# Indian Udemy Spring Boot Lesson

## Question 2: Spring Framework - Important Terminology

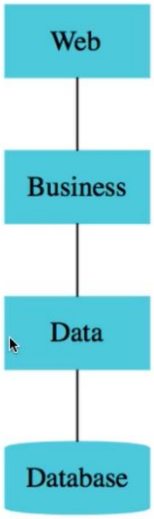
In28 Minutes

- **@Component** (..): Class managed by Spring framework
- **Dependency**: GameRunner needs GamingConsole impl!
  - GamingConsole Impl (Ex: MarioGame) is a dependency of GameRunner
- **Component Scan**: How does Spring Framework find component classes?
  - It scans packages! (`@ComponentScan("com.in28minutes")`)
- **Dependency Injection**: Identify beans, their dependencies and wire them together (provides **IOC** - Inversion of Control)
  - **Spring Beans**: An object managed by Spring Framework
  - **IoC container**: Manages the lifecycle of beans and dependencies
    - **Types:** ApplicationContext (complex), BeanFactory (simpler features - rarely used)
  - **Autowiring**: Process of wiring in dependencies for a Spring Bean


## Question 3: Does the Spring Framework really add value?

In28 Minutes

- In **Game Runner** Hello World App, we have very few classes
- BUT Real World applications **are much more complex**:
  - Multiple Layers (Web, Business, Data etc)
  - Each layer is **dependent** on the layer below it!
    - Example: Business Layer class talks to a Data Layer class
      - Data Layer class is a **dependency** of Business Layer class
    - There are thousands of such dependencies in every application!
- With Spring Framework:
  - INSTEAD of FOCUSING on objects, their dependencies and wiring
    - You can focus on the business logic of your application!
  - **Spring Framework manages the lifecycle** of objects:
    - Mark components using annotations: `@Component` (and others..)
    - Mark dependencies using `@Autowired`
    - Allow Spring Framework to do its magic!
- Ex: Controller > BusinessService (sum) > DataService (data)!



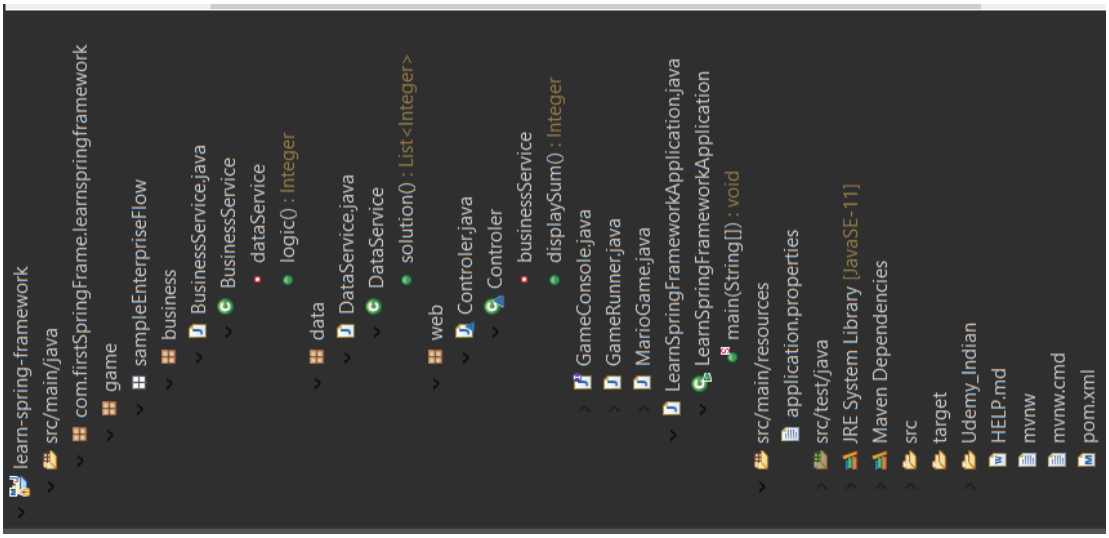
```java
/*****************************************************************************************
 NOTE: You can start using a same class with class Class and later refactor it
 using Ctrl + 1 and using move the current class to its own package and updating the package uisng
 class name follow this to expand even batter. Don't forget to organize the imports using
 Ctrl + Shift + o

 *****************************************************************************************/
package com.firstSpringFrame.learnspringframework.game.sampleEnterpriseFlow.data;

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

import org.springframework.stereotype.Component;

// Getting data from database
@Component
public class DataService {
    public List<Integer> solution() {
        return Arrays.asList(12, 15, 99, 45, 32, 18, 321);
    }
}
/*
Go to view menu
Package Presentation
Hierarchical
*/
```

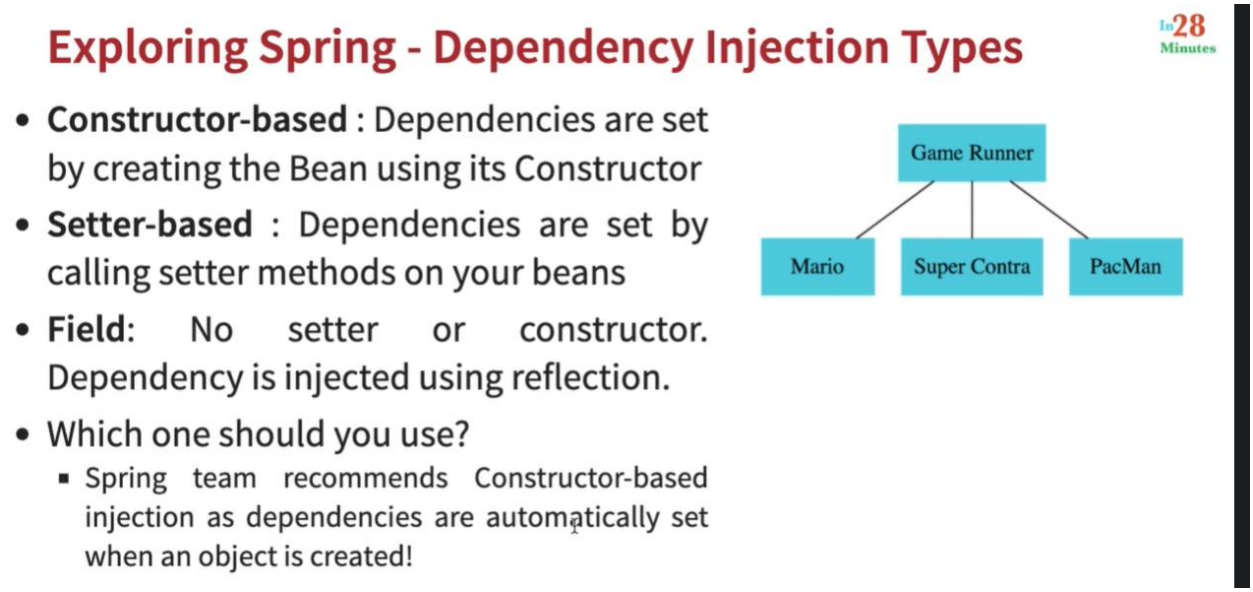
Controler.java ✕ BusinessService.java *DataService.java

```java
package com.firstSpringFrame.learnspringframework.game.sampleEnterpriseFlow.web;

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

// Send the response in the expected format

/*What @RestController will do?
Spring 4.0 introduced the @RestController annotation in order to simplify
the creation of RESTful web services. It's a convenient annotation that combines
@Controller and @ResponseBody, which eliminates the need to annotate every request
handling method of the controller class with the @ResponseBody annotation */

// Organize imports: (Ctrl + Shift, 2)

@RestController // Refer above note
class Controler {
    // "/sum" ==> 100; When the web application page has a with /sum, return with
    // response 100 in there

    @Autowired
    private BusinessService businessService;

    @GetMapping("/sum")
    public Integer displaySum() {
        return businessService.logic();
        // return businessService.buildLogic(a, b);
    }
}
```

```java
package com.firstSpringFrame.learnspringframework.game.sampleEnterpriseFlow.business;

import java.util.List;

// Business logic
@Component
public class BusinessService {
    @Autowired
    private DataService dataService;

    public Integer logic() {
        List<Integer> list = dataService.solution();
        Integer reduce = list.stream().reduce(Integer::sum).get();// .get() for list
        return reduce;
    }
}
```

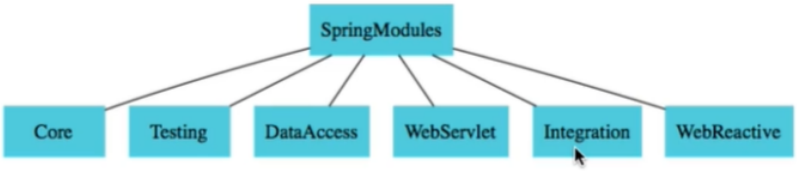
## Dependency Injection




**Spring Recommands doing dependency injection via the constructor(I think fields)**

## Spring Modules




- Spring Framework is divided into **modules**:
  - **Core**: IoC Container etc
  - **Testing**: Mock Objects, Spring MVC Test etc
  - **Data Access**: Transactions, JDBC, JPA etc
  - **Web Servlet**: Spring MVC etc
  - **Web Reactive**: Spring WebFlux etc
  - **Integration**: JMS etc
- Each application can choose the modules they want to make use of
  - They do not need to make use of all things everything in Spring framework!

What is Spring MVC?

Model-View-Controler

What is Spring MVC used for?

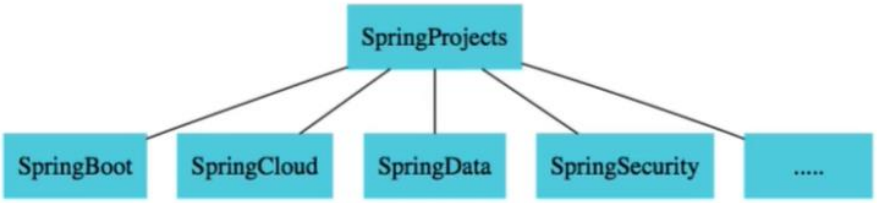
What Is Spring MVC? Spring MVC is **a library within the Spring framework that simplifies handling HTTP requests and responses**. It's built on the Servlet API and is an essential component of the Spring Framework. Jan 11, 2021

**Spring Boot** : Makes it easy to get started with Spring Based Applications (Including.. Microservices)

Traditional application development happened in more sophisticated framework called monolithic where as modern framework uses smaller application called microservices.

# Spring Projects

In 2 Min




- Spring Projects: Spring keeps evolving (REST API > Microservices > Cloud)
  - **Spring Boot**: Most popular framework to build microservices
  - **Spring Cloud**: Build cloud native applications
  - **Spring Data**: Integrate the same way with different types of databases : NoSQL and Relational
  - **Spring Integration**: Address challenges with integration with other applications
  - **Spring Security**: Secure your web application or REST API or microservice

# Spring Framework - Review

In 28 Minutes

- **Goal**: 10,000 Feet overview of Spring Framework
  - Help you understand the terminology!
    - Dependency
    - Dependency Injection (and types)
      - Autowiring
      - Spring Beans
      - Component Scan
    - IOC Container (Application Context)
  - We will play with other Spring Modules and Projects later in the course

## Getting Started with Spring Boot - Goals

In28 Minutes

- Build a Hello World App in **Modern Spring Boot** Approach
- Get **Hands-on** with Spring Boot
  - Why Spring Boot?
  - **Terminology**
    - Spring Initializr
    - Auto Configuration
    - Starter Projects
    - Actuator
    - Developer Tools

## Hands-on: Understand Power of Spring Boot

```
// http://localhost:8080/courses
[
  {
    "id": 1,
    "name": "Learn Microservices",
    "author": "in28minutes"
  }
]
```

If the user looks for ths page localhost:8080/course, we will respond it by using the JSON file as above

To DEBUG the entire process:

```
#logging.level.org.springframework=DEBUG
```

Into file application.properties

Enable all actuator endpoints:

A lot more urls will be created

```
management.endpoints.web.exposure.include=*
```

You can click on actuator/beans to view all bean that were created.

You can view on matrces to see how many requests came to the url so far.

localhost:8080/actuator/metrics/http.server.requests


```
{
    name: "http.server.requests",
    description: null,
    baseUnit: "seconds",
  - measurements: [
      - {
            statistic: "COUNT",
            value: 9,
        },
      - {
            statistic: "TOTAL_TIME",
            value: 0.73543626,
        },
      - {
            statistic: "MAX",
            value: 0.338749812,
        },
    ],
  - availableTags: [
      - {
            tag: "exception",
          - values: [
                "None"
```


So far we have been stopping and changing code and restarting by killing running instances.

Why not make it auto running while changing the code?

Just add this dependency:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
</dependency>
```

## World Before Spring Boot!

In28 Minutes

*https://github.com/in28minutes/SpringMvcStepByStep/blob/master/Step15.md#pomxml*

- Setting up Spring Web Projects **before Spring Boot was NOT easy**!
  - **Define maven dependencies** and manage versions for frameworks
    - spring-webmvc, jackson-databind, log4j etc
  - Define **web.xml** (`/src/main/webapp/WEB-INF/web.xml`)
    - Define Front Controller for Spring Framework (`DispatcherServlet`)
  - Define a **Spring context XML** file (`/src/main/webapp/WEB-INF/todo-servlet.xml`)
    - Define a Component Scan (`<context:component-scan base-package="com.in28minutes" />`)
  - **Install Tomcat** or use tomcat7-maven-plugin plugin (or any other web server)
  - Deploy and Run the application in Tomcat
- How does Spring Boot do its **Magic**?
  - Spring Boot Starter Projects
  - Spring Boot Auto Configuration

Did not understand 345 (udemy.com)

## More Spring Boot Features

In28 Minutes

- **Spring Boot Actuator**: Monitor and manage your application in your production
  - Provides a number of endpoints:
    - **beans** - Complete list of Spring beans in your app
    - **health** - Application health information
    - **metrics** - Application metrics
    - **mappings** - Details around Request Mappings

Add

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

To your pom.xml file and rerun the file and go to web and check this localhost:8080/actuator


```
{
  - _links: {
      - self: {
            href: "http://localhost:8080/actuator",
            templated: false,
        },
      - health: {
            href: "http://localhost:8080/actuator/health",
            templated: false,
        },
      - health-path: {
            href: "http://localhost:8080/actuator/health/{*path}",
            templated: true,
        },
      - info: {
            href: "http://localhost:8080/actuator/info",
            templated: false,
        },
    }
}
```

# Spring Boot Auto Configuration

- Spring Boot provides **Auto Configuration**
  - **Basic configuration** to run your application using the frameworks defined in your maven dependencies
  - Auto Configuration is **decided based on**:
    - Which frameworks are in the Class Path?
    - What is the existing configuration (Annotations etc)?
  - **An Example**: (Enable debug logging for more details):
    - If you use Spring Boot Starter Web, following are auto configured:
      - Dispatcher Servlet (`DispatcherServletAutoConfiguration`)
      - Embedded Servlet Container - Tomcat is the default (`EmbeddedWebServerFactoryCustomizerAutoConfiguration`)
      - Default Error Pages (`ErrorMvcAutoConfiguration`)
      - Bean to/from JSON conversion (`JacksonHttpMessageConvertersConfiguration`)

# Spring Boot Auto Configuration

- Spring Boot provides **Auto Configuration**
  - **Basic configuration** to run your application using the frameworks defined in your maven dependencies
  - Auto Configuration is **decided based on**:
    - Which frameworks are in the Class Path?
    - What is the existing configuration (Annotations etc)?
  - **An Example**: (Enable debug logging for more details):
    - If you use Spring Boot Starter Web, following are auto configured:
      - Dispatcher Servlet (`DispatcherServletAutoConfiguration`)
      - Embedded Servlet Container - Tomcat is the default (`EmbeddedWebServerFactoryCustomizerAutoConfiguration`)
      - Default Error Pages (`ErrorMvcAutoConfiguration`)

Final Configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
```

```
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.5.4</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.SpringBoot</groupId>
    <artifactId>LearnSpringBoot</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>LearnSpringBoot</name>
    <description>Demo project for Spring Boot</description>
    <properties>
        <java.version>11</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <scope>runtime</scope>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
            <scope>test</scope>
        </dependency>

    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

## Spring Boot vs Spring MVC vs Spring

In 28 Minutes

- **Spring Framework** Core Feature: Dependency Injection
  - @Component, @Autowired, IOC Container, ApplicationContext, Component Scan etc..
  - **Spring Modules and Spring Projects**: Good Integration with Other Frameworks (Hibernate/JPA, JUnit & Mockito for Unit Testing)
- **Spring MVC** (Spring Module): Build web applications in a decoupled approach
  - Dispatcher Servlet, ModelAndView and View Resolver etc
- **Spring Boot** (Spring Project): Build production ready applications quickly
  - **Starter Projects** - Make it easy to build variety of applications
  - **Auto configuration** - Eliminate configuration to setup Spring, Spring MVC and other projects!
  - Enable production ready non functional features:
    - Actuator : Enables Advanced Monitoring and Tracing of applications.
    - Embedded Servers - No need for separate application servers!
    - Default Error Handling

## Spring Boot - Review

In 28 Minutes

- **Goal**: 10,000 Feet overview of Spring Boot
  - Help you understand the terminology!
    - Starter Projects
    - Auto Configuration
    - Actuator
    - DevTools
- **Advantages**: Get started quickly with production ready features!

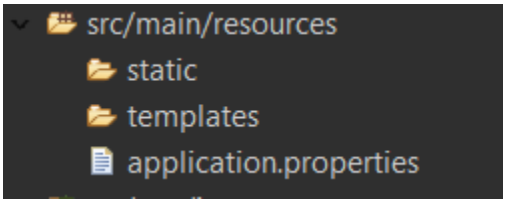
Simple REST API with Spring Boot and Spring Data JPA

- We did little configuration to communicate with the internal Database using memory of local CPU

Inside the pom.xml

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


<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
        <scope>runtime</scope>
</dependency>
```

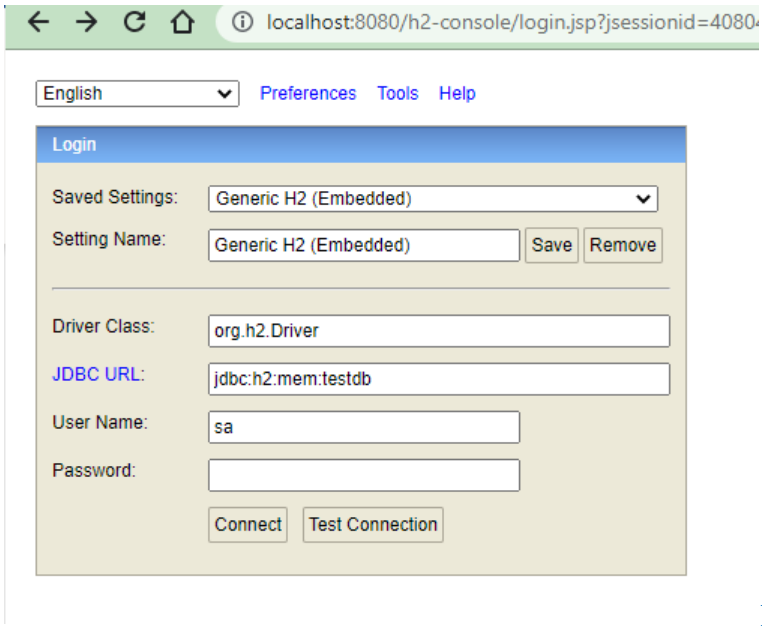
- We also did little configuration inside the properties under




```
6 spring.datasource.url=jdbc:h2:mem:testdb
```

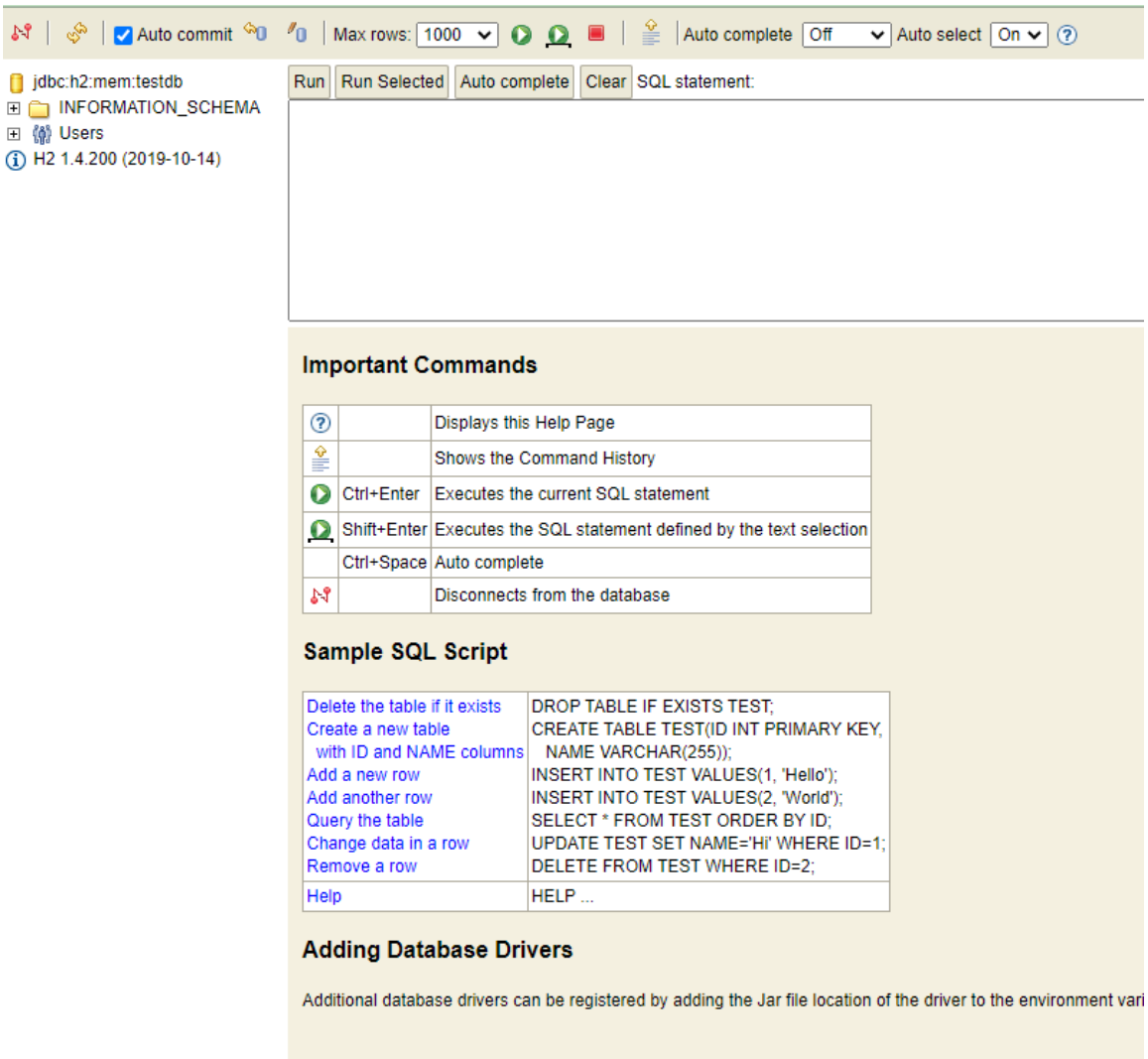
We save and run the file

- We then went to brouser and typed: local:host8080/he-console




Received this screen

- Under JDBC URL we copied and pasted the: `jdbc:h2:mem:testdb`
- Hit connect
- Received this internal database connected via local memory




This data is temperory – mostly used to learn

JDBC – Java BataBase Connectivity

# JDBC to Spring JDBC

## JDBC example

```
public void deleteTodo(int id) {
    PreparedStatement st = null;
    try {
        st = db.conn.prepareStatement(DELETE_TODO_QUERY);
        st.setInt(1, id);
        st.execute();
    } catch (SQLException e) {
        logger.fatal("Query Failed : " + DELETE_TODO_QUERY, e);
    } finally {
        if (st != null) {
            try {st.close();}
            catch (SQLException e) {}
        }
    }
}
```

## Spring JDBC example

```
public void deleteTodo(int id) {
    jdbcTemplate.update(DELETE_TODO_QUERY, id);
}
```

```
String DELETE_TODO_QUERY = "DELETE FROM TODO WHERE ID=?";
```

Using jdbc, writing code to work with database become very easier but still, you need to write a query

Within a few years, we got

JPA = Java Persistence API – When you're using JPA over Hibernate, you don't need to write queries; no need to write queries manually. What you do is you'd map Entities(objects) to tables

## JDBC to Spring JDBC to JPA to Spring Data JPA

In 28 Minutes

- **JDBC**
  - Write a lot of SQL queries!
  - And write a lot of Java code
- **Spring JDBC**
  - Write a lot of SQL queries
  - BUT lesser Java code
- **JPA**
  - Do NOT worry about queries
  - Just Map Entities to Tables!



  - Write a lot of SQL queries!
  - And write a lot of Java code
- **Spring JDBC**
  - Write a lot of SQL queries
  - BUT lesser Java code
- **JPA**
  - Do NOT worry about queries
  - Just Map Entities to Tables!

- **Spring Data JPA**
  - Let's make JPA even more simple!
  - I will take care of everything!

# JPA Example

```
@Repository
@Transactional
public class PersonJpaRepository {

  @PersistenceContext
  EntityManager entityManager;

  public Person findById(int id) {
    return entityManager.find(Person.class, id);
  }

  public Person update(Person person) {
    return entityManager.merge(person);
  }

  public Person insert(Person person) {
    return entityManager.merge(person);
  }

  public void deleteById(int id) {........
```

# Spring Data JPA Example

```
public interface TodoRepository extends JpaRepository<Todo, Integer>{
```

To manually insert data into the local computer's database using Spring

We did the following:

1. We created an empty file using src/main/resource folder with an untitled name.
2. We then did save as under the same file directory and save it as data.sql. When we save a file as .sql, it will automatically pick and refer to the existing database.
3. Then we added the values into the table as like:

```
insert into MY_TABLE(ID, AUTHOR, NAME)  values (10001, 'Tek Acharya', 'Learn Microservices')
insert into MY_TABLE(ID, AUTHOR, NAME)  values (10002, 'Heema Acharya', 'Learn Cooking')
insert into MY_TABLE(ID, AUTHOR, NAME)  values (10003, 'Prinsa Acharya', 'Learn Studying')
insert into MY_TABLE(ID, AUTHOR, NAME)  values (10004, 'Prajol Acharya', 'Learn Playing')
```

By now, we are able to populate some values into our table for us to continue learing about retrieving, updating and deleting or so called CRUD operation.

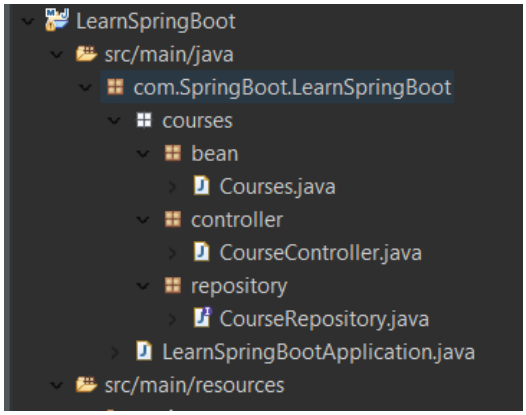
SELECT * FROM MY_TABLE;

| ID | AUTHOR | NAME |
|---|---|---|
| 10001 | Tek Acharya | Learn Microservices |
| 10002 | Heema Acharya | Learn Cooking |
| 10003 | Prinsa Acharya | Learn Studying |
| 10004 | Prajol Acharya | Learn Playing |

(4 rows, 9 ms)

Here, we will use Spring Data JPA called repository

We, then created another class called CourseRepository repository directory under com.SpringBoot.LearnSpringBoot.




Inside the repository folder, we created a CourseRepository interface with the following code to call the Java's JpaRepository

```
10 public interface CourseRepository extends JpaRepository<Courses, Long> {
11
12 }
```

After this we created a field of CourseRepository at CourseController class and autowired it for it dependencies

```
18     @Autowired
19     // private JpaRepository<Courses, Long> repository;
20     private CourseRepository repository;
```

Then I use this repository variable to retrieve data of Course table of my database.

One important thing here is that we must have a default constructor at entity class; Course

Once we created that we saved and use the following code to retrieve info from he table as:

```java
    @GetMapping("/courses")

    public List<Courses> getAllCourses() {

        // Now instead of this we will retrieve out data from repository
//      return Arrays.asList(new Courses(1, "Learn MicroServices", "Tek Acharya"),
//              new Courses(2, "Learn Spring Boot", "Prajol Acharya"));

        return repository.findAll();
    }
```

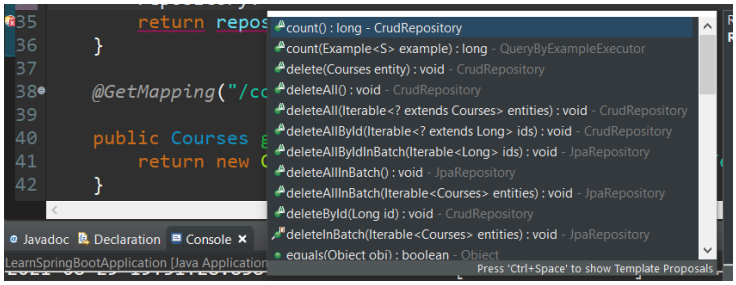
Notice, how we are able to replace hard coded values from the original code. This time we displayed the values from the database table to the web.

localhost:8080/courses


```
[
  {
    "id": 10001,
    "name": "Learn Microservices",
    "author": "Tek Acharya"
  },
  {
    "id": 10002,
    "name": "Learn Cooking",
    "author": "Heema Acharya"
  },
  {
    "id": 10003,
    "name": "Learn Studying",
    "author": "Prinsa Acharya"
  },
  {
    "id": 10004,
    "name": "Learn Playing",
    "author": "Prajol Acharya"
  }
]
```


AWESOME;

But we hve used some configurations, though. This is very important.




You can use a lot other functionalities using this repository now. Yay!!

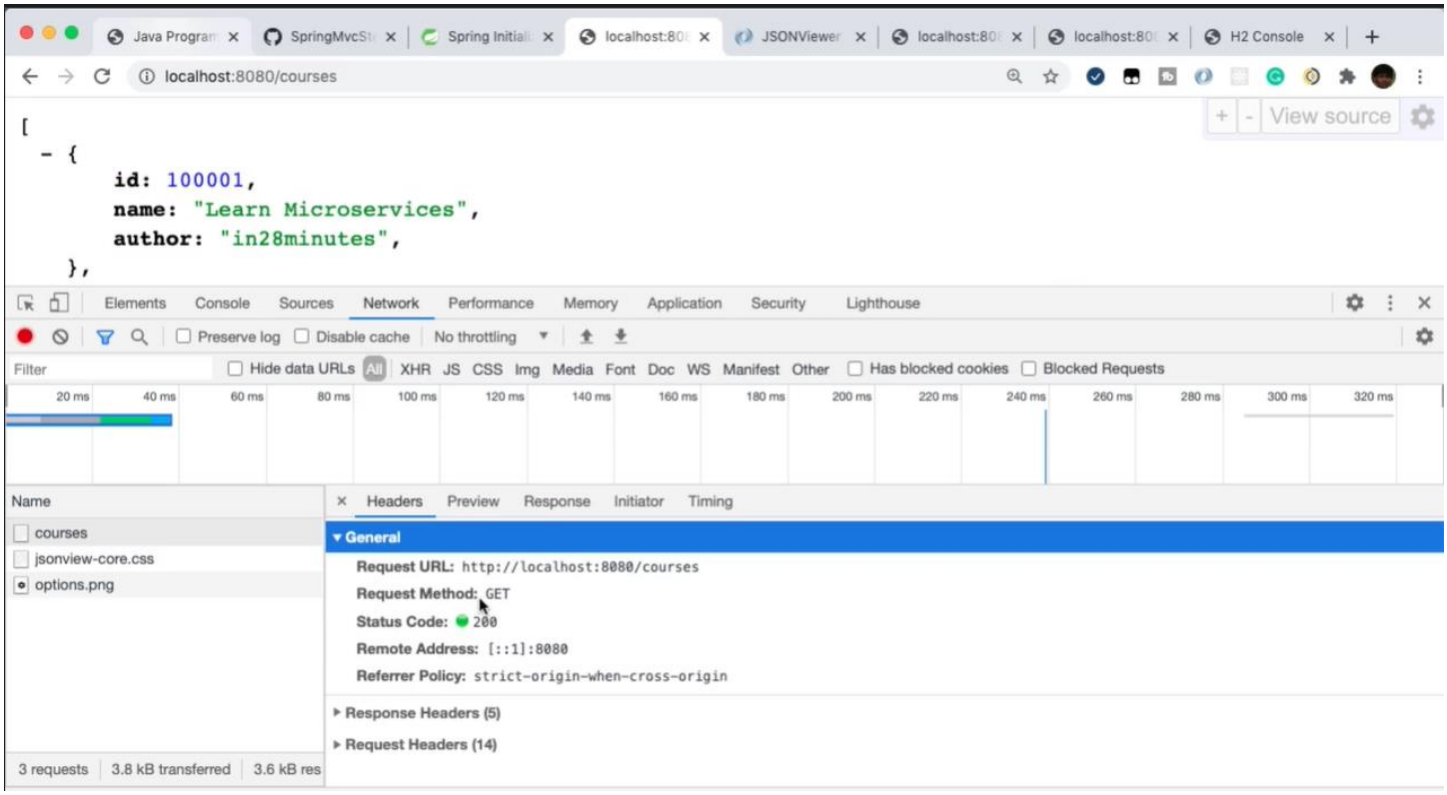
# REST API

Representational State Transfer Application Programme Interface

Is a set of constraints.

URI necessary for RESTful API. URI stands for Uniform Resourse Indentifier (Courses, Courses/1, etc.)

Whenever we are making a request using URI in the web, it uses a HTTP protocal. This typically is is GET identifier.




## REST API

In28 Minutes

- **REST API**: Architectural Style for the Web
  - **Resource**: Any information (Example: Courses)
  - **URI**: How do you identify a resource? (/courses, /courses/1)
  - You can perform actions on a resource (Create/Get/Delete/Update). Different HTTP Request Methods are used for different operations:
    - **GET** - Retrieve information (/courses, /courses/1)
    - **POST** - Create a new resource (/courses)
    - **PUT** - Update/Replace a resource (/courses/1)
    - **PATCH** - Update a part of the resource (/courses/1)
    - **DELETE** - Delete a resource (/courses/1)
  - **Representation**: How is the resource represented? (XML/JSON/Text/Video etc..)
  - **Server**: Provides the service (or API)
  - **Consumer**: Uses the service (Browser or a Front End Application)

If you take a closer look at how we are retrieving the data using URI is by hard coding as

/Courses/1 for the first data.

Instead of this, we can use {id}  for 1 to make it dynamic as ➔ /Courses/{id} and map it to id using @PathVariable annotation.

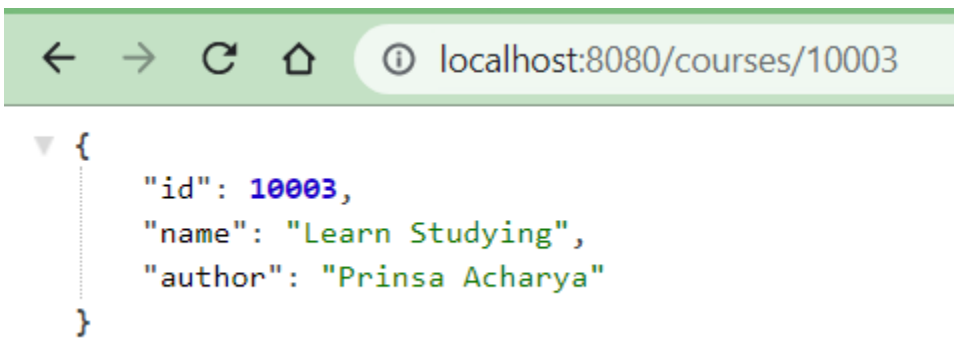
```java
    @GetMapping("/courses/{id}")

    public Courses getOneCourse(@PathVariable int id) {
        return new Courses(1, "Learn MicroServices 5020", "Tek Acharya");
    }
```

And now, we are ready to return the soft code for this instead of har coded value as we can see above.

This is how we can retrieve the individual courses from the database to the web

```java
    @GetMapping("/courses/{id}")

    public Courses getOneCourse(@PathVariable long id) {
        Optional<Courses> thisCourse = repository.findById(id);
        if (thisCourse.isEmpty()) {
            throw new RuntimeException("This course does not exists!, try it again");
        }
        return thisCourse.get();
    }
```

Here, if we provide an id that does not exists, the user will get a runtime exception as indicated by the programmer.




Notice, the id entered in the RUL

Now we will go over creating a new entry into our data into the database table

```java
// In this case we need to accept a JSON file
// Since we need to fetch the message with an API call we need the annotation of
// a message body as
// @RequestBody and map it to the Courses variable course.
// The way you can save that course entity is to use reposite.save(course);

// It seems that creating post is easy but executing it is difficult as we will
// explore below.

@PostMapping("/courses")
public void createCourse(@RequestBody Courses course) {
    repository.save(course);
}
```

Many descriptions are outlined above.

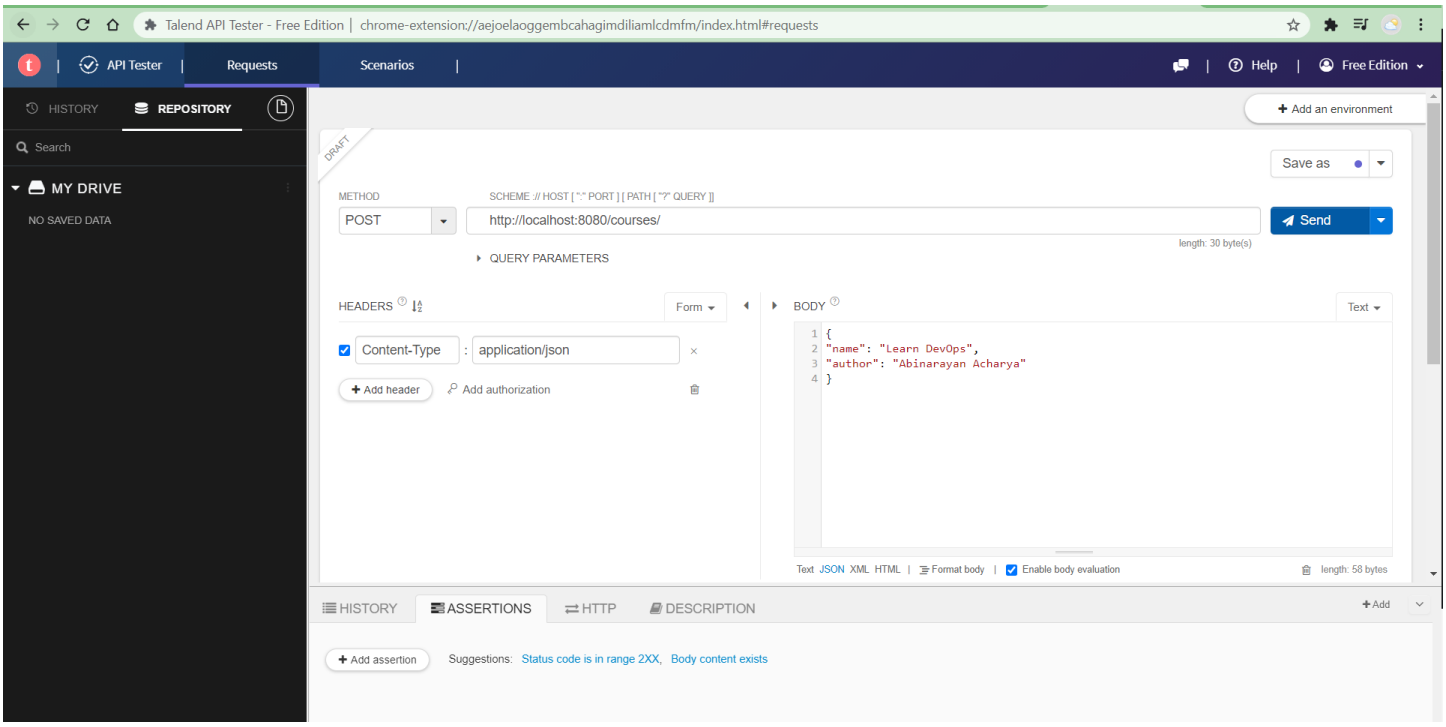
Few highlights done here are:

1. @PostMapping
2. @RequestBody
3. @RequestBody Courses course
4. Repository.save(course) to save the entry into the db table

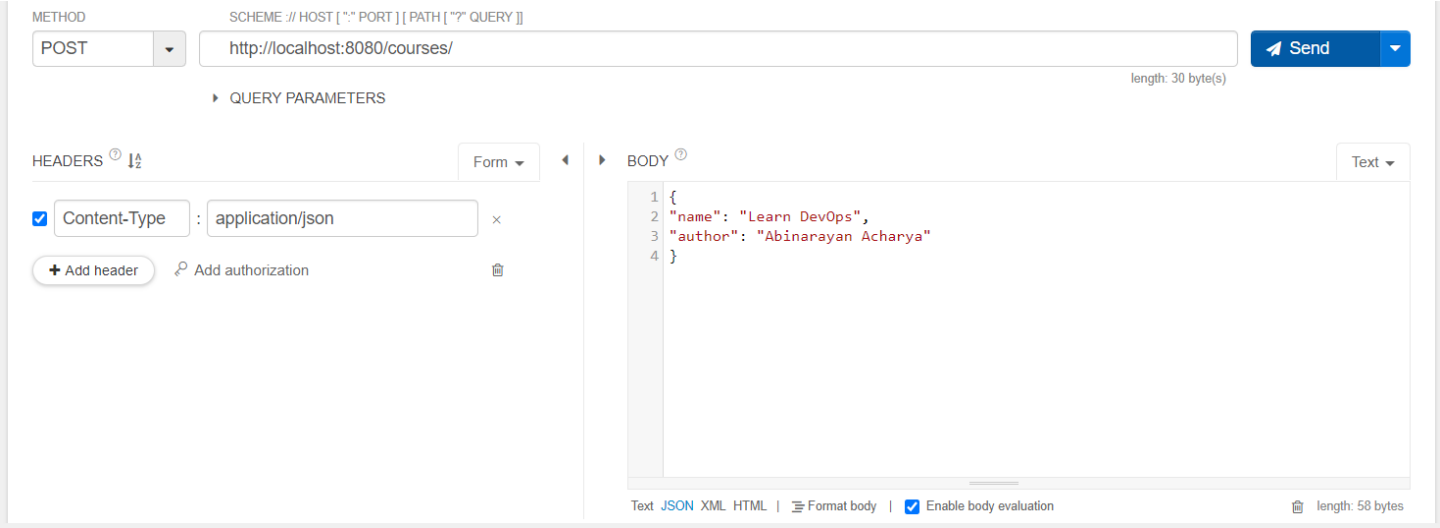
Few more things done here.

Writing a code for creating a new entry seems quite easy but to inserting the new content into our db from the web, we need to do something picular.

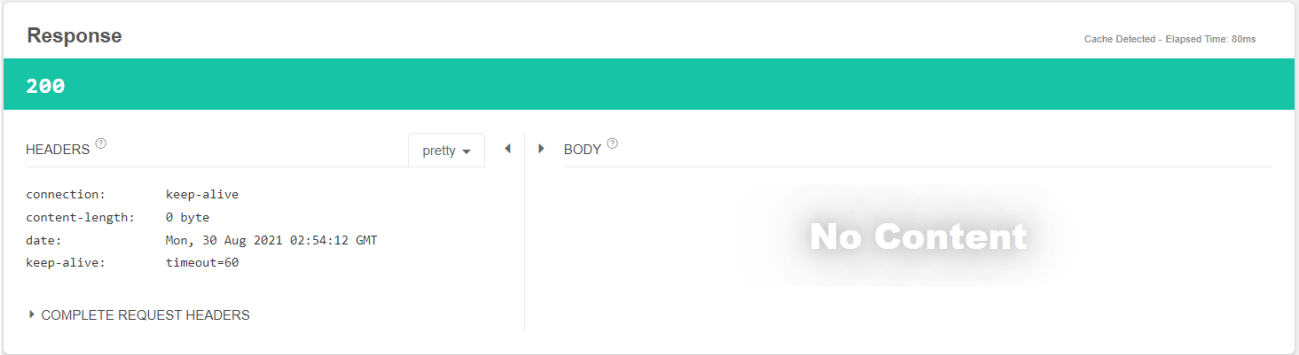
1. Downloaded a Google plugin called Talend and write data for the entry in there as

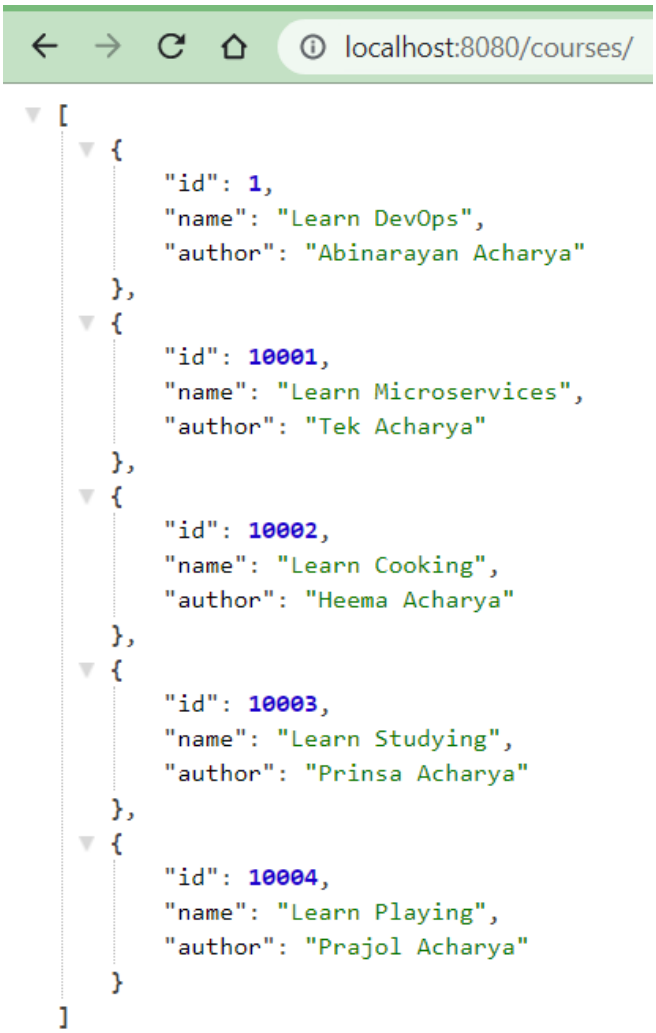



Few important things to do here is to toggle the method to POST and give the required formated body to be entered into the database.




If the post become successful, you'll get a 200 response stating that the post become successful.




Updated with a new entry. Notice that the update started with id = 1.

---

NOW we will update ino our table. Update is done by using PUT method.

The logic is the same as the POST method.

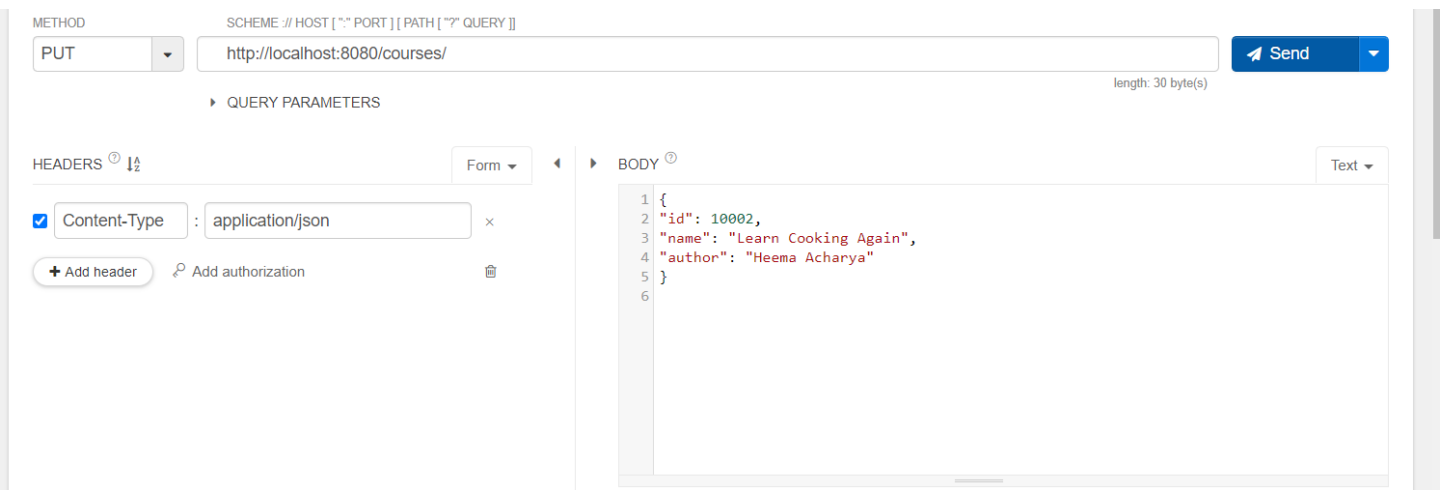
Key highlights include

```
@PutMapping("/courses")
public void updateCourse(@RequestBody Courses course) {
    repository.save(course);
}
```

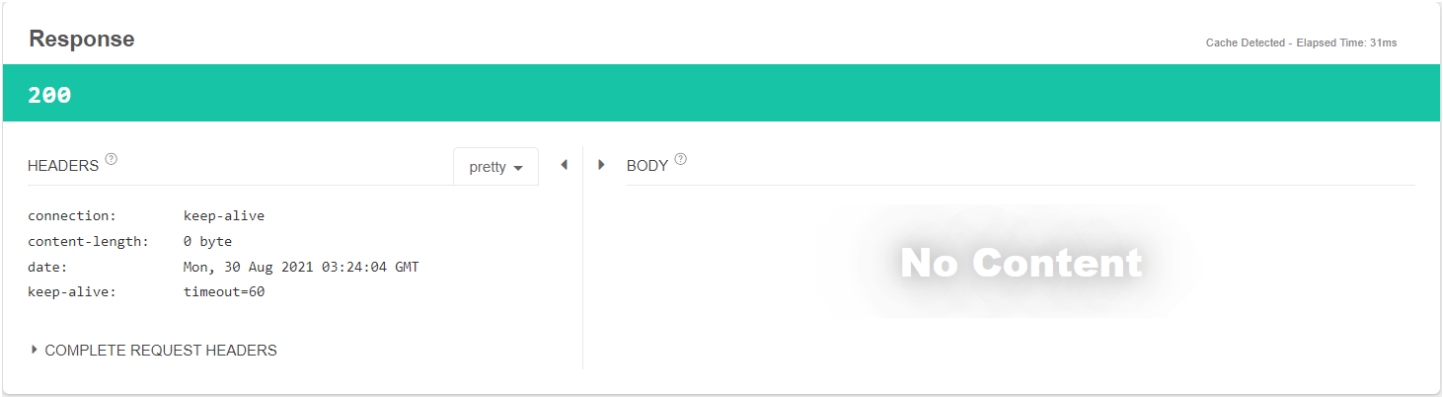
1. `@PutMapping("/courses")`
2. 
```
public void updateCourse(@RequestBody Courses course) {
    repository.save(course);
}
```
4. If the entiry exixts, it will be updated if not updated.
5. 

Let’s see this in action.

I made a little change to id 10002 in the name from “Learn Cooking” to “Learn Cooking Again”




I got 200 response stating a success.




Let’s see the data update in the web:

localhost:8080/courses/


```
[
  {
    "id": 10001,
    "name": "Learn Microservices",
    "author": "Tek Acharya"
  },
  {
    "id": 10002,
    "name": "Learn Cooking Again",
    "author": "Heema Acharya"
  },
  {
    "id": 10003,
    "name": "Learn Studying",
    "author": "Prinsa Acharya"
  },
  {
    "id": 10004,
    "name": "Learn Playing",
    "author": "Prajol Acharya"
  }
]
```


I see that the id 2 has been updated.

I also noticed that the one we just POSTed has been removed off.

Remember that the working here is a temperory just for learning. Our original data was hard-coded as.

LearnSpringBootApplication.java | Courses.java | CourseController.java | application.properties | data.sql

```
insert into MY_TABLE(ID, AUTHOR, NAME)  values (10001, 'Tek Acharya', 'Learn Microservices')
insert into MY_TABLE(ID, AUTHOR, NAME)  values (10002, 'Heema Acharya', 'Learn Cooking')
insert into MY_TABLE(ID, AUTHOR, NAME)  values (10003, 'Prinsa Acharya', 'Learn Studying')
insert into MY_TABLE(ID, AUTHOR, NAME)  values (10004, 'Prajol Acharya', 'Learn Playing')
```

Now, lets PUT a with a new Id that does not exists in the database table:

METHOD: PUT

SCHEME :// HOST [ ":" PORT ] [ PATH [ "?" QUERY ]]: http://localhost:8080/courses/

Send

length: 30 byte(s)

QUERY PARAMETERS

HEADERS — Form

Content-Type : application/json

+ Add header | Add authorization

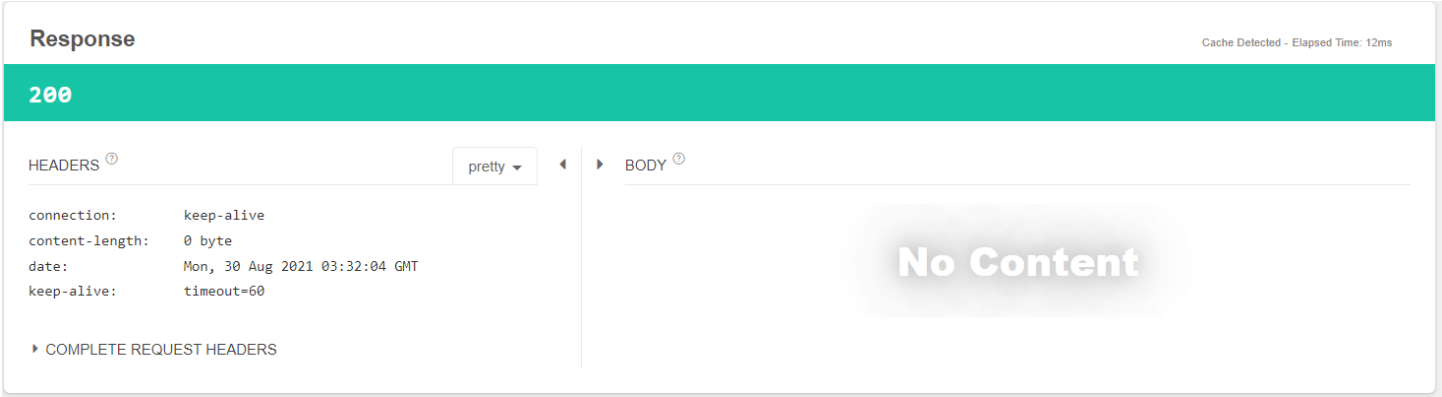
BODY — Text


```
{
"id": 10000,
"name": "I Visited America in 1999",
"author": "Hari Narayan Acharya"
}
```


Text JSON XML HTML | Format body | Enable body evaluation

length: 87 bytes

I made a simple modification to the body with a unique id and hit send




Happy return!

Now, let’s check the db table courses in the web


```
[
  {
    "id": 1,
    "name": "I Visited America in 1999",
    "author": "Hari Narayan Acharya"
  },
  {
    "id": 10001,
    "name": "Learn Microservices",
    "author": "Tek Acharya"
  },
  {
    "id": 10002,
    "name": "Learn Cooking Again",
    "author": "Heema Acharya"
  },
  {
    "id": 10003,
    "name": "Learn Studying",
    "author": "Prinsa Acharya"
  },
  {
    "id": 10004,
    "name": "Learn Playing",
    "author": "Prajol Acharya"
  }
]
```


Quite interesting to me

1. My Id provided was 1000, updated id is in 1
2. I ws expecting Id = 10002 to be updated to the hard-coded one from the .sql file.Hoerver, it stayed the recent updated.

Difference between POST and PUT ➔ POST -> no id info, PUT -> you know the id

That means we can make a little modification to our code.

FACT: providing id on the JSON file does not affect it to the data in there.

OLD

```
@PutMapping("/courses")
public void updateCourse(@RequestBody Courses course) {
    repository.save(course);
}
```

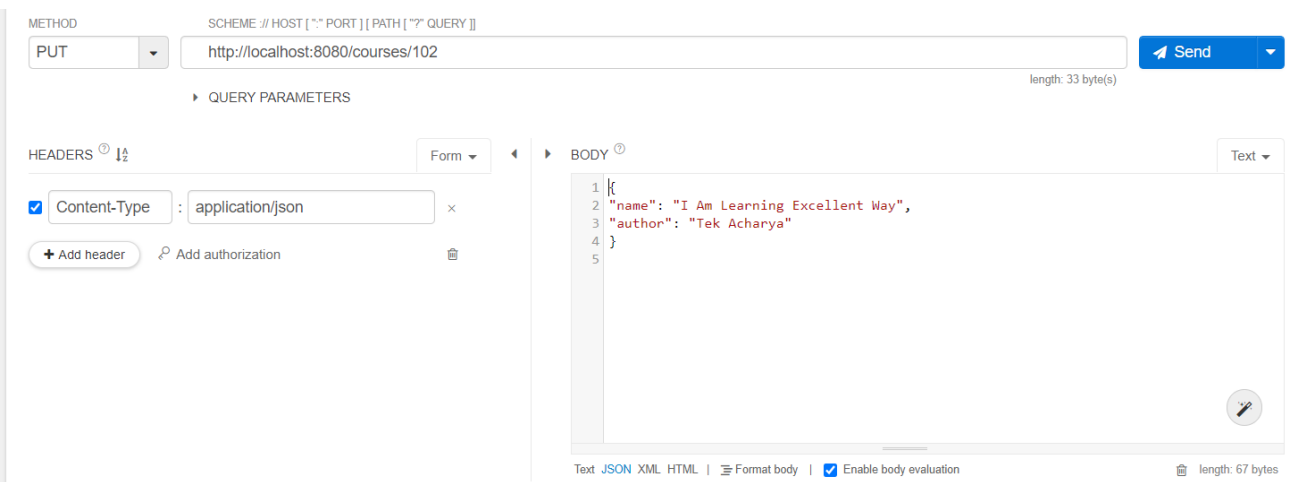
NEW

```
@PutMapping("/courses/{id}")
public void updateCourse(@PathVariable long id, @RequestBody Courses course) {
    repository.save(course);
}
```

Now, with this one, we can specify which id to fetch the update with the path variable with the id as

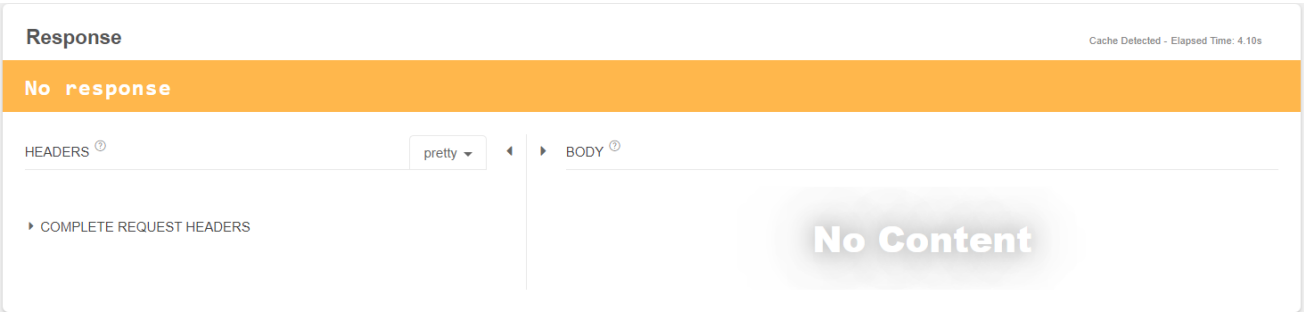
./courses/102

Let’s see



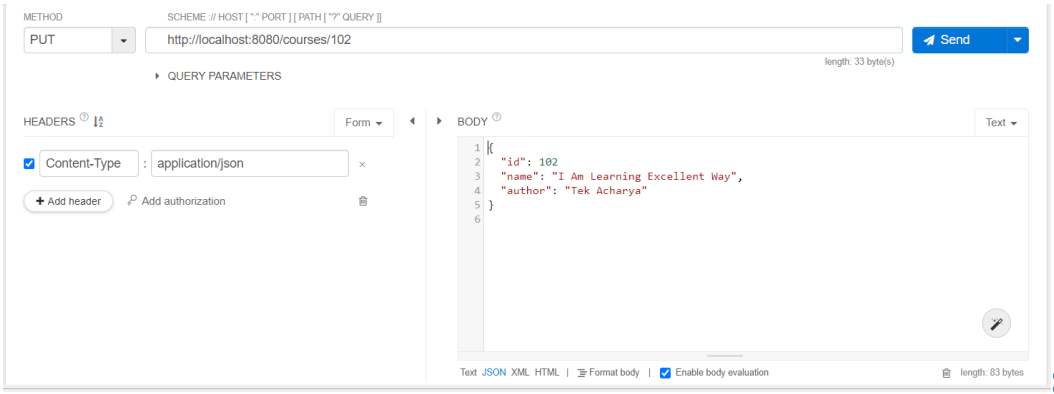

Made little modification to the body and provided the URI as courses/102

Lets hit send

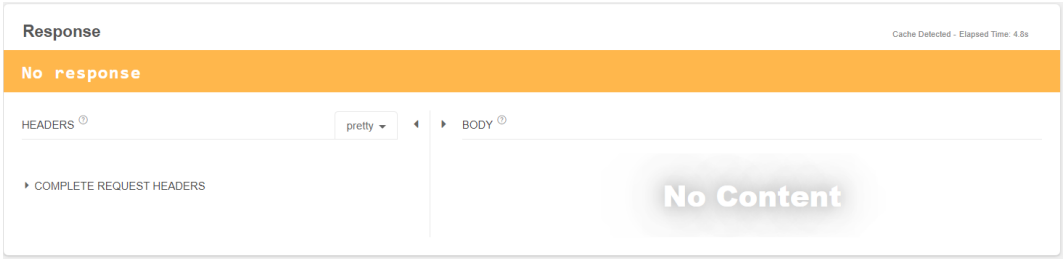



Sad

Let’s see what happened: I must provide the Id to be updated at that the body and a matching id at the URI




Send



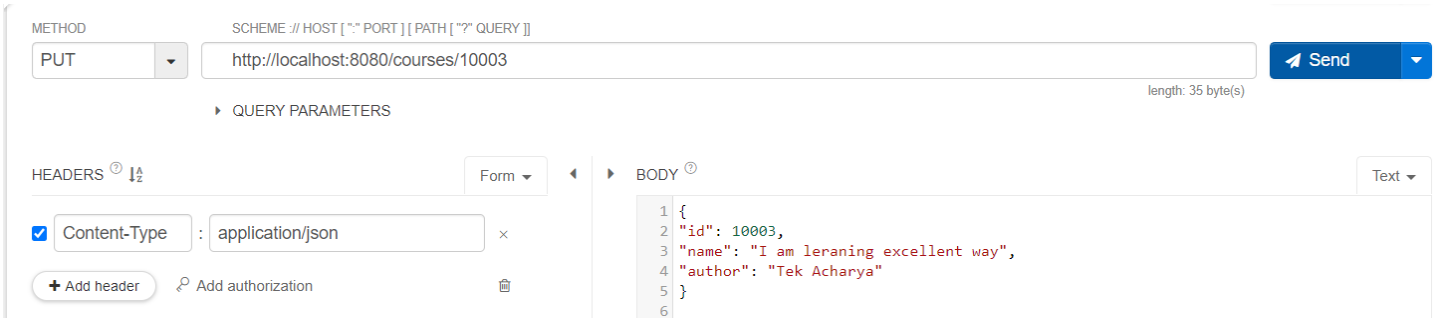

Sad

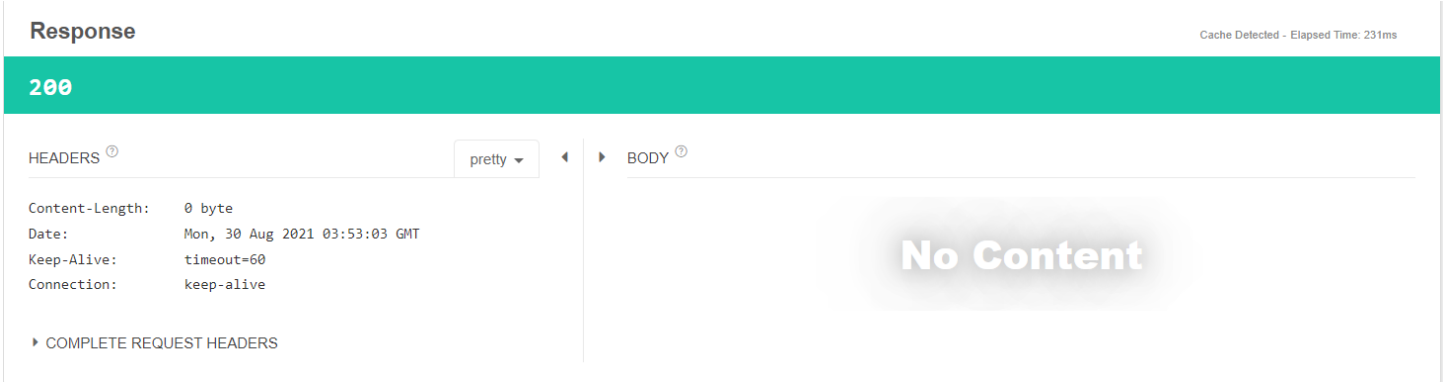
Let me try the id with the existing one and see

Dang. I forgot to save my changes to the code above.

Let me refresh all above.

I tried to update the the id 10003 and got a happy response as



localhost:8080/courses/


```
[
  {
    "id": 10001,
    "name": "Learn Microservices",
    "author": "Tek Acharya"
  },
  {
    "id": 10002,
    "name": "Learn Cooking",
    "author": "Heema Acharya"
  },
  {
    "id": 10003,
    "name": "I am leraning excellent way
    "author": "Tek Acharya"
  },
  {
    "id": 10004,
    "name": "Learn Playing",
    "author": "Prajol Acharya"
  }
]
```


10003 got updated.

Now let me try to update an non-existant id

Id used 102

METHOD: PUT

SCHEME :// HOST [ ":" PORT ] [ PATH [ "?" QUERY ]]: http://localhost:8080/courses/102

Send

length: 33 byte(s)

QUERY PARAMETERS

HEADERS — Form

Content-Type : application/json

+ Add header  Add authorization

BODY — Text


```
{
"id": 102,
"name": "I am leraning excellent way",
"author": "Tek Acharya"
}
```


**Response**

Cache Detected - Elapsed Time: 231ms

**200**

HEADERS — pretty

| | |
|---|---|
| Content-Length: | 0 byte |
| Date: | Mon, 30 Aug 2021 03:53:03 GMT |
| Keep-Alive: | timeout=60 |
| Connection: | keep-alive |

COMPLETE REQUEST HEADERS

BODY

No Content

Happy update

```
[
  {
    "id": 1,
    "name": "I am leraning excellent way",
    "author": "Tek Acharya"
  },
  {
    "id": 10001,
    "name": "Learn Microservices",
    "author": "Tek Acharya"
  },
  {
    "id": 10002,
    "name": "Learn Cooking",
    "author": "Heema Acharya"
  },
  {
    "id": 10003,
    "name": "I am leraning excellent way",
    "author": "Tek Acharya"
  },
  {
    "id": 10004,
    "name": "Learn Playing",
    "author": "Prajol Acharya"
  }
]
```


Updated id = 1

I am not providing any id in the body this time

METHOD: PUT

SCHEME :// HOST [ ":" PORT ] [ PATH [ "?" QUERY ]]: http://localhost:8080/courses/102

Send

length: 33 byte(s)

QUERY PARAMETERS

HEADERS — Form

Content-Type : application/json

+ Add header  Add authorization

BODY — Text


```
{
"name": "I am leraning excellent way",
"author": "Tek Acharya"
}
```


## Response

**200**

HEADERS — pretty — BODY

| | |
|---|---|
| connection: | keep-alive |
| content-length: | 0 byte |
| date: | Mon, 30 Aug 2021 03:58:14 GMT |
| keep-alive: | timeout=60 |

Happy

```
[
  {
      "id": 1,
      "name": "I am leraning excellent way",
      "author": "Tek Acharya"
  },
  {
      "id": 2,
      "name": "I am leraning excellent way",
      "author": "Tek Acharya"
  },
  {
      "id": 10001,
      "name": "Learn Microservices",
      "author": "Tek Acharya"
  },
  {
      "id": 10002,
      "name": "Learn Cooking",
      "author": "Heema Acharya"
  },
  {
      "id": 10003,
      "name": "I am leraning excellent way",
      "author": "Tek Acharya"
  },
  {
      "id": 10004,
      "name": "Learn Playing",
      "author": "Prajol Acharya"
  }
```


added to id =2

OOPS!

What happened.

This mean that we have no control to select the put in an id. If it exists it will update, if not it will auto increment to the default one.

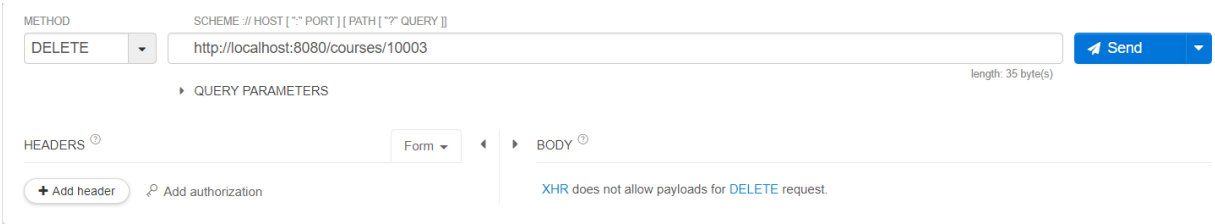
HURRAY

Finally: DELETE

```
    // DELETE
    // Pretty same as PUT method again

    @DeleteMapping("/courses/{id}")
    public void deleteCourse(@PathVariable long id) {
        repository.deleteById(id);
    }
```

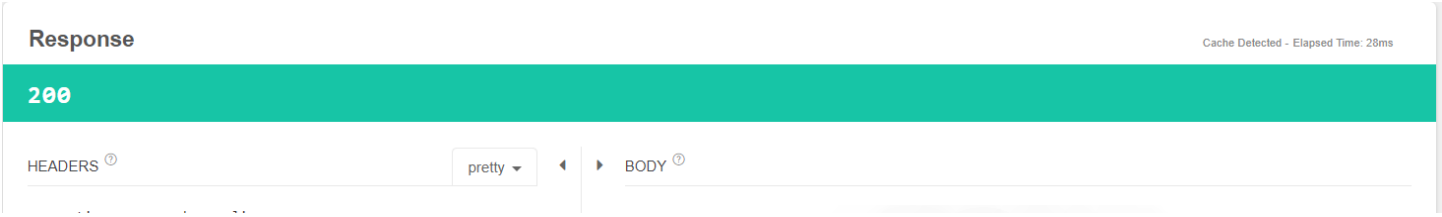
Pretty straight forward

We can also put this in try catch block or throw an error if the id does not exist.




Wanted to delete id = 10003

Hit send





```
[
  {
    "id": 10001,
    "name": "Learn Microservices",
    "author": "Tek Acharya"
  },
  {
    "id": 10002,
    "name": "Learn Cooking",
    "author": "Heema Acharya"
  },
  {
    "id": 10004,
    "name": "Learn Playing",
    "author": "Prajol Acharya"
  }
]
```


10003 is gone.

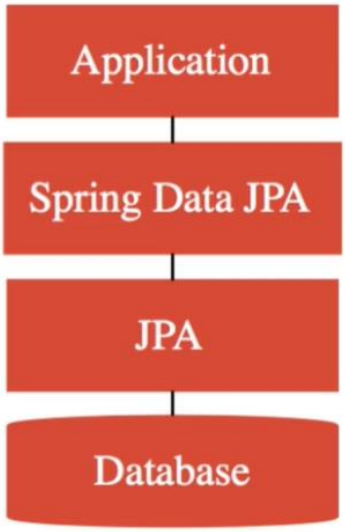
Happy Coding (08/30/2021-Sunday Night)

## REST API

- **REST API**: Architectural Style for the Web
  - **Resource**: Any information (Example: Courses)
  - **URI**: How do you identify a resource? (/courses, /courses/1)
  - You can perform actions on a resource (Create/Get/Delete/Update). Different HTTP Request Methods are used for different operations:
    - **GET** - Retrieve information (/courses, /courses/1)
    - **POST** - Create a new resource (/courses)
    - **PUT** - Update/Replace a resource (/courses/1)
    - **PATCH** - Update a part of the resource (/courses/1)
    - **DELETE** - Delete a resource (/courses/1)
  - **Representation**: How is the resource represented? (XML/JSON/Text/Video etc..)
  - **Server**: Provides the service (or API)
  - **Consumer**: Uses the service (Browser or a Front End Application)

## Spring Boot Auto Configuration Magic - Data JPA

- We added Data JPA and H2 dependencies:
  - Spring Boot Auto Configuration does some magic:
    - Initialize JPA and Spring Data JPA frameworks
    - Launch an in memory database (H2)
    - Setup connection from App to in-memory database
    - Launch a few scripts at startup (example: `data.sql`)
- **Remember** - H2 is in memory database
  - Does NOT persist data
  - Great for learning
  - BUT NOT so great for production
  - Let's see how to use MySQL next!




https://www.baeldung.com/spring-boot-war-tomcat-deploy

While working with BusinessLogic-DataSupplier-WebControler-ApplicationManager

I learned the following:

1. Just to get connected to the web API(RESTful API) we need the following

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

We can also achieve this in a different way by creating this as a local variable to

```
SpringApplication.run(SubwayApplication.class, args);
```

As

```
@SpringBootApplication
public class LearnSpringFrameworkApplication {
    public static void main(String[] args) {
        ConfigurableApplicationContext context = SpringApplication.run(LearnSpringFrameworkApplication.class, args);
        GameRunner runner = context.getBean(GameRunner.class);
        runner.runGame();
    }
}
```

Here,

```
@Component
public class GameRunner {
    private GameConsole game;

    @Autowired
    public GameRunner(GameConsole game) { // Constructor
        this.game = game;
    }

    public void runGame() {
        game.up();
        game.down();
        game.left();
        game.right();
    }
}
```

Noting carefully, I see that two important thing

```
a. @SpringBootApplication

b. SpringApplication.run(SubwayApplication.class, args)
```

c. The className and the argument for the run() function has to be the same along with .class and args parameter.

This helps us connect springboot along with the RESTful API

2. The next thing we need is the webController. This helps us connect and control to the web Controller needs:

   a. @RestController

b. @GetMapping("path")
c. A public method with a return type

Important :

1. Since no other class is dependent on this one, no @Component is required. However, since this is connecting to the web page, @RestController is needed
2. Since, this class is dependent on BusonessLogic class an @Autowired is required to the field of the class coming from BusonessLogic.

```
@RestController
public class ControlWeb {
    @Autowired
    private BusinessLogic bizLogic;
```

3. Now, since BusinessLogic gives and takes from two class this class is a @Component for one and must @Autowired to its dependent class which is DataSupplier class.

```
@Component
public class BusinessLogic {
    @Autowired
    private DataSupplier supplier;
```

4. Finally, DataSupplier class is not depending on any other class or component, it does not need any @Autowired. However, since another class (BusinessLogic) is dependent to this class it needs to be a @Component.

```
@Component
public class DataSupplier {
```

HURRAY