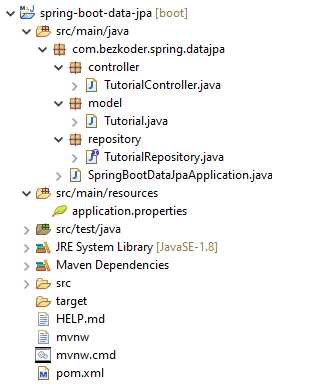
***SPRINGBOOT +SPRING DATA:***

**PROJECT STRUCTURE**

****

– Tutorial data model class corresponds to entity and table tutorials.  
– TutorialRepository is an interface that extends [JpaRepository](https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html) for CRUD methods and custom finder methods. It will be autowired in TutorialController.  
– TutorialController is a [RestController](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html) which has request mapping methods for RESTful requests such as: getAllTutorials, createTutorial, updateTutorial, deleteTutorial, findByPublished…  
– Configuration for Spring Datasource, JPA & Hibernate in **application.properties**.  
– **pom.xml** contains dependencies for Spring Boot and MySQL/PostgreSQL.

**Create & Setup Spring Boot project**

Use [Spring web tool](https://start.spring.io/) or your development tool ([Spring Tool Suite](https://spring.io/tools), Eclipse, [Intellij](https://www.jetbrains.com/idea/download/)) to create a Spring Boot project.

Then open **pom.xml** and add these dependencies:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

We also need to add one more dependency.  
– If you want to use **MySQL**:

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<scope>runtime</scope>

</dependency>

– or **PostgreSQL**:

<dependency>

<groupId>org.postgresql</groupId>

<artifactId>postgresql</artifactId>

<scope>runtime</scope>

</dependency>

## Configure Spring Datasource, JPA, Hibernate

Under src/main/resources folder, open application.properties and write these lines.

– For MySQL:

spring.datasource.url= jdbc:mysql://localhost:3306/testdb?useSSL=false

spring.datasource.username= root

spring.datasource.password= 123456

spring.jpa.properties.hibernate.dialect= org.hibernate.dialect.MySQL5InnoDBDialect

# Hibernate ddl auto (create, create-drop, validate, update)

spring.jpa.hibernate.ddl-auto= update

– For PostgreSQL:

spring.datasource.url= jdbc:postgresql://localhost:5432/testdb

spring.datasource.username= postgres

spring.datasource.password= 123

spring.jpa.properties.hibernate.jdbc.lob.non\_contextual\_creation= true

spring.jpa.properties.hibernate.dialect= org.hibernate.dialect.PostgreSQLDialect

# Hibernate ddl auto (create, create-drop, validate, update)

spring.jpa.hibernate.ddl-auto= update

* spring.datasource.username & spring.datasource.password properties are the same as your database installation.
* Spring Boot uses Hibernate for JPA implementation, we configure MySQL5InnoDBDialect for MySQL or PostgreSQLDialect for PostgreSQL
* spring.jpa.hibernate.ddl-auto is used for database initialization. We set the value to update value so that a table will be created in the database automatically corresponding to defined data model. Any change to the model will also trigger an update to the table. For production, this property should be validate.

## Define Data Model

Our Data model is Tutorial with four fields: id, title, description, published.  
In **model** package, we define Tutorial class.

model/Tutorial.java

package com.bezkoder.spring.datajpa.model;

import javax.persistence.\*;

@Entity

@Table(name = "tutorials")

public class Tutorial {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private long id;

@Column(name = "title")

private String title;

@Column(name = "description")

private String description;

@Column(name = "published")

private boolean published;

public Tutorial() {

}

public Tutorial(String title, String description, boolean published) {

this.title = title;

this.description = description;

this.published = published;

}

public long getId() {

return id;

}

public String getTitle() {

return title;

}

public void setTitle(String title) {

this.title = title;

}

public String getDescription() {

return description;

}

public void setDescription(String description) {

this.description = description;

}

public boolean isPublished() {

return published;

}

public void setPublished(boolean isPublished) {

this.published = isPublished;

}

@Override

public String toString() {

return "Tutorial [id=" + id + ", title=" + title + ", desc=" + description + ", published=" + published + "]";

}

}

– @Entity annotation indicates that the class is a persistent Java class.  
– @Table annotation provides the table that maps this entity.  
– @Id annotation is for the primary key.  
– @GeneratedValue annotation is used to define generation strategy for the primary key. GenerationType.AUTO means Auto Increment field.  
– @Column annotation is used to define the column in database that maps annotated field.

Relationship and I write a tutorial for this at:  
[Spring Boot One To Many example with JPA, Hibernate](https://www.bezkoder.com/jpa-one-to-many/)

Or add Tags with Many-to-Many Relationship:  
[Spring Boot Many to Many example with JPA, Hibernate](https://www.bezkoder.com/jpa-many-to-many/)

## Create Repository Interface

Let’s create a repository to interact with Tutorials from the database.  
In **repository** package, create TutorialRepository interface that extends JpaRepository.

repository/TutorialRepository.java

package com.bezkoder.spring.datajpa.repository;

import java.util.List;

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

import com.bezkoder.spring.datajpa.model.Tutorial;

public interface TutorialRepository extends JpaRepository<Tutorial, Long> {

List<Tutorial> findByPublished(boolean published);

List<Tutorial> findByTitleContaining(String title);

}

Now we can use JpaRepository’s methods: save(), findOne(), findById(), findAll(), count(), delete(), deleteById()… without implementing these methods.

We also define custom finder methods:  
– findByPublished(): returns all Tutorials with published having value as input published.  
– findByTitleContaining(): returns all Tutorials which title contains input title.

The implementation is plugged in by [Spring Data JPA](https://docs.spring.io/spring-data/jpa/docs/current/reference/html/) automatically.

You can modify this Repository:  
– to work with Pagination, the instruction can be found at:  
[Spring Boot Pagination & Filter example | Spring JPA, Pageable](https://bezkoder.com/spring-boot-pagination-filter-jpa-pageable/)  
– to sort/order by multiple fields with the tutorial:  
[Spring Data JPA Sort/Order by multiple Columns | Spring Boot](https://bezkoder.com/spring-data-sort-multiple-columns/)

More Derived queries at:  
[JPA Repository query example in Spring Boot](https://www.bezkoder.com/jpa-repository-query/)

Custom query with @Query annotation:  
[Spring JPA @Query example: Custom query in Spring Boot](https://www.bezkoder.com/spring-jpa-query/)

You also find way to write Unit Test for this JPA Repository at:  
[Spring Boot Unit Test for JPA Repositiory with @DataJpaTest](https://bezkoder.com/spring-boot-unit-test-jpa-repo-datajpatest/)

## Create Spring Rest APIs Controller

Finally, we create a controller that provides APIs for creating, retrieving, updating, deleting and finding Tutorials.

controller/TutorialController.java

package com.bezkoder.spring.datajpa.controller;

import java.util.ArrayList;

import java.util.List;

import java.util.Optional;

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

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

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

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

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

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

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

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

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

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

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

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

import com.bezkoder.spring.datajpa.model.Tutorial;

import com.bezkoder.spring.datajpa.repository.TutorialRepository;

@CrossOrigin(origins = "http://localhost:8081")

@RestController

@RequestMapping("/api")

public class TutorialController {

@Autowired

TutorialRepository tutorialRepository;

@GetMapping("/tutorials")

public ResponseEntity<List<Tutorial>> getAllTutorials(@RequestParam(required = false) String title) {

try {

List<Tutorial> tutorials = new ArrayList<Tutorial>();

if (title == null)

tutorialRepository.findAll().forEach(tutorials::add);

else

tutorialRepository.findByTitleContaining(title).forEach(tutorials::add);

if (tutorials.isEmpty()) {

return new ResponseEntity<>(HttpStatus.NO\_CONTENT);

}

return new ResponseEntity<>(tutorials, HttpStatus.OK);

} catch (Exception e) {

return new ResponseEntity<>(null, HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

@GetMapping("/tutorials/{id}")

public ResponseEntity<Tutorial> getTutorialById(@PathVariable("id") long id) {

Optional<Tutorial> tutorialData = tutorialRepository.findById(id);

if (tutorialData.isPresent()) {

return new ResponseEntity<>(tutorialData.get(), HttpStatus.OK);

} else {

return new ResponseEntity<>(HttpStatus.NOT\_FOUND);

}

}

@PostMapping("/tutorials")

public ResponseEntity<Tutorial> createTutorial(@RequestBody Tutorial tutorial) {

try {

Tutorial \_tutorial = tutorialRepository

.save(new Tutorial(tutorial.getTitle(), tutorial.getDescription(), false));

return new ResponseEntity<>(\_tutorial, HttpStatus.CREATED);

} catch (Exception e) {

return new ResponseEntity<>(null, HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

@PutMapping("/tutorials/{id}")

public ResponseEntity<Tutorial> updateTutorial(@PathVariable("id") long id, @RequestBody Tutorial tutorial) {

Optional<Tutorial> tutorialData = tutorialRepository.findById(id);

if (tutorialData.isPresent()) {

Tutorial \_tutorial = tutorialData.get();

\_tutorial.setTitle(tutorial.getTitle());

\_tutorial.setDescription(tutorial.getDescription());

\_tutorial.setPublished(tutorial.isPublished());

return new ResponseEntity<>(tutorialRepository.save(\_tutorial), HttpStatus.OK);

} else {

return new ResponseEntity<>(HttpStatus.NOT\_FOUND);

}

}

@DeleteMapping("/tutorials/{id}")

public ResponseEntity<HttpStatus> deleteTutorial(@PathVariable("id") long id) {

try {

tutorialRepository.deleteById(id);

return new ResponseEntity<>(HttpStatus.NO\_CONTENT);

} catch (Exception e) {

return new ResponseEntity<>(HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

@DeleteMapping("/tutorials")

public ResponseEntity<HttpStatus> deleteAllTutorials() {

try {

tutorialRepository.deleteAll();

return new ResponseEntity<>(HttpStatus.NO\_CONTENT);

} catch (Exception e) {

return new ResponseEntity<>(HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

@GetMapping("/tutorials/published")

public ResponseEntity<List<Tutorial>> findByPublished() {

try {

List<Tutorial> tutorials = tutorialRepository.findByPublished(true);

if (tutorials.isEmpty()) {

return new ResponseEntity<>(HttpStatus.NO\_CONTENT);

}

return new ResponseEntity<>(tutorials, HttpStatus.OK);

} catch (Exception e) {

return new ResponseEntity<>(HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

}

– @CrossOrigin is for configuring allowed origins.  
– @RestController annotation is used to define a controller and to indicate that the return value of the methods should be be bound to the web response body.  
– @RequestMapping("/api") declares that all Apis’ url in the controller will start with /api.  
– We use @Autowired to inject TutorialRepository bean to local variable.

## Run & Test

Run Spring Boot application with command: mvn spring-boot:run.

**tutorials** table will be automatically generated in Database.  
If you check MySQL for example, you can see things like this:

mysql> describe tutorials;

+--------------+--------------+------+-----+---------+-------+

| Field | Type | Null | Key | Default | Extra |

+--------------+--------------+------+-----+---------+-------+

| id | bigint(20) | NO | PRI | NULL | |

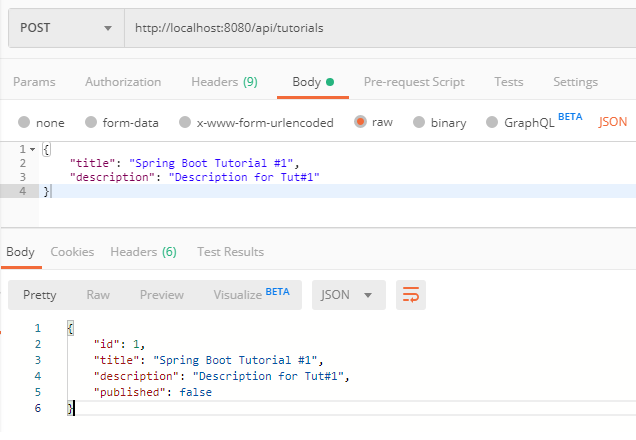
| description | varchar(255) | YES | | NULL | |

| published | bit(1) | YES | | NULL | |

| title | varchar(255) | YES | | NULL | |

+--------------+--------------+------+-----+---------+-------+

Create some Tutorials:



mysql> select \* from tutorials;

+----+-----------------------+--------------+-------------------------+

| id | description | published | title |

+----+-----------------------+--------------+-------------------------+

| 1 | Description for Tut#1 | 0 | Spring Boot Tutorial #1 |

| 2 | Description for Tut#2 | 0 | Spring Boot Tutorial #2 |

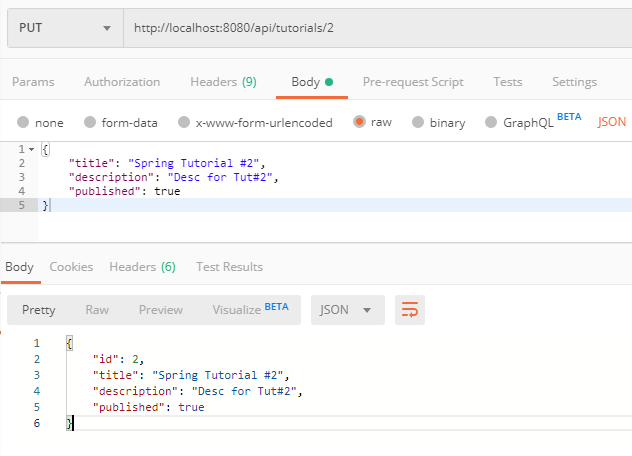
| 3 | Description for Tut#3 | 0 | Spring Boot Tutorial #3 |

| 4 | Tut#4 Description | 0 | Spring Data Tutorial #4 |

| 5 | Tut#5 Description | 0 | Spring Data Tutorial #5 |

+----+-----------------------+--------------+-------------------------+

Update some Tutorials:



mysql> select \* from tutorials;

+----+-----------------------+--------------+-------------------------+

| id | description | published | title |

+----+-----------------------+--------------+-------------------------+

| 1 | Description for Tut#1 | 0 | Spring Boot Tutorial #1 |

| 2 | Desc for Tut#2 | 1 | Spring Tutorial #2 |

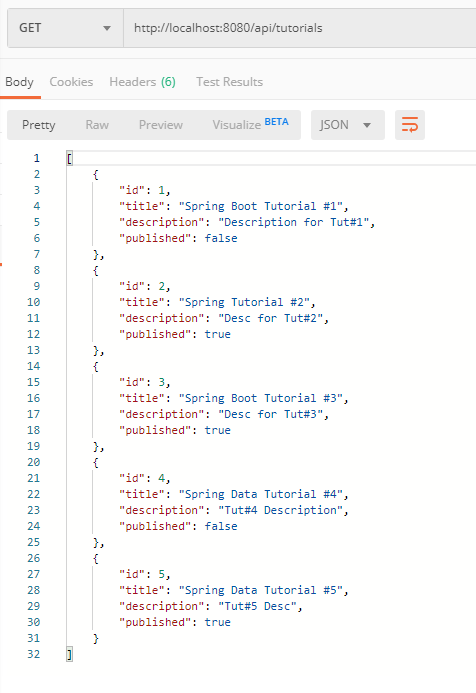
| 3 | Desc for Tut#3 | 1 | Spring Boot Tutorial #3 |

| 4 | Tut#4 Description | 0 | Spring Data Tutorial #4 |

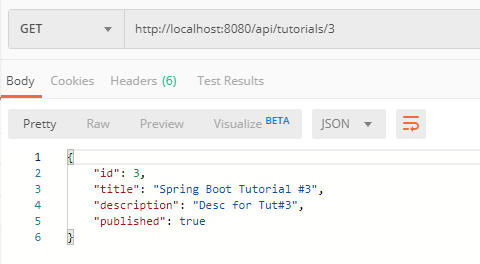
| 5 | Tut#5 Desc | 1 | Spring Data Tutorial #5 |

+----+-----------------------+--------------+-------------------------+

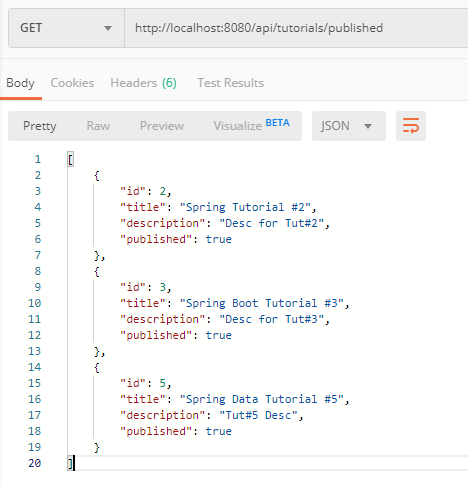
Get all Tutorials:



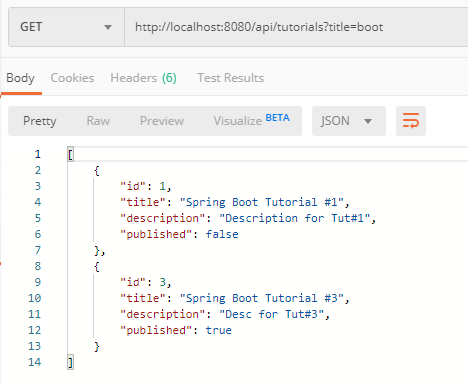
Get a Tutorial by Id:



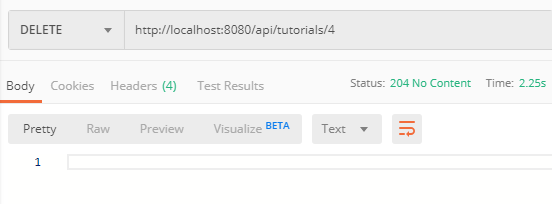
Find all ***published*** Tutorials:



Find all Tutorials which title contains ‘boot’:



Delete a Tutorial:



mysql> select \* from tutorials;

+----+-----------------------+--------------+-------------------------+

| id | description | published | title |

+----+-----------------------+--------------+-------------------------+

| 1 | Description for Tut#1 | 0 | Spring Boot Tutorial #1 |

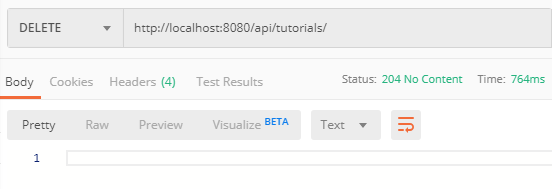
| 2 | Desc for Tut#2 | 1 | Spring Tutorial #2 |

| 3 | Desc for Tut#3 | 1 | Spring Boot Tutorial #3 |

| 5 | Tut#5 Desc | 1 | Spring Data Tutorial #5 |

+----+-----------------------+--------------+-------------------------+

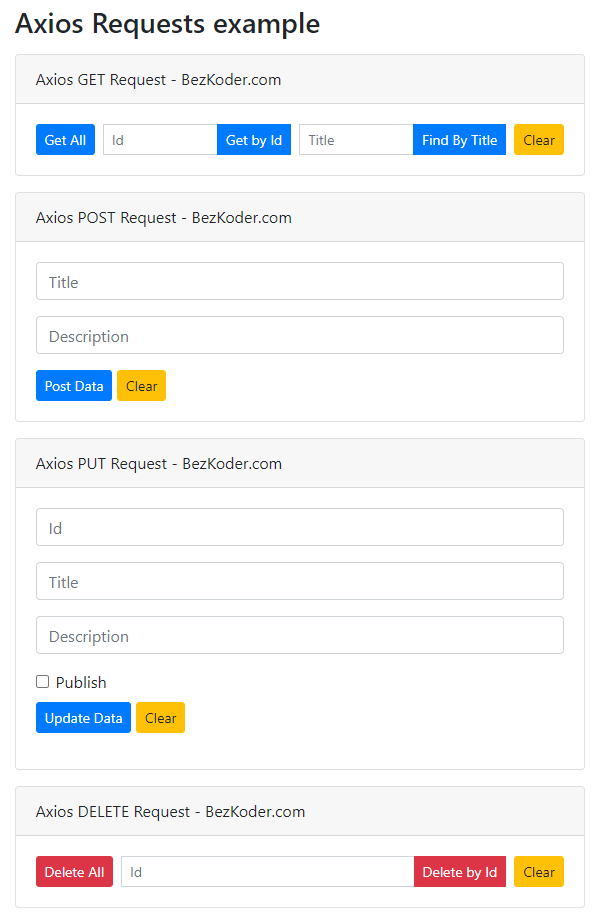
Delete all Tutorials:



mysql> select \* from tutorials;

Empty set (0.00 sec)

You can use the [Simple HTTP Client using Axios](https://www.bezkoder.com/axios-request/) to check it.



Or: [Simple HTTP Client using Fetch API](https://www.bezkoder.com/javascript-fetch/)

## Conclusion

Today we’ve built a Rest CRUD API using Spring Boot, Spring Data JPA, Hibernate, Maven to interact with MySQL/PostgreSQL.

We also see that JpaRepository supports a great way to make CRUD operations and custom finder methods without need of boilerplate code.