@Ref: <https://www.callicoder.com/spring-boot-rest-api-tutorial-with-mysql-jpa-hibernate/>

Spring Boot has taken Spring framework to the next level. It has drastically reduced the configuration and setup time required for spring projects.

You can setup a project with almost zero configuration and start building the things that actually matter to your application.

If you are new to Spring boot and want to get started with it quickly, then this blog post is for you.

In this post, we’ll build a Restful CRUD API for a simple note-taking application. A Note can have a title and some content. We’ll first build the apis to create, retrieve, update and delete a Note, and then test them using postman.

So, Let’s get started!

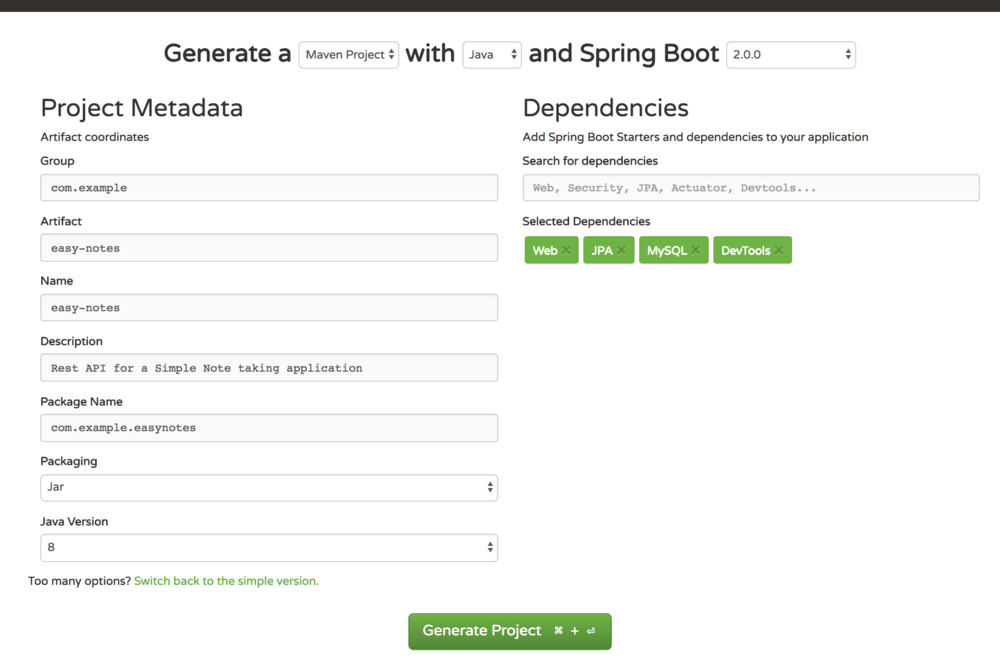
## Creating the Project

Spring Boot provides a web tool called [Spring Initializer](http://start.spring.io/) to bootstrap an application quickly. Just go to [http://start.spring.io](http://start.spring.io/) and follow the steps below to generate a new project.

**Step 1** : Click **Switch to full version** on [http://start.spring.io](http://start.spring.io/) page.

**Step 2** : Enter the details as follows -

* Group : com.example
* Artifact : easy-notes
* Name : easy-notes
* Description : Rest API for a Simple Note Taking Application
* Package Name : com.example.easynotes
* Packaging : jar (This is the default value)
* Java Version : 1.8 (Default)
* Dependencies : Web, JPA, MySQL, DevTools

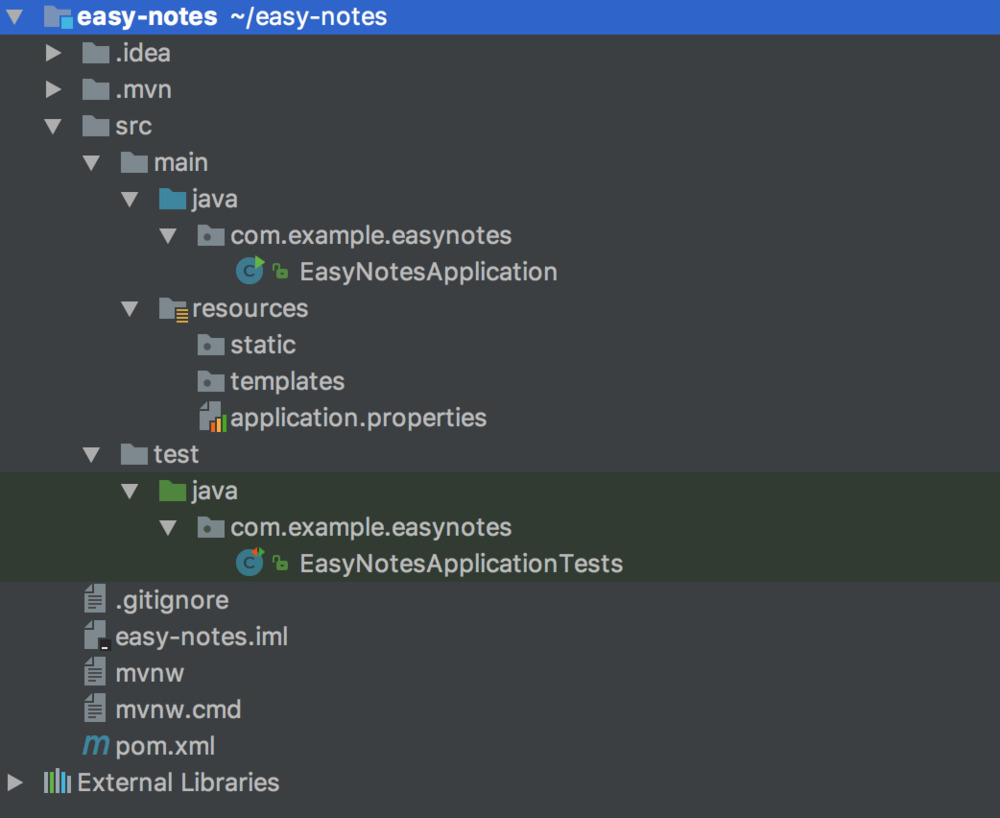


Once all the details are entered, click **Generate Project** to generate and download your project. Spring Initializer will generate the project with the details you have entered and download a zip file with all the project folders.

Next, Unzip the downloaded zip file and import it into your favorite IDE.

## Exploring the Directory Structure

Following is the directory structure of our Note taking application -



Let’s understand the details of some of the important files and directories -

**1. EasyNotesApplication**

This is the main entry point of our Spring Boot application.

package com.example.easynotes;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class EasyNotesApplication {

public static void main(String[] args) {

SpringApplication.run(EasyNotesApplication.class, args);

}

}

It contains a simple annotation called @SpringBootApplication which is a combination of the following more specific spring annotations -

* [@Configuration](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/annotation/Configuration.html) : Any class annotated with @Configuration annotation is bootstrapped by Spring and is also considered as a source of other bean definitions.
* [@EnableAutoConfiguration](https://docs.spring.io/spring-boot/docs/1.2.1.RELEASE/api/org/springframework/boot/autoconfigure/EnableAutoConfiguration.html) : This annotation tells Spring to automatically configure your application based on the dependencies that you have added in the pom.xml file.

For example, If spring-data-jpa is in the classpath, then it automatically tries to configure a DataSource by reading the database properties from application.properties file.

* [@ComponentScan](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/annotation/ComponentScan.html) : It tells Spring to scan and bootstrap other components defined in the current package (com.example.easynotes) and all the sub-packages.

The main() method calls Spring Boot’s SpringApplication.run() method to launch the application.

**2. resources/**

This directory, as the name suggests, is dedicated to all the static resources, templates and property files.

* **resources/static** - contains static resources such as css, js and images.
* **resources/templates** - contains server-side templates which are rendered by Spring.
* **resources/application.properties** - This file is very important. It contains application-wide properties. Spring reads the properties defined in this file to configure your application. You can define server’s default port, server’s context path, database URLs etc, in this file.

You can refer [this page](https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html) for common application properties used in Spring Boot.

**3. EasyNotesApplicationTests** - Define unit and integration tests here.

**4. pom.xml** - contains all the project dependencies

## Configuring MySQL Database

As I pointed out earlier, Spring Boot tries to auto-configure a DataSource if spring-data-jpa is in the classpath by reading the database configuration from application.properties file.

So, we just have to add the configuration and Spring Boot will take care of the rest.

Open application.properties file and add the following properties to it.

## Spring DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)

spring.datasource.url = jdbc:mysql://localhost:3306/notes\_app?useSSL=false

spring.datasource.username = root

spring.datasource.password = root

## Hibernate Properties

# The SQL dialect makes Hibernate generate better SQL for the chosen database

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

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

spring.jpa.hibernate.ddl-auto = update

You will need to create a database named **notes\_app** in MySQL, and change the spring.datasource.username & spring.datasource.password properties as per your MySQL installation.

In the above properties file, the last two properties are for hibernate. Spring Boot uses Hibernate as the default JPA implementation.

The property spring.jpa.hibernate.ddl-auto is used for database initialization. I’ve used the value **“update”** for this property.

It does two things -

* When you define a domain model, a table will automatically be created in the database and the fields of the domain model will be mapped to the corresponding columns in the table.
* Any change to the domain model will also trigger an update to the table. For example, If you change the name or type of a field, or add another field to the model, then all these changes will be reflected in the mapped table as well.

Using update for spring.jpa.hibernate.ddl-auto property is fine for development. But, For production, You should keep the value of this property to **“validate”**, and use a database migration tool like [Flyway](https://www.callicoder.com/spring-boot-flyway-database-migration-example/) for managing changes in the database schema.

## Creating the Note model

All right! Let’s now create the Note model. Our Note model has following fields -

* **id**: Primary Key with Auto Increment.
* **title**: The title of the Note. (NOT NULL field)
* **content**: Note’s content. (NOT NULL field)
* **createdAt**: Time at which the Note was created.
* **updatedAt**: Time at which the Note was updated.

Now, let’s see how we can model this in Spring. Create a new package called model inside com.example.easynotes and add a class named Note.java with following contents -

package com.example.easynotes.model;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

import org.springframework.data.annotation.CreatedDate;

import org.springframework.data.annotation.LastModifiedDate;

import org.springframework.data.jpa.domain.support.AuditingEntityListener;

import javax.persistence.\*;

import javax.validation.constraints.NotBlank;

import java.util.Date;

@Entity

@Table(name = "notes")

@EntityListeners(AuditingEntityListener.class)

@JsonIgnoreProperties(value = {"createdAt", "updatedAt"},

allowGetters = true)

public class Note implements Serializable {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@NotBlank

private String title;

@NotBlank

private String content;

@Column(nullable = false, updatable = false)

@Temporal(TemporalType.TIMESTAMP)

@CreatedDate

private Date createdAt;

@Column(nullable = false)

@Temporal(TemporalType.TIMESTAMP)

@LastModifiedDate

private Date updatedAt;

// Getters and Setters ... (Omitted for brevity)

}

* All your domain models must be annotated with **@Entity** annotation. It is used to mark the class as a persistent Java class.
* **@Table** annotation is used to provide the details of the table that this entity will be mapped to.
* **@Id** annotation is used to define the primary key.
* **@GeneratedValue** annotation is used to define the primary key generation strategy. In the above case, we have declared the primary key to be an Auto Increment field.
* **@NotBlank** annotation is used to validate that the annotated field is not null or empty.
* **@Column** annotation is used to define the properties of the column that will be mapped to the annotated field. You can define several properties like name, length, nullable, updateable etc.

By default, a field named createdAt is mapped to a column named created\_at in the database table. i.e. all camel cases are replaced with underscores.

If you want to map the field to a different column, you can specify it using -

@Column(name = "created\_on")

private String createdAt;

* **@Temporal** annotation is used with java.util.Date and java.util.Calendar classes. It converts the date and time values from Java Object to compatible database type and vice versa.
* **@JsonIgnoreProperties** annotation is a Jackson annotation. Spring Boot uses Jackson for Serializing and Deserializing Java objects to and from JSON.

This annotation is used because we don’t want the clients of the rest api to supply the createdAtand updatedAt values. If they supply these values then we’ll simply ignore them. However, we’ll include these values in the JSON response.

## Enable JPA Auditing

In our Note model we have annotated createdAt and updatedAt fields with @CreatedDate and @LastModifiedDate annotations respectively.

Now, what we want is that these fields should automatically get populated whenever we create or update an entity.

To achieve this, we need to do two things -

**1. Add Spring Data JPA’s AuditingEntityListener to the domain model.**

We have already done this in our Note model with the annotation @EntityListeners(AuditingEntityListener.class).

**2. Enable JPA Auditing in the main application.**

Open EasyNotesApplication.java and add @EnableJpaAuditing annotation.

@SpringBootApplication

@EnableJpaAuditing

public class EasyNotesApplication {

public static void main(String[] args) {

SpringApplication.run(EasyNotesApplication.class, args);

}

}

## Creating NoteRepository to access data from the database

The next thing we’re gonna do is create a repository to access Note’s data from the database.

Well, Spring Data JPA has got us covered here. It comes with a [JpaRepository](https://docs.spring.io/autorepo/docs/spring-data-jpa/current/api/org/springframework/data/jpa/repository/JpaRepository.html) interface which defines methods for all the CRUD operations on the entity, and a default implementation of JpaRepositorycalled [SimpleJpaRepository](http://docs.spring.io/autorepo/docs/spring-data-jpa/current/api/org/springframework/data/jpa/repository/support/SimpleJpaRepository.html).

Cool! Let’s create the repository now. First, Create a new package called repository inside the base package com.example.easynotes. Then, create an interface called NoteRepository and extend it from JpaRepository -

package com.example.easynotes.repository;

import com.example.easynotes.model.Note;

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

@Repository

public interface NoteRepository extends JpaRepository<Note, Long> {

}

Note that, we have annotated the interface with @Repository annotation. This tells Spring to bootstrap the repository during component scan.

Great! That is all you have to do in the repository layer. You will now be able to use JpaRepository’s methods like save(), findOne(), findAll(), count(), delete() etc.

You don’t need to implement these methods. They are already implemented by Spring Data JPA’s SimpleJpaRepository. This implementation is plugged in by Spring automatically at runtime.

Checkout all the methods available from [SimpleJpaRepository’s documentation](http://docs.spring.io/autorepo/docs/spring-data-jpa/current/api/org/springframework/data/jpa/repository/support/SimpleJpaRepository.html).

## Creating Custom Business Exception

We’ll define the Rest APIs for creating, retrieving, updating, and deleting a Note in the next section.

The APIs will throw a ResourceNotFoundException whenever a Note with a given id is not found in the database.

Following is the definition of ResourceNotFoundException. (I’ve created a package named exceptioninside com.example.easynotes to store this exception class) -

package com.example.easynotes.exception;

import org.springframework.http.HttpStatus;

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

@ResponseStatus(value = HttpStatus.NOT\_FOUND)

public class ResourceNotFoundException extends RuntimeException {

private String resourceName;

private String fieldName;

private Object fieldValue;

public ResourceNotFoundException( String resourceName, String fieldName, Object fieldValue) {

super(String.format("%s not found with %s : '%s'", resourceName, fieldName, fieldValue));

this.resourceName = resourceName;

this.fieldName = fieldName;

this.fieldValue = fieldValue;

}

public String getResourceName() {

return resourceName;

}

public String getFieldName() {

return fieldName;

}

public Object getFieldValue() {

return fieldValue;

}

}

Notice the use of @ResponseStatus annotation in the above exception class. This will cause Spring boot to respond with the specified HTTP status code whenever this exception is thrown from your controller.

## Creating NoteController

**The Final Step** - We’ll now create the REST APIs for creating, retrieving, updating and deleting a Note.

First, create a new package controller inside com.example.easynotes. Then, create a new class NoteController.java with the following contents -

package com.example.easynotes.controller;

import com.example.easynotes.exception.ResourceNotFoundException;

import com.example.easynotes.model.Note;

import com.example.easynotes.repository.NoteRepository;

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

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import javax.validation.Valid;

import java.util.List;

@RestController

@RequestMapping("/api")

public class NoteController {

@Autowired

NoteRepository noteRepository;

// Get All Notes

// Create a new Note

// Get a Single Note

// Update a Note

// Delete a Note

}

**@RestController** annotation is a combination of Spring’s @Controller and @ResponseBodyannotations.

The @Controller annotation is used to define a controller and the @ResponseBody annotation is used to indicate that the return value of a method should be used as the response body of the request.

**@RequestMapping("/api")** declares that the url for all the apis in this controller will start with /api.

Let’s now look at the implementation of all the apis one by one.

#### 1. Get All Notes (GET /api/notes)

// Get All Notes

@GetMapping("/notes")

public List<Note> getAllNotes() {

return noteRepository.findAll();

}

The above method is pretty straightforward. It calls JpaRepository’s findAll() method to retrieve all the notes from the database and returns the entire list.

Also, The @GetMapping("/notes") annotation is a short form of @RequestMapping(value="/notes", method=RequestMethod.GET).

#### 2. Create a new Note (POST /api/notes)

// Create a new Note

@PostMapping("/notes")

public Note createNote(@Valid @RequestBody Note note) {

return noteRepository.save(note);

}

The @RequestBody annotation is used to bind the request body with a method parameter.

The @Valid annotation makes sure that the request body is valid. Remember, we had marked Note’s title and content with @NotBlank annotation in the Note model?

If the request body doesn’t have a title or a content, then spring will return a 400 BadRequest error to the client.

#### 3. Get a Single Note (Get /api/notes/{noteId})

// Get a Single Note

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

public Note getNoteById(@PathVariable(value = "id") Long noteId) {

return noteRepository.findById(noteId)

.orElseThrow(() -> new ResourceNotFoundException("Note", "id", noteId));

}

The @PathVariable annotation, as the name suggests, is used to bind a path variable with a method parameter.

In the above method, we are throwing a ResourceNotFoundException whenever a Note with the given id is not found.

This will cause Spring Boot to return a 404 Not Found error to the client (Remember, we had added a @ResponseStatus(value = HttpStatus.NOT\_FOUND) annotation to the ResourceNotFoundExceptionclass).

#### 4. Update a Note (PUT /api/notes/{noteId})

// Update a Note

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

public Note updateNote(@PathVariable(value = "id") Long noteId,

@Valid @RequestBody Note noteDetails) {

Note note = noteRepository.findById(noteId)

.orElseThrow(() -> new ResourceNotFoundException("Note", "id", noteId));

note.setTitle(noteDetails.getTitle());

note.setContent(noteDetails.getContent());

Note updatedNote = noteRepository.save(note);

return updatedNote;

}

#### 5. Delete a Note (DELETE /api/notes/{noteId})

// Delete a Note

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

public ResponseEntity<?> deleteNote(@PathVariable(value = "id") Long noteId) {

Note note = noteRepository.findById(noteId)

.orElseThrow(() -> new ResourceNotFoundException("Note", "id", noteId));

noteRepository.delete(note);

return ResponseEntity.ok().build();

}

## Running the Application

We’ve successfully built all the apis for our application. Let’s now run the app and test the apis.

Just go to the root directory of the application and type the following command to run it -

$ mvn spring-boot:run

The application will start at Spring Boot’s default tomcat port 8080.

Great! Now, It’s time to test our apis using postman.

## Testing the APIs

<http://localhost:8080/api/notes/>

