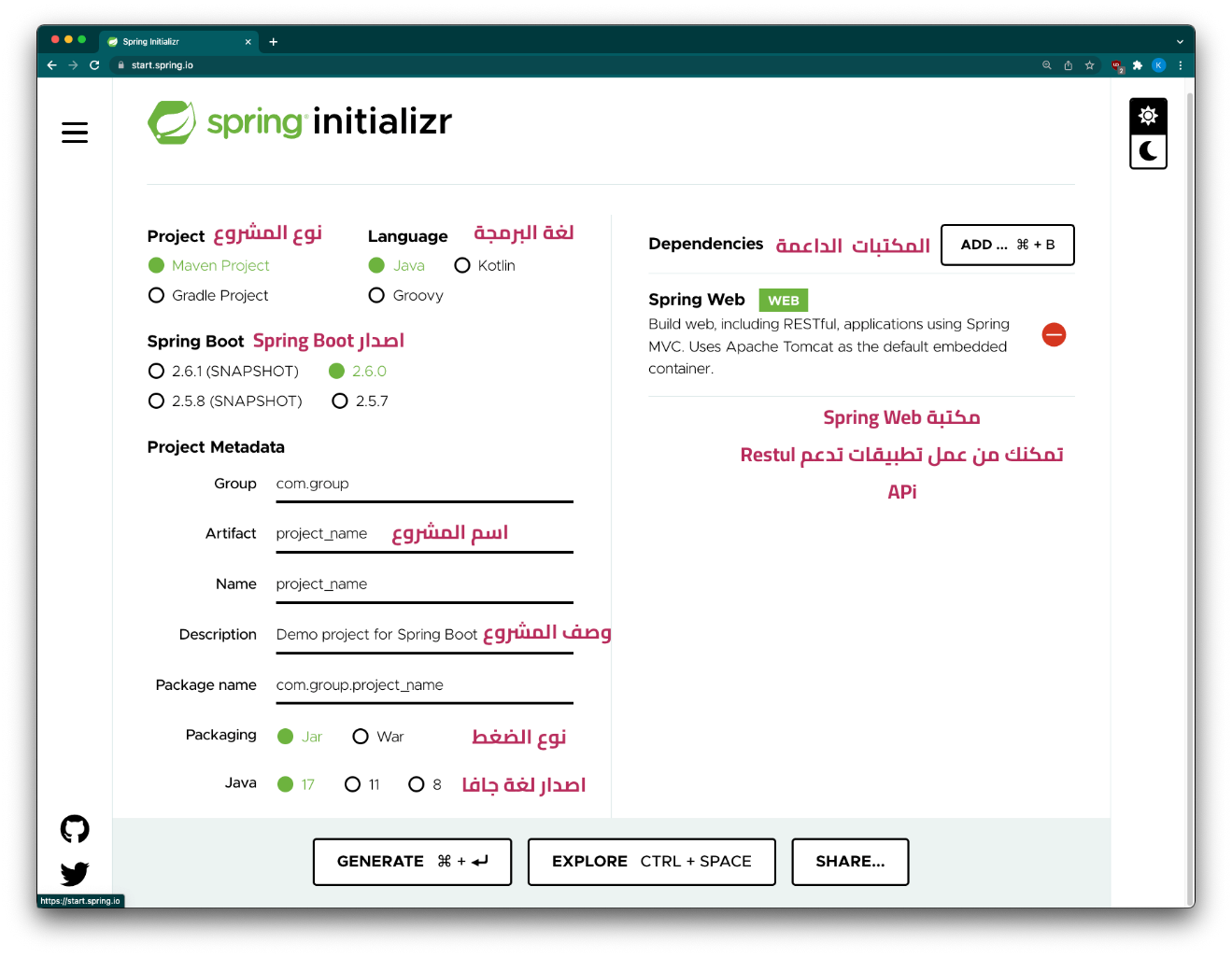
* توفير تجربة بدء أسرع بشكل جذري عن طريق استخدام Spring.io.
* يدعم أكثر من لغة مثل Kotlin و Java.
* توفير تجربة تطوير أسرع عن طريق الاستغناء عن بعض المتطلبات مثل استخدام ملفات XML. متطلبات التشغيل
* يتطلب استخدام Spring Boot تثبيت Java ويدعم حاليًا إصدارات Java 8 و Java 11 و Java 17.
* استخدام محرر نصوص، مثل VScode ويمكنك استخدام محررات أخرى مثل IntelliJ IDEA.
* تثبيت منظم مكتبات (Package Manager) للغة Java مثل Maven أو Gradle.

**تثبيت متطلبات التشغيل**

* نقوم بتثبيت Java عن طريق تثبيت Java Development Kit
* نقوم بتثبيت IntelliJ IDEA

**1.2** **بناء مشروع Spring Boot عن طريق Spring Boot Initializr**

من أجل القيام بأول مشروع Spring Boot سوف نتجه إلى موقع Spring Boot Initializr والذي سيمكننا من إنشاء مشروع Spring Boot بشكل بسيط وسريع.

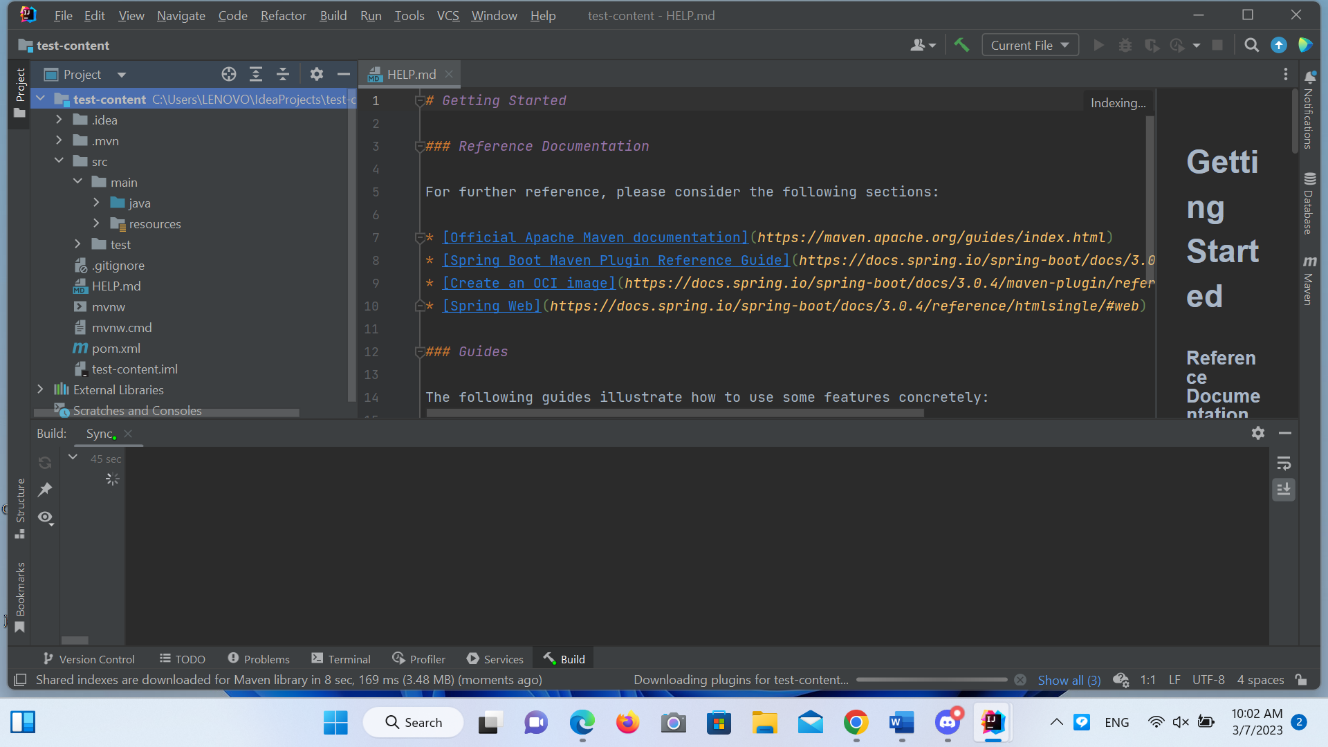


**البيانات المهمة:**

1. نوع المشروع
   1. مشروع Maven: سوف ينشئ مشروع يحتوي على Maven كمنظم لملفات المشروع (اختيارنا).
   2. مشروع Gradle: سوف ينشئ مشروع يحتوي على Gradle كمنظم لملفات المشروع.
2. إصدار Spring Boot من الأفضل اختيار أخر نسخة مستقرة.
3. نوع الضغط
   1. نوع الضغط Jar.
   2. نوع الضغط War.
4. إصدار لغة Java.
5. المكتبات الداعمة
   1. قمنا بإضافة مكتبة Spring Web والتي من خلالها يمكننا بناء تطبيقات Web.

**ملفات المشروع**

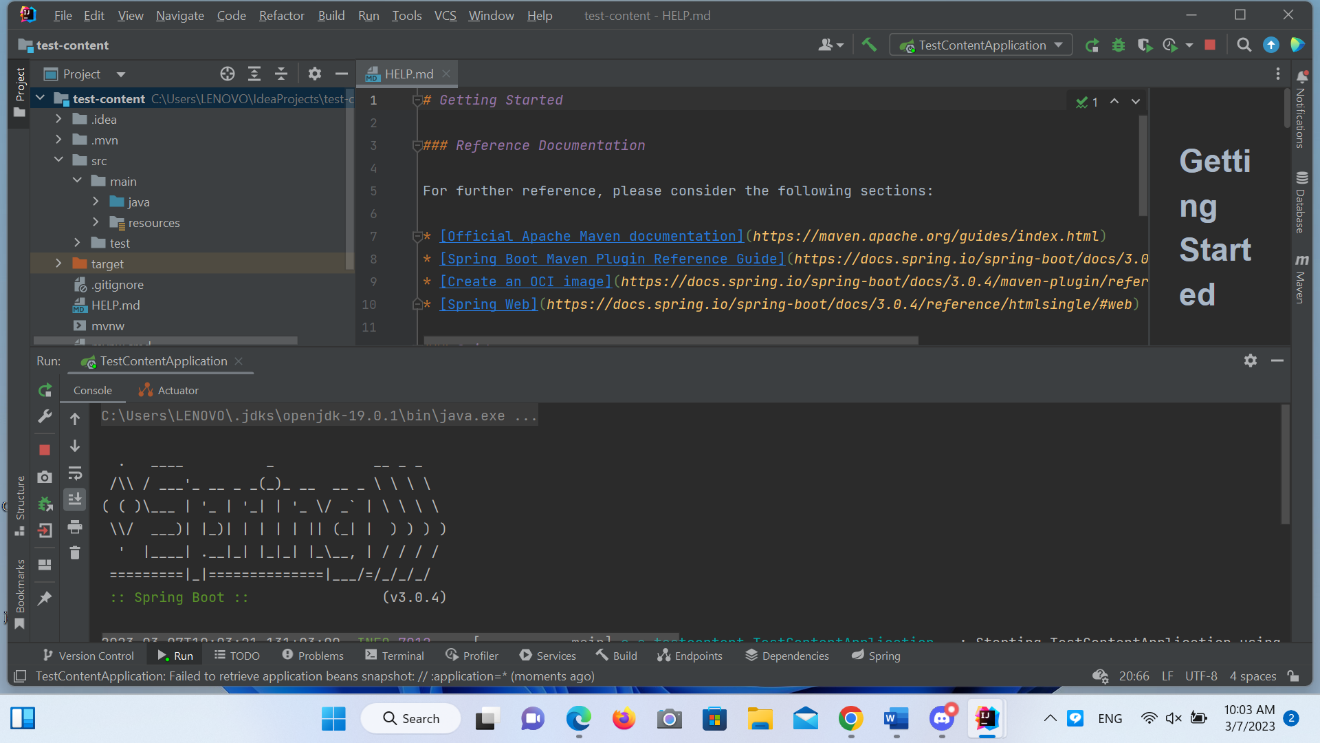
بعد تحميل الملف من موقع Spring Boot Initializr سوف نحصل على مشروع يحتوي على هذا الترتيب:

****

* ملف pom.xml  هو الملف الخاص بإضافة المكتبات المساندة ويمكننا من خلاله إضافة مكتبات جديدة إلى مشروعنا.
* مجلد main هو المجلد المسؤول عن إضافة ملفات البرمجة الخاصة بالمشروع.

**تشغيل المشروع**

لتشغيل المشروع علينا الضغط على زر Run ، بعدها سيقوم maven بتشغيل المشروع



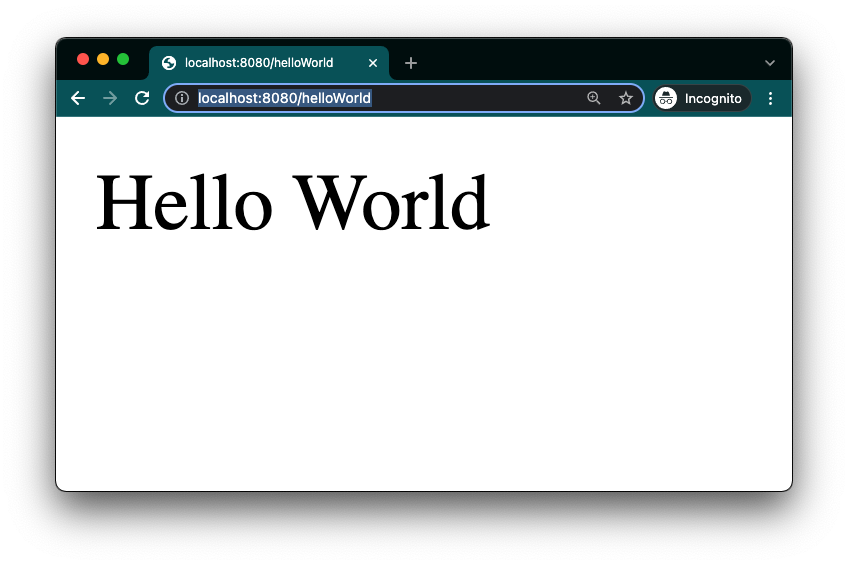
**1.3 إضافة Hello World Api**

**في البداية سوف نتجه إلى ملف java داخل الmain ونقوم بإضافة التعديلات التالية :**

package com.example.testcontent;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
@SpringBootApplication  
@RestController  
public class TestContentApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(TestContentApplication.class, args);  
 }  
  
 @GetMapping("/hello-world")  
 public String helloWord() {  
 return "Hello world!!";  
  
  
 }  
}

* قمنا بإضافة  Annotaionباسم Rest Controller وهو المسؤول عن تحديد ما إذا كان هذا  Class لديه خواص Controller مثل استقبال الطلبات Request، وسنتعرف عليه بشكل أكبر لاحقًا.
* قمنا بإضافة Annotaion باسم  GetMapping ثم قمنا بإضافة  method اسمها  helloWorld وظيفة GetMapping أنها تبين أن  method  عبارة عن getRequest ويمكننا الوصول إليها عن طريق هذا المسار "helloWorld/"

والآن سنقوم بتشغيل المشروع مرة أخرى ونتجه إلى الرابط التالي في المتصفح، وهو رابط localhost؛ وذلك لأن المشروع يعمل على مستضيف من نفس الجهاز: http://localhost:8080/helloWorld



**1.4 إيجابيات Spring boot مقارنة ب Spring**

* يقوم بإنشاء تطبيقات Spring قائمة بذاتها يمكن البدء باستخدامها Java-.Jar
* لا توجد متطلبات لتكوين XML.
* يوفر أداة CLI لتطوير واختبار تطبيق Spring Boot.
* يزيد من الإنتاجية ويقلل من وقت التطوير.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Spring Context .2**

Boot Spring لديها سياق خاص لعمل البرنامج و يعتمد السياق اوContext) (Spring على مايسمى .Beans

على سبيل المثال لدينا دالة () :message

public void message(){ System.*out*.println("hey");

}

لن يتم تشغيل الدالة الا بعد استدعائها في دالة main في السياق المعتاد، ولكن إذا أردنا اتباع سياق spring الخاص نقوم

باستخدام مايسمى ب Beans .Spring

**Spring Beans .3**

beans Spring هي كائنات Java يتم إدارتها بواسطة container .Spring

@Bean

public void message(){ System.*out*.println("hey");

}

عندما نقوم بإضافة @Bean اعلى الداله، سيقوم Spring بتشغيل واستدعاء العديد من الدوال واالسطر في سياقه

الخاص وسيقوم باستدعاء الداله () message تلقائي.

**لا يمكننا التنبؤ بالسياق الخاص ب Spring في حال كانت لدينا دالتان او أكثر.**

@Bean

public void message(){ System.*out*.println("hey from 1");

}

@Bean

public void message2(){ System.*out*.println("hey from 2");

}

**Spring Container .4**

حاوية Spring هي المسؤولة عن إنشاء وتكوين وتجميع Beans .Spring على سبيل المثال لو قمنا بانشاء دالة تقوم بارجاع قيمة نصية

.

@Bean

public String getName(){ return "Maha";

}

سيقوم Spring باستدعاء الدالة وتنفيذها ومن ثم تخزين القيمة النصية في Container .Spring

لو قمنا بانشاء دالة تقوم باستقبال قيمة نصية وقمنا بإضافة @Bean سيقوم Spring بأخذ القيمة النصية المخزنة في

Container وتمريرها للدالة.

@Bean

public String printName(String name){ System.*out*.println(name(;

return name;

}

// Output:

// Maha

لدينا طريقتين لتحديد أي قيمة نصية يتم تمريرها في حال كانت لدينا أكثر من قيمة نصية مخزنة في :Container

-1 باستخدام @Primary

@Bean @Primary

public String getName(){ return "Maha";

}

بوضع @Primary فوق الدالة تعني ان Spring سيقوم دائما˝ باستعمال القيمة النصية التي تم ارجاعها من هذه ال

.Bean

-2 باستخدام @Qualifier مع قيمة نصية فريدة

@Bean @Qualifier("1")

public String getName(){ return "Maha";

}

@Bean @Qualifier("2")

public String getName2(){ return "majd";

}

@Bean

public String printName(@Qualifier("1")String name){ System.*out*.println(name);

return name;

}

//Output:

//Maha

بوضع (“1”) @Qualifier في Parameter الدالة تعني ان Spring سيقوم بأخذ القيمة النصية المرجعة من ال

.@Qualifier (“1”) ب المعرف Bean

**5.التعليقات التوضيحية لـ Spring Boot (Annotations)**

التعليقات التوضيحية لـ Spring Boot هي شكل من أشكال البيانات الوصفية التي توفر بيانات حول البرنامج. بمعنى آخر ، يتم استخدام التعليقات التوضيحية لتوفير معلومات تكميلية حول البرنامج. إنه ليس جزءًا من التطبيق الذي نطوره. ليس له تأثير مباشر على تشغيل الكود الذي يعلقون عليه. لا يغير عمل البرنامج المترجم.

جميع الـ annotations مرفقة في المرجع التالي :

<https://www.javatpoint.com/spring-boot-tutorial>

ـــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــ

**6.ملفات JSON (JavaScript object notation )**

سميَ بهذا الاسم لمشابهته طريقة إنشاء الكائنات بلغة JavaScript.

**6.1 ماهو** **JSON** : هو ملف نصي يستخدم لتمثيل البيانات وتبادلها بين التطبيقات المختلفة، ومن ذلك نستنتج انه فقط طريقة لتمثيل البيانات وتناقلها وليس لغة برمجة بحد ذاته. كما يكون امتداد الملف .json .

* **مثال:**  fileName.json

**6.2 مميزاتها :** استخدمت ملفات JSON على نطاق واسع لسهولة كتابتها واستخدامها. ومن استخداماتها الاساسية أن تكون حلقة ربط بين الخادم ولغة البرمجة بحيث يتم تناقل البيانات بينهم بهذه الصيغة, ويتم كتابة الملف باستخدام الأقواس المتعرجة { }.

**6.3 أنواع البيانات المتاحة:** تقبل ملفات json أنواع بيانات محددة وهي: String, numbers, null, boolean, arrays and objects.

**نصية (String) : سلسلة من الأحرف تمثل قيمة ما.**

* **مثال:**  {"title": "Hello World"}

**رقم (Number) : هو رقم صحيح** **أو حقيقي او عشري وقد يحتوي على اشارة.**

* **مثال:**  {"age": 20}

**قيمة منطقية (Boolean): : قيمة قد تكون اما صحيحة(true) او خاطئة (false). وهي مستخدمة بكثرة لاعتماد الحاسب عليها.**

* **مثال:**  {"isAnimal": true}

**قيمة فارغة (Null) : عدم وجود قيمة فعلية للمتغير.**

* **مثال:**  {"job" : null}

**كعدم وجود وظيفة للشخص.**

**مصفوفة (Array): هي سلسلة من القيم المترابطة مخزنة في متغير.**

* **مثال:**  {"oddNumbers" : [ 1, 3, 5, 7, 9 ]}

**الكائن (object): مجموعة من اسم للمتغيرات والقيم الخاصة بها.**

**مثال:** {"employee":{ "id" : 100, "name" : "Sami" }}

**6.4 المفتاح والقيمة**

**المفتاح**: يمثل اسم خاص لقيمة البيانات ويتم وضعه بين علاماتي التنصيص " " ويجب أن يكون متبوعا بـ نقطتان رأسيتان : ليفصل بين الاسم وقيمة البيانات. كما تكون المفاتيح عباره عن نصوص (string) مثل: Book ,Company, Pen.

**القيمة**: تمثّل البيانات ويمكن أن تكون اكثر من نوع: اسم, رقم, مصفوفه... \*يجب تكون القيم أنواع متاح استخدامها مثل: (string, number, object, array, Boolean or null)

* **مثال :**  { "Name": "Ahmed" }

وبين أزواج المفاتيح والقيم المتعددة يتم الفصل بينهم بوضع علامة فاصله ( , )

* **مثال :**  {" Name": "Ali" , "Age": 23 , "id": 4433}

**6.5 كائنات JSON المتداخلة** (**Nested JSON Objects**):

أي يمكن ان يحتوي الكائن على كائن آخر بداخلة كما في المثال التالي:

* **مثال :**  { "Book" : { " BookName": " Java programming", " price": 200, " AuthorName": { " firstName": " xxxxxx", " secondName": " xxxxxx", "AuthorMajor": ["CS", "IT", "IS"] } }

**مثال شامل :** لنأخذ الطالب كمثال نعرض به ما تعلمناه بطريقة ملفات JSON.

{"name":{ "FristName":"Ghaida" ,"LastName":"Albohairy" }, "id":437005 , "major":"Information Technology", "courses": [2,1] , "isSenior":False , "GPA": null }

ـــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــ

**7. وحدة التحكم ( :(Controller Layer**

هي واجهة الخادم (Server) التي من خلالها يمكن استقبال الطلبات عبر البوابات وارسال النتيجة ، ولكي يقوم بمهمته لابد أن تحتوي كل بوابة على ثلاثة أشياء :

1.تحديد نوع الدالة : تحديد نوع الطلب GET, POST, PUT , DELETE باستخدام التعليق التوضيحي @....mapping

2. تحديد URL : الذي من خلاله يمكن طلب الصفحة

3. تحديد end point: الدالة المسؤولة عن استقبال الطلب وارسال النتيجة

@RestController  
@RequestMapping("/api/v1/message")  
public class TodoController {  
  
 @GetMapping("/message")  
 public String getMessage(){  
 return “Hey from Spring Boot”;  
 }

|  |  |
| --- | --- |
| @RestController | يوضح أن هذا class نوعه controller ويقوم بإرجاع نتيجة من نوع JSON |
| @RequstMapping | لوضع API محدد لهذا controller |
| @GetMapping | لتحديد نوع Method |
| (“/message”) | تحديد URL للوصول لهذه الصفحة |

**7.1 مفهوم REST :**

يعتبر REST، أو REpresentational State Transfer، أسلوبًا معماريًا لتوفير معايير بين أنظمة الكمبيوتر على الويب، مما يسهل على الأنظمة التواصل مع بعضها البعض. تتميز الأنظمة المتوافقة مع REST، والتي غالبًا ما تسمى أنظمة RESTful، بكيفية كونها عديمة الحالة وتفصل اهتمامات العميل (Client) والخادم (Server). سوف ندرس ما تعنيه هذه المصطلحات ولماذا تعتبر خصائص مفيدة للخدمات على الويب. فصل العميل (Client) والخادم (Server) في النمط المعماري REST، يمكن تنفيذ العميل (Client) وتنفيذ الخادم (Server) بشكل مستقل دون معرفة كل منهما بالآخر. هذا يعني أنه يمكن تغيير الكود الموجود على جانب العميل (Client) في أي وقت دون التأثير على تشغيل الخادم (Server)، ويمكن تغيير الكود الموجود على جانب الخادم (Server) دون التأثير على عمل العميل (Client). طالما أن كل جانب يعرف تنسيق الرسائل التي سيتم إرسالها إلى الآخر، فيمكن الاحتفاظ بها معيارية ومنفصلة. بفصل مخاوف واجهة المستخدم عن مخاوف تخزين البيانات (Data)، نقوم بتحسين مرونة الواجهة عبر الأنظمة الأساسية وتحسين قابلية التوسع من خلال تبسيط مكونات الخادم (Server). بالإضافة إلى ذلك، يتيح الفصل لكل مكون القدرة على التطور بشكل مستقل. باستخدام واجهة REST، يصل العملاء المختلفون إلى نفس نقاط نهاية REST، ويقومون بنفس الإجراءات، ويتلقون نفس الاستجابات. الأنظمة التي تتبع نموذج REST هي أنظمة عديمة الحالة (Stateless)، مما يعني أن الخادم (Server) لا يحتاج إلى معرفة أي شيء عن حالة العميل (Client) والعكس صحيح. بهذه الطريقة، يمكن لكل من الخادم (Server) والعميل (Client) فهم أي رسالة مستلمة، حتى بدون رؤية الرسائل السابقة.

هناك 4 أفعال HTTP أساسية نستخدمها في طلبات التفاعل مع الموارد في نظام REST: (HTTP Verbs)

1. GET : استرداد مورد معين (بواسطة المعرف) أو مجموعة من الموارد
2. POST : إنشاء مورد جديد
3. PUT: تحديث مورد معين (بواسطة المعرف)
4. DELETE :إزالة مورد معين بواسطة المعرف

ـــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــ

**7.2 مفهوم** **CRUD (Create,Read,Update,Delete):**

الإنشاء والقراءة والتحديث والحذف (CRUD) هي الوظائف الأساسية الأربع التي يجب أن تكون النماذج قادرة على القيام بها، على الأكثر. عندما نبني واجهات برمجة التطبيقات، نريد أن توفر نماذجنا أربعة أنواع أساسية من الوظائف. يجب أن يكون النموذج قادرًا على إنشاء الموارد وقراءتها وتحديثها وحذفها. غالبًا ما يشير علماء الكمبيوتر إلى هذه الوظائف بالاختصار CRUD. يجب أن يتمتع النموذج بالقدرة على أداء هذه الوظائف الأربع على الأكثر حتى يكتمل. إذا كان لا يمكن وصف إجراء من خلال إحدى هذه العمليات الأربع، فمن المحتمل أن يكون نموذجًا خاصًا به. يعد نموذج CRUD شائعًا في إنشاء تطبيقات الويب، لأنه يوفر إطارًا لا يُنسى لتذكير المطورين بكيفية إنشاء نماذج كاملة قابلة للاستخدام.

**مثال :** لنتخيل نظامًا لتتبع كتب المكتبة.

في قاعدة بيانات المكتبة الافتراضية هذه يمكننا أن نتخيل أنه سيكون هناك مورد كتب، والذي سيخزن أشياء من الكتب. لنفترض أن كائن الكتاب يشبه هذا:

“book”: {

"id": <Integer>,

“title”: <String>,

“author”: <String>,

}

لجعل نظام المكتبة هذا قابلاً للاستخدام، نود التأكد من وجود آليات واضحة لإكمال عمليات: CRUD

**عملية الإنشاء**  **Create** : سيتكون هذا من وظيفة نسميها عند إضافة كتاب مكتبة جديد إلى الفهرس. البرنامج الذي يستدعي الوظيفة سيوفر قيم “title” و “author” ، بعد استدعاء هذه الوظيفة، يجب أن يكون هناك إدخال جديد في مورد الكتب المقابل لهذا الكتاب الجديد. بالإضافة إلى ذلك، يتم تعيين معرّف فريد للإدخال الجديد، والذي يمكن استخدامه للوصول إلى هذا المورد لاحقًا.

**عملية القراءة** **Read**  :سيتكون هذا من وظيفة سيتم استدعاؤها لرؤية جميع الكتب الموجودة حاليًا في الفهرس. لن يؤدي استدعاء الوظيفة هذا إلى تغيير الكتب الموجودة في الكتالوج - بل سيؤدي ببساطة إلى استرداد المورد وعرض النتائج. سيكون لدينا أيضًا وظيفة لاسترداد كتاب واحد، يمكننا توفير العنوان أو المؤلف ومرة أخرى، لن يتم تعديل هذا الكتاب، بل سيتم استرجاعه فقط.

**عملية التعديل أو التحديث** **Update** : يجب أن تكون هناك وظيفة لاستدعاء عندما يجب تغيير المعلومات حول الكتاب. سيقوم البرنامج الذي يستدعي الوظيفة بتزويد القيم الجديدة لكل من “title” و “author” بعد استدعاء الوظيفة، سيحتوي الإدخال المقابل في مورد الكتب على الحقول الجديدة المتوفرة.

**عملية الحذف** **Delete** : يجب أن تكون هناك وظيفة لاستدعاء لإزالة كتاب مكتبة من الفهرس. سيوفر البرنامج الذي يستدعي الوظيفة قيمة واحدة أو أكثر من title و author لتحديد الكتاب، ثم تتم إزالة هذا الكتاب من مصدر الكتب. بعد استدعاء هذه الوظيفة، يجب أن يحتوي مورد الكتب على جميع الكتب التي كان لديه من قبل، باستثناء الكتاب الذي تم حذفه للتو.

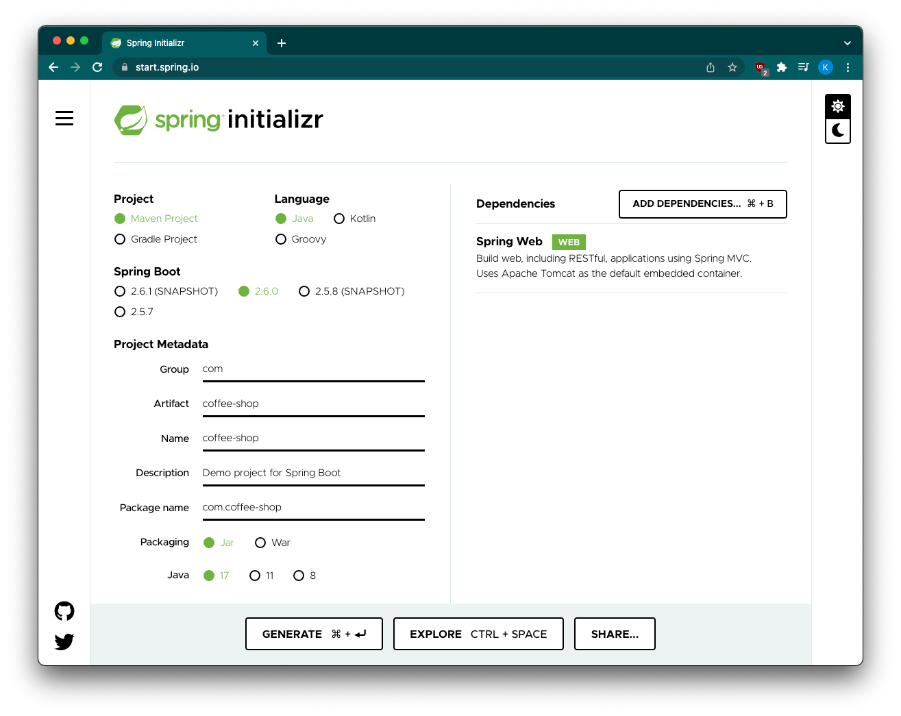
**7.3 بناء REST Api :**

**المتطلبات :** برنامج Postman لاختبار الروابط .

في هذا الجزء سنقوم ببناء مشروع يحتوي على الأربع عناصر الأساسية في أي مشروع Api وهي:

* get وهو النوع الخاص باستقبال البيانات.
* postوهو النوع الخاص بإضافة بيانات جديدة.
* put وهو النوع الخاص بتعديل البيانات.
* delete وهو النوع الخاص بحذف البيانات.

**بناء مشروع Spring Boot عن طريق Spring Boot Initializer**



**مثال : إنشاء مجلد خاص بالقهوة وصفاتها :**

في البداية سنقوم بإنشاء المجلد الخاص بالقهوة والذي يحتوي على جميع الملفات الخاصة بالقهوة، ولكن قبل البدء بكتابته سنتعرف على أفضل الممارسات في تنظيم المشروع.

com

+- example

+- myapplication

+- MyApplication.java

|

+- class-name

| +- ClassName.java

| +- ClassNameController.java

|

**إنشاء ملف coffee.java الخاص بالقهوة :**

بداخل هذا المجلد سنقوم بإنشاء ملف java جديد وسيكون باسم Coffee.java وبداخله الصفات المتعلقة بالقهوة مثل الاسم و السعر... إلى آخره.

package com.coffeeshop.coffee;  
public class Coffee {  
 private final long id;  
 private String name;  
 private double price;  
 public Coffee(long id, String name, double price) {  
 this.id = id;  
 this.name = name;  
 this.price = price;  
 }  
 public long getId() {  
 return id;  
 }  
 public String getName() {  
 return name;  
 }  
 public void setName(String name) {  
 this.name = name;  
 }  
 public double getPrice() {  
 return price;  
 }  
 public void setPrice(double price) {  
 this.price = price;  
 }  
 @Override  
 public String toString() {  
 return "Coffee [id=" + id + ", name=" + name + ", price=" + price + "]";  
 }  
}

يحتوي ملف coffee.java على ثلاثة صفات :

* صفة id: وهي ترمز إلى رقم خاص بكل قهوة ونوعها long.
* صفة name: وهي صفة خاصة باسم القهوة ونوعها String.
* صفة price: وهي صفة خاصة بسعر القهوة ونوعها double.

وتم إضافة setter و getter للصفات وكذلك تم إضافة toString.

**إنشاء ملف CoffeeController.java**

ملف coffeeController.java هو الملف المسؤول عن إدارة طلبات Rest Api Class الخاصة، وسوف يحتوي على الأربع  requests الأساسية: Get , Post ,Put , Delete

من المهم استخدام @RestController عند بدايته وذلك لتحديد أنه كلاس خاص بمعالجة Rest Requests

package com.coffeeshop.coffee;  
  
import org.springframework.web.bind.annotation.RestController;  
 @RestController  
 public class CoffeeController {  
  
 }

الآن نضيف بياناتcoffee  وسوف تكون من نوع ArrayList <Coffee>  وباسم coffees وسوف نسند له قيمة new ArrayList<>()

Array<Coffee> coffees = new ArrayList<>();

**7.3.1طلبات Get :**

سنقوم الآن بإنشاء أول Api لنا وسيكون من نوع get وسيكون خاص باستعراض أنواع القهوة التي لدينا ، حالياً سيقوم بإرجاع مصفوفة فارغة .

**7.3.2 طلبات Post :**

تختلف طلبات Post عن Get بسبب أن Post تحتوي على Body ويمكننا من وضع بيانات فيها وتقوم Post بإضافة وإرسال البيانات. الآن سنقوم بإضافة postApi آخر وسيكون وظيفته إضافة قهوة إلى مجموعة القهوة التي نملكها. في البداية سوف نقوم بكتابة  methodالخاصة بهذه الوظيفة:

@PostMapping()  
public ResponseEntity<Coffee> addCoffee(@RequestBody Coffee c) {  
 coffees.add(c);  
 return ResponseEntity.*status*(201).body(new ApiResponse("Coffee Added"));  
}

استخدمنا @PostMapping() من أجل تحديد أن هذا الطلب هو Post أو الرابط الخاص بهذه الوظيفة، كما قمنا بإضافة واستخدام @RequestBody وذلك لإضافة بيانات القهوة الجديدة. وبداخل هذه الدالة قمنا بإضافة القهوة c الموجودة بداخل @RequestBody .

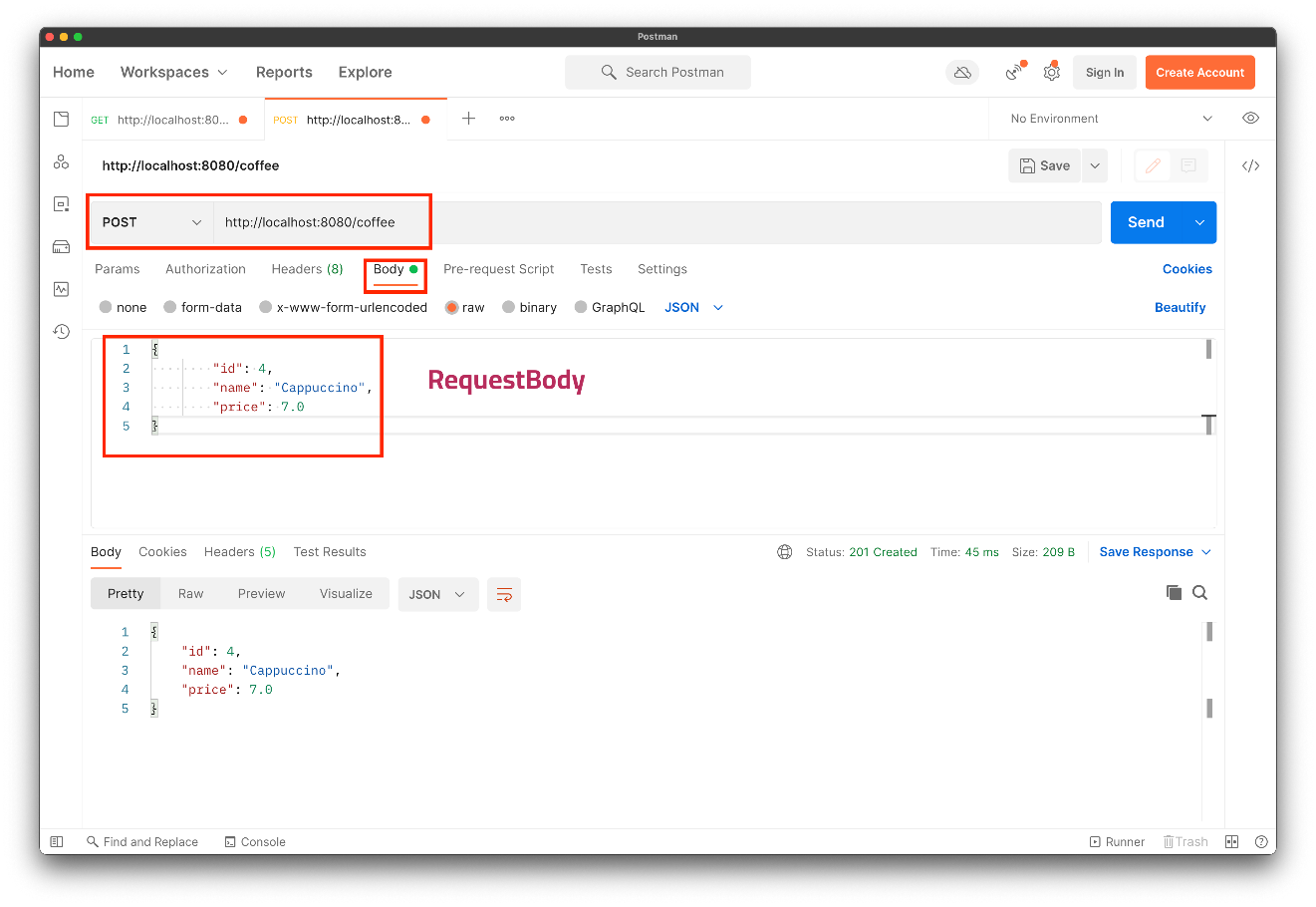
ولتحسين Api سنقوم باستخدام  ResponseEntity  وذلك لكي نعيد إشارة Httpststus والتي سنتعرف عليها بشكل أكبر في وقت لاحق .

وكذلك قمنا بإنشاء كلاس ApiResponse ومعرف بداخله message من نوع String لاستخدامه في إرجاع message داخل الBody

يمكننا طلب Api عن طريق الرابط <http://localhost:8080>

ولكن سنقوم بتعديل المسار إلى http://localhost:8080/coffee  وذلك لتمييز أن هذا هو المسار الخاص بجلب المعلومات الخاصة بالقهوة. ويمكننا ذلك عن طريق إضافة

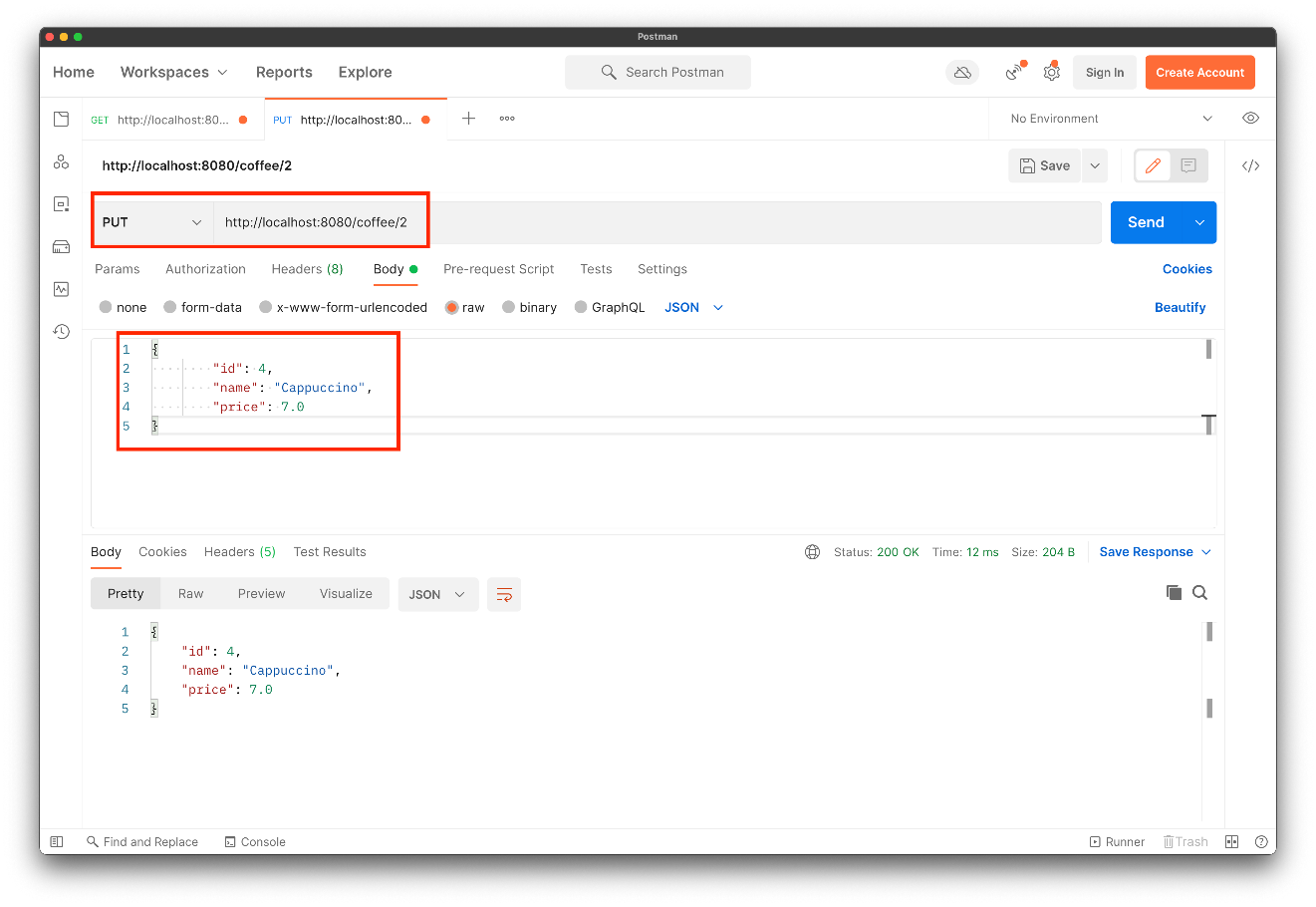
@PostMapping(“/coffee)

**7.3.3 طلبات PUT :**

تقوم طلبات Put بتعديل البيانات الموجودة. سنقوم الآن بإضافة putApi وسيكون وظيفته التعديل على بيانات القهوة الموجودة. في البداية سنقوم بكتابة method الخاصة بهذه الوظيفة:

public ResponseEntity updateCoffee(@PathVariable Integer id,@RequestBody Coffee c) {  
 coffees.set(id,c);  
 return ResponseEntity.*status*(200).body(new ApiResponse("Coffee Updated"));

* استخدمنا  @PutMapping("/{id}") من أجل تحديد المسار أو الرابط الخاص بهذه الوظيفة وقمنا بإضافة /{id} وذلك لإرسال هذه القيمة إلى Parameter الخاص بهذه الدالة، مثال: http://localhost:8080/coffee/1 ويمثل رقم ١ id الخاص بالقهوة .
* قمنا باستخدام @PathVariable وذلك لجعل id يأخذ قيمته من الرابط.
* قمنا باستخدام @RequestBody وذلك لإضافة بيانات القهوة الجديدة.

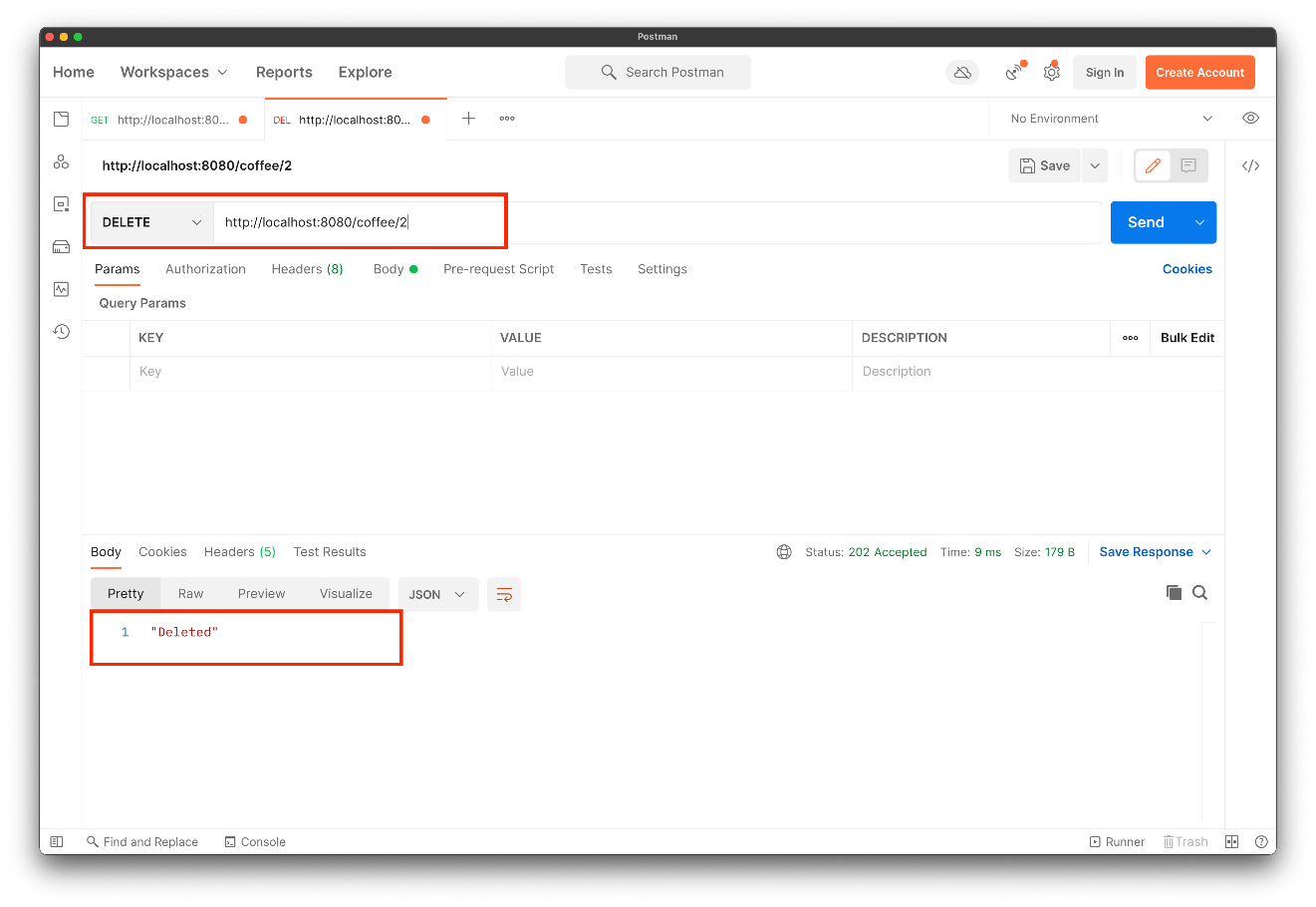


**7.3.4 طلبات Delete :**

تقوم طلبات Delete بحذف البيانات. سنقوم الآن بإضافة deleteApi وسيكون وظيفته حذف قهوة موجودة بين القهوة التي لدينا. في البداية سنقوم بكتابة method الخاصة بهذه الوظيفة:

public ResponseEntity deleteCoffee(@PathVariable Integer id) {  
 coffees.remove(id);  
 return ResponseEntity.*status*(200).body(new ApiResponse("Deleted"));

* ستخدمنا @DeleteMapping("/{id}") من أجل تحديد المسار أو الرابط الخاص بهذه الوظيفة وقمنا بإضافة /{id} وذلك لإرسال هذه القيمة إلى Parameter الخاص بهذه الدالة مثال: http://localhost:8080/coffee/1 ويمثل رقم ١ id الخاص بالقهوة .
* قمنا باستخدام @PathVariable وذلك لجعل id يأخذ قيمته من الرابط.



**7.4 مكتبة Lombok :**

هي مكتبة جافا تُستخدم لتقليل / إزالة الشفرة المعيارية وحفظ الوقت للمطورين أثناء التطوير عن طريق استخدام بعض التعليقات التوضيحية فقط. بالإضافة إلى ذلك ، فإنه يزيد أيضًا من قابلية قراءة الكود المصدري ويوفر المساحة.

**تعليقات Lombok التوضيحية :** توفر Lombok مجموعة من التعليقات التوضيحية لجعل حياتنا البرمجية أسهل.

لنلقي نظرة على التعليقات التوضيحية القليلة الأكثر استخدامًا في Lombok /

**@Getter and @Setter:**  : ينشئ التعليق التوضيحي Getter طريقة getter بنوع وصول كعام والذي يقوم ببساطة بإرجاع الحقل وباسم ()getName إذا كان اسم الحقل هو "Name". ينشئ التعليق التوضيحي Setter طريقة تعيين بنوع الوصول كعموم والذي يعيد الفراغ ويأخذ معلمة واحدة لتعيين القيمة للحقل. سيكون للمُعيِّن الافتراضي اسم ()setName إذا كان اسم الحقل هو "Name".

**@NoArgsConstructor:** يتم استخدام هذا التعليق التوضيحي لإنشاء مُنشئ بدون وسيطات. له جسم فارغ ولا يفعل شيئًا. يتم استخدامه بشكل عام مع بعض المُنشئ ذي المعلمات الأخرى قيد الاستخدام. يكون مطلوبًا عندما تريد إنشاء كائن للفئة من خلال عدم تمرير أي وسائط في المنشئ.

**@AllArgsConstructor:** يتم استخدام هذا التعليق التوضيحي لإنشاء مُنشئ ذي parameters إنه مطلوب عندما تريد إنشاء كائن للفئة عن طريق تمرير القيم الأولية للحقول في المنشئ.

**@Data:** هذا التعليق التوضيحي عبارة عن تعليق توضيحي مختصر لمجموعات التعليقات التوضيحية ToString وGetter وSetter وRequiredArgsConstructor في تعليق توضيحي واحد .

ـــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــ

**8. التحقق - Validation**

هو التحقق من صحة الإدخال، المعروف أيضًا باسم التحقق من صحة البيانات، هو اختبار أي إدخال (أو بيانات) يقدمها مستخدم مقابل المعايير المتوقعة. يمنع التحقق البيانات الخبيثة أو غير المؤهلة من الدخول إلى النظام. كما يجب أن تتحقق التطبيقات من جميع المدخلات التي يتم إدخالها في النظام وتتحقق من صحتها لمنع الهجمات والأخطاء.

**8.1 مكتبة** **The Bean Validation API :**

هي إحدى مواصفات Java التي تُستخدم لتطبيق قيود على نموذج الكائن عبر التعليقات التوضيحية ، ويمكننا استخدامها من خلال اختيارها من قائمة المكتبات عند بدء المشروع . يمكننا التحقق من صحة طول ورقم وصيغة القيمة المدخلة وما إلى ذلك. يمكننا أيضًا تقديم عمليات تحقق مخصصة.

|  |  |
| --- | --- |
| **التعليق** | **الوصف** |
| @NotNull | يحدد أن القيمة لا يمكن أن تكون خالية |
| @Min | يحدد أن الرقم يجب أن يكون مساويًا أو أكبر من القيمة المحددة |
| @Max | يحدد أن الرقم يجب أن يكون مساويًا أو أقل من القيمة المحددة |
| @Size | يحدد أن الحجم يجب أن يكون مساويًا للقيمة المحددة |
| @Pattern | يحدد أن التسلسل يتبع specified regular expression |
| @NotEmpty | التحقق من أن القيمة ليست فارغة؛ يمكن تطبيقها على قيم String او Collection او Map او Array |
| @NotBlank | يمكن تطبيقه فقط على القيم النصية والتحقق من أن الخاصية ليست خالية أو مسافة بيضاء |
| @Positive & @PositiveOrZero | تنطبق على القيم الرقمية وتتحقق من أنها موجبة تمامًا أو صفر |
| @Negative & @NegativeOrZero | تنطبق على القيم الرقمية وتتحقق من أنها سالبة تمامًا أو صفر |
| @Past & @PastOrPresent | التحقق من أن قيمة التاريخ في الماضي أو الحاضر؛ يمكن تطبيقه على أنواع المتغيرات بما في ذلك تلك المضافة في Java 8 |
| @Future & @FutureOrPresent | التحقق من أن قيمة التاريخ في المستقبل، أو الحاضر |
| @Email | التحقق من أن القيمة بريد إلكتروني صالح |
| @AssertTrue | التحقق من ان القيمة المدخلة True |
| @AssertFalse | التحقق من ان القيمة المدخلة False |

**8.2 رموز حالة الاستجابة**  **HTTP status codes -**

تشير رموز حالة الاستجابة إلى ما إذا كان قد تم إكمال طلب HTTP معين بنجاح. يتم تجميع الردود في خمس فئات:

|  |  |
| --- | --- |
| **الرمز** | **الحالة** |
| 1XX | رموز الحالة: طلبات إعلامية |
| 2XX | رموز الحالة: الطلبات الناجحة |
| 3XX | رموز الحالة: عمليات إعادة التوجيه |
| 4XX | رموز الحالة: أخطاء العميل |
| 5XX | رموز الحالة: أخطاء الخادم |

**رمز الحالة 1XX طلبات إعلامية**

|  |  |
| --- | --- |
| **الرمز** | **الحالة** |
| 100 | متابعة: كل شيء على ما يرام حتى الآن ويجب على العميل متابعة الطلب أو تجاهله إذا كان قد انتهى بالفعل |
| 101 | تبديل البروتوكولات: البروتوكول الذي يقوم الخادم بالتبديل إليه حسب طلب العميل الذي أرسل الرسالة بما في ذلك عنوان طلب الترقية |
| 102 | قيد المعالجة: وافق الخادم على الطلب الكامل، لكنه لا يزال قيد المعالجة |
| 103 | تلميحات مبكرة: السماح لوكيل المستخدم ببدء تحميل الموارد مسبقًا أثناء تحضير الخادم للاستجابة |

**رمز الحالة 2XX الطلبات الناجحة**

|  |  |
| --- | --- |
| **الرمز** | **الحالة** |
| 200 | حسنًا: طلب ناجح |
| 201 | تم الإنشاء: أقر الخادم بالمورد الذي تم إنشاؤه |
| 202 | مقبول: تم استلام طلب العميل، ولكن الخادم لا يزال يقوم بمعالجته |
| 203 | معلومات غير موثوقة: الاستجابة التي أرسلها الخادم إلى العميل ليست هي نفسها عندما أرسلها الخادم |
| 204 | لا يوجد محتوى: قام الخادم بمعالجة الطلب، ولكنه لا يقدم أي محتوى |
| 205 | إعادة تعيين المحتوى: يجب على العميل تحديث نموذج المستند |
| 206 | المحتوى الجزئي: يرسل الخادم جزءًا فقط من المورد |
| 207 | الحالة المتعددة: نص الرسالة التالي هو افتراضيًا رسالة XML ويمكن أن يحتوي على عدد من رموز الاستجابة المنفصلة |

**رمز الحالة 3XX عمليات اعادة التوجيه**

|  |  |
| --- | --- |
| **الرمز** | **الحالة** |
| 300 | اختيارات متعددة: الطلب الذي قدمه العميل له عدة استجابات محتملة |
| 301 | تم النقل بشكل دائم: يخبر الخادم العميل أن المورد الذي يبحثون عنه قد تم نقله بشكل دائم إلى عنوان URL آخر. سيتم إعادة توجيه جميع المستخدمين والروبوتات إلى عنوان URL الجديد. إنه رمز حالة مهم جدًا لتحسين محركات البحث |
| 302 | تم العثور عليها: تم نقل موقع ويب أو صفحة إلى عنوان URL مختلف مؤقتًا. إنه رمز حالة آخر يتعلق بـ SEO |
| 303 | انظر أخرى: يخبر هذا الرمز العميل أن الخادم لا يعيد توجيههم إلى المورد المطلوب، ولكن إلى صفحة أخرى |
| 304 | غير معدّل: لم يتم تغيير المورد المطلوب منذ الإرسال السابق |
| 305 | استخدام الوكيل: لا يمكن للعميل الوصول إلى المورد المطلوب إلا من خلال وكيل تم تقديمه في الاستجابة |
| 307 | إعادة توجيه مؤقتة: يخبر الخادم العميل أن المورد الذي يبحث عنه قد تمت إعادة توجيهه مؤقتًا إلى عنوان URL آخر. إنها ذات صلة بأداء تحسين محركات البحث |
| 308 | إعادة التوجيه الدائم: يخبر الخادم العميل أن المورد الذي يبحث عنه قد تمت إعادة توجيهه إلى عنوان URL آخر |

**رمز الحالة 4XX أخطاء العميل**

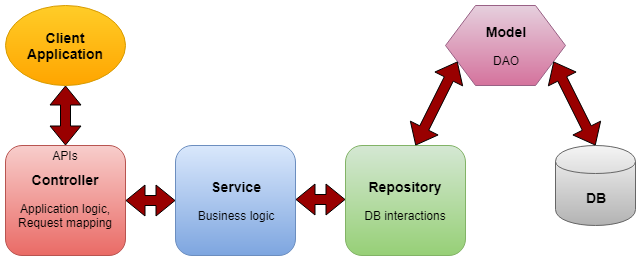
|  |  |
| --- | --- |
| **الرمز** | **الحالة** |
| 400 | طلب خاطئ: يرسل العميل طلبًا يحتوي على بيانات غير كاملة أو بيانات سيئة الإنشاء أو بيانات غير صالحة |
| 401 | غير مصرح به: التفويض مطلوب للعميل للوصول إلى المورد المطلوب |
| 403 | ممنوع: المورد الذي يحاول العميل الوصول إليه محظور |
| 404 | غير موجود: يمكن الوصول إلى الخادم، ولكن الصفحة المحددة التي يبحث عنها العميل ليست كذلك |
| 405 | الطريقة غير مسموح بها: استلم الخادم الطلب وتعرف عليه، لكنه رفض طريقة الطلب المحددة |
| 406 | غير مقبول: لا يدعم موقع الويب أو تطبيق الويب طلب العميل ببروتوكول معين |
| 407 | مطلوب مصادقة الوكيل: رمز الحالة هذا مشابه لـ 401 غير مصرح به. الاختلاف الوحيد هو أن التفويض يجب أن يتم بواسطة وكيل |
| 408 | مهلة الطلب: انتهت صلاحية الطلب الذي أرسله العميل إلى خادم موقع الويب |
| 409 | التعارض: يتعارض الطلب الذي تم إرساله مع العمليات الداخلية للخادم |
| 410 | ذهب: تم ​​مسح المورد الذي يريد العميل الوصول إليه نهائيًا |

**رمز الحالة 5XX أخطاء الخادم**

|  |  |
| --- | --- |
| **الرمز** | **الحالة** |
| 500 | خطأ داخلي في الخادم: واجه الخادم موقفًا لا يمكنه التعامل معه أثناء معالجة طلب العميل |
| 501 | لم يتم التنفيذ: لا يعرف الخادم طريقة الطلب التي أرسلها العميل أو يمكنه حلها |
| 502 | مدخل غير صالح: كان الخادم يعمل كبوابة أو خادم وكيل وتلقى رسالة غير صالحة من خادم وارد |
| 503 | الخدمة غير متوفرة: قد يكون الخادم معطلاً ولا يمكن معالجة طلب العميل. يعد رمز حالة HTTP أحد أكثر مشكلات الخادم شيوعًا التي يمكنك مواجهتها على الويب |
| 511 | مطلوب مصادقة الشبكة: يحتاج العميل إلى المصادقة على الشبكة قبل أن يتمكن من الوصول إلى المورد |

**9. طبقة الخدمة – Service Layer**

طبقة الخدمة هي طبقة تسهل الاتصال بين وحدة التحكم وطبقة المستودع. بالإضافة إلى ذلك، يتم تخزين منطق الأعمال فيها. طبقة الخدمة تحدد الدوال التي توفرها، وكيفية الوصول إليها، وما يجب تمريره والحصول عليه في المقابل. قد يبدأ منطق الأعمال بسيطًا في البداية، ولا يقدم أكثر من عمليات CRUD بسيطة، لكن هذا لا يعني أنه سيبقى على هذا النحو إلى الأبد. بمجرد أن تحتاج إلى التعامل مع منطق معقد أكثر، لن يعد الأمر متعلق بطلبات التوجيه من وحدة التحكم مباشرة إلى المستودع فقط، ولكن أيضًا التحقق من الوصول وتصفية البيانات. قد تحتاج الطلبات إلى التحقق من الصحة والتحقق من التناسق قبل الوصول إلى قاعدة البيانات، ويمكن تطبيق قواعد وعمليات إضافية، حتى تحصل خدماتك على قيمة أكبر بمرور الوقت. وإحدى مزايا طبقة الخدمة هي إمكانية إعادة استخدام الدوال في أكثر من وحدة تحكم.



يمكن استخدام التعليق التوضيحي **@Service**للإشارة إلى ان الـ class يؤدي خدمة أو وظيفة معينة في تطبيق Spring Boot. ستستخدمه لتحديد أن الـ class ينفذ مهام منطق الأعمال أو معالجة البيانات.

**10.مقدمة في قواعد البيانات ( Data bases)**

**10.1 مفهوم قواعد البيانات :**

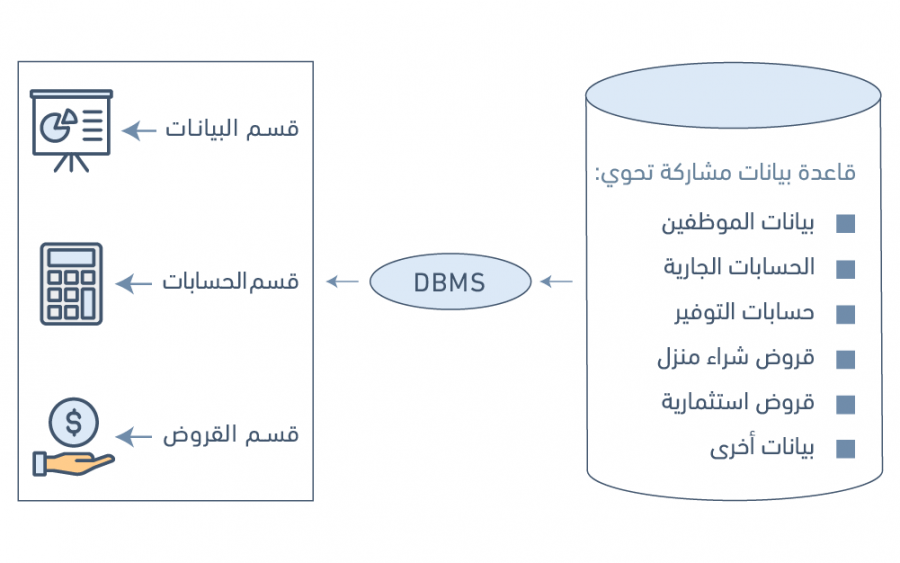
مجموعة من البيانات المنظمة والمرتبة بطريقة يسهل الوصول إليها وإداراتها والتعديل عليها .

**10.2 مميزات استخدام قواعد البيانات :**

* حفظ أكبر كم من المعلومات بطريقة منظمة .
* سهولة تحديث المعلومات وإجراء أي تعديلات عليها
* خصوصية البيانات المحفوظة حيث لا يمكن لأحد الاطلاع عليها .
* إمكانية الوصول إليها بسرعة والاسترجاع التلقائي
* القيام بعمليات الإضافة والحذف والمسح والتعديل عليها .

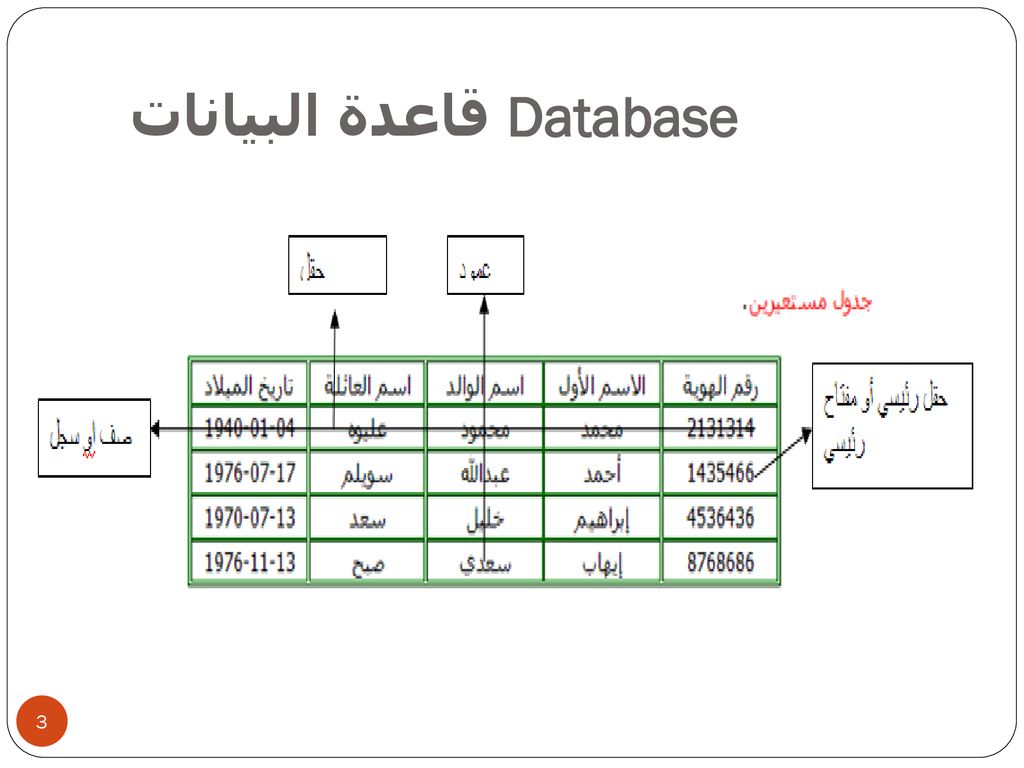
**10.3 نظام إدارة قواعد البيانات (DBMS (:**

هو عبارة عن مجموعة من البرامج التي تدير بنية قاعدة البيانات وتتحكم في الوصول إلى البيانات المخزنة في قاعدة البيانات .



**10.4 قاعدة البيانات العلاقية :** هي مجموعة من المعلومات المنظمة بشكل صحيح مع علاقات محددة بوضوح بحيث يمكن الوصول إليها واسترجاعها بسهولة.  ويتم تخزينها داخل الجداول .

**الجداول :** هي هيكل لشكل البيانات ونوعها داخل صفوف وأعمدة .

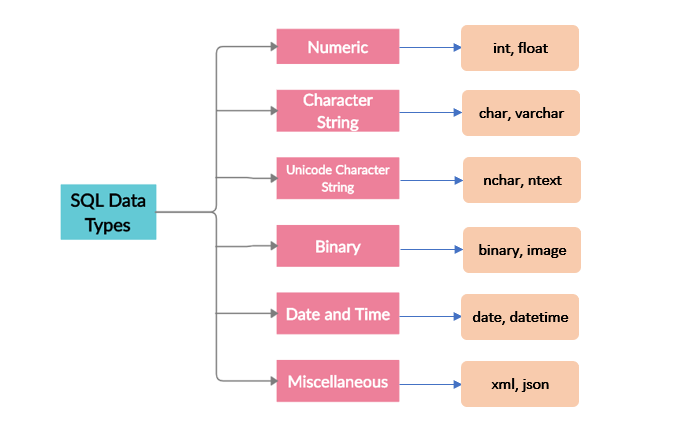


**أشهر الأمثلة استخداماً :**

* PostgreSQL
* Oracle
* MySQL

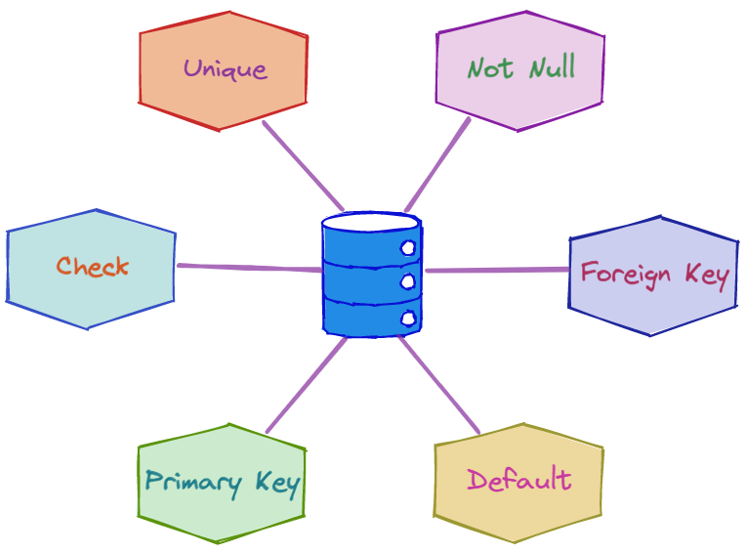
**نوع البيانات التي يمكن تخزينها في قاعدة البيانات العلاقية**

* 1. بيانات لها هيكلة واضحة
  2. بيانات لديها علاقات فيما بينها.

**10.5 أنواع البيانات :**

**10.6 القيود ( (constraints:**

هي قواعد يضعها صانع الجدول عند الجدول أو العامود ليتحكم بالقيم

**أنواع القيود :**

**ماهو المفتاح الأساسي والأجنبي ؟**

المفتاح الأساسي primary key) ): هو صفة فريدة خاصة لجدول محدد .

المفتاح الأجنبي foreign key) (: هو مفتاح رئيسي من جدول آخر للتعبير عن وجود علاقة بينهما.

**10.7 أدوات الاستعلام : (Query tools)**

هو نظام يمكننا من الاتصال ب قاعدة البيانات العلاقية وكتابة أوامرSQL عليها .

**أمثلة شهيرة الاستخدام :**

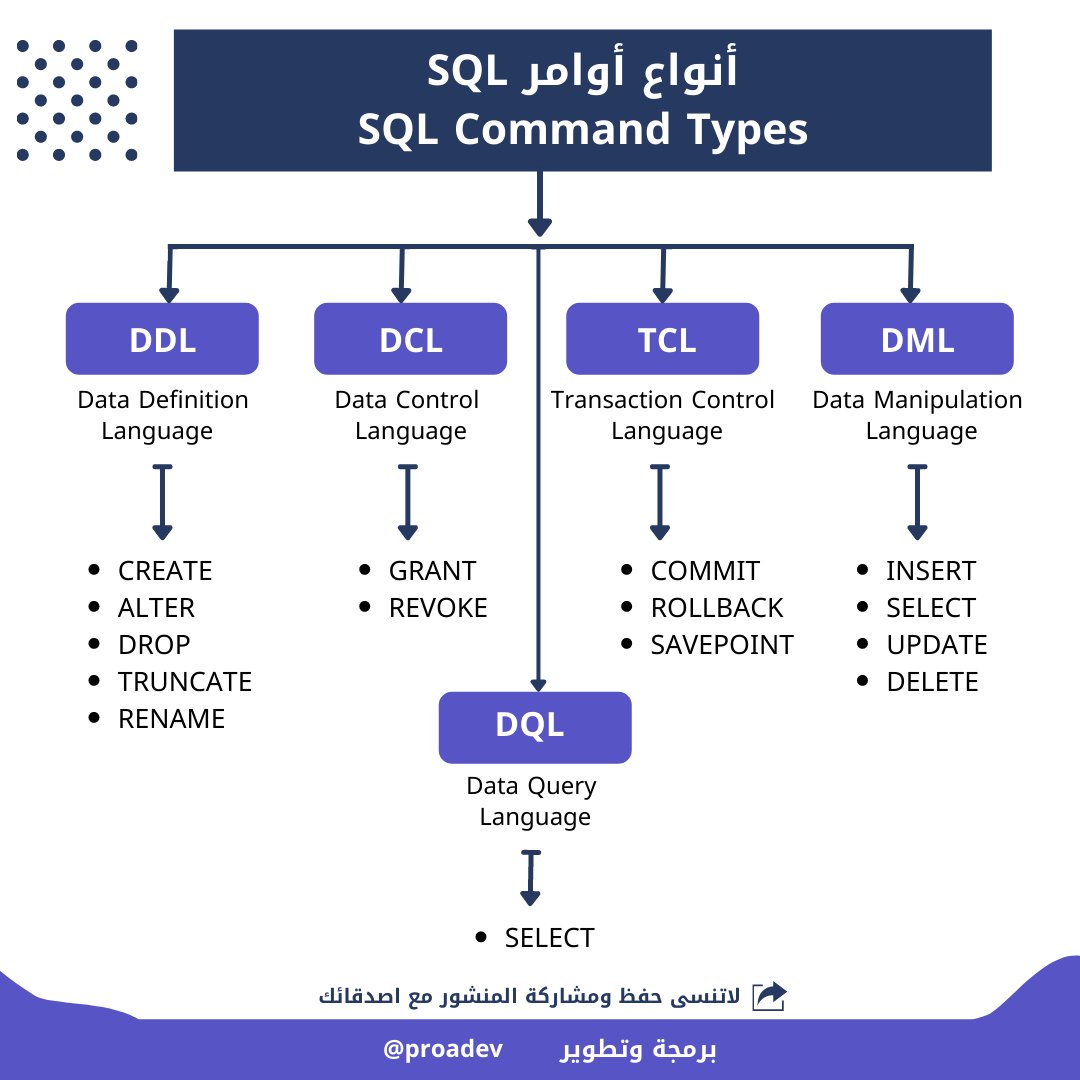
* Php MyAdmin
* SQL server
* MySQL Workbench
* DataGrip

**10.8 لغة الاستعلام الهيكلية (SQL ) :**

هي لغة برمجه لديها عمليات لتحديد، ترتيب، تعديل وإجراء العمليات الحسابية على بيانات محفوظه في أنظمة قواعد البيانات العلائقية (RDBMS).

**ما تمكننا هذه اللغة من فعله :**

1. إنشاء قواعد البيانات والجداول
2. إضافة البيانات (C)
3. الاستعلام عن بيانات داخل الجدول (R)
4. التعديل على البيانات (U)
5. حذف البيانات (D)



ـــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــ

**11. إدارة البيانات العلاقية باستخدام JPA**

**11.1** **مفهوم ( Object Relational Mapping (ORM**

ORM هو أسلوب يستخدم في إنشاء "جسر" بين البرامج الكائنية وقواعد البيانات العلائقية. بعبارة أخرى، يمكنك رؤية ORM على أنها الطبقة التي تربط البرمجة الموجهة للكائنات (OOP) بقواعد البيانات العلائقية.

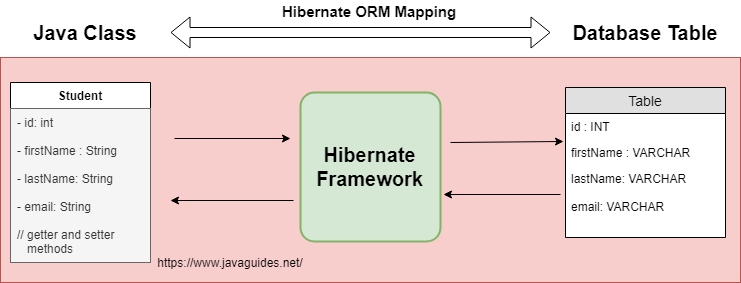
عند التفاعل مع قاعدة بيانات باستخدام لغات OOP، سيتعين عليك إجراء عمليات مختلفة مثل إنشاء بيانات من قاعدة بيانات وقراءتها وتحديثها وحذفها (CRUD). يمكنك استخدام SQL لإجراء هذه العمليات في قواعد البيانات العلائقية. استخدام SQL لهذا الغرض ليس بالضرورة فكرة سيئة، ولكن ORM وادواته يساعد في تبسيط التفاعل بين قواعد البيانات العلائقية ولغات OOP المختلفة.

**11.2** **أداة ORM Tool - ORM**

أداة ORM هي برنامج مصمم لمساعدة مطوري OOP على التفاعل مع قواعد البيانات العلائقية. لذا بدلاً من إنشاء برنامج ORM خاص بك من البداية، يمكنك الاستفادة من هذه الأدوات.

**Hibernate 11.3**

Hibernate هي أداة مفتوحة المصدر لـ ORM توفر إطار عمل لتعيين نماذج المجال الكائنية (object-oriented domain models) إلى قواعد البيانات العلاقية لتطبيقات الويب. يرشد إطار عمل Hibernate ORM لتعيين classes Java إلى جداول قاعدة البيانات، وأنواع بيانات Java لأنواع بيانات SQL ويوفر الاستعلام والاسترجاع.

 يقلل Hibernate سطور الاوامر البرمجية من خلال الحفاظ على تعيين جدول الكائنات نفسه ويعيد النتيجة إلى التطبيق في شكل كائنات Java. إنه يعفي المبرمج من المعالجة اليدوية للبيانات المستمرة، وبالتالي تقليل وقت التطوير وتكلفة الصيانة.

**11.4 مفهوم JPA (Java Persistence API  )**

JPA هي إحدى مواصفات Java لإدارة البيانات العلاقية في تطبيقات Java. يسمح لنا بالوصول إلى البيانات واستمرارها بين كائن Javaوقاعدة البيانات العلاقية. يعمل JPA كجسر بين نماذج المجال الكائنية وأنظمة قواعد البيانات العلاقية (ORM).

نظرًا لأن JPA هي مجرد مواصفات، فهي لا تقوم بأي عملية بمفردها; يتطلب التنفيذ. لذا، فإن أدوات ORM مثل Hibernate تنفذ مواصفات JPA لاستمرار البيانات.

**11.5** **ربط مشروع بقاعدة البيانات Connect Spring Project to DB**

يوفر Spring Boot مكتبات (Dependencies) لتوصيل تطبيق Spring بقاعدة البيانات العلاقية بكفاءة. وهي:

* Spring Data JPA
* MySQL Driver

وللربط مع قاعدة البيانات يجب إضافة بعض التكوينات الى ملف application.properties:

spring.datasource.url=jdbc:mysql://localhost:3306/schoolDB  
spring.datasource.username=root  
spring.datasource.password=  
spring.jpa.database-platform=org.hibernate.dialect.MySQL5Dialect  
  
spring.jpa.show-sql=true  
spring.jpa.generate-ddl=true  
spring.jpa.hibernate.ddl-auto=create-drop  
  
server.error.include-message=*always*server.error.include-stacktrace=*always*

ولتعريف كيان بسيط بقاعدة بيانات، قم بإضافة تعليقات توضيحية لكائن :Java

@Data @NoArgsConstructor @AllArgsConstructor @Entity  
public class Student {  
  
 @Id  
 private Integer id;  
 private String name;  
}

تم إضافة تعليق توضيحي على Student Class باستخدام@Entity ، للإشارة إلى أنه أحد كيانات JPA(نظرًا لعدم وجود التعليق التوضيحي @Table ، فمن المفترض أن هذا الكيان قد تم تعيينه لجدول يسمى student).

تمت إضافة التعليق التوضيحي @Id على id حتى يتعرف JPA عليها على أنها معرف الكائن.

**ــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــ**

**12. وحدة المستودع Repository Layer) )**

يوفر لنا Spring boot مستودعًا مسؤولاً عن إجراء عمليات مختلفة على الكائن. لإنشاء أي مستودع للـClass ، يتعين علينا استخدام التعليق التوضيحي @Repository.

قم بإنشاء Repository Interface يعمل مع كيان Student على النحو التالي:

@Repository  
public interface StudentRepository extends JpaRepository<Student, Integer> {

}

يستخدم التعليق التوضيحي @Repositoryللإشارة إلى أن Class يوفر آلية للتخزين والاسترجاع والبحث والتحديث والحذف على الكائنات.

يقوم StudentRepository بوراثة interface JpaRepository. يتم تحديد نوع الكيان والمعرف الذي يعمل معه، في الـ generic parameters في JpaRepository. بوراثة JpaRepository، يرث StudentRepository عدة طرق للعمل مع Student، بما في ذلك طرق حفظ كيانات Student وحذفها والعثور عليها.

**12.1 انواع الـ Query باستخدام :Jpa**

**النوع الاول:**

بإستخدام الدوال الجاهزة، يوفر لنا Jpa عدة دوال جاهزة مثل: save و delete

**النوع الثاني:**

بإنشاء الدوال الجاهزة

|  |  |  |
| --- | --- | --- |
| **Keyword** | **Sample** | **JPQL snippet** |
| findBy | findByAppleName | where a.apple\_name= ?1 |
| And | findByLastnameAndFirstname | … where x.lastname = ?1 and x.firstname = ?2 |
| Or | findByLastnameOrFirstname | … where x.lastname = ?1 or x.firstname = ?2 |
| Is,Equals | findByFirstname, findByFirstnameIs, findByFirstnameEquals | … where x.firstname = ?1 |
| Between | findByStartDateBetween | … where x.startDate between ?1 and ?2 |
| LessThan | findByAgeLessThan | … where x.age < ?1 |
| LessThanEqual | findByAgeLessThanEqual | … where x.age <= ?1 |
| GreaterThan | findByAgeGreaterThan | … where x.age > ?1 |
| GreaterThanEqual | findByAgeGreaterThanEqual | … where x.age >= ?1 |
| After | findByStartDateAfter | … where x.startDate > ?1 |
| Before | findByStartDateBefore | … where x.startDate < ?1 |
| IsNull | findByAgeIsNull | … where x.age is null |
| IsNotNull,NotNull | findByAge(Is)NotNull | … where x.age not null |
| Like | findByFirstnameLike | … where x.firstname like ?1 |
| NotLike | findByFirstnameNotLike | … where x.firstname not like ?1 |
| StartingWith | findByFirstnameStartingWith | … where x.firstname like ?1 (parameter bound with appended %) |
| EndingWith | findByFirstnameEndingWith | … where x.firstname like ?1 (parameter bound with prepended %) |
| Containing | findByFirstnameContaining | … where x.firstname like ?1 (parameter bound wrapped in %) |
| OrderBy | findByAgeOrderByLastnameDesc | … where x.age = ?1 order by x.lastname desc |
| Not | findByLastnameNot | … where x.lastname <> ?1 |
| In | findByAgeIn(Collection<Age> ages) | … where x.age in ?1 |
| NotIn | findByAgeNotIn(Collection<Age> ages) | … where x.age not in ?1 |
| True | findByActiveTrue() | … where x.active = true |
| False | findByActiveFalse() | … where x.active = false |
| IgnoreCase | findByFirstnameIgnoreCase | … where UPPER(x.firstame) = UPPER(?1) |

**3- النوع الثالث:**

بكتابة Java Persistence Query language (JPQL)

JPQLهي لغة الاستعلام المحددة في مواصفات JPA. يتم استخدامها لإنشاء استعلامات مقابل الكيانات المخزنة في قاعدة البيانات العلاقية. تم تطوير JPQL على أساس بناء جملة SQL. لكنها لن تؤثر على قاعدة البيانات بشكل مباشر.

@Repository  
public interface StudentRepository extends JpaRepository<Student, Integer> {  
  
 @Query("select s from Student s where s.name = ?1")  
 Student findByName(String name);  
  
}

**ـــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــ**

**13. نصائح التحكم (Controller Advise )**

**13.1 مفهوم الاستثناءات في جافا :**

الاستثناء (**Exceptions**) هو عبارة عن خطأ يحدث أثناء تشغيل البرنامج يؤدي إلى إيقافه بشكل غير طبيعي .

ظهور خطأ يؤدي إلى إيقاف البرنامج هو أمر سيئ جداً لأنه يؤدي إلى نفور عدد كبير من المستخدمين وعدم رغبتهم في العودة إلى استخدام هذا البرنامج مجدداً ، ولابد أن نجعل البرنامج يلتقط هذه الأخطاء في حال حدثت ، لجعل البرنامج شغال دائماً في نظر المستخدم و لا يظهر له أي أخطاء.

**13.2 بعض الأسباب لحدوث استثناء في البرنامج :**

* في حال إدخال قيمة لا تتطابق مع نوع المتغير الذي ستخزن فيه .
* في حال إدخال رقم index غير موجود في مصفوفة أو متغير نوعه String .
* في حال كان البرنامج يتصل بالشبكة وفجأة انقطع الاتصال .
* في حال كان البرنامج يحاول قراءة معلومات من ملف نصي وكان هذا الملف غير موجود.

**13.3 بناء الاستثناءات في جافا :**

تم تقسيم الاستثناءات في جافا إلى عدة أنواع وكل نوع تم تمثيله في كلاس منعزل ، وجميع هذه الكلاسات ترث من كلاس أساسي اسمه (Exception ) .

**13.4 الاستثناءات في Spring Boot :**

لكي يستطيع السيرفر التعامل مع هذه الاستثناءات لابد من إنشاء كلاس يسمى Controller Advice وهو طبقة التحكم المسؤولة عن التعامل مع الأخطاء التي قد تحدث أثناء عمل السيرفر لتجنب توقفه من خلال استقباله لهذه الأخطاء وإرسال نتيجة بنوع الخطأ ومكانه ، ولابد من تعريف Methods داخل هذا الكلاس لتتعامل مع كل نوع من أنواع هذه الأخطاء . وسنقوم بإضافة التعليق التوضيحي @**RestControllerAdvise** ليأخذ الكلاس خاصية التعامل مع الاستثناءات .

// Our Exception  
@ExceptionHandler(value = ApiException.class)  
public ResponseEntity ApiException(ApiException e){  
 String message=e.getMessage();  
 return ResponseEntity.*status*(400).body(message);  
}  
  
// Server Validation Exception  
@ExceptionHandler(value = MethodArgumentNotValidException.class)  
public ResponseEntity<ApiResponse> MethodArgumentNotValidException(MethodArgumentNotValidException e) {  
 String msg = e.getFieldError().getDefaultMessage();  
 return ResponseEntity.*status*(400).body(new ApiResponse(msg));  
}  
  
// SQL Constraint Exception  
@ExceptionHandler(value = SQLIntegrityConstraintViolationException.class)  
public ResponseEntity<ApiResponse> SQLIntegrityConstraintViolationException(SQLIntegrityConstraintViolationException e){  
 String msg=e.getMessage();  
 return ResponseEntity.*status*(400).body(new ApiResponse(msg));  
}  
  
// Method not allowed Exception  
@ExceptionHandler(value = HttpRequestMethodNotSupportedException.class)  
public ResponseEntity<ApiResponse> HttpRequestMethodNotSupportedException(HttpRequestMethodNotSupportedException e) {  
 String msg = e.getMessage();  
 return ResponseEntity.*status*(400).body(new ApiResponse(msg));  
}  
  
// Json parse Exception  
@ExceptionHandler(value = HttpMessageNotReadableException.class)  
public ResponseEntity<ApiResponse> HttpMessageNotReadableException(HttpMessageNotReadableException e){  
 String msg = e.getMessage();  
 return ResponseEntity.*status*(400).body(new ApiResponse(msg));  
}  
  
// TypesMissMatch Exception  
@ExceptionHandler(value = MethodArgumentTypeMismatchException.class)  
public ResponseEntity<ApiResponse> MethodArgumentTypeMismatchException(MethodArgumentTypeMismatchException e) {  
 String msg = e.getMessage();  
 return ResponseEntity.*status*(400).body(new ApiResponse(msg));  
}

**13.5 إنشاء استثناء Exception :**

يمكننا كذلك انشاء Exception خاص وبداخله رسالتنا الخاصة من خلال إنشاء كلاس اسمه كما هو واضح في الأعلى ApiException ويرث من كلاس RuntimeException

public class ApiException extends RuntimeException {  
 public ApiException(String message){  
 super(message);  
 }  
}

وسيمكننا الآن أن نرمي استثناء في أي مكان نتوقع فيه الخطأ مع رسالتنا الخاصة .

**مثال على ذلك :** نريد أن نبحث عن Blog في قاعدة البيانات بواسطة الid الخاص بها وفي حال لم تكن هذا المدونة موجودة سنقوم برمي استثناء :

public Blog getBlogbyId(Integer id){  
 Blog blog=blogRepository.findBlogById(id);  
 if(blog==null){  
 throw new ApiException("wrong ID");  
 }  
 return blog;  
  
}

**ـــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــ**

**14. أنواع العلاقات في JPA Relationship – JPA**

العلاقة، في سياق قواعد البيانات، هي حالة توجد بين جدولي قاعدة بيانات علاقية عندما يحتوي أحدهما على مفتاح خارجي يشير إلى المفتاح الأساسي للجدول الآخر. تسمح العلاقات لقواعد البيانات العلاقية بتقسيم البيانات وتخزينها في جداول مختلفة، مع ربط عناصر البيانات.

هناك 4 أنواع للعلاقات، وهي: OneToOne – OneToMany – ManyToOne - ManyToMany.

في البداية، لنعرف بعض العلاقات في المثال:

* Teacher – Course
* Teacher – Address
* Student – Course

**OneToMany/ManyToOne 14.1**

كما يوحي اسمها، فهي علاقة تربط كيانًا بالعديد من الكيانات الأخرى.

في مثالنا، سيكون هذا مدرسًا ودوراته التدريبية. يمكن للمدرس أن يعطي دورات متعددة، لكن الدورة يتم تقديمها من قبل مدرس واحد فقط (من منظور ManyToOne - العديد من الدورات لمعلم واحد).

قبل الغوص في تفاصيل كيفية تعيين هذه العلاقة، دعنا ننشئ كياناتنا:

@Setter @Getter  
@AllArgsConstructor @NoArgsConstructor @Entity  
public class Teacher {  
  
 @Id

@GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Integer id;  
 @NotEmpty(message = "Name must not be null")  
 @Column(nullable = false)  
 private String name;

}

@Setter @Getter  
@AllArgsConstructor @NoArgsConstructor @Entity  
public class Course {  
  
 @Id

@GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Integer id;  
 @NotEmpty(message = "Title must not be null")  
 @Column(nullable = false)  
 private String title;

}

الآن، يجب أن تتضمن حقول Teacher Class قائمة بالـ Courses نظرًا لأننا نرغب في تعيين هذه العلاقة في قاعدة بيانات، ويجب تعيين حقل Teacher في Course Class ايضاً، فسنقوم باستخدام التعليق التوضيحي @OneToMany في Teacher Class واستخدام @ManyToOne في :Course Class

@Setter @Getter  
@AllArgsConstructor @NoArgsConstructor @Entity  
public class Teacher {  
  
 @Id

@GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Integer id;  
 @NotEmpty(message = "Name must not be null")  
 @Column(nullable = false)  
 private String name;

@OneToMany(mappedBy = "teacher")  
 private Set<Course> courses;

}

@Setter @Getter  
@AllArgsConstructor @NoArgsConstructor @Entity  
public class Course {  
  
 @Id

@GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Integer id;  
 @NotEmpty(message = "Title must not be null")  
 @Column(nullable = false)  
 private String title;

@ManyToOne  
 @JoinColumn(name = "teacher\_id" , referencedColumnName = "id")  
 @JsonIgnore  
 private Teacher teacher;

}

كيف تعكس JPA هذه العلاقة في قاعدة البيانات؟ بشكل عام، لهذا النوع من العلاقات، يجب علينا استخدام مفتاح خارجي في جدول.

يقوم JPA بهذا من أجلنا، بالنظر إلى مدخلاتنا حول كيفية التعامل مع العلاقة. يتم ذلك عبر التعليق التوضيحي @JoinColumn ، سيؤدي استخدام هذا التعليق التوضيحي إلى إخبار JPA بأن جدول Course يجب أن يحتوي على عمود مفتاح خارجي teacher\_id يشير إلى عمود معرف جدول Teacher.

المهم أن نلاحظ هنا استخدام علامة mappedBy في التعليق التوضيحي OneToMany. بدونها، لن يكون لدينا علاقة ثنائية الاتجاه. سيكون لدينا علاقتان في اتجاه واحد. يقوم كلا الكيانين بتعيين المفاتيح الخارجية للكيان الآخر.

باستخدامه، نخبر JPA أن الحقل تم تعيينه بالفعل بواسطة كيان آخر. تم تعيينه بواسطة حقل Teacher في كيان Course.

@JsonIgnoreيتم استخدامه على مستوى الحقل لتمييز خاصية أو قائمة بالخصائص التي سيتم تجاهلها.

**OneToOne 14.2**

إنها علاقة تربط كيان واحد بسجل واحد بالضبط في كيان آخر.

في مثالنا، سيكون هذا مدرسًا وعنوانه. لكل مدرس عنوان واحد فقط، وكل عنوان تابع لمدرس واحد فقط.

@Setter @Getter  
@AllArgsConstructor @NoArgsConstructor @Entity  
public class Teacher {  
  
 @Id

@GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Integer id;  
 @NotEmpty(message = "Name must not be null")  
 @Column(nullable = false)  
 private String name;

@OneToMany(mappedBy = "teacher")  
 private Set<Course> courses;

@OneToOne(cascade = CascadeType.*ALL* , mappedBy = "teacher")  
 @PrimaryKeyJoinColumn  
 private Address address;

}

@Setter @Getter  
@AllArgsConstructor @NoArgsConstructor @Entity  
public class Address {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Integer id;  
 @NotEmpty(message = "Name must not be null")  
 @Column(nullable = false)  
 private String name;  
  
 @OneToOne  
 @JoinColumn(name = "teacher\_id" , referencedColumnName = "id")  
 @MapsId  
 @JsonIgnore  
 private Teacher teacher;  
}

نقوم بتعيين ارتباط حقيقي OneToOne باستخدام المفاتيح الأساسية المشتركة.

يوضح التعليق التوضيح @PrimaryKeyJoinColumn أنه يتم استخدام المفتاح الأساسي للكيان كقيمة مفتاح خارجي للكيان المرتبط.

يتم استخدام @MapsId إذا كان المفتاح الأساسي للكيان المرتبط هو نفسه المفتاح الأساسي للكيان الاصلي. من خلال تحديد التعليق التوضيحي، يمكنك ملء المفتاح الأساسي للكيان المرتبط تلقائيًا من للكيان الأصلي.

تعني كلمة Cascading عندما نقوم بتنفيذ بعض الإجراءات على الكيان المستهدف، فسيتم تطبيق نفس الإجراء على الكيان المرتبط.

**ManyToMany 14.3**

في قاعدة البيانات، تتضمن علاقة ManyToMany جدولًا متوسطًا يشير إلى كلا الجدولين الآخرين.

لحسن الحظ بالنسبة لنا ، تقوم JPA بمعظم العمل ، وعلينا فقط إلقاء بعض التعليقات التوضيحية هناك ، وهي تتولى الباقي نيابةً عنا.

لذلك، على سبيل المثال، ستكون علاقة ManyToMany هي العلاقة بين الطالب والدورة التدريبية حيث يمكن للطالب حضور دورات متعددة ، ويمكن أن يتبع الدورة العديد من الطلاب.

@Data @NoArgsConstructor @AllArgsConstructor @Entity  
public class Student {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Integer id;  
 private String name;  
  
 @ManyToMany(mappedBy = "students")  
 @JsonIgnore  
 private List<Course> courses;  
}

@Setter @Getter  
@AllArgsConstructor @NoArgsConstructor @Entity  
public class Course {  
  
 @Id

@GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Integer id;  
 @NotEmpty(message = "Title must not be null")  
 @Column(nullable = false)  
 private String title;

@ManyToOne  
 @JoinColumn(name = "teacher\_id" , referencedColumnName = "id")  
 @JsonIgnore  
 private Teacher teacher;

@ManyToMany  
 private List<Student> students;

}

**ـــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــ**

**15. أمان الخادم (Spring Security )**

عبارة عن إطار مصادقة قوي وقابل للتخصيص بدرجة كبيرة وإطار عمل للتحكم في الوصول.

إنه المعيار الواقعي لتأمين التطبيقات القائمة على Spring .

و هو إطار عمل يركز على توفير كل من المصادقة (authentication) وتعني أنه يمكن للمستخدم الوصول إلى النظام الحالي ، والترخيص (authorization) و يعني تحديد الحقوق الوظيفية التي يتمتع بها المستخدم.

مثل جميع مشاريع Spring ، تكمن القوة الحقيقية لـ Spring Security في مدى سهولة توسيعها لتلبية المتطلبات المخصصة.

**15.1 مميزاته :**

* دعم شامل وقابل للتوسيع لكل من المصادقة والترخيص
* لحماية من الهجمات مثل session fixation ، clickjacking ، cross site request forgery ، وما إلى ذلك
* تكامل Servlet API
* تكامل اختياري مع Spring Web MVC
* حماية CSRF
* دعم تكوين جافا

**15.2** **المصادقة (Authentication ) :**

المصادقة هي عملية تتحقق من أن شخصًا ما هو ما يقوله. تستخدم أنظمة التكنولوجيا عادةً شكلاً من أشكال المصادقة لتأمين الوصول إلى تطبيق أو بياناته ، وعادةً ما يتم ذلك من خلال التحقق من اسم المستخدم وكلمة المرور. على سبيل المثال ، عندما تحتاج إلى الوصول إلى موقع أو خدمة عبر الإنترنت ، فعادة ما يتعين عليك إدخال اسم المستخدم وكلمة المرور الخاصين بك. ثم تتم مقارنة اسم المستخدم وكلمة المرور اللذين أدخلتهما مع السجل الموجود في قاعدة البيانات الخاصة به. إذا كانت المعلومات التي أرسلتها متطابقة ، يفترض النظام أنك مستخدم صالح ويمنحك حق الوصول.

**15.3** **الترخيص (Authorization ) :**

الترخيص هو عملية الأمان التي تحدد مستوى وصول المستخدم أو الخدمة. في التكنولوجيا ، نستخدم الترخيص لمنح المستخدمين أو الخدمات إذنًا للوصول إلى بعض البيانات أو تنفيذ إجراء معين. على سبيل المثال ، العملاء الذين لديهم حق الوصول إلى الواجهة الأمامية للموقع الخاص بك ، والمسؤولين الذين لديهم حق الوصول إلى منطقة إدارة منفصلة. يحتاج كلا النوعين من المستخدمين إلى تسجيل الدخول ، ولكن مجرد المصادقة لا توضح شيئًا عما يُسمح لهم بالقيام به في نظامك. وبالتالي ، تحتاج أيضًا إلى التحقق من أذونات المستخدم المصادق عليه ، أي أنك تحتاج إلى تخويل المستخدم.

من الضروري ملاحظة الاختلاف هنا بين المصادقة والترخيص. تتحقق المصادقة من المستخدم قبل السماح له بالوصول ، ويحدد الترخيص ما يمكنه فعله بمجرد أن يمنحه النظام حق الوصول.

ولكي يستطيع spring تحديد الترخيص لمسجل الدخول لابد من إضافة التعليق التوضيحي @AthanticationPrincipel في بوابة الطلب

كمثال

@PostMapping()  
public ResponseEntity <ApiResponse> addTodos(@AuthenticationPrincipal MyUser myUser, @RequestBody Todo todo){  
 todoService.addTodo(myUser.getId(),todo);  
 return ResponseEntity.*status*(HttpStatus.*OK*).body(new ApiResponse("New Todo added !",201));  
}

**16.****4 طريقة إضافة Spring Security إلى المشروع :**

1.إضافة Spring Security Dependency

2.انشاء مجلد config

3.انشاء ملف SecurityConfig الذي يحتوي على جميع التكوينات

4.اضافة Configuration@ وهو التعليق التوضيحي الذي يشير إلى أن الفئة لديها أساليب تعريف Bean. لذلك يمكن أن تقوم حاوية Spring بمعالجة الفصل وإنشاء Spring Beans لاستخدامها في التطبيق. هذا التعليق التوضيحي جزء من إطار العمل الأساسي Spring

5.اضافة EnableWebSecurity@ وهو التعليق التوضيحي الذي يؤدي إلى تطبيق تكوين الأمان الافتراضي لـ Spring

6.إعادة تعريف دالة authenticationProvider

@Configuration  
@EnableWebSecurity  
@RequiredArgsConstructor  
public class SecurityConfig {  
  
 private final MyUserDetailsService myUserDetailsService;  
  
 @Bean  
 public DaoAuthenticationProvider authenticationProvider(){  
 DaoAuthenticationProvider authenticationProvider=new DaoAuthenticationProvider();  
 authenticationProvider.setUserDetailsService(myUserDetailsService);  
 authenticationProvider.setPasswordEncoder(new BCryptPasswordEncoder());  
 return authenticationProvider;  
 }

7. انشاء MyUserDetailsService الذي يقوم بتنفيذ واجهة UserDetailsService

8. تعريف دالة loadUserByUsername

@Service  
@RequiredArgsConstructor  
public class MyUserDetailsService implements UserDetailsService {  
  
  
  
 private final AuthRepository authRepository;  
 @Override  
 public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {  
 MyUser myUser=authRepository.findMyUserByUsername(username);  
 if(myUser==null){  
 throw new UsernameNotFoundException("Wrong username or password");  
 }  
 return myUser;  
 }

9.جعل MyUser تنفذ واجهة UserDetails وتعريف جميع الدوال الخاصة بالواجهة.

@Entity  
@Setter  
@Getter  
@AllArgsConstructor  
@NoArgsConstructor  
public class MyUser implements UserDetails {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Integer id;  
 private String username;  
 private String password;  
 private String role;  
  
 @Override  
 public Collection<? extends GrantedAuthority> getAuthorities() {  
 return Collections.*singleton*(new SimpleGrantedAuthority(this.role));  
 }  
  
 @Override  
 public boolean isAccountNonExpired() {  
 return true;  
 }  
  
 @Override  
 public boolean isAccountNonLocked() {  
 return true;  
 }  
  
 @Override  
 public boolean isCredentialsNonExpired() {  
 return true;  
 }  
  
 @Override  
 public boolean isEnabled() {  
 return true;  
 }  
}

10.إعادة تعريف دالة securityFilterChain(HttpSecurity http)داخل ملف SecurityConfig

@Configuration  
@EnableWebSecurity  
@RequiredArgsConstructor  
public class SecurityConfig {  
  
 private final MyUserDetailsService myUserDetailsService;  
  
 @Bean  
 public DaoAuthenticationProvider authenticationProvider(){  
 DaoAuthenticationProvider authenticationProvider=new DaoAuthenticationProvider();  
 authenticationProvider.setUserDetailsService(myUserDetailsService);  
 authenticationProvider.setPasswordEncoder(new BCryptPasswordEncoder());  
 return authenticationProvider;  
 }  
  
 @Bean  
 public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception{  
 http.csrf().disable()  
 .sessionManagement()  
 .sessionCreationPolicy(SessionCreationPolicy.*IF\_REQUIRED*)  
 .and()  
 .authenticationProvider(authenticationProvider())  
 .authorizeHttpRequests()  
 .requestMatchers(HttpMethod.*POST*,"/api/v1/auth/register").permitAll()  
 .requestMatchers("/api/v1/auth/admin").hasAuthority("ADMIN")  
 .anyRequest().authenticated()  
 .and()  
 .logout().logoutUrl("/api/v1/auth/logout")  
 .deleteCookies("JSESSIONID")  
 .invalidateHttpSession(true)  
 .and()  
 .httpBasic();  
 return http.build();  
 }  
  
  
}

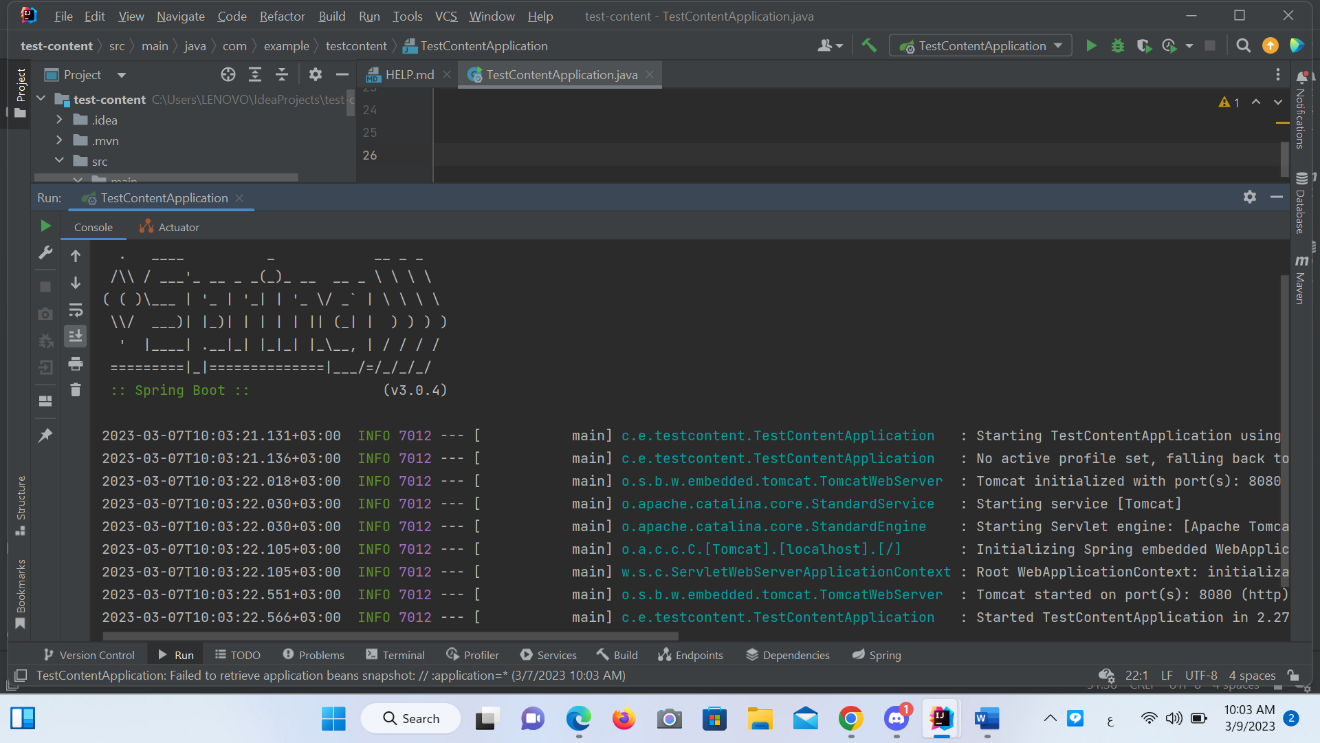
ـــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــ

**17. السجلات والتسجيل في Spring Boot(Logging and Logs)**

يستخدم Spring Boot تسجيل Apache Commons لجميع عمليات التسجيل الداخلية. توفر التكوينات الافتراضية لـ Spring Boot دعمًا لاستخدام Java Util Logging و Log4j2 و Logback. باستخدام هذه ، يمكننا تكوين تسجيل وحدة التحكم وكذلك تسجيل الملف.

**17.1** **تنسيق السجل ((Logs format**

**تنسيق السجلات الافتراضي يكون بالشكل التالي :**

****

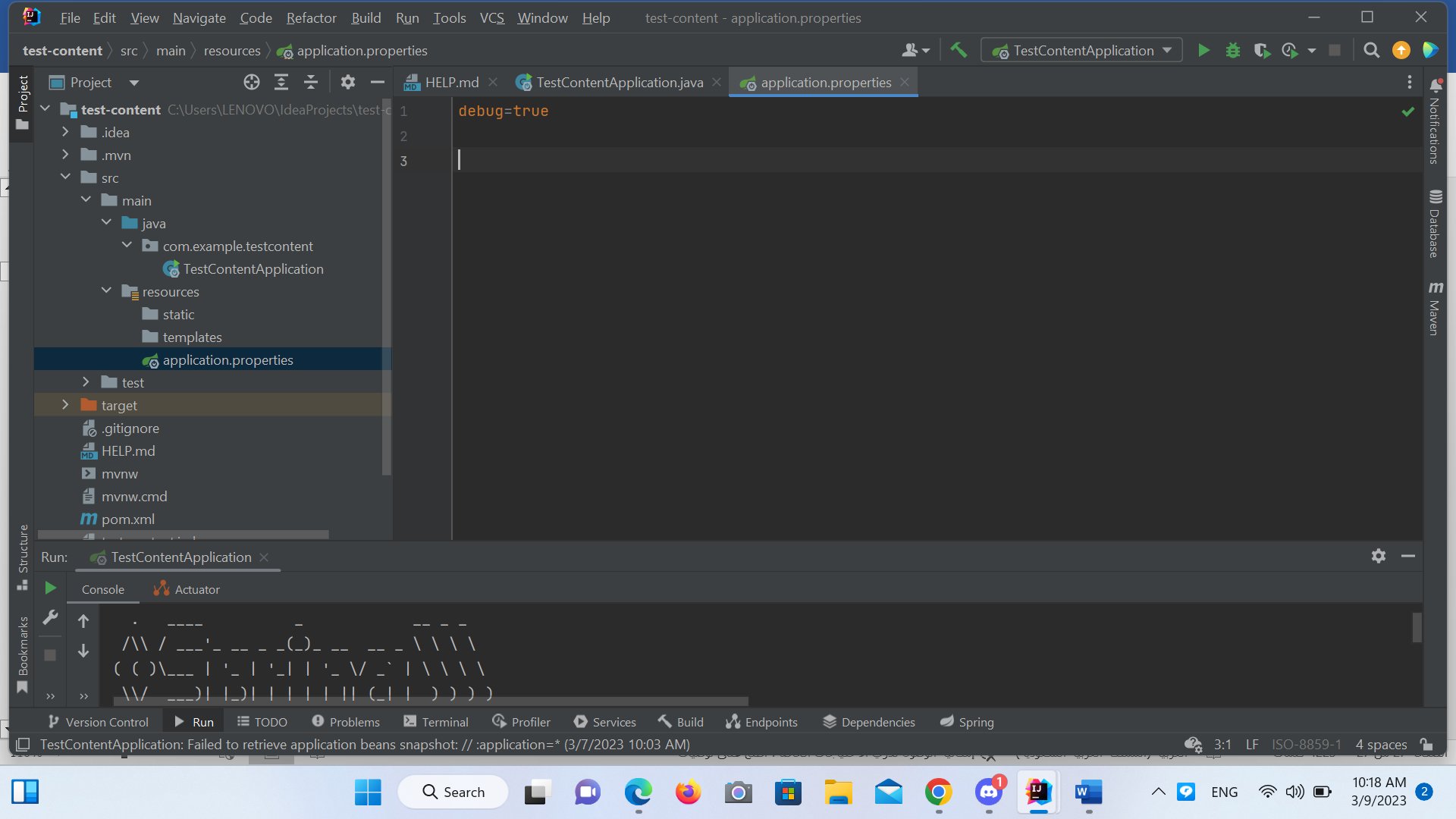
**ال**Time اللذان يوضحان تاريخ ووقت التسجيل .

والذي يحتوي على :

1. التاريخ والوقت اللذات يوضحان تاريخ ووقت السجل
2. مستوى السجل INFO, ERROR or WARN
3. معرف العملية Process ID
4. --- وهو مجرد فاصل
5. اسم الموضوع محاط بأقواس مربعة []
6. اسم المسجل الذي يظهر اسم فئة المصدر
7. رسالة السجل والتي تأتي بعد :

ستتم طباعة رسائل السجل الافتراضية في نافذة وحدة التحكم. بشكل افتراضي ، ستتم طباعة رسائل السجل "INFO" و "ERROR" و "WARN" في ملف السجل.

وإذا أردت السماح بطباعة رسائل السجل "Debug" أو "”Traceنستطيع تفعيله عن من داخل ملف aapplication.properties



وسنقوم الآن بإنشاء LoggingController حتى نرى طريقة استخدامنا لسجلات خاصة بنا ويمكننا بالتالي استخدامها بداخل مشروعنا بكل سهولة .

@RestController  
public class LoggingController {  
  
 Logger logger = LoggerFactory.*getLogger*(LoggingController.class);  
  
 @RequestMapping("/")  
 public String index() {  
 logger.trace("A TRACE Message");  
 logger.debug("A DEBUG Message");  
 logger.info("An INFO Message");  
 logger.warn("A WARN Message");  
 logger.error("An ERROR Message");  
  
 return "Howdy! Check out the Logs to see the output...";  
 }  
 }  
}

وبمجرد تشغيل خادم الويب سنستطيع رؤية هذه السجلات ورسائلها .

**ـــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــ**

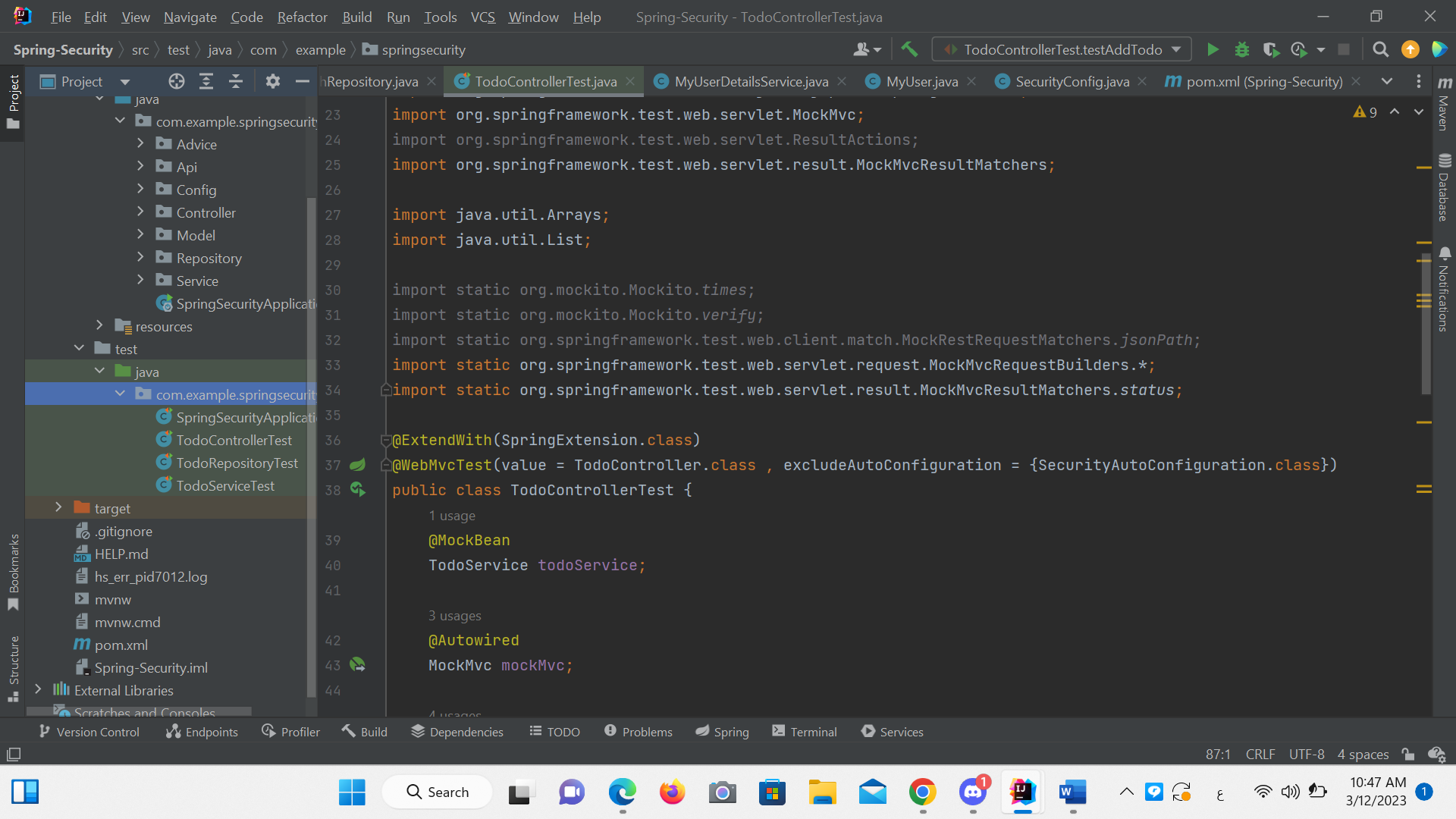
**18. اختبار الخادم باستخدام مكتبة Junit 5**

JUnit هي واحدة من أكثر أطر اختبار الوحدات شيوعًا في نظام Java البيئي. يحتوي إصدار JUnit 5 على عدد من الابتكارات المثيرة ، بهدف دعم الميزات الجديدة في Java 8 وما فوق ، بالإضافة إلى تمكين العديد من أنماط الاختبار المختلفة.

يعد إعداد JUnit 5.x.0 أمرًا بسيطًا جدًا ، نحتاج فقط إلى إضافة التبعية التالية إلى ملف pom.xml الخاص بنا:

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
 <exclusions>  
 <exclusion>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 </exclusion>  
 </exclusions>  
</dependency>  
  
<dependency>  
 <groupId>org.junit.jupiter</groupId>  
 <artifactId>junit-jupiter-api</artifactId>  
 <scope>test</scope>  
</dependency>  
  
<dependency>  
 <groupId>org.junit.jupiter</groupId>  
 <artifactId>junit-jupiter-engine</artifactId>  
 <scope>test</scope>  
</dependency>

ويمكننا اختبار كل وحدة في داخل كل طبقة في المشروع ، في البداية سنقوم بإنشاء test class لكل class نود اختباره في داخل مجلد test الموجود بشكل تلقائي داخل المشروع



**18.1 مثال لاختبار طبقة التحكم (Controller layer)**

سنقوم باختبار هذه الوحدة :

@GetMapping("/get-all")  
public ResponseEntity<List<Todo>> getAllTodos(){  
 return ResponseEntity.status(HttpStatus.OK).body(todoService.getTodos());  
}

في البداية سنقوم بإنشاء كلاس TodoControllerTest داخل مجلد test ثم سنقوم بكتابة التعليقات التوضيحية الخاصة بهذا الكلاس ليتم اختباره بشكل سليم مع وضع قيم يتم اختبارها وكأنها موجودة بداخل قاعدة البيانات لتساعد في اختبار الوحدات باستخدام التعليق التوضيحي @BeforeEachوكذلك استخدام تعليق @MockBean للتعامل مع طبقة الخادم المتصلة بطبقة التحكم دون الخوض في تفاصيلها

@ExtendWith(SpringExtension.class)  
@WebMvcTest(value = TodoController.class , excludeAutoConfiguration = {SecurityAutoConfiguration.class})  
public class TodoControllerTest {  
 @MockBean  
 TodoService todoService;  
  
 @Autowired  
 MockMvc mockMvc;  
  
 Todo todo1,todo2,todo3;  
 MyUser myUser;  
  
 ApiResponse apiResponse;  
  
 List<Todo> todos,todoList;  
  
 @BeforeEach  
 void setUp() {  
 myUser=new MyUser(1,"Maha" , "12345" , "ADMIN" , null);  
 todo1 = new Todo(1, "todo1", myUser );  
 todo2 = new Todo(2 , "todo2", myUser );  
 todo3 = new Todo(3 , "todo3", myUser );  
 todos= Arrays.*asList*(todo1);  
 todoList=Arrays.*asList*(todo2);  
  
  
  
 }

والآن سنقوم بإنشاء دالة تقوم باختبار الوحدة السابقة مع كتابة التعليق التوضيحي @Test لتفعيل خاصية الاختبار

@Test  
public void GetAllTodo() throws Exception {  
 Mockito.*when*(todoService.getTodos()).thenReturn(todos);  
 mockMvc.perform(*get*("/api/v1/todo/get-all"))  
 .andExpect(*status*().isOk())  
 .andExpect(MockMvcResultMatchers.*jsonPath*("$", Matchers.*hasSize*(1))) .andExpect(MockMvcResultMatchers.*jsonPath*("$[0].title").value("todo1"));  
}

**ثم سنقوم بتشغيل زر التشغيل Run الذي بجانب الدالة لنتأكد أن الاختبار يعمل وأن حالته success**

**18.2 مثال لاختبار طبقة الخدمة (Service layer)**

سنقوم باختبار هذه الوحدة :

@PostMapping()  
public ResponseEntity <ApiResponse> addTodos(@AuthenticationPrincipal MyUser myUser, @RequestBody Todo todo){  
 todoService.addTodo(myUser.getId(),todo);  
 return ResponseEntity.*status*(HttpStatus.*OK*).body(new ApiResponse("New Todo added !",201));  
}

في البداية سنقوم بإنشاء كلاس TodoServiceTest داخل مجلد test ثم سنقوم بكتابة التعليقات التوضيحية الخاصة بهذا الكلاس ليتم اختباره بشكل سليم مع وضع قيم يتم اختبارها وكأنها موجودة بداخل قاعدة البيانات لتساعد في اختبار الوحدات باستخدام التعليق التوضيحي @BeforeEachوكذلك استخدام تعليق @Mock للتعامل مع طبقة المستودع المتصلة بطبقة الخدمة دون الخوض في تفاصيلها

@ExtendWith(MockitoExtension.class)  
public class TodoServiceTest {  
  
 @InjectMocks  
 TodoService todoService;  
 @Mock  
 TodoRepository todoRepository;  
 @Mock  
 AuthRepository authRepository;  
  
 MyUser user;  
  
 Todo todo1,todo2,todo3;  
  
 List<Todo> todos;  
  
 @BeforeEach  
 void setUp() {  
 user=new MyUser(null,"majd","123","Admin", null);  
 todo1=new Todo(null,"todo1",user);  
 todo2=new Todo(null,"todo2",user);  
 todo3=new Todo(null,"todo3",null);  
  
 todos=new ArrayList<>();  
 todos.add(todo1);  
 todos.add(todo2);  
 todos.add(todo3);  
 }

والآن سنقوم بإنشاء دالة تقوم باختبار الوحدة السابقة مع كتابة التعليق التوضيحي @Test لتفعيل خاصية الاختبار

@Test  
public void AddTodoTest(){  
  
 *when*(authRepository.findMyUserById(user.getId())).thenReturn(user);  
  
 todoService.addTodo(user.getId(),todo3);  
 *verify*(authRepository,*times*(1)).findMyUserById(user.getId());  
 *verify*(todoRepository,*times*(1)).save(todo3);  
}

**18.3 مثال لاختبار وحدة في طبقة المستودع ((repository layer**

سنقوم باختبار هذه الوحدتين

public interface TodoRepository extends JpaRepository<Todo,Integer> {  
 Todo findTodoById(Integer id);  
  
 List<Todo> findAllByMyUser(MyUser myUser);  
}

في البداية سنقوم بإنشاء كلاس TodoRepositoryTest داخل مجلد test ثم سنقوم بكتابة التعليقات التوضيحية الخاصة بهذا الكلاس ليتم اختباره بشكل سليم مع وضع قيم يتم اختبارها وكأنها موجودة بداخل قاعدة البيانات لتساعد في اختبار الوحدات باستخدام التعليق التوضيحي @BeforeEach

@ExtendWith(SpringExtension.class)  
@DataJpaTest  
@AutoConfigureTestDatabase(replace =AutoConfigureTestDatabase.Replace.*NONE* )  
public class TodoRepositoryTest {  
  
 @Autowired  
 TodoRepository todoRepository;  
  
 MyUser user;  
  
 Todo todo1,todo2,todo3;  
  
 List<Todo> todos;  
  
 @BeforeEach  
 void setUp() {  
 user=new MyUser(null,"majd","123","Admin", null);  
 todo1=new Todo(null,"todo1",user);  
 todo2=new Todo(null,"todo2",user);  
 todo3=new Todo(null,"todo3",user);  
  
 }

والآن سنقوم بإنشاء دالتين تقوم باختبار الوحدات السابقة مع كتابة التعليق التوضيحي @Test لتفعيل خاصية الاختبار

@Test  
public void findTodoByIdTest(){  
  
 todoRepository.save(todo1);  
 Todo todo=todoRepository.findTodoById(todo1.getId());  
 Assertions.*assertThat*(todo).isEqualTo(todo1);  
  
}  
@Test  
public void findAllByMyUserTest(){  
  
 todoRepository.save(todo1);  
 todoRepository.save(todo2);  
 todoRepository.save(todo3);  
  
 List<Todo> todoList=todoRepository.findAllByMyUser(user);  
  
 Assertions.*assertThat*(todoList.get(0).getMyUser().getId()).isEqualTo(user.getId());  
  
}

**19. نشر المشروع (Deployment )**

بعد أن تعرفنا على كيفية بناء خادم باستخدام Spring Boot بشكل كامل ستكون خطوتنا الأخيرة هو نشر هذا الخادم على أحد مزودين الخدمة ومن أشهر الأمثلة Heroku والذي سنتعرف كيف يمكننا استخدامه من خلال الخطوات التالية :

**19.1 أولاً :تجهيز المشروع على Heroku**

* إنشاء حساب على <https://www.heroku.com/>
* تثبيت git عن طريق الرابط <https://git-scm.com/downloads>
* تثبيت node.js عن طريق الرابط <https://nodejs.org/en/>
* تسجيل الدخول عن طريق الأمر npx heroku login

**19.2 ثانياً : تجهيز المشروع باستخدام Git**

استخدم الأوامر التالية بداخل ملفات مشروعك

git init

git add .

git commit -m "first commit"

**19.3 ثالثاً :نشر التطبيق باستخدام Heroku**

انشئ تطبيق Heroku

npx heroku create <your\_unique\_app\_name>

الامر السابق ينشئ تطبيق heroku و يجعلك تختار اسم المشروع

انشر التطبيق باستخدام

git push heroku master

**وهنا بعض المصادر الإضافية**

* <https://devcenter.heroku.com/articles/deploying-spring-boot-apps-to-heroku>
* <https://www.youtube.com/watch?v=KDK5xXPJVIg>

**انتهى ،،**