Spring Boot, JPA/Hibernate, PostgreSQL example with Maven

Last modified: January 9, 2021 (https://bezkoder.com/spring-boot-postgresql-example/) bezkoder (https://bezkoder.com/author/bezkoder/) Spring (https://bezkoder.com/category/spring/)

In this tutorial, we're gonna build a Spring Boot Rest CRUD API example with Maven that use Spring Data JPA/Hibernate to interact with PostgreSQL database. You'll know:

- How to configure Spring Data, JPA, Hibernate to work with PostgreSQL Database
- How to define Data Models and Repository interfaces
- Way to create Spring Rest Controller to process HTTP requests
- Way to use Spring Data JPA to interact with PostgreSQL Database

More Practice:
– Spring Boot, Spring Security, PostgreSQL: JWT Authentication example (https://bezkoder.com/spring-boot-security-postgresql-jwt-authentication/)
– Spring Boot Rest XML example – Web service with XML Response (https://bezkoder.com/spring-boot-rest-xml/)
– Spring Boot Multipart File upload example (https://bezkoder.com/spring-boot-file-upload/)
– Spring Boot Pagination and Sorting example (https://bezkoder.com/spring-boot-pagination-sorting-example/)

Fullstack:
– Spring Boot + Vue.js example: Build a CRUD App (https://bezkoder.com/spring-boot-vue-js-crud-example/)
– Spring Boot + Angular + PostgreSQL example: Build a CRUD App (https://bezkoder.com/angular-10-spring-

boot-postgresql/)
– Spring Boot + React + PostgreSQL example: Build a CRUD App (https://bezkoder.com/spring-boot-react-postgresql/)

Exception Handling:
– Spring Boot @ControllerAdvice & @ExceptionHandler example (https://bezkoder.com/spring-boot-controlleradvice-exceptionhandler/)
– @RestControllerAdvice example in Spring Boot (https://bezkoder.com/spring-boot-restcontrolleradvice/)

Testing: Spring Boot Unit Test for JPA Repositiory with @DataJpaTest (https://bezkoder.com/spring-boot-unit-test-jpa-repo-datajpatest/)

Contents [hide]

Overview of Spring Boot, PostgreSQL example with Maven

We will build a Spring Boot + PostgreSQL + Rest CRUD API for a Tutorial application in that:

- Each Tutotial has id, title, description, published status.
- Apis help to create, retrieve, update, delete Tutorials.
- Apis also support custom finder methods such as find by published status or by title.

These are APIs that we need to provide:

Methods	Urls	Actions
POST	/api/tutorials	create new Tutorial
GET	/api/tutorials	retrieve all Tutorials
GET	/api/tutorials/:id	retrieve a Tutorial by `:id`
PUT	/api/tutorials/:id	update a Tutorial by `:id`
DELETE	/api/tutorials/:id	delete a Tutorial by `:id`

Methods	Urls	Actions
DELETE	/api/tutorials	delete all Tutorials
GET	/api/tutorials/published	find all published Tutorials
GET	/api/tutorials?title=[keyword]	find all Tutorials which title contains `keyword`

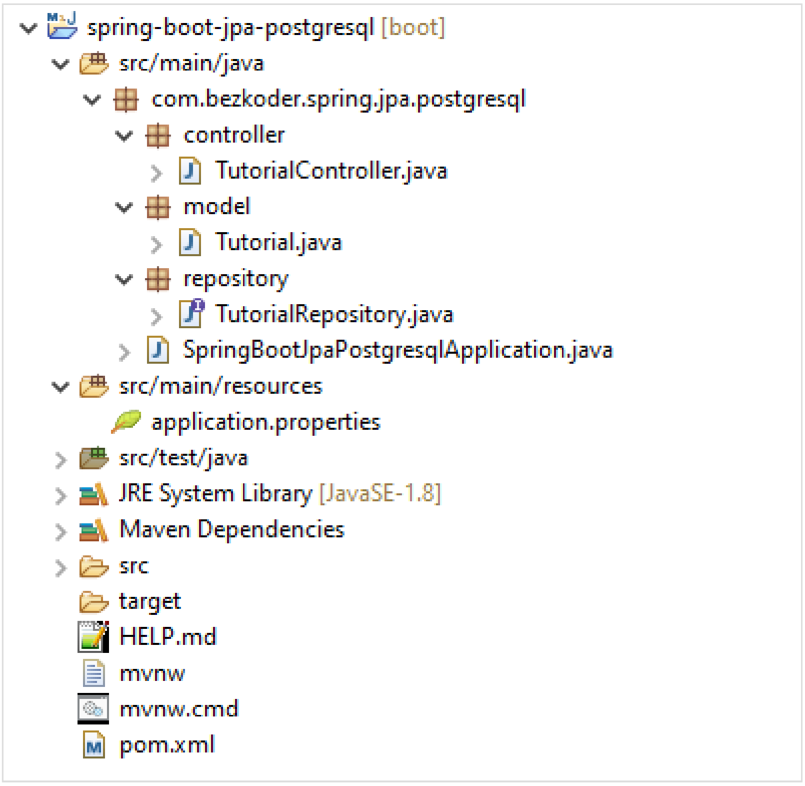
– We make CRUD operations & finder methods with Spring Data JPA's `JpaRepository` .

– The database will be PostgreSQL by configuring project dependency & datasource.

Technology

- Java 8
- Spring Boot 2 (with Spring Web MVC, Spring Data JPA)
- PostgreSQL
- Maven 3.6.1

Maven Project Structure



```
spring-boot-jpa-postgresql [boot]
  src/main/java
    com.bezkoder.spring.jpa.postgresql
      controller
        TutorialController.java
      model
        Tutorial.java
      repository
        TutorialRepository.java
      SpringBootJpaPostgresqlApplication.java
  src/main/resources
    application.properties
  src/test/java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

Let me explain it briefly.

– `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 PostgreSQL.

Create & Setup Spring Boot project

Use Spring web tool (http://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 Maven 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 for **PostgreSQL**:

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
```

Connect to PostgreSQL database

Under **src/main/resources** folder, open *application.properties* and configure Spring Data Source, JPA/Hibernate:

```
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 `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

```java
package com.bezkoder.spring.jpa.postgresql.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 + ", p
    }
}
```

– `@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.

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.jpa.postgresql.repository;

import java.util.List;

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

import com.bezkoder.spring.jpa.postgresql.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/)
– or 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/)

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.jpa.postgresql.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.jpa.postgresql.model.Tutorial;
import com.bezkoder.spring.jpa.postgresql.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)
        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(), fa
            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, @RequestB
        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 PostgreSQL for example, you can see things like this:

```
testdb=# \d tutorials
             Table "public.tutorials"
   Column    |          Type          | Modifiers
-------------+------------------------+-----------
 id          | bigint                 | not null
 description | character varying(255) |
 published   | boolean                |
 title       | character varying(255) |
Indexes:
    "tutorials_pkey" PRIMARY KEY, btree (id)
```

Create some Tutorials:

```
testdb=# SELECT * FROM tutorials;
 id |    description    | published |            title
----+-------------------+-----------+------------------------------
  1 | Tut#1 Description | f         | Spring Boot Tut#1
  2 | Tut#2 Description | f         | PostgreSQL Tut#2
  3 | Tut#3 Description | f         | Spring Data JPA Tut#3
  4 | Tut#4 Description | f         | Maven Tut#4
  5 | Tut#5 Description | f         | Spring Boot PostgreSQL Tut#5
(5 rows)
```

Update some Tutorials:

```
testdb=# SELECT * FROM tutorials;
 id |    description    | published |            title
----+-------------------+-----------+------------------------------
  3 | Tut#3 Description | f         | Spring Data JPA Tut#3
  5 | Tut#5 Description | f         | Spring Boot PostgreSQL Tut#5
  2 | Desc for Tut#2    | t         | PostgreSQL DB Tut#2
  4 | Desc for Tut#4    | t         | Maven Tut#4
  1 | Desc for Tut#1    | t         | Spring Boot Tut#1
(5 rows)
```

Get all Tutorials:

Get a Tutorial by Id:

Find all ***published*** Tutorials:

Find all Tutorials which title contains ‘ring’:

Delete a Tutorial:

```
testdb=# SELECT * FROM tutorials;
 id |    description    | published |            title
----+-------------------+-----------+-----------------------------
  3 | Tut#3 Description | f         | Spring Data JPA Tut#3
  5 | Tut#5 Description | f         | Spring Boot PostgreSQL Tut#5
  2 | Desc for Tut#2    | t         | PostgreSQL DB Tut#2
  1 | Desc for Tut#1    | t         | Spring Boot Tut#1
(4 rows)
```

Delete all Tutorials:

```
testdb=# SELECT * FROM tutorials;
 id | description | published | title
----+-------------+-----------+-------
(0 rows)
```

Conclusion

Today we've built a Spring Boot PostgreSQL example with Rest CRUD API using Maven & Spring Data JPA, Hibernate.

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

If you want to add Pagination to this Spring project, you can find the instruction at:
Spring Boot Pagination & Filter example | Spring JPA, Pageable (https://bezkoder.com/spring-boot-pagination-filter-jpa-pageable/)

To sort/order by multiple fields:
Spring Data JPA Sort/Order by multiple Columns | Spring Boot (https://bezkoder.com/spring-data-sort-multiple-columns/)

Handle Exception for this Rest APIs is necessary:
– Spring Boot @ControllerAdvice & @ExceptionHandler example (https://bezkoder.com/spring-boot-controlleradvice-exceptionhandler/)
– @RestControllerAdvice example in Spring Boot (https://bezkoder.com/spring-boot-restcontrolleradvice/)

Or way to write Unit Test for the JPA Repository:
Spring Boot Unit Test for JPA Repositiory with @DataJpaTest (https://bezkoder.com/spring-boot-unit-test-jpa-repo-datajpatest/)

Happy learning! See you again.

Further Reading

- Spring Boot, Spring Security, PostgreSQL: JWT Authentication example (https://bezkoder.com/spring-boot-security-postgresql-jwt-authentication/)
- Spring Data JPA Reference Documentation (https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#reference)
- Spring Boot Pagination and Sorting example (https://bezkoder.com/spring-boot-pagination-sorting-example/)

Fullstack examples:
– Spring Boot + Vue.js example: Build a CRUD App (https://bezkoder.com/spring-boot-vue-js-crud-example/)
– Spring Boot + Angular + PostgreSQL example: Build a CRUD App (https://bezkoder.com/angular-10-spring-boot-postgresql/)
– Spring Boot + React + PostgreSQL example: Build a CRUD App (https://bezkoder.com/spring-boot-react-postgresql/)

Source Code

You can find the complete source code for this tutorial on Github (https://github.com/bezkoder/spring-boot-jpa-postgresql).

crud (https://bezkoder.com/tag/crud/) hibernate (https://bezkoder.com/tag/hibernate/)
postgresql (https://bezkoder.com/tag/postgresql/) rest api (https://bezkoder.com/tag/rest-api/)
spring boot (https://bezkoder.com/tag/spring-boot/)
spring data jpa (https://bezkoder.com/tag/spring-data-jpa/)

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

☐

Save my name, email, and website in this browser for the next time I comment.

POST COMMENT

< Dart/Flutter Map, HashMap Tutorial with Examples (https://bezkoder.com/dart-map/)

Spring Boot + React + PostgreSQL example: Build a CRUD App > (https://bezkoder.com/spring-boot-react-postgresql/)

Search...

FOLLOW US

(https://facebook.com/bezkoder) (https://www.youtube.com/channel/UCp0mx9RH0JxaFsmMvKXA86Q) (https://github.com/bezkoder)

TOOLS

Json Formatter (https://bezkoder.com/json-formatter/)