**Spring Boot Tutorials & Protips**

# Spring Boot Optional Path Variables

February 17, 2020 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

It is quite common in Spring Boot to perform more than one task in a single controller method. In this quick article, you'll learn how to use optional path variables in Spring Boot to handle more than one type of request.

By design, in Spring Boot, it is not possible to have optional path variables. Let us say you have the following controller method that handles GET requests on the /todos/{id} endpoint:

@GetMapping(value = {"/todos", "/todos/{id}"})

public @ResponseBody Object fetchTodos(@PathVariable Long id) {

if (id == null) {

// return single todo

return todoRespository.findById(id);

} else {

// return all todos

return todoRespository.findAll();

}

}

We want to make sure that the above controller method works for both /todos and /todos/1 requests. If the id path variable is present, we just fetch a todo by ID and send it back to the client. Otherwise, simply returns all todos.

Right now, if you skip the id path variable and call the /todos endpoint directly, Spring Boot will throw an exception. One way to avoid this exception is to **have two methods**, one with the path variable and then another without as shown below:

// two methods - not recommended

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

public @ResponseBody Todo fetchTodoById(@PathVariable Long id) {

return todoRespository.findById(id);

}

@GetMapping("/todos")

public @ResponseBody List<Todo> fetchAllTodos() {

return todoRespository.findAll();

}

Technically, the above approach should work fine but what if you have a bunch of optional path variables? This approach can quickly lead to a large number of (almost) duplicate code-blocks.

## Using required Attribute

If you are using Spring Boot 2 and higher, the easiest and efficient way to make any @PathVariable optional is by setting the required attribute to false as shown below:

@GetMapping(value = {"/todos", "/todos/{id}"})

public @ResponseBody Object fetchTodos(@PathVariable(required = false) Long id) {

if (id == null) {

return todoRespository.findById(id);

} else {

return todoRespository.findAll();

}

}

That's all you need to do. Now, the above method can handle both types of requests without any exception.

## Using Java 8 Optional Class

Another way to make a path variable optional in Spring Boot (with Spring 4+) is by using the Java 8 Optional class:

@GetMapping(value = {"/todos", "/todos/{id}"})

public @ResponseBody Object fetchTodos(@PathVariable Optional<Long> id) {

if (id.isPresent()) {

return todoRespository.findById(id.get());

} else {

return todoRespository.findAll();

}

}

The Optional class is a special container object which may or may not contain a non-null value. If a value is present, the isPresent() method will return true and get() will return the value.

# RestTemplate GET Request with Parameters and Headers

November 09, 2019 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

In this article, you will learn how to make different kinds of HTTP GET requests by using the [RestTemplate](https://attacomsian.com/blog/http-requests-resttemplate-spring-boot) class in a Spring Boot application.

## Simple GET Request

To make a GET HTTP request, you can use either getForObject() or getForEntity() method. Here is an example that uses the getForObject() method to fetch the user information as a [JSON](https://attacomsian.com/blog/what-is-json) string:

// request url

String url = "https://jsonplaceholder.typicode.com/posts/1";

// create an instance of RestTemplate

RestTemplate restTemplate = new RestTemplate();

// make an HTTP GET request

String json = restTemplate.getForObject(url, String.class);

// print json

System.out.println(json);

## GET Request with Request Parameters

If you want to pass query parameters, you can either append them directly to URL or use placeholders. Here is an example of GET request made with **query parameters appended** to the URL:

// request url

String url = "https://google.com/search?q=java";

// create an instance of RestTemplate

RestTemplate restTemplate = new RestTemplate();

// make an HTTP GET request

String html = restTemplate.getForObject(url, String.class);

Similarly, you can add **placeholders** to the URL for query parameters:

// request url

String url = "https://google.com/search?q={q}";

// create an instance of RestTemplate

RestTemplate restTemplate = new RestTemplate();

// make an HTTP GET request

String html = restTemplate.getForObject(url, String.class, "java");

## GET Request with Parameters and Headers

To add custom request headers to HTTP GET request, you should use the generic exchange() method provided by the RestTemplate class.

The following GET request is made with **query parameters and request headers**:

// request url

String url = "https://jsonplaceholder.typicode.com/posts/{id}";

// create an instance of RestTemplate

RestTemplate restTemplate = new RestTemplate();

// create headers

HttpHeaders headers = new HttpHeaders();

// set `Content-Type` and `Accept` headers

headers.setContentType(MediaType.APPLICATION\_JSON);

headers.setAccept(Collections.singletonList(MediaType.APPLICATION\_JSON));

// example of custom header

headers.set("X-Request-Source", "Desktop");

// build the request

HttpEntity request = new HttpEntity(headers);

// make an HTTP GET request with headers

ResponseEntity<String> response = restTemplate.exchange(

url,

HttpMethod.GET,

request,

String.class,

1

);

// check response

if (response.getStatusCode() == HttpStatus.OK) {

System.out.println("Request Successful.");

System.out.println(response.getBody());

} else {

System.out.println("Request Failed");

System.out.println(response.getStatusCode());

}

## GET Request with Basic Authentication

The following example demonstrates how to add [basic authentication to RestTemplate](https://attacomsian.com/blog/resttemplate-basic-authentication) GET request:

// request url

String url = "https://jsonplaceholder.typicode.com/posts";

// create an instance of RestTemplate

RestTemplate restTemplate = new RestTemplate();

// create headers

HttpHeaders headers = new HttpHeaders();

// add basic authentication header

headers.setBasicAuth("username", "password");

// build the request

HttpEntity request = new HttpEntity(headers);

// make an HTTP GET request with headers

ResponseEntity<String> response = restTemplate.exchange(

url,

HttpMethod.GET,

request,

String.class

);

// check response

if (response.getStatusCode() == HttpStatus.OK) {

System.out.println("Request Successful.");

System.out.println(response.getBody());

} else {

System.out.println("Request Failed");

System.out.println(response.getStatusCode());

}

## GET Request with Response Mapped to Java Object

Using RestTemplate, you can also **map the JSON response directly to a Java object**. Let us first create a simple model class:

**Post.java**

public class Post implements Serializable {

private int userId;

private int id;

private String title;

private String body;

public Post() {

}

public Post(int userId, int id, String title, String body) {

this.userId = userId;

this.id = id;

this.title = title;

this.body = body;

}

// getters and setters, equals(), toString() .... (omitted for brevity)

}

Now we can simply use the Post class as response type in the exchange() method:

// request url

String url = "https://jsonplaceholder.typicode.com/posts/1";

// create an instance of RestTemplate

RestTemplate restTemplate = new RestTemplate();

// create headers

HttpHeaders headers = new HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_JSON);

headers.setAccept(Collections.singletonList(MediaType.APPLICATION\_JSON));

// build the request

HttpEntity request = new HttpEntity(headers);

// make an HTTP GET request with headers

ResponseEntity<Post> response = restTemplate.exchange(

url,

HttpMethod.GET,

request,

Post.class

);

// check response

if (response.getStatusCode() == HttpStatus.OK) {

System.out.println("Request Successful.");

System.out.println(response.getBody());

} else {

System.out.println("Request Failed");

System.out.println(response.getStatusCode());

}

Check out the [Making HTTP Requests using RestTemplate in Spring Boot](https://attacomsian.com/blog/http-requests-resttemplate-spring-boot) guide for more RestTemplate examples.

# RestTemplate POST Request with JSON and Headers

November 09, 2019 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

In this article, you will learn how to make different kinds of HTTP POST requests by using the [RestTemplate](https://attacomsian.com/blog/http-requests-resttemplate-spring-boot) class in a Spring Boot application.

An HTTP POST request is used to create a new resource. The RestTemplate class provides several template methods like postForObject(), postForEntity(), and postForLocation() for making POST requests. The first two methods are very similar to what I have discussed before in [RestTemplate's GET request](https://attacomsian.com/blog/spring-boot-resttemplate-get-request-parameters-headers) tutorial. The last method returns the location of the newly created resource instead of returning the full resource.

## Simple POST Request

To make a simple HTTP POST request using RestTemplate, you can use the postForEntity method and pass the request body parameters as a map object:

// request url

String url = "https://reqres.in/api/users";

// create an instance of RestTemplate

RestTemplate restTemplate = new RestTemplate();

// request body parameters

Map<String, String> map = new HashMap<>();

map.put("name", "John Doe");

map.put("job", "Java Developer");

// send POST request

ResponseEntity<Void> response = restTemplate.postForEntity(url, map, Void.class);

// check response

if (response.getStatusCode() == HttpStatus.OK) {

System.out.println("Request Successful");

} else {

System.out.println("Request Failed");

}

## POST Request with JSON and Headers

To make a POST request with the JSON request body, we need to set the Content-Type request header to application/json. The following example demonstrates how to make an HTTP POST request with JSON request body:

// request url

String url = "https://jsonplaceholder.typicode.com/posts";

// create an instance of RestTemplate

RestTemplate restTemplate = new RestTemplate();

// create headers

HttpHeaders headers = new HttpHeaders();

// set `content-type` header

headers.setContentType(MediaType.APPLICATION\_JSON);

// set `accept` header

headers.setAccept(Collections.singletonList(MediaType.APPLICATION\_JSON));

// request body parameters

Map<String, Object> map = new HashMap<>();

map.put("userId", 1);

map.put("title", "Spring Boot 101");

map.put("body", "A powerful tool for building web apps.");

// build the request

HttpEntity<Map<String, Object>> entity = new HttpEntity<>(map, headers);

// send POST request

ResponseEntity<String> response = restTemplate.postForEntity(url, entity, String.class);

// check response

if (response.getStatusCode() == HttpStatus.CREATED) {

System.out.println("Request Successful");

System.out.println(response.getBody());

} else {

System.out.println("Request Failed");

System.out.println(response.getStatusCode());

}

## POST Request with Basic Authentication

The following example demonstrates how to [add basic authentication to RestTemplate](https://attacomsian.com/blog/resttemplate-basic-authentication) POST request:

// request url

String url = "https://reqres.in/api/login";

// create an instance of RestTemplate

RestTemplate restTemplate = new RestTemplate();

// create headers

HttpHeaders headers = new HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_JSON);

headers.setAccept(Collections.singletonList(MediaType.APPLICATION\_JSON));

// add basic authentication

headers.setBasicAuth("username", "password");

// build the request

HttpEntity request = new HttpEntity(headers);

// send POST request

ResponseEntity<String> response = restTemplate.postForEntity(url, request, String.class);

// check response

if (response.getStatusCode() == HttpStatus.OK) {

System.out.println("Login Successful");

} else {

System.out.println("Login Failed");

}

## POST Request with Response Mapped to Java Object

RestTemplate allows you to map the JSON response directly to a Java object. Let us first create a simple Post class:

public class Post implements Serializable {

private int userId;

private int id;

private String title;

private String body;

public Post() {

}

public Post(int userId, int id, String title, String body) {

this.userId = userId;

this.id = id;

this.title = title;

this.body = body;

}

// getters and setters, equals(), toString() .... (omitted for brevity)

}

We can now use the above class to map the JSON response, as shown below:

// request url

String url = "https://jsonplaceholder.typicode.com/posts";

// create an instance of RestTemplate

RestTemplate restTemplate = new RestTemplate();

// create headers

HttpHeaders headers = new HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_JSON);

headers.setAccept(Collections.singletonList(MediaType.APPLICATION\_JSON));

// create a post object

Post post = new Post(101, 1, "Spring Boot 101",

"A powerful tool for building web apps.");

// build the request

HttpEntity<Post> request = new HttpEntity<>(post, headers);

// send POST request

ResponseEntity<Post> response = restTemplate.postForEntity(url, request, Post.class);

// check response

if (response.getStatusCode() == HttpStatus.CREATED) {

System.out.println("Post Created");

System.out.println(response.getBody());

} else {

System.out.println("Request Failed");

System.out.println(response.getStatusCode());

}

Check out the [Making HTTP Requests using RestTemplate in Spring Boot](https://attacomsian.com/blog/http-requests-resttemplate-spring-boot) guide for more RestTemplate examples.

# Making HTTP Requests using RestTemplate in Spring Boot

July 25, 2019 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

The RestTemplate class in Spring Framework is a synchronous HTTP client for making HTTP requests to consume RESTful web services. It exposes a simple and easy-to-use template method API for sending an HTTP request and also handling the HTTP response. The RestTemplate class also provides aliases for all supported HTTP request methods, such as GET, POST, PUT, DELETE, and OPTIONS.

In this tutorial, we will learn how to use the Spring REST client — **RestTemplate** — for sending HTTP requests in a Spring Boot application. For all our examples, we will use [JSONPlaceholder](https://jsonplaceholder.typicode.com/" \t "_blank" \o "Open in new window) fake REST API to mimic real application scenarios.

## Dependencies

Since the RestTemplate class is a part of the Spring Web project, we only need spring-boot-starter-web dependency. Add the following dependency to your Gradle project's build.gradle file:

implementation 'org.springframework.boot:spring-boot-starter-web'

If you are using Maven, add the following dependency to your pom.xml file:

<dependency>

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

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

</dependency>

## GET Request

Let's start with a simple example to retrieve a list of posts using RestTemplate's getForObject() method:

**RestService.java**

@Service

public class RestService {

private final RestTemplate restTemplate;

public RestService(RestTemplateBuilder restTemplateBuilder) {

this.restTemplate = restTemplateBuilder.build();

}

public String getPostsPlainJSON() {

String url = "https://jsonplaceholder.typicode.com/posts";

return this.restTemplate.getForObject(url, String.class);

}

}

Notice the response returned by the getForObject() method. It is a plain [JSON](https://attacomsian.com/blog/what-is-json) string. We can easily [parse this JSON string into an object](https://attacomsian.com/blog/processing-json-spring-boot) using Jackson.

### Get Response as Object

We can also map the response directly to a model class. Let us first create a model class:

**Post.java**

public class Post implements Serializable {

private int userId;

private int id;

private String title;

private String body;

// getters and setters

}

Now we can simply use the Post class as response type in getForObject() method:

public Post[] getPostsAsObject() {

String url = "https://jsonplaceholder.typicode.com/posts";

return this.restTemplate.getForObject(url, Post[].class);

}

### URL Parameters

If you want to pass the query parameters, just append them to URL:

String url = "https://jsonplaceholder.typicode.com/posts?userId=2";

Another way is to use placeholders in the URL for parameters:

public Post getPostWithUrlParameters() {

String url = "https://jsonplaceholder.typicode.com/posts/{id}";

return this.restTemplate.getForObject(url, Post.class, 1);

}

### Response Handing

If you want to manipulate the response (like checking HTTP status code), use getForEntity() method instead like below:

public Post getPostWithResponseHandling() {

String url = "https://jsonplaceholder.typicode.com/posts/{id}";

ResponseEntity<Post> response = this.restTemplate.getForEntity(url, Post.class, 1);

if(response.getStatusCode() == HttpStatus.OK) {

return response.getBody();

} else {

return null;

}

}

### Custom Request Headers

If you want to set the request headers like content-type, accept, or any custom header, use generic exchange() method:

public Post getPostWithCustomHeaders() {

String url = "https://jsonplaceholder.typicode.com/posts/{id}";

// create headers

HttpHeaders headers = new HttpHeaders();

// set `accept` header

headers.setAccept(Collections.singletonList(MediaType.APPLICATION\_JSON));

// set custom header

headers.set("x-request-source", "desktop");

// build the request

HttpEntity request = new HttpEntity(headers);

// use `exchange` method for HTTP call

ResponseEntity<Post> response = this.restTemplate.exchange(url, HttpMethod.GET, request, Post.class, 1);

if(response.getStatusCode() == HttpStatus.OK) {

return response.getBody();

} else {

return null;

}

}

**Quick Guide:** Check out [RestTemplate GET Request with Parameters and Headers](https://attacomsian.com/blog/spring-boot-resttemplate-get-request-parameters-headers) for more GET request examples.

## POST Request

A POST request is used to create a new resource. The RestTemplate class offers several template methods like postForObject(), postForEntity(), and postForLocation() for making POST request. The first two methods are quite similar to what we have discussed above in terms of response format. The last method returns the location of the newly created resource instead of returning the full resource.

Let us make use of the postForEntity() method to create a new post:

public Post createPost() {

String url = "https://jsonplaceholder.typicode.com/posts";

// create headers

HttpHeaders headers = new HttpHeaders();

// set `content-type` header

headers.setContentType(MediaType.APPLICATION\_JSON);

// set `accept` header

headers.setAccept(Collections.singletonList(MediaType.APPLICATION\_JSON));

// create a map for post parameters

Map<String, Object> map = new HashMap<>();

map.put("userId", 1);

map.put("title", "Introduction to Spring Boot");

map.put("body", "Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications.");

// build the request

HttpEntity<Map<String, Object>> entity = new HttpEntity<>(map, headers);

// send POST request

ResponseEntity<Post> response = this.restTemplate.postForEntity(url, entity, Post.class);

// check response status code

if (response.getStatusCode() == HttpStatus.CREATED) {

return response.getBody();

} else {

return null;

}

}

Alternatively, we can also use postForObject() method to create a new post:

public Post createPostWithObject() {

String url = "https://jsonplaceholder.typicode.com/posts";

// create headers

HttpHeaders headers = new HttpHeaders();

// set `content-type` header

headers.setContentType(MediaType.APPLICATION\_JSON);

// set `accept` header

headers.setAccept(Collections.singletonList(MediaType.APPLICATION\_JSON));

// create a post object

Post post = new Post(1, "Introduction to Spring Boot",

"Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications.");

// build the request

HttpEntity<Post> entity = new HttpEntity<>(post, headers);

// send POST request

return restTemplate.postForObject(url, entity, Post.class);

}

**Quick Guide:** Check out [RestTemplate POST Request with JSON and Headers](https://attacomsian.com/blog/spring-boot-resttemplate-post-request-json-headers) for more POST request examples.

## PUT Request

The RestTemplate class provides put() method that can be used to update a resource:

public void updatePost() {

String url = "https://jsonplaceholder.typicode.com/posts/{id}";

// create headers

HttpHeaders headers = new HttpHeaders();

// set `content-type` header

headers.setContentType(MediaType.APPLICATION\_JSON);

// set `accept` header

headers.setAccept(Collections.singletonList(MediaType.APPLICATION\_JSON));

// create a post object

Post post = new Post(4, "New Title", "New Body");

// build the request

HttpEntity<Post> entity = new HttpEntity<>(post, headers);

// send PUT request to update post with `id` 10

this.restTemplate.put(url, entity, 10);

}

The put() method does not return anything. If you want to process the response, use generic exchange() method instead:

public Post updatePostWithResponse() {

String url = "https://jsonplaceholder.typicode.com/posts/{id}";

// create headers

HttpHeaders headers = new HttpHeaders();

// set `content-type` header

headers.setContentType(MediaType.APPLICATION\_JSON);

// set `accept` header

headers.setAccept(Collections.singletonList(MediaType.APPLICATION\_JSON));

// create a post object

Post post = new Post(4, "New Title", "New Body");

// build the request

HttpEntity<Post> entity = new HttpEntity<>(post, headers);

// send PUT request to update post with `id` 10

ResponseEntity<Post> response = this.restTemplate.exchange(url, HttpMethod.PUT, entity, Post.class, 10);

// check response status code

if (response.getStatusCode() == HttpStatus.OK) {

return response.getBody();

} else {

return null;

}

}

## DELETE Request

To delete an existing resource, you can use delete() method:

public void deletePost() {

String url = "https://jsonplaceholder.typicode.com/posts/{id}";

// send DELETE request to delete post with `id` 10

this.restTemplate.delete(url, 10);

}

## HEAD Request

The RestTemplate class offers headForHeaders() method to retrieve headers:

public HttpHeaders retrieveHeaders() {

String url = "https://jsonplaceholder.typicode.com/posts";

// send HEAD request

return this.restTemplate.headForHeaders(url);

}

## OPTIONS Request

Let us use optionsForAllow() method to get a list of all supported HTTP operations:

public Set<HttpMethod> allowedOperations() {

String url = "https://jsonplaceholder.typicode.com/posts";

// send HEAD request

return this.restTemplate.optionsForAllow(url);

}

## Error Handling

If there is an error during the execution of the request or the server returns a non-successful HTTP error (4xx or 5xx), RestTemplate will throw an exception. You can catch the HttpStatusCodeException in catch block to get the response body and headers:

public String unknownRequest() {

try {

String url = "https://jsonplaceholder.typicode.com/404";

return this.restTemplate.getForObject(url, String.class);

} catch (HttpStatusCodeException ex) {

// raw http status code e.g `404`

System.out.println(ex.getRawStatusCode());

// http status code e.g. `404 NOT\_FOUND`

System.out.println(ex.getStatusCode().toString());

// get response body

System.out.println(ex.getResponseBodyAsString());

// get http headers

HttpHeaders headers= ex.getResponseHeaders();

System.out.println(headers.get("Content-Type"));

System.out.println(headers.get("Server"));

}

return null;

}

**Quick Guide:** Learn more about [handling errors while using the RestTemplate](https://attacomsian.com/blog/spring-boot-resttemplate-error-handling) in a Spring Boot application.

## RestTemplate Connection Timeout

There are two types of timeouts: connection timeout and read time out. By default, RestTemplate has infinite timeouts. But we can change this behavior by using RestTemplateBuilder class for setting the connection and read timeouts:

public RestService(RestTemplateBuilder restTemplateBuilder) {

// set connection and read timeouts

this.restTemplate = restTemplateBuilder

.setConnectTimeout(Duration.ofSeconds(500))

.setReadTimeout(Duration.ofSeconds(500))

.build();

}

**Source code:** Download the complete source code from [GitHub](https://github.com/attacomsian/code-examples/tree/master/spring-boot/rest-template) available under MIT license.

## Conclusion

That's all folks for using Spring Framework's RestTemplate class to call remote RESTful web services in a Spring Boot application. We talked about almost all important HTTP verbs and used RestTemplate to make requests for all of them.

If you are interested in learning more, check out [processing JSON data in Spring Boot](https://attacomsian.com/blog/processing-json-spring-boot) guide. It will introduce you to Jackson that is frequently used with RestTemplate for parsing unknown JSON data.

# RestTemplate Basic Authentication Example

September 21, 2019 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

In an [earlier article](https://attacomsian.com/blog/http-requests-resttemplate-spring-boot), I talked about how to make HTTP requests to consume RESTful web services by using the Spring Framework RestTemplate class.

In this short article, you will learn how to **add basic authentication to the requests** made by RestTemplate in a Spring Boot application.

In basic HTTP authentication, the outgoing HTTP request contains an authorization header in the following form:

Authorization: Basic <credentials>

Where credentials is a [base64 encoded string](https://attacomsian.com/blog/base64-encode-decode-java) that is created by combing both user name and password with a colon (:).

There are multiple ways to add this authorization HTTP header to a RestTemplate request.

## Add Basic Authentication to a Single Request

The simplest way to add basic authentication to a request is to create an instance of HttpHeaders, set the Authorization header value, and then pass it to the RestTemplate. Here is an example:

try {

// request url

String url = "https://jsonplaceholder.typicode.com/posts";

// create auth credentials

String authStr = "username:password";

String base64Creds = Base64.getEncoder().encodeToString(authStr.getBytes());

// create headers

HttpHeaders headers = new HttpHeaders();

headers.add("Authorization", "Basic " + base64Creds);

// create request

HttpEntity request = new HttpEntity(headers);

// make a request

ResponseEntity<String> response = new RestTemplate().exchange(url, HttpMethod.GET, request, String.class);

// get JSON response

String json = response.getBody();

} catch (Exception ex) {

ex.printStackTrace();

}

In the latest version of Spring Framework (5.1 and higher), it is no longer required to manually set the authorization header. You can use the setBasicAuth() method from HttpHeaders to pass the login credentials:

// create headers

HttpHeaders headers = new HttpHeaders();

headers.setBasicAuth("username", "password");

The setBasicAuth() method will automatically create the base64 encoded string and set the authorization header.

## Add Basic Authentication to All Requests

Sometimes you want to add basic HTTP authentication to all requests to consume secure RESTful web services. It is not a good approach to manually set the authorization header for each request.

Fortunately, Spring Boot provides RestTemplateBuilder class to configure and create an instance of RestTemplate. It includes a number of convenience methods that can be used to create a customized RestTemplate instance.

To use the RestTemplateBuilder, simply inject it to the class where you want to use the RestTemplate HTTP client:

@Service

public class RestService {

private final RestTemplate restTemplate;

public RestService(RestTemplateBuilder restTemplateBuilder) {

this.restTemplate = restTemplateBuilder

.basicAuthentication("username", "password")

.build();

}

}

Now, the basic authentication will be added to all requests sent through above restTemaplate instance. You do not need to set the authorization header.

Instead of autowiring the RestTemplateBuilder, you can set the following bean in your Spring Boot main application class:

@Bean

RestOperations restTemplateBuilder(RestTemplateBuilder restTemplateBuilder) {

return restTemplateBuilder.basicAuthentication("username", "password").build();

}

It will make sure that the basic authentication is added to each and every request that is made by the RestTemplate HTTP client. This solution is not recommended if you are calling different APIs as it would add an authorization header to unwanted requests.

**Read Next:** [Spring Boot RestTemplate Error Handling](https://attacomsian.com/blog/spring-boot-resttemplate-error-handling)

# Spring Boot RestTemplate Error Handling

November 08, 2019 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

In [an earlier article](https://attacomsian.com/blog/http-requests-resttemplate-spring-boot), I wrote about making HTTP requests using the RestTemplate class in a Spring Boot application.

In this quick article, you'll learn **how to handle the errors thrown by the RestTemplate** during the execution of an HTTP request.

## Default Error Handling

By default, if there is an error during the execution of the request or the server returns a non-successful HTTP status code (4xx or 5xx), RestTemplate will throw one of the following exceptions:

* HttpClientErrorException — For HTTP status code 4xx
* HttpServerErrorException — For HTTP status code 5xx
* UnknownHttpStatusCodeException — In case of an unknown HTTP status code

All these exceptions extend a common base class called RestClientResponseException that contains actual HTTP response data.

## Error Handling using a Try-Catch Block

The simplest way to add a custom error handler is to use a try-catch block to catch the HttpStatusCodeException exception. From the HttpStatusCodeException instance, you can then get the response status code, body, and headers, as shown below:

try {

// request url

String url = "https://reqres.in/api/unknown/23";

// create an instance of RestTemplate

RestTemplate restTemplate = new RestTemplate();

// make an HTTP GET request

ResponseEntity<String> response = restTemplate.getForEntity(url, String.class);

} catch (HttpStatusCodeException ex) {

// raw http status code e.g `404`

System.out.println(ex.getRawStatusCode());

// http status code e.g. `404 NOT\_FOUND`

System.out.println(ex.getStatusCode().toString());

// get response body

System.out.println(ex.getResponseBodyAsString());

// get http headers

HttpHeaders headers = ex.getResponseHeaders();

System.out.println(headers.get("Content-Type"));

System.out.println(headers.get("Server"));

}

## Implementing a Custom Error Handler

Sometimes, a try-catch block is not enough to handle errors as it is not scalable when the number of HTTP request increases.

You may want to create a reusable custom error handler by implementing the ResponseErrorHandler interface, as follows:

**MyErrorHandler.java**

package com.attacomsian.runner;

import org.springframework.http.HttpHeaders;

import org.springframework.http.HttpStatus;

import org.springframework.http.client.ClientHttpResponse;

import org.springframework.web.client.DefaultResponseErrorHandler;

import org.springframework.web.client.ResponseErrorHandler;

import java.io.IOException;

public class MyErrorHandler implements ResponseErrorHandler {

@Override

public boolean hasError(ClientHttpResponse response) throws IOException {

return new DefaultResponseErrorHandler().hasError(response);

}

@Override

public void handleError(ClientHttpResponse response) throws IOException {

if (response.getStatusCode().series() == HttpStatus.Series.SERVER\_ERROR) {

// handle 5xx errors

// raw http status code e.g `500`

System.out.println(response.getRawStatusCode());

// http status code e.g. `500 INTERNAL\_SERVER\_ERROR`

System.out.println(response.getStatusCode());

} else if (response.getStatusCode().series() == HttpStatus.Series.CLIENT\_ERROR) {

// handle 4xx errors

// raw http status code e.g `404`

System.out.println(response.getRawStatusCode());

// http status code e.g. `404 NOT\_FOUND`

System.out.println(response.getStatusCode());

// get response body

System.out.println(response.getBody());

// get http headers

HttpHeaders headers = response.getHeaders();

System.out.println(headers.get("Content-Type"));

System.out.println(headers.get("Server"));

}

}

}

You can now create an instance of MyErrorHandler and pass it to the RestTemplate class:

// request url

String url = "https://reqres.in/api/unknown/23";

// create an instance of RestTemplate

RestTemplate restTemplate = new RestTemplate();

// set custom error handler

restTemplate.setErrorHandler(new MyErrorHandler());

// make an HTTP GET request

ResponseEntity<String> response = restTemplate.getForEntity(url, String.class);

Check out the [Making HTTP Requests using RestTemplate in Spring Boot](https://attacomsian.com/blog/http-requests-resttemplate-spring-boot) guide for more RestTemplate examples.

# Getting Started with Spring Data JPA

October 01, 2019 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

[Spring Data](https://spring.io/projects/spring-data) is a popular project which provides an easy-to-use and consistent, Spring-based programming model for storing and retrieving data from a data store. This allows you to connect with different data access technologies, relational and non-relational databases, map-reduce frameworks, and cloud-based data services.

[Spring Data JPA](https://spring.io/projects/spring-data-jpa) (Java Persistence API) is an extension of Spring Data that makes it easy to connect with relational databases to implement JPA based repositories.

## What is Java Persistence API?

[Java Persistence API](https://jcp.org/en/jsr/detail?id=338) is a specification that defines an object-relational mapping (ORM) standard for storing, accessing, and managing Java objects in a relational database.

While originally intended for use with relational/SQL databases only, JPA's ORM model has been since extended for use with NoSQL data stores as well. At the moment, two most popular implementations of JPA's specification are [Hibernate](https://hibernate.org/) and [EclipseLink](https://www.eclipse.org/eclipselink" \t "_blank" \o "Open in new window).

Spring Data JPA is **not a JPA provider** but just **an extra layer of abstraction on top** of an existing JPA provider such as Hibernate. This means that it uses all features defined by the JPA specification such as the entity and association mappings, the entity lifecycle management, and JPA's query capabilities.

On top of this, Spring Data JPA defines its own cool features such as no-code repositories and the ability to generate queries based on method names. Thus, eliminating the need for writing too much boilerplate code for executing simple queries.

## Why Spring Data JPA?

Although we can use any JPA implementation like Hibernate or EclipseLink directly in our project, using Spring Data JPA gives us additional benefits. This significantly reduces the boilerplate code and makes the overall development much faster.

The extra layer on top of the JPA specification also allows us to build Spring-powered applications that use JPA for data access layers.

Here is a list of features that makes the Spring Data JPA a go-to choice:

### 1. Repository Abstraction

The real strength of Spring Data JPA lies in the repository abstraction provided by the [Spring Data Commons](https://github.com/spring-projects/spring-data-commons) project. It hides the data store specific implementation details and allows you to write your business logic code on a higher abstraction level.

You only need to **learn how to use Spring Data repository interfaces** without worrying about the underlying implementation of the repository abstraction.

**Spring Data Commons** provides the following repository interfaces:

* Repository — Central repository marker interface. Captures the domain type to manage as well as the domain type's id type.
* CrudRepository — Interface for generic CRUD operations on a repository for a specific type.
* PagingAndSortingRepository — Extension of CrudRepository to provide additional methods to retrieve entities using the pagination and sorting abstraction.
* QuerydslPredicateExecutor — Interface to allow execution of [QueryDSL](http://www.querydsl.com/" \t "_blank" \o "Open in new window) Predicate instances. It is not a repository interface.

**Spring Data JPA** provides the following additional repository interfaces:

* JpaRepository — JPA specific extension of Repository interface. It combines all methods declared by the Spring Data Commons repository interfaces behind a single interface.
* JpaSpecificationExecutor — It is not a repository interface. It allows the execution of Specifications based on the JPA criteria API.

I'll discuss how to use these repositories in more detail in the next section as well as in the coming weeks. Here is a quick example of a repository that extends the CrudRepository interface:

public interface UserRepository extends CrudRepository<User, Long> {}

Things you can do by using the above UserRepository interface:

* Persist, update and remove one or multiple User entities.
* Find one or more users by their primary keys.
* Count, get and remove all users.
* Check if a User exists with a given primary key.

You might be wondering how it is possible when we didn't write any such method. The answer is the CrudRepository interface that defines all these methods, I mentioned before and makes them available for you.

### 2. Less Boilerplate Code

The best thing about Spring Data JPA is the **default implementation of each method** defined in its repository interfaces. This means that you do not need to write a lot of boilerplate code for CRUD methods. You only need to extend the JPA specific repository interface — JpaRepository and that's it. Spring Data JPA will ensure that CRUD methods are available for standard data access.

### 3. Auto-Generated Queries

Another cool feature of Spring Data JPA is the **auto-generation of database queries** based on method names. When you implement a Spring Data JPA repository interface, it analyses all the methods defined by the entity class and tries to generate the queries automatically from the method names.

Auto-generated queries may not be well-suited for complex use cases. But for simple scenarios, these queries are very useful. You just define a method on your repository interface with a name that starts with find...By. Spring then parses the method name and creates a query for it automatically.

Following is an example of a query that loads a User entity with a given name. Behind the scene, Spring generates a JPQL query based on the method name, sets the provided method parameters as bind parameter values, executes the query and returns the result:

public interface UserRepository extends CrudRepository<User, Long> {

User findByName(String name);

}

## Using Spring Data JPA with Spring Boot

As we have discussed above, Spring Data JPA makes the implementation of your data access layer much easier by reducing the boilerplate code. So that you easily build a Spring-based application using any data access technology.

In this section, I'll show you **how to add and configure Spring Data JPA in a Spring Boot application** using Hibernate as a persistence provider.

Follow the below steps to add Spring Data JPA support to your Spring Boot project.

### Step 1 — Add Dependencies

You only need spring-boot-start-data-jpa dependency to enable JPA support in a Spring Boot application. Add the following dependency to your build.gradle file:

implementation 'org.springframework.boot:spring-boot-starter-data-jpa'

If you are using Maven, add the following dependencies to your pom.xml file:

<dependency>

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

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

</dependency>

Spring Boot Starter Data JPA includes everything — all required dependencies and activates the default configuration. To start a Spring Boot application from scratch, use the [Spring Initializr](https://start.spring.io/) tool to easily bootstrap your application with required dependencies.

### Step 2 — Configure Data Source

Spring Boot automatically **configures Hibernate as a default JPA provider**. So it's no longer required to explicitly configure the EntityManagerFactory bean unless we want to customize it.

Similarly, if you are using an in-memory database such as H2, HSQLDB, or Apache Derby, Spring Boot will auto-configure the DataSource bean for you as well. You only need to make sure that the corresponding database dependency is available in the classpath.

#### H2 Configuration

Just add the following H2 database dependency to your Gradle project's build.gradle file:

runtimeOnly 'com.h2database:h2'

For Maven, add the following dependency to your pom.xml file:

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

That's it. Spring Boot will automatically configure the DataSource bean for you when it will detect the H2 database dependency in the classpath.

#### MySQL Configuration

If you want to use the MySQL database with Spring Data JPA, then you need to do a bit extra work. First of all, add the following MySQL driver dependency to your build.gradle file:

runtimeOnly 'mysql:mysql-connector-java'

For Maven project, include the following dependency to your pom.xml file:

<dependency>

<groupId>mysql</groupId>

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

<scope>runtime</scope>

</dependency>

Next, you need to define the DataSource configuration. There are two ways to do this. You can either define it as a bean in a Java class with @Configuration annotation or use the Spring Boot properties file to define data source properties.

Here is how the Java-based data source configuration looks like in a Spring Boot project:

@Bean

public DataSource dataSource() {

DriverManagerDataSource dataSource = new DriverManagerDataSource();

dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");

dataSource.setUsername("root");

dataSource.setPassword("rootpass");

dataSource.setUrl("jdbc:mysql://localhost:3306/testdb?createDatabaseIfNotExist=true&useSSL=false");

return dataSource;

}

To configure the data source using a properties file, you need to define the following properties in your application.properties or application.yml file:

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.username=root

spring.datasource.password=rootpass

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

Spring Boot will automatically configure a DataSource bean based on the above properties.

### Step 3 — Define an Entity

Let us define a simple entity class for storing User objects:

**User.java**

package com.attacomsian.jpa.domains;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private Long id;

private String name;

private String email;

public User() {

}

public User(String name, String email) {

this.name = name;

this.email = email;

}

// getters and setters removed for brevity

}

The above User class has three attributes (id, name, and email) and two constructors. The no-argument constructor is only required for the JPA. The other constructor is the one you should use to create instances of User to be saved to the database.

The User class is annotated with @Entity, indicating that it is a JPA entity. Since no @Table annotation is provided, it is assumed that this entity is mapped to a table named User.

The id property is annotated with @Id so that JPA recognizes it as the object’s ID. The id property is also annotated with @GeneratedValue to indicate that the ID should be generated automatically.

The other two properties, name and email, are left unannotated. It means that they are mapped to columns that have the same names as the properties themselves.

### Step 4 — Create a Repository

The next step is to create a repository interface for the above User entity:

**UserRepository.java**

package com.attacomsian.jpa.repositories;

import com.attacomsian.jpa.domains.User;

import org.springframework.data.repository.CrudRepository;

import java.util.List;

public interface UserRepository extends CrudRepository<User, Long> {

List<User> findByName(String name);

User findByEmail(String email);

}

UserRepository extends the CrudRepository interface provided by the Spring Data Commons project. The type of entity and ID that it works with, User and Long, are specified in the generic parameters on CrudRepository. By extending CrudRepository, UserRepository inherits several methods for saving, deleting, and finding User entities.

Spring Data JPA also allows you to define other query methods by declaring their method signature. For example, UserRepository declares two additional methods: findByName() and findByEmail().

In a typical Java application, you have to write a class that implements UserRepository interface methods. However, it is no longer required with Spring Data JPA. It will create the repository implementation automatically, at runtime, from the repository interface. That is what makes Spring Data JPA so much powerful.

### Step 4 — Create an Application Class

Let us create the Spring Boot main application class to play with the above UserRepository interface. Since we don't need the web part of Spring Boot, we will implement the CommandLineRunner interface to make it a [console application](https://attacomsian.com/blog/spring-boot-console-application).

**Application.java**

package com.attacomsian.jpa;

import com.attacomsian.jpa.domains.User;

import com.attacomsian.jpa.repositories.UserRepository;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.Bean;

@SpringBootApplication

public class Application {

private static final Logger log = LoggerFactory.getLogger(Application.class);

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

@Bean

public CommandLineRunner userDemo(UserRepository userRepository) {

return (args) -> {

// create users

userRepository.save(new User("John Doe", "john.doe@example.com"));

userRepository.save(new User("Emma Watson", "emma.watson@example.com"));

userRepository.save(new User("Seno Reta", "seno.reta@example.com"));

userRepository.save(new User("Mike Hassan", "mike.hassan@example.com"));

// fetch all users

log.info("Users found with findAll():");

log.info("---------------------------");

for (User user : userRepository.findAll()) {

log.info(user.toString());

}

log.info("");

// fetch user by id

User user = userRepository.findById(1L).get();

log.info("User found with findById(1L):");

log.info("-----------------------------");

log.info(user.toString());

log.info("");

// fetch user by email address

User userWithEmail = userRepository.findByEmail("john.doe@example.com");

log.info("User found with findByEmail('john.doe@example.com'):");

log.info("----------------------------------------------------");

log.info(userWithEmail.toString());

log.info("");

// delete all users

userRepository.deleteAll();

// confirm users deletion

log.info("Total users after deletion with :");

log.info("--------------------------");

log.info(userRepository.count() + " users are in DB");

log.info("");

};

}

}

The above class is pretty much self-explanatory. We are using the @SpringBootApplication annotation on our main class to activate the auto-configuration. The main() method uses Spring Boot’s SpringApplication.run() method to launch an application.

We have also defined a CommandLineRunner bean method that gets UserRepository on runtime through dependency injection. Inside this method, we first create and save some users via the save() method. Next, we call findAll() to fetch all User objects from the database. Then it calls findById() and findByEmail() methods to get a single User by its ID and email address respectively. Finally, it calls deleteAll() to remove all users from the database. To verify that all users are successfully deleted, we call the count() method to get the number of entities.

**Note:** By default, Spring Boot enables JPA repository support and looks in the package (and its sub-packages) where the class with @SpringBootApplication annotation is located. If your application doesn't follow the default project structure, you need to configure your repositories package using the @EnableJpaRepositories annotation.

### Step 5 — Run the Application

You can run the application from the command line through Gradle or Maven. If you use Gradle, you can run the application by typing:

$ ./gradlew bootRun

If you use Maven, you can run the application by running the following command:

$ ./mvnw spring-boot:run

Once the application is started, you should see the following output:

Users found with findAll():

---------------------------

User{id=1, name='John Doe', email='john.doe@example.com'}

User{id=2, name='Emma Watson', email='emma.watson@example.com'}

User{id=3, name='Seno Reta', email='seno.reta@example.com'}

User{id=4, name='Mike Hassan', email='mike.hassan@example.com'}

User found with findById(1L):

-----------------------------

User{id=1, name='John Doe', email='john.doe@example.com'}

User found with findByEmail('john.doe@example.com'):

----------------------------------------------------

User{id=1, name='John Doe', email='john.doe@example.com'}

Total users after deletion with :

--------------------------

0 users are in DB

# Dynamic Queries with Spring Data JPA Specifications

October 20, 2019 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

[Spring Data JPA](https://attacomsian.com/blog/getting-started-spring-data-jpa) allows us to connect with different relational databases through a common interface to create JPA (Java Persistence API) based repositories. We can easily create database queries by using [method names](https://attacomsian.com/blog/derived-query-methods-spring-data-jpa), the [@Query annotation](https://attacomsian.com/blog/spring-data-jpa-query-annotation), and [named queries](https://attacomsian.com/blog/spring-data-jpa-named-queries) to retrieve data from the database.

The Spring Data repository abstraction is a powerful feature that hides the store-specific implementation details and allows us to write business logic code at a higher abstraction level. All we need to do is only extend any of the [Spring Data repositories](https://attacomsian.com/blog/spring-data-jpa-repositories) to take the full advantage of this feature. A typical repository interface looks something like below:

public interface DogRepository extends JpaRepository<Dog, Long> {

Dog findByName(String name);

List<Dog> findByColor(String color, Sort sort);

Page<Dog> findByAgeGreaterThan(int age, Pageable pageable)

}

The first method simply finds and returns a single dog with a given name. The second method returns a list of dogs with a given color and applies the [dynamic sorting](https://attacomsian.com/blog/spring-data-jpa-sorting#sorting-query-results-using-dynamic-sorting) to the results through the given Sort object. Finally, the last method returns a Page of dogs that are older than the given age.

Although the **above approach looks really convenient** (you don't need to write a single line of implementation code to execute queries), it has at least **two drawbacks**:

1. **The number of query methods are fixed** as we cannot dynamically define a new method on runtime. As the application grows, these query methods increase quickly making it difficult to maintain the persistence layer.
2. **Each query method defines a fixed set of criteria**. We have predefined the number and type of parameters that cannot be changed dynamically on runtime.

For larger and complex applications, we need a robust and flexible query generation strategy to handle different types of use cases. That's where the Spring Data JPA specifications come in.

## **Spring Data JPA Specifications**

Spring Data JPA **Specifications allow us to create dynamic database queries** by using the JPA Criteria API. It defines a specification as a predicate over an entity. Here is how the Specification interface looks like:

public interface Specification<T> extends Serializable {

@Nullable

Predicate toPredicate(Root<T> root, CriteriaQuery<?> query,

CriteriaBuilder criteriaBuilder);

// ... other methods

}

As you can see above, there is only one abstract method toPredicate() which returns an instance of Predicate. In the following sections, you'll **learn how to use this interface to write dynamic queries** for different use cases.

## Create an Application

Let us start with creating a simple Spring Boot application with the [Spring Data JPA and H2](https://attacomsian.com/blog/spring-data-jpa-h2-database) in-memory database. Our application has just one Movie entity class, as shown below:

**Movie.java**

package com.attacomsian.jpa.domains;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import java.io.Serializable;

@Entity

public class Movie implements Serializable {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private Long id;

private String title;

private String genre;

private double rating;

private double watchTime;

private int releaseYear;

public Movie() {

}

public Movie(String title, String genre, double rating, double watchTime, int releaseYear) {

this.title = title;

this.genre = genre;

this.rating = rating;

this.watchTime = watchTime;

this.releaseYear = releaseYear;

}

// getters and setters, equals(), testing(), ... (omitted for brevity)

}

The next step is to create a repository interface called MovieRepository to retrieve data from the database. To use Specifications, we also need to extend our repository interface from the JpaSpecificationExecutor interface. This interface provides methods to execute Specifications.

Here is how our repository interface looks like:

**MovieRepository.java**

package com.attacomsian.jpa.repositories;

import com.attacomsian.jpa.domains.Movie;

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

import org.springframework.data.repository.CrudRepository;

public interface MovieRepository extends CrudRepository<Movie, Long>,

JpaSpecificationExecutor<Movie> {

// TODO: add queries

}

## Create Specifications

Let us now start with the most interesting part of this article — creating specifications to execute dynamic queries for searching movies in the database.

The first step is to create a simple enum class that defines different search operations:

**SearchOperation.java**

package com.attacomsian.jpa.repositories.specs;

public enum SearchOperation {

GREATER\_THAN,

LESS\_THAN,

GREATER\_THAN\_EQUAL,

LESS\_THAN\_EQUAL,

NOT\_EQUAL,

EQUAL,

MATCH,

MATCH\_START,

MATCH\_END,

IN,

NOT\_IN

}

Next, create the SearchCriteria class that represents a single search criteria:

**SearchCriteria.java**

package com.attacomsian.jpa.repositories.specs;

public class SearchCriteria {

private String key;

private Object value;

private SearchOperation operation;

public SearchCriteria() {

}

public SearchCriteria(String key, Object value, SearchOperation operation) {

this.key = key;

this.value = value;

this.operation = operation;

}

// getters and setters, toString(), ... (omitted for brevity)

}

As you can see above, the SearchCriteria class holds a basic representation of a single constraint:

* key — Represents the entity field name, i.e. title, genre, etc.
* value — Represents the parameter value, i.e. Troy, Action, etc.
* operation — Indicates the search operation, i.e. equality, match, comparison, etc.

Finally, let us create a MovieSpecification class that extends the Specification interface and provides an implementation for the toPredicate() method:

**MovieSpecification.java**

package com.attacomsian.jpa.repositories.specs;

import com.attacomsian.jpa.domains.Movie;

import org.springframework.data.jpa.domain.Specification;

import javax.persistence.criteria.CriteriaBuilder;

import javax.persistence.criteria.CriteriaQuery;

import javax.persistence.criteria.Predicate;

import javax.persistence.criteria.Root;

import java.util.ArrayList;

import java.util.List;

public class MovieSpecification implements Specification<Movie> {

private List<SearchCriteria> list;

public MovieSpecification() {

this.list = new ArrayList<>();

}

public void add(SearchCriteria criteria) {

list.add(criteria);

}

@Override

public Predicate toPredicate(Root<Movie> root, CriteriaQuery<?> query, CriteriaBuilder builder) {

//create a new predicate list

List<Predicate> predicates = new ArrayList<>();

//add add criteria to predicates

for (SearchCriteria criteria : list) {

if (criteria.getOperation().equals(SearchOperation.GREATER\_THAN)) {

predicates.add(builder.greaterThan(

root.get(criteria.getKey()), criteria.getValue().toString()));

} else if (criteria.getOperation().equals(SearchOperation.LESS\_THAN)) {

predicates.add(builder.lessThan(

root.get(criteria.getKey()), criteria.getValue().toString()));

} else if (criteria.getOperation().equals(SearchOperation.GREATER\_THAN\_EQUAL)) {

predicates.add(builder.greaterThanOrEqualTo(

root.get(criteria.getKey()), criteria.getValue().toString()));

} else if (criteria.getOperation().equals(SearchOperation.LESS\_THAN\_EQUAL)) {

predicates.add(builder.lessThanOrEqualTo(

root.get(criteria.getKey()), criteria.getValue().toString()));

} else if (criteria.getOperation().equals(SearchOperation.NOT\_EQUAL)) {

predicates.add(builder.notEqual(

root.get(criteria.getKey()), criteria.getValue()));

} else if (criteria.getOperation().equals(SearchOperation.EQUAL)) {

predicates.add(builder.equal(

root.get(criteria.getKey()), criteria.getValue()));

} else if (criteria.getOperation().equals(SearchOperation.MATCH)) {

predicates.add(builder.like(

builder.lower(root.get(criteria.getKey())),

"%" + criteria.getValue().toString().toLowerCase() + "%"));

} else if (criteria.getOperation().equals(SearchOperation.MATCH\_END)) {

predicates.add(builder.like(

builder.lower(root.get(criteria.getKey())),

criteria.getValue().toString().toLowerCase() + "%"));

} else if (criteria.getOperation().equals(SearchOperation.MATCH\_START)) {

predicates.add(builder.like(

builder.lower(root.get(criteria.getKey())),

"%" + criteria.getValue().toString().toLowerCase()));

} else if (criteria.getOperation().equals(SearchOperation.IN)) {

predicates.add(builder.in(root.get(criteria.getKey())).value(criteria.getValue()));

} else if (criteria.getOperation().equals(SearchOperation.NOT\_IN)) {

predicates.add(builder.not(root.get(criteria.getKey())).in(criteria.getValue()));

}

}

return builder.and(predicates.toArray(new Predicate[0]));

}

}

The MovieSpecification class allows you to combine multiple specifications to filter the movies using multiple constraints. Using this class, you can easily generate different kinds of database queries dynamically.

## Test the Application

Now is the time to create the main application class to test Specifications by writing dynamic search queries:

**Application.java**

package com.attacomsian.jpa;

import com.attacomsian.jpa.domains.Movie;

import com.attacomsian.jpa.repositories.MovieRepository;

import com.attacomsian.jpa.repositories.specs.MovieSpecification;

import com.attacomsian.jpa.repositories.specs.SearchCriteria;

import com.attacomsian.jpa.repositories.specs.SearchOperation;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.Bean;

import org.springframework.data.domain.Page;

import org.springframework.data.domain.PageRequest;

import org.springframework.data.domain.Pageable;

import org.springframework.data.domain.Sort;

import java.util.Arrays;

import java.util.List;

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

@Bean

public CommandLineRunner specificationsDemo(MovieRepository movieRepository) {

return args -> {

// create new movies

movieRepository.saveAll(Arrays.asList(

new Movie("Troy", "Drama", 7.2, 196, 2004),

new Movie("The Godfather", "Crime", 9.2, 178, 1972),

new Movie("Invictus", "Sport", 7.3, 135, 2009),

new Movie("Black Panther", "Action", 7.3, 135, 2018),

new Movie("Joker", "Drama", 8.9, 122, 2018),

new Movie("Iron Man", "Action", 8.9, 126, 2008)

));

// search movies by `genre`

MovieSpecification msGenre = new MovieSpecification();

msGenre.add(new SearchCriteria("genre", "Action", SearchOperation.EQUAL));

List<Movie> msGenreList = movieRepository.findAll(msGenre);

msGenreList.forEach(System.out::println);

// search movies by `title` and `rating` > 7

MovieSpecification msTitleRating = new MovieSpecification();

msTitleRating.add(new SearchCriteria("title", "black", SearchOperation.MATCH));

msTitleRating.add(new SearchCriteria("rating", 7, SearchOperation.GREATER\_THAN));

List<Movie> msTitleRatingList = movieRepository.findAll(msTitleRating);

msTitleRatingList.forEach(System.out::println);

// search movies by release year < 2010 and rating > 8

MovieSpecification msYearRating = new MovieSpecification();

msYearRating.add(new SearchCriteria("releaseYear", 2010, SearchOperation.LESS\_THAN));

msYearRating.add(new SearchCriteria("rating", 8, SearchOperation.GREATER\_THAN));

List<Movie> msYearRatingList = movieRepository.findAll(msYearRating);

msYearRatingList.forEach(System.out::println);

// search movies by watch time >= 150 and sort by `title`

MovieSpecification msWatchTime = new MovieSpecification();

msWatchTime.add(new SearchCriteria("watchTime", 150, SearchOperation.GREATER\_THAN\_EQUAL));

List<Movie> msWatchTimeList = movieRepository.findAll(msWatchTime, Sort.by("title"));

msWatchTimeList.forEach(System.out::println);

// search movies by `title` <> 'white' and paginate results

MovieSpecification msTitle = new MovieSpecification();

msTitle.add(new SearchCriteria("title", "white", SearchOperation.NOT\_EQUAL));

Pageable pageable = PageRequest.of(0, 3, Sort.by("releaseYear").descending());

Page<Movie> msTitleList = movieRepository.findAll(msTitle, pageable);

msTitleList.forEach(System.out::println);

};

}

}

The above example demonstrates how to use Specifications to dynamically generate different database queries. You can also apply sorting and pagination to query results in the same way as you apply to other query generation strategies.

You can even **combine** multiple Specifications together to create a new one on the fly. To do so, Spring Data JPA's Specification interface provides and() and or() methods to concatenate different Specifications. There is also a where() method that makes the expression more readable.

Here is example that combines the above msTitle and msYearRating Specifications to create a new one:

// combine using `AND` operator

List<Movie> movies = movieRepository.findAll(Specification.where(msTitle).and(msYearRating));

// combine using `OR` operator

List<Movie> movies = movieRepository.findAll(Specification.where(msTitle).or(msYearRating));

**Source Code:** Download the complete source code from [GitHub](https://github.com/attacomsian/code-examples/tree/master/spring-data-jpa/jpa-specifications) available under MIT license.

## Conclusion

That's all folks. In this article, you have learned how to use Spring Data Specifications along with JPA Criteria API to dynamically generate database queries.

Specifications provide us a robust and flexible approach of creating database queries to handle complex use cases. This tutorial covers almost all basic operations that can be used to implement a powerful search feature.

# Spring Data JPA Composite Primary Key Mapping Example

October 19, 2019 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

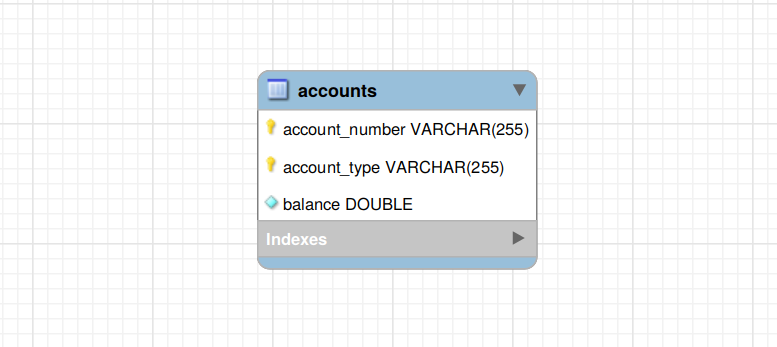
In an [earlier article](https://attacomsian.com/blog/spring-data-jpa-many-to-many-mapping), I wrote about how to map many-to-many entity relationship using [Spring Data JPA](https://attacomsian.com/blog/getting-started-spring-data-jpa) and MySQL in a Spring Boot project. We created a join table to hold the primary keys of both relationship tables. These keys act as a composite primary for the join database table.

Simply put, a **composite primary key** — also known as a composite key — is **a key that contains two or more columns to form a primary key** for the table.

In this article, you'll **learn how to map a composite primary key** in a Spring Boot project using Spring Data JPA's both @IdClass and @EmbeddedId annotations.

## Mapping Composite Key using @IdClass Annotation

Let us consider an application that manages different types of bank accounts. Each bank account has an account number and type (i.e. checking or saving), among other information. Now we want to create a compound key using this information to uniquely identify each account in the database.



In the above Entity-Relationship (ER) diagram, the accounts table has a composite primary key, which consists of two columns:

1. account\_number
2. account\_type

To map this database relationship using Spring Data JPA, we need to create a separate composite primary key class with both these primary key columns:

**AccountId.java**

package com.attacomsian.jpa.composite.domains;

import java.io.Serializable;

import java.util.Objects;

public class AccountId implements Serializable {

private String accountNumber;

private String accountType;

public AccountId() {

}

public AccountId(String accountNumber, String accountType) {

this.accountNumber = accountNumber;

this.accountType = accountType;

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

AccountId accountId = (AccountId) o;

return accountNumber.equals(accountId.accountNumber) &&

accountType.equals(accountId.accountType);

}

@Override

public int hashCode() {

return Objects.hash(accountNumber, accountType);

}

}

**Note:** The composite primary key class must be public, contains a no-argument constructor, defines both equals() and hashCode() methods, and implements the Serializable interface.

The next step is to create an Account entity class that declares all attributes of AccountId and annotate them with the @Id annotation:

**Account.java**

package com.attacomsian.jpa.composite.domains;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.IdClass;

import javax.persistence.Table;

import java.io.Serializable;

@Entity

@Table(name = "accounts")

@IdClass(AccountId.class)

public class Account implements Serializable {

@Id

private String accountNumber;

@Id

private String accountType;

private double balance;

public Account() {

}

public Account(String accountNumber, String accountType, double balance) {

this.accountNumber = accountNumber;

this.accountType = accountType;

this.balance = balance;

}

// getters and setters, equals(), toString() .... (omitted for brevity)

}

As you can see above, we have annotated the Account class with @IdClass to specify a composite primary key class that is mapped to multiple fields of the entity. The @Id annotation is then used to indicate all properties which are a part of the compound key.

With the @IdClass annotation, you can easily query data without using the name of the composite key class. Let us first create a repository interface for this purpose:

**AccountRepository.java**

package com.attacomsian.jpa.composite.repositories;

import com.attacomsian.jpa.composite.domains.Account;

import com.attacomsian.jpa.composite.domains.AccountId;

import org.springframework.data.repository.CrudRepository;

public interface AccountRepository extends CrudRepository<Account, AccountId> {

// TODO: add queries

}

Here is how you can write a simple [derived query](https://attacomsian.com/blog/derived-query-methods-spring-data-jpa) to fetch all bank accounts by a given account type:

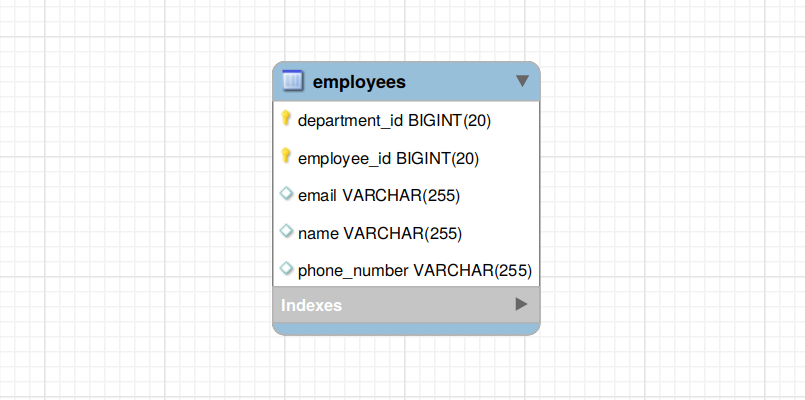
List<Account> findByAccountType(String accountType);

## Mapping Composite Key using @EmbeddedId Annotation

In additional to @IdClass, Spring Data JPA provides another annotation — @EmbeddedId — to define a composite primary key.

Let us consider another example application that manages employees of a company with multiple departments. Each employee has a unique ID within his own department. But the same ID can be assigned to a different employee in another department. So we cannot uniquely identify an employee just by his employee ID.

To uniquely identify an employee, we need to know his employee ID as well as his department ID. As you can see in the below Entity-Relationship (ER) diagram, the employees table contains a composite primary key that includes both employee\_id and department\_id columns:



To map the above relationship using Spring Data JPA, you need to create a separate composite primary key class annotated with @Embeddable:

**EmployeeId.java**

package com.attacomsian.jpa.composite.domains;

import javax.persistence.Column;

import javax.persistence.Embeddable;

import java.io.Serializable;

import java.util.Objects;

@Embeddable

public class EmployeeId implements Serializable {

@Column(name = "employee\_id")

private Long employeeId;

@Column(name = "department\_id")

private Long departmentId;

public EmployeeId() {

}

public EmployeeId(Long employeeId, Long departmentId) {

this.employeeId = employeeId;

this.departmentId = departmentId;

}

public Long getEmployeeId() {

return employeeId;

}

public void setEmployeeId(Long employeeId) {

this.employeeId = employeeId;

}

public Long getDepartmentId() {

return departmentId;

}

public void setDepartmentId(Long departmentId) {

this.departmentId = departmentId;

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

EmployeeId that = (EmployeeId) o;

return employeeId.equals(that.employeeId) &&

departmentId.equals(that.departmentId);

}

@Override

public int hashCode() {

return Objects.hash(employeeId, departmentId);

}

}

The next step is to create the Employee class and embed the above composite primary class into it by using the @EmbeddedId annotation:

**Employee.java**

package com.attacomsian.jpa.composite.domains;

import javax.persistence.Column;

import javax.persistence.EmbeddedId;

import javax.persistence.Entity;

import javax.persistence.Table;

import java.io.Serializable;

@Entity

@Table(name = "employees")

public class Employee implements Serializable {

@EmbeddedId

private EmployeeId employeeId;

private String name;

@Column(unique = true)

private String email;

private String phoneNumber;

public Employee() {

}

public Employee(EmployeeId employeeId, String name, String email, String phoneNumber) {

this.employeeId = employeeId;

this.name = name;

this.email = email;

this.phoneNumber = phoneNumber;

}

// getters and setters, equals(), toString() .... (omitted for brevity)

}

In the above Employee class, we have specified the composite key class using the @EmbeddedId annotation and mark it as a primary key of the entity.

Next, create a repository interface for retrieving Employee entities from the database, as shown below:

**EmployeeRepository.java**

package com.attacomsian.jpa.composite.repositories;

import com.attacomsian.jpa.composite.domains.Employee;

import com.attacomsian.jpa.composite.domains.EmployeeId;

import org.springframework.data.repository.CrudRepository;

public interface EmployeeRepository extends ­­­­­­­­ {

// TODO: add queries

}

Let us create another derived query to fetch all employees by a given department ID:

List<Employee> findByEmployeeIdDepartmentId(Long departmentId);

## Testing Composite Primary Key Mapping

Finally, let us create the main application class to test the composite primary key mapping:

**Application.java**

package com.attacomsian.jpa;

import com.attacomsian.jpa.composite.domains.Account;

import com.attacomsian.jpa.composite.domains.AccountId;

import com.attacomsian.jpa.composite.domains.Employee;

import com.attacomsian.jpa.composite.domains.EmployeeId;

import com.attacomsian.jpa.composite.repositories.AccountRepository;

import com.attacomsian.jpa.composite.repositories.EmployeeRepository;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.Bean;

import java.util.List;

import java.util.Optional;

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

@Bean

public CommandLineRunner mappingDemo(AccountRepository accountRepository,

EmployeeRepository employeeRepository) {

return args -> {

// ======= `@IdClass` Annotation =======

// create new accounts

accountRepository.save(new Account("458666", "Checking", 4588));

accountRepository.save(new Account("458689", "Checking", 2500));

accountRepository.save(new Account("424265", "Saving", 100000));

// fetch accounts by a given type

List<Account> accounts = accountRepository.findByAccountType("Checking");

accounts.forEach(System.out::println);

// fetch account by composite key

Optional<Account> account = accountRepository.findById(new AccountId("424265", "Saving"));

account.ifPresent(System.out::println);

// ======= `@EmbeddedId` Annotation =======

// create new employees

employeeRepository.save(new Employee(new EmployeeId(100L, 10L),

"John Doe", "john@example.com", "123456"));

employeeRepository.save(new Employee(new EmployeeId(101L, 20L),

"Emma Ali", "emma@example.com", "654321"));

// fetch employees by a given department id

List<Employee> employees = employeeRepository.findByEmployeeIdDepartmentId(20L);

employees.forEach(System.out::println);

// fetch employee by composite key

Optional<Employee> employee = employeeRepository.findById(new EmployeeId(100L, 10L));

employee.ifPresent(System.out::println);

};

}

}

In the main application class, we used both AccountRepository and EmployeeRepository repositories to test our implementation of a composite primary key with @IdClass and @EmbeddedId.

The next step is to run the application to see the output. For Gradle, execute the following command to start the application:

$ ./gradlew bootRun

For Maven, type the following command to launch the application:

$ ./mvnw spring-boot:run

When the application starts, you should see the following output printed on the console:

Account{accountNumber='458666', accountType='Checking', balance=4588.0}

Account{accountNumber='458689', accountType='Checking', balance=2500.0}

Account{accountNumber='424265', accountType='Saving', balance=100000.0}

Employee{employeeId=EmployeeId{employeeId=101, departmentId=20}, name='Emma Ali', email='emma@example.com', phoneNumber='654321'}

Employee{employeeId=EmployeeId{employeeId=100, departmentId=10}, name='John Doe', email='john@example.com', phoneNumber='123456'}

## @IdClass vs @EmbeddedId

The main difference between @IdClass and @EmbeddedId annotations is that with @IdClass, you need to specify the primary key columns twice — once in the composite primary key class and then again in the entity class with the @Id annotation.

The @EmbeddedId annotation is more verbose than @IdClass as you can access the entire primary key object using the field access method. This also gives a clear notion of the fields that are part of the composite key because they are all aggregated in a class that is only accessible through a field access method.

Another difference between @IdClass and @EmbeddedId is when it comes to creating [custom JPQL queries](https://attacomsian.com/blog/spring-data-jpa-query-annotation).

For example, with @IdClass, the query is a little simpler:

SELECT a.accountType FROM Account a

With @EmbeddedId, you have to write more text for a similar query:

SELECT e.employeeId.departmentId FROM Employee e

The @IdClass annotation can be a preferred choice over @EmbeddedId in situations where the composite primary key class is not accessible or comes in from another module or legacy code. For such scenarios, where you cannot modify the composite key class, the @IdClass annotation is the only way-out.

**Source Code:** Download the complete source code from [GitHub](https://github.com/attacomsian/code-examples/tree/master/spring-data-jpa/jpa-mappings) available under MIT license.

## Conclusion

That's all folks for handling a composite primary key mapping using Spring Data JPA's @IdClass and @EmbeddedId annotations. You've learned about two different approaches to handle compound keys.

We've also discussed the differences between the @IdClass and the @EmbeddedId annotations. If we need to access parts of the composite key, just use @IdClass, but for scenarios where you frequently use the composite key as an object, @EmbeddedId should be preferred.

# Spring Data JPA One To Many Relationship Mapping Example

October 13, 2019 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

In the [previous article](https://attacomsian.com/blog/spring-data-jpa-one-to-one-mapping), I wrote about how to define and use a one-to-one relationship with [Spring Data JPA](https://attacomsian.com/blog/getting-started-spring-data-jpa) and MySQL in a Spring Boot application.

In this article, you'll **learn how to map a one-to-many database relationship** using Spring Data JPA in a Spring Boot and MySQL application.

## Dependencies

We need both spring-data-starter-data-jpa and mysql-connector-java dependencies to use [Spring Data JPA with the MySQL database](https://attacomsian.com/blog/accessing-data-spring-data-jpa-mysql) in Spring Boot.

If you are using Gradle, add the following dependencies to your build.gradle file:

implementation 'org.springframework.boot:spring-boot-starter-data-jpa'

runtimeOnly 'mysql:mysql-connector-java'

If you are using Maven, include the following dependencies to your pom.xml file:

<dependency>

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

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

</dependency>

<dependency>

<groupId>mysql</groupId>

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

<scope>runtime</scope>

</dependency>

Want to create a new Spring Boot project from scratch? Just use [Spring Initializr](https://start.spring.io/) web tool to bootstrap a new application with the above dependencies.

## Configure MySQL Database

Spring Boot auto-configures the DataSource bean for in-memory databases like [H2 database](https://attacomsian.com/blog/spring-data-jpa-h2-database#configure-h2-database), HSQLDB, and Apache Derby. Since we are using MySQL, we need to explicitly define the database connection properties in a properties file.

Open the application.properties file in your favorite editor and add the following properties:

# MySQL connection properties

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.username=root

spring.datasource.password=mypass

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

# Log JPA queries

# Comment this in production

spring.jpa.show-sql=true

# Drop and create new tables (create, create-drop, validate, update)

# Only for testing purpose - comment this in production

spring.jpa.hibernate.ddl-auto=create

# Hibernate SQL dialect

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

Spring Boot will automatically configure a DataSource bean after reading the above properties. Don't forget to change the spring.datasource.username and spring.datasource.password properties to match your MySQL database installation.

The hibernate property spring.jpa.hibernate.ddl-auto = create will automatically create database tables based on the entity classes in your application on application startup.

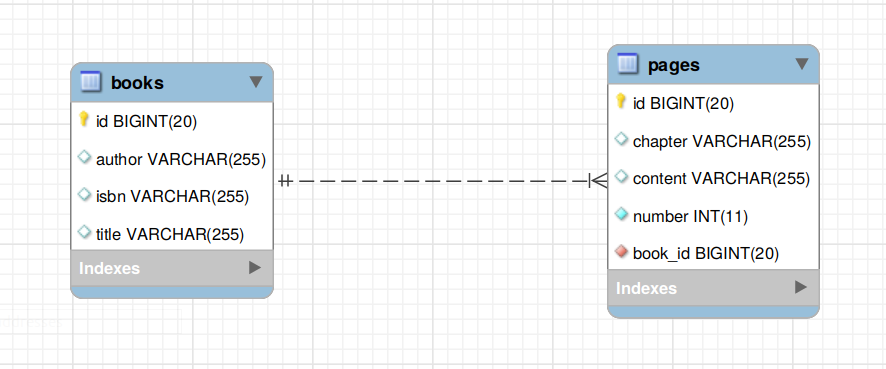
The createDatabaseIfNotExist=true configuration property, we have included in spring.datasource.url, makes sure that the database schema is automatically created if it doesn't already exist.

## One-To-Many Relationship

A one-to-many relationship refers to the relationship between two entities/tables A and B in which **one element/row of A may only be linked to many elements/rows of B, but a member of B is linked to only one element/row of A**.

For instance, think of A as a book, and B as pages. A book can have many pages but a page can only exist in one book, forming a one-to-many relationship. The **opposite of one-to-many is many-to-one** relationship.

Let us model the above relationship in the database by creating two tables, one for the books and another for the pages, as shown below in an Entity-Relationship (ER) diagram:



The one-to-many relationship is defined by the foreign key book\_id in the pages table.

## Create Entities

The next step is to create the Book and Page entities and define the one-to-many relationship mapping, as shown below:

**Book.java**

package com.attacomsian.jpa.one2many.domains;

import javax.persistence.\*;

import java.io.Serializable;

import java.util.Set;

@Entity

@Table(name = "books")

public class Book implements Serializable {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String title;

private String author;

@Column(unique = true)

private String isbn;

@OneToMany(mappedBy = "book", fetch = FetchType.LAZY,

cascade = CascadeType.ALL)

private Set<Page> pages;

public Book() {

}

public Book(String title, String author, String isbn) {

this.title = title;

this.author = author;

this.isbn = isbn;

}

// getters and setters, equals(), toString() .... (omitted for brevity)

}

**Page.java**

package com.attacomsian.jpa.one2many.domains;

import javax.persistence.\*;

import java.io.Serializable;

@Entity

@Table(name = "pages")

public class Page implements Serializable {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private int number;

private String content;

private String chapter;

@ManyToOne(fetch = FetchType.LAZY, optional = false)

@JoinColumn(name = "book\_id", nullable = false)

private Book book;

public Page() {

}

public Page(int number, String content, String chapter, Book book) {

this.number = number;

this.content = content;

this.chapter = chapter;

this.book = book;

}

// getters and setters, equals(), toString() .... (omitted for brevity)

}

Both Book and Page classes are annotated with the Entity annotation to indicate that they are JPA entities.

The @Table annotation is used to specify the name of the database table that should be mapped to this entity.

The id attributes are annotated with both @Id and @GeneratedValue annotations. The former annotation indicates that they are the primary keys of the entities. The latter annotation defines the primary key generation strategy. In the above case, we have declared that the primary key should be an AUTO INCREMENT field.

### @OneToMany Annotation

A one-to-many relationship between two entities is defined by using the @OneToMany annotation in Spring Data JPA. It declares the mappedBy element to indicate the entity that owns the bidirectional relationship. Usually, the child entity is one that owns the relationship and the parent entity contains the @OneToMany annotation.

### @ManyToOne Annotation

The @ManyToOne annotation is used to define a many-to-one relationship between two entities in Spring Data JPA. The child entity, that has the join column, is called the owner of the relationship defined using the @ManyToOne annotation.

### @JoinColumn Annotation

The @JoinColumn annotation is used to specify the foreign key column in the owner of the relationship. The inverse-side of the relationship sets the mappedBy attribute to indicate that the relationship is owned by the other entity.

## Create Repositories

Let us now define the repository interfaces to store and access the data from the database. We'll be extending our repositories from Spring Data JPA's CrudRepository interface that provides methods for generic CRUD operations.

**BookRepository.java**

package com.attacomsian.jpa.one2many.repositories;

import com.attacomsian.jpa.one2many.domains.Book;

import org.springframework.data.repository.CrudRepository;

public interface BookRepository extends CrudRepository<Book, Long> {

Book findByIsbn(String isbn);

}

**PageRepository.java**

package com.attacomsian.jpa.one2many.repositories;

import com.attacomsian.jpa.one2many.domains.Book;

import com.attacomsian.jpa.one2many.domains.Page;

import org.springframework.data.domain.Sort;

import org.springframework.data.repository.CrudRepository;

import java.util.List;

public interface PageRepository extends CrudRepository<Page, Long> {

List<Page> findByBook(Book book, Sort sort);

}

In the above repositories, we also defined some [derived query methods](https://attacomsian.com/blog/derived-query-methods-spring-data-jpa) like findByIsbn() to fetch a book by its ISBN number.

That's it. You have successfully defined a one-to-many relationship mapping in Spring Data JPA. You don't need to implement the above interfaces thanks to Spring Data JPA.

## Create an Application Class

Let us now create the main application class for the Spring Boot [console application](https://attacomsian.com/blog/spring-boot-console-application) to test our one-to-many relationship mapping:

**Application.java**

package com.attacomsian.jpa;

import com.attacomsian.jpa.one2many.domains.Book;

import com.attacomsian.jpa.one2many.domains.Page;

import com.attacomsian.jpa.one2many.repositories.BookRepository;

import com.attacomsian.jpa.one2many.repositories.PageRepository;

import com.attacomsian.jpa.one2one.domains.Address;

import com.attacomsian.jpa.one2one.domains.User;

import com.attacomsian.jpa.one2one.repositories.AddressRepository;

import com.attacomsian.jpa.one2one.repositories.UserRepository;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.Bean;

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

@Bean

public CommandLineRunner mappingDemo(BookRepository bookRepository,

PageRepository pageRepository) {

return args -> {

// create a new book

Book book = new Book("Java 101", "John Doe", "123456");

// save the book

bookRepository.save(book);

// create and save new pages

pageRepository.save(new Page(1, "Introduction contents", "Introduction", book));

pageRepository.save(new Page(65, "Java 8 contents", "Java 8", book));

pageRepository.save(new Page(95, "Concurrency contents", "Concurrency", book));

};

}

}

## Run the Application

Next, run the application to see the output. If you are using Gradle, execute the following command to start the application:

$ ./gradlew bootRun

For Maven, type the following command to launch the application:

$ ./mvnw spring-boot:run

Once the application is started, you should see the following lines printed on the console:

Hibernate: drop table if exists books

Hibernate: drop table if exists pages

Hibernate: create table books (id bigint not null auto\_increment, author varchar(255), isbn varchar(255), title varchar(255), primary key (id)) engine=InnoDB

Hibernate: create table pages (id bigint not null auto\_increment, chapter varchar(255), content varchar(255), number integer not null, book\_id bigint not null, primary key (id)) engine=InnoDB

Hibernate: insert into books (author, isbn, title) values (?, ?, ?)

Hibernate: insert into pages (book\_id, chapter, content, number) values (?, ?, ?, ?)

Hibernate: insert into pages (book\_id, chapter, content, number) values (?, ?, ?, ?)

Hibernate: insert into pages (book\_id, chapter, content, number) values (?, ?, ?, ?)

...

**Source Code:** Download the complete source code from [GitHub](https://github.com/attacomsian/code-examples/tree/master/spring-data-jpa/jpa-mappings) available under MIT license.

## Conclusion

That's all folks. In this article, you've learned how to map and use a one-to-many relationship in Spring Data JPA and Hibernate.

Don't forget to join the [mailing list](https://attacomsian.com/newsletter) if you want to be the first to know when new tutorials are available.

# Spring Data JPA Many To Many Relationship Mapping Example

October 18, 2019 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

In previous articles, I wrote about how to use [one-to-one](https://attacomsian.com/blog/spring-data-jpa-one-to-one-mapping) and [one-to-many](https://attacomsian.com/blog/spring-data-jpa-one-to-many-mapping) relationship mappings in [Spring Data JPA](https://attacomsian.com/blog/getting-started-spring-data-jpa).

In this article, you'll **learn how to map a many-to-many bidirectional relationship** using Spring Data JPA and MySQL in a Spring Boot application. Many-to-many relationships are one of the most commonly used association mappings. They require an additional database table to hold the primary keys of both relationship tables.

Let us create a new project from scratch and learn how to implement a many-to-many relationship mapping using Spring Data JPA and MySQL.

## Dependencies

To use [Spring Data JPA with the MySQL database](https://attacomsian.com/blog/accessing-data-spring-data-jpa-mysql) in Spring Boot, you need both spring-data-starter-data-jpa and mysql-connector-java dependencies.

Add the following dependencies to your Gradle project's build.gradle file:

implementation 'org.springframework.boot:spring-boot-starter-data-jpa'

runtimeOnly 'mysql:mysql-connector-java'

If you are using Maven, include the following dependencies to your pom.xml file:

<dependency>

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

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

</dependency>

<dependency>

<groupId>mysql</groupId>

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

<scope>runtime</scope>

</dependency>

If you are starting a new project, just use [Spring Initializr](https://start.spring.io/) web tool to bootstrap a new Spring Boot application with the above-mentioned dependencies.

## Configure MySQL Database

Spring Boot automatically configures DataSource for in-memory databases like [H2 database](https://attacomsian.com/blog/spring-data-jpa-h2-database#configure-h2-database), HSQLDB, and Apache Derby. For the MySQL database, you need to explicitly define the database connection properties in a properties file.

Open the application.properties file and copy and paste the following properties:

# MySQL connection properties

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.username=root

spring.datasource.password=mypass

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

# Log JPA queries

# Comment this in production

spring.jpa.show-sql=true

# Drop and create new tables (create, create-drop, validate, update)

# Only for testing purpose - comment this in production

spring.jpa.hibernate.ddl-auto=create

# Hibernate SQL dialect

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

Spring Boot will automatically configure a DataSource bean based on the above properties. Make sure to change the spring.datasource.username and spring.datasource.password properties to match your MySQL database installation.

The hibernate property spring.jpa.hibernate.ddl-auto = create will automatically create database tables based on the entity classes when the application starts.

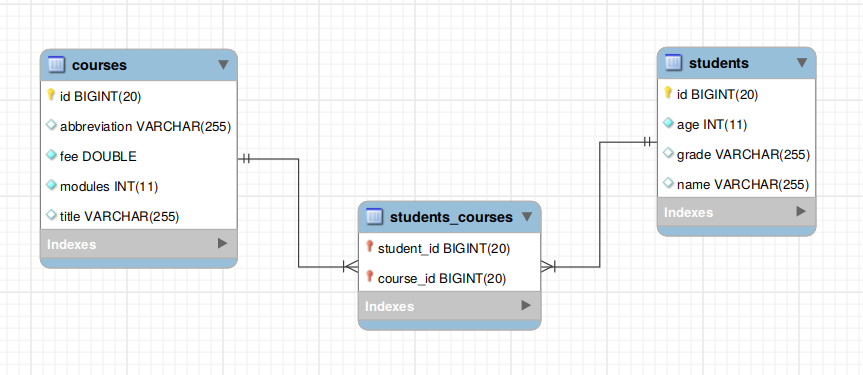
The createDatabaseIfNotExist=true configuration property, included in spring.datasource.url, makes sure that the database schema is automatically created if it doesn't already exist.

## Many-To-Many Relationship

A many-to-many relationship refers to the relationship between two entities/tables A and B in which **one element/row of A may only be associated with many elements/rows of B and vice versa**.

A typical example of such a many-to-many relationship is the relationship between **students** and **courses**. A student can enroll in multiple courses and a course can also have multiple students, thus forming a many-to-many relationship.

To model the above relationship in the database, **you need to create three tables**, one each for both students and courses, and another one for holding relationship keys, as shown below in the Entity-Relationship (ER) diagram:



students\_courses is a join table that contains two foreign keys, student\_id and course\_id, to reference both students and courses database tables. Both these foreign keys also act as a composite primary key for the students\_courses table.

## Create Entities

We need to create two entity classes, Student and Course, to map the above many-to-many relationship. You don't need to create a separate entity class for the join table.

Here is how our entity classes look like:

**Student.java**

package com.attacomsian.jpa.many2many.domains;

import javax.persistence.\*;

import java.io.Serializable;

import java.util.Set;

@Entity

@Table(name = "students")

public class Student implements Serializable {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private int age;

private String grade;

@ManyToMany(fetch = FetchType.LAZY, cascade = CascadeType.PERSIST)

@JoinTable(name = "students\_courses",

joinColumns = {

@JoinColumn(name = "student\_id", referencedColumnName = "id",

nullable = false, updatable = false)},

inverseJoinColumns = {

@JoinColumn(name = "course\_id", referencedColumnName = "id",

nullable = false, updatable = false)})

private Set<Course> courses = new HashSet<>();

public Student() {

}

public Student(String name, int age, String grade) {

this.name = name;

this.age = age;

this.grade = grade;

}

// getters and setters, equals(), toString() .... (omitted for brevity)

}

**Course.java**

package com.attacomsian.jpa.many2many.domains;

import javax.persistence.\*;

import java.io.Serializable;

import java.util.Set;

@Entity

@Table(name = "courses")

public class Course implements Serializable {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String title;

private String abbreviation;

private int modules;

private double fee;

@ManyToMany(mappedBy = "courses", fetch = FetchType.LAZY)

private Set<Student> students = new HashSet<>();

public Course() {

}

public Course(String title, String abbreviation, int modules, double fee) {

this.title = title;

this.abbreviation = abbreviation;

this.modules = modules;

this.fee = fee;

}

// getters and setters, equals(), toString() .... (omitted for brevity)

}

Both Student and Course classes are annotated with the Entity annotation to indicate that they are JPA entities.

The @Table annotation is used to specify the name of the database table that should be mapped to this entity.

The id attributes are annotated with both @Id and @GeneratedValue annotations. The former annotation indicates that they are the primary keys of the entities. The latter annotation defines the primary key generation strategy. In the above case, we have declared that the primary key should be an AUTO INCREMENT field.

### @ManyToMany Annotation

A many-to-many relationship between two entities is defined by using the @ManyToMany annotation in Spring Data JPA. It uses the mappedBy attribute to indicate the entity that owns the bidirectional relationship. In a bidirectional relationship, the @ManyToMany annotation is defined in both entities but only one entity can own the relationship. We've picked the Student class as an owner of the relationship in the above example.

### @JoinTable Annotation

The @JoinTable annotation defines the join table between two entities on the owner's side of the relationship. We have used this annotation to define the students\_courses table. If the @JoinTable annotation is left out, the default values of the annotation elements apply. The name of the join table is supposed to be the table names of the associated primary tables concatenated together (owning side first) using an underscore.

## Create Repositories

The next step is to define the repository interfaces for storing and accessing the data from the database. We'll extend our repositories from Spring Data JPA's CrudRepository interface that provides methods for generic CRUD operations.

**StudentRepository.java**

package com.attacomsian.jpa.many2many.repositories;

import com.attacomsian.jpa.many2many.domains.Student;

import org.springframework.data.repository.CrudRepository;

import java.util.List;

public interface StudentRepository extends CrudRepository<Student, Long> {

List<Student> findByNameContaining(String name);

}

**CourseRepository.java**

package com.attacomsian.jpa.many2many.repositories;

import com.attacomsian.jpa.many2many.domains.Course;

import org.springframework.data.repository.CrudRepository;

import java.util.List;

public interface CourseRepository extends CrudRepository<Course, Long> {

List<Course> findByTitleContaining(String title);

List<Course> findByFeeLessThan(double fee);

}

We also defined a few [derived query methods](https://attacomsian.com/blog/derived-query-methods-spring-data-jpa) like findByFeeLessThan() to retrieve all courses that have a fee less than the given value and more.

That's it. You are done with defining a many-to-many relationship mapping in Spring Data JPA.

## Create an Application Class

Now is the time to create the main application class for our Spring Boot [console application](https://attacomsian.com/blog/spring-boot-console-application) to test our many-to-many relationship mapping:

**Application.java**

package com.attacomsian.jpa;

import com.attacomsian.jpa.many2many.domains.Course;

import com.attacomsian.jpa.many2many.domains.Student;

import com.attacomsian.jpa.many2many.repositories.CourseRepository;

import com.attacomsian.jpa.many2many.repositories.StudentRepository;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.Bean;

import java.util.Arrays;

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

@Bean

public CommandLineRunner mappingDemo(StudentRepository studentRepository,

CourseRepository courseRepository) {

return args -> {

// create a student

Student student = new Student("John Doe", 15, "8th");

// save the student

studentRepository.save(student);

// create three courses

Course course1 = new Course("Machine Learning", "ML", 12, 1500);

Course course2 = new Course("Database Systems", "DS", 8, 800);

Course course3 = new Course("Web Basics", "WB", 10, 0);

// save courses

courseRepository.saveAll(Arrays.asList(course1, course2, course3));

// add courses to the student

student.getCourses().addAll(Arrays.asList(course1, course2, course3));

// update the student

studentRepository.save(student);

};

}

}

## Run the Application

Next, run the application to see the output. If you are using Gradle, execute the following command to start the application:

$ ./gradlew bootRun

For Maven, type the following command to launch the application:

$ ./mvnw spring-boot:run

Once the application is started, you should see the following lines printed on the console:

Hibernate: drop table if exists courses

Hibernate: drop table if exists students

Hibernate: drop table if exists students\_courses

Hibernate: create table courses (id bigint not null auto\_increment, abbreviation varchar(255), fee double precision not null, modules integer not null, title varchar(255), primary key (id)) engine=InnoDB

Hibernate: create table students (id bigint not null auto\_increment, age integer not null, grade varchar(255), name varchar(255), primary key (id)) engine=InnoDB

Hibernate: create table students\_courses (student\_id bigint not null, course\_id bigint not null, primary key (student\_id, course\_id)) engine=InnoDB

Hibernate: insert into students (age, grade, name) values (?, ?, ?)

Hibernate: insert into courses (abbreviation, fee, modules, title) values (?, ?, ?, ?)

Hibernate: insert into courses (abbreviation, fee, modules, title) values (?, ?, ?, ?)

Hibernate: insert into courses (abbreviation, fee, modules, title) values (?, ?, ?, ?)

Hibernate: select student0\_.id as id1\_4\_0\_, student0\_.age as age2\_4\_0\_, student0\_.grade as grade3\_4\_0\_, student0\_.name as name4\_4\_0\_ from students student0\_ where student0\_.id=?

Hibernate: select courses0\_.student\_id as student\_1\_5\_0\_, courses0\_.course\_id as course\_i2\_5\_0\_, course1\_.id as id1\_2\_1\_, course1\_.abbreviation as abbrevia2\_2\_1\_, course1\_.fee as fee3\_2\_1\_, course1\_.modules as modules4\_2\_1\_, course1\_.title as title5\_2\_1\_ from students\_courses courses0\_ inner join courses course1\_ on courses0\_.course\_id=course1\_.id where courses0\_.student\_id=?

Hibernate: select course0\_.id as id1\_2\_0\_, course0\_.abbreviation as abbrevia2\_2\_0\_, course0\_.fee as fee3\_2\_0\_, course0\_.modules as modules4\_2\_0\_, course0\_.title as title5\_2\_0\_ from courses course0\_ where course0\_.id=?

Hibernate: select course0\_.id as id1\_2\_0\_, course0\_.abbreviation as abbrevia2\_2\_0\_, course0\_.fee as fee3\_2\_0\_, course0\_.modules as modules4\_2\_0\_, course0\_.title as title5\_2\_0\_ from courses course0\_ where course0\_.id=?

Hibernate: select course0\_.id as id1\_2\_0\_, course0\_.abbreviation as abbrevia2\_2\_0\_, course0\_.fee as fee3\_2\_0\_, course0\_.modules as modules4\_2\_0\_, course0\_.title as title5\_2\_0\_ from courses course0\_ where course0\_.id=?

Hibernate: insert into students\_courses (student\_id, course\_id) values (?, ?)

Hibernate: insert into students\_courses (student\_id, course\_id) values (?, ?)

Hibernate: insert into students\_courses (student\_id, course\_id) values (?, ?)

...

**Source Code:** Download the complete source code from [GitHub](https://github.com/attacomsian/code-examples/tree/master/spring-data-jpa/jpa-mappings) available under MIT license.

## Conclusion

That's all folks. In this article, you've learned how to map and use a many-to-many relationship using Spring Data JPA and MySQL in a Spring Boot application.

Don't forget to [subscribe for updates](https://attacomsian.com/newsletter) if you want to be the first to know when new tutorials are available.

# Spring Data JPA One To One Relationship Mapping Example

October 12, 2019 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

[Spring Data JPA](https://attacomsian.com/blog/getting-started-spring-data-jpa) provides excellent support to create Spring-powered applications that communicate with different relational databases to implement JPA based repositories.

[Java Persistence API](https://jcp.org/en/jsr/detail?id=338) (JPA) is just a specification that defines an object-relational mapping (ORM) standard for storing, accessing, and managing Java objects in a relational database. [Hibernate](http://hibernate.org/) is the most popular and widely used implementation of JPA specifications. By default, Spring Data JPA uses Hibernate as a JPA provider.

Hibernate provides a framework for **mapping relational database tables to entity classes** in your application. You can describe the relationships between these entities in a similar way as you define relationships among the tables in your relational database.

In this article, you'll **learn how to create a one-to-one mapping between two entities** using Spring Data JPA in a Spring Boot and MySQL application.

## Dependencies

You just need spring-data-starter-data-jpa and mysql-connector-java dependencies to use [Spring Data JPA with the MySQL database](https://attacomsian.com/blog/accessing-data-spring-data-jpa-mysql).

For a Gradle project, add the following dependencies to your build.gradle file:

implementation 'org.springframework.boot:spring-boot-starter-data-jpa'

runtimeOnly 'mysql:mysql-connector-java'

For a Maven project, include the following dependencies to your pom.xml file:

<dependency>

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

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

</dependency>

<dependency>

<groupId>mysql</groupId>

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

<scope>runtime</scope>

</dependency>

If you want to create a new Spring Boot project from scratch, just use [Spring Initializr](https://start.spring.io/) web tool to bootstrap a new application with the above dependencies.

## Configure MySQL Database

By default, Spring Boot automatically configures the DataSource bean for in-memory databases like [H2 database](https://attacomsian.com/blog/spring-data-jpa-h2-database#configure-h2-database). But for MySQL, we need to manually define the database connection details in a properties file.

Open your application.properties file and add the following properties:

# MySQL connection properties

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.username=root

spring.datasource.password=mypass

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

# Log JPA queries

# Comment this in production

spring.jpa.show-sql=true

# Drop and create new tables (create, create-drop, validate, update)

# Only for testing purpose - comment this in production

spring.jpa.hibernate.ddl-auto=create

# Hibernate SQL dialect

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

Spring Boot will read the above properties and will auto-configure a DataSource bean for you. Make sure to change the spring.datasource.username and spring.datasource.password properties to match your MySQL credentials.

The hibernate property spring.jpa.hibernate.ddl-auto = create will automatically create database tables based on the entity classes in your application on startup.

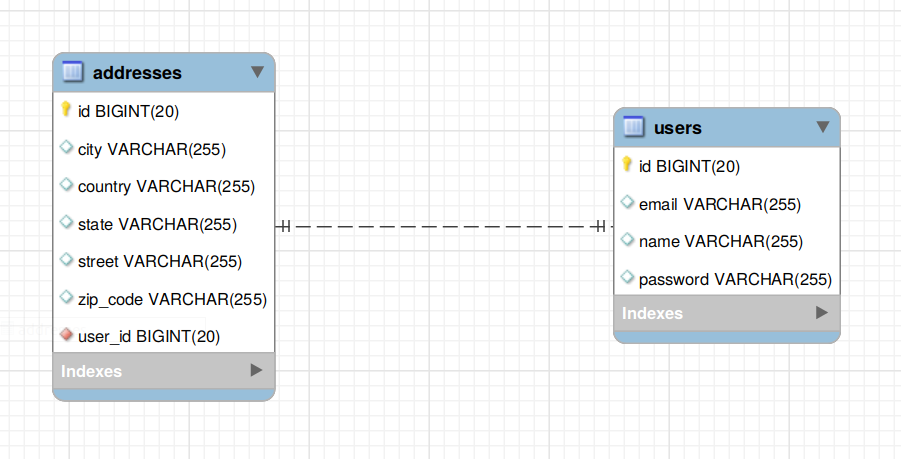
We have also included the createDatabaseIfNotExist=true configuration property in spring.datasource.url to automatically create the database if it doesn't already exist.

## One-To-One Relationship

A one-to-one relationship refers to the relationship between two entities/database tables A and B in which **only one element/row of A may only be linked to one element/row of B**, and vice versa.

Let us consider an application scenario where you want to store users' information along with their addresses. We want to make sure that a user can have just one address, and an address can only be associated with a single user.

We can map the above requirement as a one-to-one relationship between the user and the address entities, as shown in the following Entity-Relationship (ER) diagram:



The one-to-one relationship is defined by using a foreign key called user\_id in the addresses table.

## Create Entities

You need to create the following two entity classes to map the above database tables.

### User Entity

Let us define the below User entity in your application to map the users database table:

package com.attacomsian.jpa.one2one.domains;

import javax.persistence.\*;

import java.io.Serializable;

@Entity

@Table(name = "users")

public class User implements Serializable {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

private String password;

@OneToOne(mappedBy = "user", fetch = FetchType.LAZY,

cascade = CascadeType.ALL)

private Address address;

public User() {

}

public User(String name, String email, String password) {

this.name = name;

this.email = email;

this.password = password;

}

// getters and setters, equals(), toString() .... (omitted for brevity)

}

### Address Entity

To map the addresses table, you need to define the following Address entity:

package com.attacomsian.jpa.one2one.domains;

import javax.persistence.\*;

import java.io.Serializable;

@Entity

@Table(name = "addresses")

public class Address implements Serializable {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String street;

private String city;

private String state;

private String zipCode;

private String country;

@OneToOne(fetch = FetchType.LAZY, optional = false)

@JoinColumn(name = "user\_id", nullable = false)

private User user;

public Address() {

}

public Address(String street, String city, String state, String zipCode,

String country) {

this.street = street;

this.city = city;

this.state = state;

this.zipCode = zipCode;

this.country = country;

}

// getters and setters, equals(), toString() .... (omitted for brevity)

}

Both User and Address classes are annotated with Entity to indicate that they are JPA entities.

The @Table annotation is used to specify the name of the database table that should be mapped to this entity.

The id attributes are annotated with both @Id and @GeneratedValue annotations. The former annotation indicates that they are the primary keys of the entities. The latter annotation defines the primary key generation strategy. In the above case, we have declared that the primary key should be an AUTO INCREMENT field.

### @OneToOne Annotation

In Spring Data JPA, a one-to-one relationship between two entities is declared by using the @OneToOne annotation. It accepts the following parameters:

* fetch — Defines a strategy for fetching data from the database. By default, it is EAGER which means that the data must be eagerly fetched. We have set it to LAZY to fetch the entities lazily from the database.
* cascade — Defines a set of cascadable operations that are applied to the associated entity. CascadeType.ALL means to apply all cascading operations to the related entity. Cascading operations are applied when you delete or update the parent entity.
* mappedBy — Defines the entity that owns the relationship which is the Address entity in our case.
* optional — Defines whether the relationship is optional. If set to false then a non-null relationship must always exist.

In a bidirectional relationship, we have to specify the @OneToOne annotation in both entities. But only one entity is the owner of the association. Usually, the child entity is one that owns the relationship and the parent entity is the inverse side of the relationship.

### @JoinColumn Annotation

The @JoinColumn annotation is used to specify the foreign key column in the owner of the relationship. The inverse-side of the relationship sets the @OneToOne's mappedBy parameter to indicate that the relationship is mapped by the other entity.

The @JoinColumn accepts the following two important parameters, among others:

* name — Defines the name of the foreign key column.
* nullable — Defines whether the foreign key column is nullable. By default, it is true.

## Create Repositories

Finally, let us create our repository interfaces to save and retrieve User and Address entities from the database. We'll extend our repositories from Spring Data JPA's CrudRepository interface which provides generic CRUD methods out-of-the-box.

### UserRepository Interface

Here is how our UserRepository interface looks like:

package com.attacomsian.jpa.one2one.repositories;

import com.attacomsian.jpa.one2one.domains.User;

import org.springframework.data.repository.CrudRepository;

public interface UserRepository extends CrudRepository<User, Long> {

}

### AddressRepository Interface

The next step is to create the following AddressRepository interface:

package com.attacomsian.jpa.one2one.repositories;

import com.attacomsian.jpa.one2one.domains.Address;

import org.springframework.data.repository.CrudRepository;

public interface AddressRepository extends CrudRepository<Address, Long> {

}

That's all you need to do to define a one-to-one mapping in Spring Data JPA. You're now ready to perform the CRUD operations on User and Address entities without implementing the above interfaces. This is what makes Spring Data JPA a very powerful tool.

## Create an Application Class

Let us now create the main application for our Spring Boot [console application](https://attacomsian.com/blog/spring-boot-console-application) to test our one-to-one relationship mapping:

**Application.java**

package com.attacomsian.jpa;

import com.attacomsian.jpa.one2one.domains.Address;

import com.attacomsian.jpa.one2one.domains.User;

import com.attacomsian.jpa.one2one.repositories.AddressRepository;

import com.attacomsian.jpa.one2one.repositories.UserRepository;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.Bean;

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

@Bean

public CommandLineRunner mappingDemo(UserRepository userRepository,

AddressRepository addressRepository) {

return args -> {

// create a user instance

User user = new User("John Doe", "john.doe@example.com", "1234abcd");

// create an address instance

Address address = new Address("Lake View 321", "Berlin", "Berlin",

"95781", "DE");

// set child reference

address.setUser(user);

// set parent reference

user.setAddress(address);

// save the parent

// which will save the child (address) as well

userRepository.save(user);

};

}

}

We've defined a CommandLineRunner interface bean in the main application class. This interface provides a run() method that is automatically invoked by Spring Boot after the application context is loaded.

## Run the Application

Now is the time to run the application to see the output. If you are using Gradle, execute the following command to start the application:

$ ./gradlew bootRun

For Maven, type the following command to launch the application:

$ ./mvnw spring-boot:run

Once the application is started, you should see the following lines printed on the console:

Hibernate: drop table if exists addresses

Hibernate: drop table if exists users

Hibernate: create table addresses (id bigint not null auto\_increment, city varchar(255), country varchar(255), state varchar(255), street varchar(255), zip\_code varchar(255), user\_id bigint not null, primary key (id)) engine=InnoDB

Hibernate: create table users (id bigint not null auto\_increment, email varchar(255), name varchar(255), password varchar(255), primary key (id)) engine=InnoDB

Hibernate: insert into users (email, name, password) values (?, ?, ?)

Hibernate: insert into addresses (city, country, state, street, user\_id, zip\_code) values (?, ?, ?, ?, ?, ?)

...

**Source Code:** Download the complete source code from [GitHub](https://github.com/attacomsian/code-examples/tree/master/spring-data-jpa/jpa-mappings) available under MIT license.

# Pagination with Spring Data JPA

October 12, 2019 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

In the [last article](https://attacomsian.com/blog/spring-data-jpa-sorting), I wrote about how to sort the query results in Spring Data JPA using both static and dynamic sorting techniques.

In the modern web, the response time of your website is a critical factor for higher search engine ranking. Visitors are expecting pages to load quickly and only show the relevant information. For example, if you own an e-commerce website with tens of thousands of products, this means only displaying a small number of products at once, and not all of them.

To help you deal with such situations, **Spring Data JPA provides the concepts of pagination**. It makes it easy to deal with a large amount of data in the most efficient way.

In this article, you will **learn how to paginate the query results** in Spring Data JPA. I will show you how pagination works with [derived](https://attacomsian.com/blog/derived-query-methods-spring-data-jpa) and [custom](https://attacomsian.com/blog/spring-data-jpa-query-annotation) queries.

## Create an Application

Let us create a simple Spring Boot application with the [Spring Data JPA and H2](https://attacomsian.com/blog/spring-data-jpa-h2-database) in-memory database. It only contains a single entity called Person:

**Person.java**

package com.attacomsian.jpa.domains;

import javax.persistence.\*;

@Entity

public class Person {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private Long id;

private String firstName;

private String lastName;

private int age;

public Person() {

}

public Person(String firstName, String lastName, int age) {

this.firstName = firstName;

this.lastName = lastName;

this.age = age;

}

// getters and setters, equals(), toString(), ... (omitted for brevity)

}

Here is how our PersonRespository looks like:

**PersonRepository.java**

package com.attacomsian.jpa.repositories;

import com.attacomsian.jpa.domains.Person;

import org.springframework.data.repository.PagingAndSortingRepository;

public interface PersonRepository extends PagingAndSortingRepository<Person, Long> {

// TODO: add paging methods

}

## Spring Data's Pageable Interface

Just like the special Sort parameter, we used for the [dynamic sorting](https://attacomsian.com/blog/spring-data-jpa-sorting#sorting-query-results-using-dynamic-sorting), Spring Data JPA supports another special parameter called Pageable for paginating the query results. These special parameters are automatically recognized by the Spring Data infrastructure to apply pagination and sorting to database queries dynamically.

The Pageable interface contains the information about the requested page such as the size and the number of the page. It provides the following methods, among others, to add paging to statically define queries:

public interface Pageable {

// returns the current page number (zero-based)

int getPageNumber();

// returns the size of the page

int getPageSize();

// returns the sorting parameters

Sort getSort();

// ... other methods

}

Whenever you want to apply pagination to query results, all you need to do is just add Pageable to the query method definition as a parameter and set the return by Page<T>:

Page<Person> findByLastName(String lastName, Pageable pageable);

While calling the above method, you need to create an object of Pageable and pass it to the invoked repository method. The simplest way to create an instance of Pageable is to use the PageRequest class which provides the implementation of Pageable methods:

Pageable pageable = PageRequest.of(0, 10);

This will create a page request for the first page (page index is zero-based) with 10 as the size of the page to be returned.

You can even apply [dynamic sorting](https://attacomsian.com/blog/spring-data-jpa-sorting#sorting-query-results-using-dynamic-sorting) by using the Pageable instance as shown below:

// pageable instance with dynamic sorting

Pageable pageable = PageRequest.of(0, 10, Sort.by("age").descending());

// mutlple sort parameters

Pageable pageable = PageRequest.of(0, 10, Sort.by("age").descending()

.and(Sort.by("lastName").ascending()));

## Spring Data's Slice and Page

The example, we have defined above, returns a Page object. A Page contains information about the total number of elements and pages available in the database. It is because the Spring Data JPA triggers a count query to calculate the number of elements.

Depending on the database you are using in your application, it might become expensive as the number of items increased. To avoid this costly count query, you should instead return a Slice. Unlike a Page, a Slice only knows about whether the next slice is available or not. This information is sufficient to walk through a larger result set.

Both Slice and Page are part of Spring Data JPA, where Page is just a sub-interface of Slice with a couple of additional methods. You should use Slice if you don't need the total number of items and pages. A good example of such a scenario is when you only need Next Page and Previous Page buttons. Here is an example query that returns a Slice:

Slice<Person> findByAgeGreaterThan(int age, Pageable pageable);

## Paginating Query Results with Pageable

Now we know what's the Pageable interface and what are the differences between a Page and a Slice, let us create database queries that accept a Pageable object as a parameter to paginate their results.

The PagingAndSortingRepository interface, we are extending above, provides us the findAll(Pageable pageable) method to paginate all Person entities available in the database. All you need to do is just create an instance of Pageable and pass it to this method to get a Page.

However, if you want to get a Slice or a List, you need to define your own queries in the repository interface.

### Derived Queries

Applying pagination to [derived queries](https://attacomsian.com/blog/spring-data-jpa-sorting#sort-query-results-using-static-sorting) is a straightforward task. You just need to pass the Pageable interface as a parameter to any derived query and set the desired return type as shown below:

Page<Person> findByFirstName(String firstName, Pageable pageable);

Slice<Person> findByAgeBetween(int start, int end, Pageable pageable);

List<Person> findByLastNameIsNotNull(Pageable pageable);

### Custom Queries with @Query Annotation

To apply pagination to JPQL queries defined using the [@Query annotation](https://attacomsian.com/blog/spring-data-jpa-query-annotation), Spring Data JPA allows you to pass the Pageable interface as a parameter:

@Query("SELECT p FROM Person p WHERE p.lastName = ?1")

Page<Person> findByLastNameJPQL(String lastName, Pageable pageable);

@Query("SELECT p FROM Person p WHERE p.age < :age")

Page<Person> findByAgeLessThanJPQL(@Param("age") int page, Pageable pageable);

To use pagination with native SQL queries declared using the @Query annotation, you need to **define the count query by yourself**, as shown in the following example:

@Query(value = "SELECT \* FROM Person p WHERE p.firstName = :firstName",

countQuery = "SELECT count(\*) Person p WHERE p.firstName = :firstName",

nativeQuery = true)

Page<Person> findByFirstNameNativeSQL(@Param("firstName") String firstName, Pageable pageable);

### Named Queries

You can also paginate the query results of [named queries](https://attacomsian.com/blog/spring-data-jpa-named-queries) that use JPQL to define the query statements.

Let us first declare a JPQL named query by using the [@NamedQuery annotation](https://attacomsian.com/blog/spring-data-jpa-named-queries#namedquery-annotation):

@NamedQuery(name = "Person.findByFirstNameNamed",

query = "SELECT p FROM Person p WHERE p.firstName = ?1")

public class Person {

// ...

}

Now you can reference the above named query in the PersonRepository interface and pass Pageable as a special parameter:

Page<Person> findByFirstNameNamed(String firstName, Pageable pageable);

Similar to native SQL custom queries, **native named queries also require a separate count query** in order to use the Pageable interface for paging the query results. Let us define a native SQL named query:

@NamedNativeQuery(name = "Person.findByLastNameNativeNamed",

query = "SELECT \* FROM Person p WHERE p.firstName = :firstName")

public class Person {

// ...

}

To reference the above native SQL named query in your repository interface, you must add a @Query annotation on the repository method and set the nativeQuery attribute to true along with the count query as shown below:

@Query(countQuery = "SELECT count(\*) Person p WHERE p.lastName = :lastName",

nativeQuery = true)

Page<Person> findByLastNameNativeNamed(@Param("lastName") String lastName, Pageable pageable);

**Note:** Dynamic sorting is not supported by named queries. So whiling calling the above repository methods, your Pageable instance should not contain a Sort object.

## Using the Pageable Interface

The following examples show how to create an instance of Pageable and then call different query methods that paginate their results:

// get all persons by last name

Pageable pageable = PageRequest.of(0, 3);

Page<Person> personPage = personRepository.findByLastName("Doe", pageable);

// get all persons sorted by their age in the descending order

Pageable pageable2 = PageRequest.of(0, 5, Sort.by("age").descending());

Page<Person> personPage2 = personRepository.findAll(pageable2);

If you want to **skip the pagination** for query methods that require an instance of Pageable, just use the Pageable.unpaged() method:

Slice<Person> personSlice = personRepository.findByAgeBetween(20, 60, Pageable.unpaged());

**Source Code:** Download the complete source code from [GitHub](https://github.com/attacomsian/code-examples/tree/master/spring-data-jpa/jpa-pagination) available under MIT license.

## Conclusion

In this article, you have learned how to applying pagination to query results for different kinds of queries in Spring Data JPA.

We also discussed the Pageable interface and the Slice and Page differences in detail. Check out the [Sorting Query Results in Spring Data JPA](https://attacomsian.com/blog/spring-data-jpa-sorting) if you want to learn more about the sorting functionality in Spring Data JPA.

# How to Use Spring Data JPA Named Queries

October 08, 2019 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

[Spring Data JPA](https://attacomsian.com/blog/getting-started-spring-data-jpa) provides multiple ways of writing and executing different kinds of queries to fetch data from the database. In the last two articles, I have explained how to:

* create [derived queries](https://attacomsian.com/blog/derived-query-methods-spring-data-jpa) by referencing the name of the method.
* use the [@Query annotation](https://attacomsian.com/blog/spring-data-jpa-query-annotation) to declare JPQL and native SQL queries.

Both **these approaches are good** for small to medium applications that have **only a small number of queries**. As soon as your application grows with tens of hundreds of different kinds of queries scattered in many classes, it becomes harder to maintain such a large number of queries in Java codebase.

To make developers' life easier, Spring Data JPA provides the concept of **named queries** that can be defined by using a properties file, annotations, or an XML file. You can group related queries in one place and refer them in your code by their names. Spring Data JPA will take care of all the boilerplate code required to execute these queries.

In this article, you will learn **how to create and reference named JPQL and native SQL queries** in a Spring Data JPA repository.

## Create an Application

Let us extend our existing [Spring Data JPA and H2 database](https://attacomsian.com/blog/spring-data-jpa-h2-database) application to add another entity and a repository interface.

First of all, add the following Book entity that holds book information:

**Book.java**

package com.attacomsian.jpa.named.domains;

import javax.persistence.\*;

@Entity

public class Book {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private Long id;

private String title;

@Column(unique = true)

private String isbn;

private int pages;

public Book() {

}

public Book(String title, String isbn, int pages) {

this.title = title;

this.isbn = isbn;

this.pages = pages;

}

// getters and setters, equals(), toString(), ... (omitted for brevity)

}

Now create the BookRepository.java interface for storing and retrieving the data from the database:

**BookRepository.java**

package com.attacomsian.jpa.named.repositories;

import com.attacomsian.jpa.named.domains.Book;

import org.springframework.data.repository.CrudRepository;

public interface BookRepository extends CrudRepository<Book, Long> {

// TODO: add named query methods

}

## Creating Named Queries

We can create named queries by using an external properties file, Java annotations, or an XML file. Spring Data JPA supports both JPQL and native SQL named queries.

The definition of a named query is pretty much straightforward. It has two parts: the **name** and the **query**.

The **name must be unique** within your application persistence context. To reference the named query in your Spring Data JPA repository, its name should start with the name of the entity class, followed by a dot (.), and the name of the repository method. Here is an example:

{EntityName}.{RepositoryMethodName}

// Examples

1. Book.findByIsbn

2. Book.findByPagesGreaterThan

**Note:** You do not need to follow these naming conventions if you want to execute named queries programmatically using EntityManager.

The **query part** of the named query **must contain a valid JPQL or native SQL statement**.

### 1. Using a Properties File

You can define named queries by using a properties file called jpa-named-queries.properties inside the META-INF folder of your classpath.

In a Spring Boot project, by default, this folder is not available. You need to first create META-INF folder inside /src/main/resources/.

Here is how you can declare JPQL named queries in the jpa-named-queries.properties file:

# find all books order by title descending

Book.findAllNamedFile=SELECT b FROM Book b ORDER BY b.title DESC

# find books by title

Book.findByTitleNamedFile=SELECT b FROM Book b WHERE b.title = ?1

# find a book by isbn

Book.findByIsbnNamedFile=SELECT b FROM Book b WHERE b.isbn = ?1

As you can see above, we set the name of the named query as the property name and the query as the value of the property.

Similarly, you can define native SQL named queries inside the properties file:

# native SQL query to find books by title

Book.findByTitleNativeNamedFile=SELECT \* FROM book b WHERE b.title = :title

### 2. Using the orm.xml File

If you don't want to use a properties file, you can create an orm.xml file inside the same META-INF folder for declaring named queries. Spring Data JPA supports the following two XML elements to enable automatic invocation of named queries:

1. <named-query /> — Use this XML element for defining a JPQL named query.
2. <named-native-query /> — This XML element is used to declare a native SQL named query. If your native name query returns an entity, you have to specify the entity class by using the result-class attribute.

Let us create the orm.xml file and add some named queries:

<?xml version="1.0" encoding="UTF-8"?>

<entity-mappings version="2.0" xmlns="http://java.sun.com/xml/ns/persistence/orm"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm

http://java.sun.com/xml/ns/persistence/orm\_2\_0.xsd ">

<!--find all books order by pages descending-->

<named-query name="Book.findAllXML">

<query>SELECT b FROM Book b ORDER BY b.pages DESC</query>

</named-query>

<!--find books by title-->

<named-query name="Book.findByTitleContainingXML">

<query>SELECT b FROM Book b WHERE b.title LIKE ?1</query>

</named-query>

<!--native SQL query to find a book by isbn-->

<named-native-query name="Book.findByIsbnNativeXML"

result-class="com.attacomsian.jpa.named.domains.Book">

<query>SELECT \* FROM book b WHERE b.isbn = :isbn</query>

</named-native-query>

</entity-mappings>

### 3. Using Annotations

In addition to the above two methods, you can also declare named queries by annotating your entities with the following annotations:

* @NamedQuery — This annotation is used to define a JPQL named query.
* @NamedNativeQuery — Use this annotation to declare a native SQL named query.

Both @NamedQuery and @NamedNativeQuery annotations can be used separately or together on the entity class. These annotations define the name of the query as well as the query string. If your native SQL named query returns an entity, you need explicitly indicate it by using the resultClass element.

#### @NamedQuery Annotation

The @NamedQuery annotation accepts two elements, name and query, as follows:

@NamedQuery(name = "Book.findAllJPQL",

query = "SELECT b FROM Book b ORDER BY b.title DESC")

public class Book {

// ...

}

Multiple JPQL named queries can be defined together using the @NamedQueries annotation, as shown below:

@NamedQueries({

@NamedQuery(name = "Book.findAllJPQL",

query = "SELECT b FROM Book b ORDER BY b.title DESC"),

@NamedQuery(name = "Book.findByTitleJPQL",

query = "SELECT b FROM Book b WHERE b.title = ?1"),

@NamedQuery(name = "Book.findByTitleAndPagesGreaterThanJPQL",

query = "SELECT b FROM Book b WHERE b.title = :title AND b.pages > :pages")

})

public class Book {

// ...

}

The @NamedQuery annotation is very much similar to the [@Query](https://attacomsian.com/blog/spring-data-jpa-query-annotation) annotation. The only difference is the former is used to define finder queries directly on the entity class, whereas the latter is used in the repository interface method.

#### @NamedNativeQuery Annotation

The @NamedNativeQuery annotation is used for defining native SQL named queries. A native SQL query is more powerful and flexible than a JPQL query. The underlying persistence provider, Hibernate for Spring Boot, doesn't parse these queries and sends them directly to the database. This allows you to use all SQL features supported by your database.

A native SQL named query is declared in almost the same way as you define a named JPQL query. However, there are two important differences:

1. The value of the query attribute must be a valid SQL statement instead of a JPQL statement.
2. You need to indicate the return type of the query by using the resultClass attribute.

Here is an example of native SQL named query defined using the NamedNativeQuery annotation:

@NamedNativeQuery(name = "Book.findAllNative",

query = "SELECT \* FROM book b ORDER BY b.title DESC",

resultClass = Book.class)

public class Book {

// ...

}

Multiple native SQL named queries can be declared using the @NamedNativeQueries annotation, as follows:

@NamedNativeQueries({

@NamedNativeQuery(name = "Book.findAllNative",

query = "SELECT \* FROM book b ORDER BY b.title DESC",

resultClass = Book.class),

@NamedNativeQuery(name = "Book.findByIsbnNative",

query = "SELECT \* FROM book b WHERE b.isbn = :isbn",

resultClass = Book.class)

})

public class Book {

// ...

}

## Executing Named Queries using EntityManager

The simplest way to execute a named query is by using the JPA's EntityManager.createNamedQuery() method. You can run both JPQL and native SQL queries in the same way.

Just inject the EntityManager object to your controller or any other business class:

@PersistenceContext

private EntityManager em;

Now you call the createNamedQuery() method on the EntityManager with the name of the named query you want to execute as a parameter:

Query q = em.createNamedQuery("Book.findByTitleJPQL");

This gives you an instance of a Query interface. Now call the setParameter() method on the returned interface for each binding parameter used in your query:

q.setParameter(1, "Java 101");

After specifying bind parameters, call the getSingleResult() (for a single result) or getResultSet() (for multiple results) method on the Query interface to execute the query:

// execute query

List<Book> books = q.getResultList();

That's all for creating and executing a JPQL named query. You can also execute a native SQL named query in a similar way:

Query q = em.createNamedQuery("Book.findByIsbnNative");

q.setParameter("isbn", "145804");

// execute query

Book book = (Book) q.getSingleResult();

## Referencing Named Queries in a Spring Data Repository

While creating and executing a named query using EntityManager is straightforward, it may not be the most efficient way. Spring Data JPA allows you to reference your named queries a repository interface.

If you have followed the Spring Data JPA's naming conventions, I [explained before](https://attacomsian.com/blog/spring-data-jpa-named-queries#creating-named-queries) in this article, **referencing a named query is very simple**. All you need to do is just **create a method** in the repository interface **with the same name** as of named query, pass the correct method parameters, and specify the return type of the query method.

If the named query is a native SQL query, you also need to annotate the query method with the @Query annotation and set the value of its nativeQuery attribute to true.

Let us reference all named queries, we have created in the above section, in our BookRepository interface:

// named queries declared in `jpa-named-queries.properties` file

List<Book> findAllNamedFile();

List<Book> findByTitleNamedFile(String title);

Book findByIsbnNamedFile(String isbn);

@Query(nativeQuery = true)

List<Book> findByTitleNativeNamedFile(@Param("title") String title);

// named queries declared in `orm.xml` file

List<Book> findAllXML();

List<Book> findByTitleContainingXML(String title);

@Query(nativeQuery = true)

List<Book> findByIsbnNativeXML(@Param("isbn") String isbn);

// named queries declared with `@NamedQuery`

List<Book> findAllJPQL();

List<Book> findByTitleJPQL(String title);

List<Book> findByTitleAndPagesGreaterThanJPQL(@Param("title") String title, @Param("pages") int pages);

// named queries declared with `@NamedNativeQuery`

@Query(nativeQuery = true)

List<Book> findAllNative();

@Query(nativeQuery = true)

List<Book> findByIsbnNative(@Param("isbn") String isbn);

You can now inject a BookRepository instance in your controller or any other class and call the repository methods to execute the named queries:

// create books

bookRepository.save(new Book("Java 101", "145804", 450));

bookRepository.save(new Book("Spring Bot", "48524", 289));

// list all books

List<Book> books = bookRepository.findAllXML();

// fetch a single book

Book book = bookRepository.findByIsbnNamedFile("145804");

// multiple parameters

List<Book> moreBooks = bookRepository.findByTitleAndPagesGreaterThanJPQL("Spring Bot", 150);

As you can see above, calling a named query repository method is the same as you call a [derived query](https://attacomsian.com/blog/derived-query-methods-spring-data-jpa) or a [custom query](https://attacomsian.com/blog/spring-data-jpa-query-annotation) defined by using @Query.

## Sorting Named Queries Results

Named queries support [static sorting](https://attacomsian.com/blog/spring-data-jpa-sorting#sort-query-results-using-static-sorting) mechanism to sort the query results. You can append the ORDER BY clause to any named JPQL or native SQL query.

We already added static sorting to several named queries, we have declared above. For instance, the Book.findAllJPQL() named query sorts all books by their title field in the descending order.

Unfortunately, [dynamic sorting](https://attacomsian.com/blog/spring-data-jpa-sorting#sorting-query-results-using-dynamic-sorting)**is currently not supported** by Spring Data JPA named queries.

**Source Code:** Download the complete source code from [GitHub](https://github.com/attacomsian/code-examples/tree/master/spring-data-jpa/jpa-queries) available under MIT license.

## Conclusion

Named queries are one of the three options we have discussed in this Spring Data JPA tutorial series. The other two are [derived query](https://attacomsian.com/blog/derived-query-methods-spring-data-jpa) and the [@Query annotation](https://attacomsian.com/blog/spring-data-jpa-query-annotation). Spring Data JPA allows you to define both JPQL and native SQL named queries.

In this article, you have **learned about three different ways of defining name queries** in Spring Data JPA. You can either use a properties file, JPA annotations, or an XML file for this purpose.

The simplest and the most widely used approach is the JPA @NamedQuery and @NamedNativeQuery annotations. These annotations accept at least two parameters, the name and the query. The name must be unique within the application context and followed the {EntityName}.{RepositoryMethodName} naming convention. The query attribute must contain a valid JPQL or an SQL statement depending upon the type of the name query (JPQL vs native SQL).

To execute a named query, you can either use JPA's EntityManager or reference it in your repository interface and then execute it by invoking the repository method.

# Sorting Query Results with Spring Data JPA

October 09, 2019 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

In previous articles, I have talked about creating and using [derived](https://attacomsian.com/blog/derived-query-methods-spring-data-jpa), [custom](https://attacomsian.com/blog/spring-data-jpa-query-annotation), and [named](https://attacomsian.com/blog/spring-data-jpa-named-queries) queries in Spring Data JPA to interact with the database. I also explained how to apply simple ordering to query results while retrieving data.

In this article, you will learn **how to sort the query results** with [Spring Data JPA](https://attacomsian.com/blog/getting-started-spring-data-jpa). There are **two ways** to achieve this in Spring Data JPA:

1. **Static Sorting** — Add an ORDER BY clause to your JPQL or native SQL query
2. **Dynamic Sorting** — Add the special parameter Sort to your repository method

Both of these approaches work fine. The only difference is static sorting is **fixed and defined beforehand** with attributes names and sorting directions. On the other hand, dynamic sorting enables you to **dynamically choose any attribute** or even a combination of multiple attributes for sorting the query results.

## Create an Application

To demonstrate Spring Data JPA sorting capabilities, let us first create a new [Spring Boot application](https://attacomsian.com/blog/spring-data-jpa-h2-database) that uses the H2 in-memory database for storing and retrieving data. If you want to use the MySQL database as a data store, follow [this tutorial](https://attacomsian.com/blog/accessing-data-spring-data-jpa-mysql).

For the sake of simplicity, our example application has just one entity — Employee:

**Employee.java**

package com.attacomsian.jpa.domains;

import javax.persistence.\*;

@Entity

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private Long id;

private String firstName;

private String lastName;

private int age;

private double salary;

public Employee() {

}

public Employee(String firstName, String lastName, int age, double salary) {

this.firstName = firstName;

this.lastName = lastName;

this.age = age;

this.salary = salary;

}

// getters and setters, equals(), toString(), ... (omitted for brevity)

}

The next step is to add the EmployeeRepository interface to retrieve Employee entities:

**EmployeeRepository.java**

package com.attacomsian.jpa.repositories;

import com.attacomsian.jpa.domains.Employee;

import org.springframework.data.repository.PagingAndSortingRepository;

public interface EmployeeRepository extends PagingAndSortingRepository<Employee, Long> {

// TODO: add query methods

}

In the above code, we are extending the PagingAndSortingRepository interface. This is an extension of CrudRepository and provides additional methods for pagination and sorting such as findAll(Sort sort).

## Sort Query Results using Static Sorting

Static sorting refers to the **mechanism where the retrieved data is always sorted by specified columns** and directions. The columns and sort directions are defined at the development time and cannot be changed at runtime.

### Derived Queries

For [derived queries](https://attacomsian.com/blog/spring-data-jpa-sorting#sort-query-results-using-static-sorting), Spring Data JPA allows us to apply static sorting by adding the OrderBy keyword to the method name along with the property name and sort direction (Asc or Desc).

Let us use the method naming conventions to create a derived query that returns all Employee entities in the ascending order of their salary:

List<Employee> findByOrderBySalaryAsc();

The Asc keyword is **optional** as OrderBy, by default, sorts the results in the ascending order. The above query can be rewritten as follows:

List<Employee> findByOrderBySalary();

To **reverse** the sort direction, just add the Desc keyword after the property name:

List<Employee> findByOrderBySalaryDesc();

With the sort query, you can **use all other Spring Data JPA keywords**. Let us create another query method that returns employees by their last name and sorts them by their age in the descending order:

List<Employee> findByLastNameOrderByAgeDesc(String lastName);

To **sort query results by multiple attributes**, all you need to do is just reference multiple properties after the OrderBy clause and specify their sort directions. For example, the following derived query sorts the employees by their last name in the ascending order and by their salary in the descending order:

List<Employee> findByOrderByLastNameAscSalaryDesc();

### Custom Queries with @Query Annotation

If you are using the [@Query annotation](https://attacomsian.com/blog/spring-data-jpa-query-annotation) to define custom JPQL or native SQL queries, adding support for static ordering is even simpler and straightforward. All you need to do is just add the ORDER BY clause to the query definition.

The following **JPQL query** returns all employees with the salary greater than the specified amount and sorted by their age in the descending order:

@Query("SELECT e FROM Employee e WHERE e.salary > ?1 ORDER BY e.age DESC")

List<Employee> findBySalaryGreaterThanJPQL(double salary);

In a similar way, you can apply static sorting to a **native SQL query**. The following query returns employees by their first name sorted by salary in the ascending order:

@Query(value = "SELECT \* FROM Employee e WHERE e.fistName = :firstName ORDER BY e.salary ASC",

nativeQuery = true)

List<Employee> findByFirstNameNativeSQL(@Param("firstName") String firstName);

### Named Queries

Applying static sorting to [named queries](https://attacomsian.com/blog/spring-data-jpa-sorting) is very much similar to the above custom queries. All you need to do is just add the ORDER BY clause to any JPQL or native SQL named query.

The following **JPQL named query** returns employees who are older than the given age after sorting them by their first name in the ascending order:

@NamedQuery(name = "Employee.findByAgeGreaterThanNamedJPQL",

query = "SELECT e FROM Employee e WHERE e.age > :age ORDER BY e.firstName ASC")

public class Employee {

// ...

}

Here is another example of **native SQL named query** that returns all employees after sorting them by their age in the descending order:

@NamedNativeQuery(name = "Employee.findAllNamedNativeSQL",

query = "SELECT \* FROM Employee e ORDER BY e.age DESC")

public class Employee {

// ...

}

## Sorting Query Results using Dynamic Sorting

Unlike static sorting, dynamic sorting provides **more flexibility in choosing sort columns and directions**. Spring Data JPA allows you to add a special Sort parameter to your query method. The Sort class is just a specification that provides sorting options for database queries.

By using dynamic sorting, you can **choose the sorting column and direction at runtime** to sort the query results. The EmployeeRepository interface already inherits the findAll(Sort sort) method from PagingAndSortingRepository that returns all Employee entities sorted by the given options.

### Derived Queries

You can easily define additional derived and custom query methods that accept the special Sort parameter. The following [derived query](https://attacomsian.com/blog/derived-query-methods-spring-data-jpa) returns employees by their last name and also accepts the Sort parameter

List<Employee> findByLastName(String lastName, Sort sort);

Let us define another derived query that returns employees by their first name and the salary less than the given value:

List<Employee> findByFirstNameAndSalaryLessThan(String lastName, double salary, Sort sort);

### Custom Queries with @Query Annotation

Similar to derived queries, [custom queries](https://attacomsian.com/blog/spring-data-jpa-query-annotation) also accept the Sort parameter to apply sorting to query results. The following **JPQL query** accepts the Sort parameter for sorting employees whose salaries fall in a certain range:

@Query("SELECT e FROM Employee e WHERE e.salary > ?1 AND e.salary < ?2")

List<Employee> findBySalaryRange(double start, double end, Sort sort);

Currently, Spring Data JPA **does not support dynamic sorting for native SQL queries**, as it would require the manipulation of the actual SQL query defined, which cannot be done reliably by the Spring Data JPA.

### Using the Sort Parameter

Just defining queries with the Sort parameter is not enough. You have to create a Sort object to specify the entity attributes, you want to use for sorting, and their directions, and then pass this object as a parameter to call the above query methods.

The following example calls findAll() to sort all Employee entities found in the database by their age in the descending order:

Iterable<Employee> emps = employeeRepository.findAll(Sort.by("age").descending());

The real benefit of using dynamic sorting is that you **do not need to modify the query** method if you decide to change the sort options. All you need to do is just update the Sort object.

Let us extend the above example to include one more sorting attribute:

Iterable<Employee> emps = employeeRepository.findAll(Sort.by("age", "salary").descending());

Sometimes, we need **different sorting order for each sort attribute**. You can easily do this using the Sort class. The following example sorts the query results in the descending order by using the salary property and in the ascending order by using the firstName property:

Sort sort = Sort.by("salary").descending().and(Sort.by("firstName"));

List<Employee> employees = employeeRepository.findByLastName("Doe", sort);

There is **no explicit limit** for the number of sorting attributes you can use in a single query. You can use as many as you want:

Sort sort = Sort.by("salary").descending().and(Sort.by("firstName"))

.and(Sort.by("age").descending()).and(Sort.by("lastName").ascending());

List<Employee> employees = employeeRepository.findBySalaryRange(100, 10000, sort);

If you **want to skip the sorting** for a query that has a Sort parameter, just use the Sort.unsorted() method:

Iterable<Employee> emps = employeeRepository.findAll(Sort.unsorted());

**Source code:** Download the complete source code from [GitHub](https://github.com/attacomsian/code-examples/tree/master/spring-data-jpa/jpa-sorting) available under MIT license.

## Conclusion

In this article, you have learned how to apply sorting to query results by using Spring Data JPA. There are two ways to sort the query results in Spring Data JPA: static and dynamic sorting.

For static sorting, we add the ORDER BY clause directly to the query definition at the development time. Therefore the query result is always sorted by defined columns and sorting directions.

Dynamic sorting is more flexible than static sorting. We can choose the sort columns and order at runtime.

# Spring Data JPA Custom Queries using @Query Annotation

October 06, 2019 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

In an [earlier article](https://attacomsian.com/blog/derived-query-methods-spring-data-jpa), I explained how to create and use derived query methods to retrieve data from the database in Spring Data JPA. This article is an extension of the previous article to learn how to use the @Query annotation to define JPQL (Java Persistence Query Language) and native queries in [Spring Data JPA](https://attacomsian.com/blog/getting-started-spring-data-jpa).

**Derived queries** are good as long as they are not complex. As the number of query parameters goes beyond 3 to 4, you need a more flexible strategy. Then add sorting, filtering, comparison, and result size limiting keywords to derived queries, it would have become practically impossible to read and maintain these queries.

For such complicated situations, you should rather use the Spring Data JPA's @Query annotation to **define a custom JPQL or native SQL query**.

The @Query annotation defines queries directly on repository methods. This gives you full flexibility to run any query without following the method naming conventions.

Let us get started with the creation of a sample Spring Data JPA application.

## Create an Application

We have already created a [Spring Data JPA and H2 database](https://attacomsian.com/blog/spring-data-jpa-h2-database) application for the previous tutorial. Let us extend this application to add another entity and a repository interface.

Add the following Note entity to store notes:

**Note.java**

package com.attacomsian.jpa.custom.domains;

import javax.persistence.\*;

import java.util.Date;

@Entity

public class Note {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private Long id;

private String title;

private int priority;

private boolean featured;

private Date created;

public Note() {

}

public Note(String title, int priority, boolean featured, Date created) {

this.title = title;

this.priority = priority;

this.featured = featured;

this.created = created;

}

// getters and setters, equals(), toString(), ... (omitted for brevity)

}

Next, add the following NoteRepository interface to save and fetch notes from the database:

**NoteRepository.java**

package com.attacomsian.jpa.custom.repositories;

import com.attacomsian.jpa.custom.domains.Note;

import org.springframework.data.repository.CrudRepository;

public interface NoteRepository extends CrudRepository<Note, Long> {

// TODO: add custom methods

}

## JPQL vs Native SQL Query

According to [Wikipedia](https://en.wikipedia.org/wiki/Java_Persistence_Query_Language):

The ***Java Persistence Query Language*** (JPQL) is a ***platform-independent object-oriented query language*** defined as part of the Java Persistence API (JPA) specification. JPQL is used to make queries against entities stored in a relational database. It is ***heavily inspired by SQL***, and its queries resemble SQL queries in syntax, but ***operate against JPA entity objects*** rather than directly with database tables.

Clearly, JPQL is just an object-oriented way of defining queries based on entity attributes. Spring Data JPA supports both JPQL as well as native SQL queries. The JPA implementation you use, which is Hibernate by default, will then execute the query and return the result. The only downside of using JPQL is that it supports a subset of the SQL standard. So, it may not be a great choice for complex queries.

Let us define a custom query using JPQL to retrieve all featured notes:

@Query("SELECT n FROM Note n WHERE n.featured = true")

List<Note> findByActiveNotes();

// SELECT clause is optional - it can be omitted

@Query("FROM Note n WHERE n.featured = true")

List<Note> findByActiveNotes();

The same query can be written in native SQL format. All you need to do is just set the nativeQuery attribute value to true and define the native SQL query in the value attribute of the @Query annotation:

@Query(value = "SELECT \* FROM Notes n WHERE n.featured = 1",

nativeQuery = true)

List<Note> findByFeaturedNotesNative();

## Positional vs Named Bind Parameters

Bind parameters act as a placeholder in a custom query that must be replaced with actual values before the query gets executed. There are two ways to bind parameters in Spring Data JPA. You can either use **positional** (also called indexed) or **named** bind parameters. Spring Data JPA treats both JPQL and native SQL queries bind parameters in the same way.

### Positional Parameters

A positional bind parameter is referenced by its position in the query. They are defined with ? followed by a number that specifies the position (?1, ?2, etc.). Spring Data JPA will automatically set the bind parameter values. It replaces the value of each method parameter with the value of a bind parameter in the same position. For example, the first method parameter value becomes the bind parameter value at position 1, the second method parameter value becomes the bind parameter value at position 2, and so on.

Here is an example of positional-based bind parameters:

// single bind parameter

@Query("SELECT n FROM Note n WHERE n.title = ?1")

List<Note> findByTitlePositionalBind(String title);

// multiple bind parameters

@Query("SELECT n FROM Note n WHERE n.title = ?1 AND n.featured = ?2")

List<Note> findByTitleAndFeaturedPositionalBind(String title, boolean featured);

### Named Parameters

Named bind parameters are another way of passing method parameter values to the query bind parameters. A named bind parameter starts with : followed by the name of the parameter. This gives you the liberty to pass method parameters in any order without worrying about their position.

You can use the @Param annotation to specify the name of the bind parameter in the method definition. Each method parameter annotated with @Param must have a corresponding bind parameter in the JPQL or SQL query. Named parameters are easier to read and less error-prone.

Let us have an example to demonstrate how to use named bind parameters:

// single named parameter

@Query("SELECT n FROM Note n WHERE n.title = :title")

List<Note> findByTitleNamedBind(@Param("title") String title);

// multiple named parameters

@Query("SELECT n FROM Note n WHERE n.title = :title AND n.featured = :featured")

List<Note> findByTitleAndFeaturedNamedBind(@Param("featured") boolean featured,

@Param("title") String title);

Named bind parameters are especially useful when you want to use a single method parameter value multiple times in the query. A common example is a search query:

@Query("SELECT u FROM User u WHERE " +

"lower(u.name) LIKE lower(CONCAT('%', :keyword, '%')) OR " +

"lower(u.email) LIKE lower(CONCAT('%', :keyword, '%'))")

List<User> searchUsers(@Param("keyword") String keyword);

## JPQL Selection Queries

The next step is to write some JPQL selection queries to retrieve data from the database. We'll be writing custom queries for the following JPA keywords using the @Query annotation:

* And / Or
* Equality
* Ignore Case
* Not Equal
* Like / Contains / Starts With / Ends With
* Less Than / Greater Than
* Between
* Before / After
* Null / Not Null
* In

// And / Or

@Query("SELECT n FROM Note n WHERE n.title = ?1 AND n.featured = ?2 OR n.priority = ?3")

List<Note> findByTitleAndFeaturedOrPriority(String title, boolean featured, int priority);

// Equality

@Query("SELECT n FROM Note n WHERE n.title = ?1")

List<Note> findByTitle(String title);

// Ignore Case

@Query("SELECT n FROM Note n WHERE lower(n.title) = lower(?1) ")

List<Note> findByTitleIgnoreCase(String title);

// Not Equal

@Query("SELECT n FROM Note n WHERE n.title <> ?1")

List<Note> findByTitleNotEqual(String title);

// Like / Contains / Starts With / Ends With

@Query("SELECT n FROM Note n WHERE n.title LIKE ?1")

List<Note> findByTitleLike(String pattern);

// Less Than

@Query("SELECT n FROM Note n WHERE n.priority < ?1")

List<Note> findByPriorityLessThan(int priority);

// Greater Than

@Query("SELECT n FROM Note n WHERE n.priority > ?1")

List<Note> findByPriorityGreaterThan(int priority);

// Between

@Query("SELECT n FROM Note n WHERE n.priority BETWEEN ?1 AND ?2")

List<Note> findByPriorityBetween(int start, int end);

// Before

@Query("SELECT n FROM Note n WHERE n.created < ?1")

List<Note> findByCreatedBefore(Date before);

// After

@Query("SELECT n FROM Note n WHERE n.created > ?1")

List<Note> findByCreatedAfter(Date before);

// Null

@Query("SELECT n FROM Note n WHERE n.title IS NULL")

List<Note> findByTitleIsNull();

// Not Null

@Query("SELECT n FROM Note n WHERE n.title IS NOT NULL")

List<Note> findByTitleIsNotNull();

// In

@Query("SELECT n FROM Note n WHERE n.priority IN ?1")

List<Note> findByPriorityIn(Set<Integer> priorities);

## Sorting Custom Query Results

Just like derived queries, you can add sorting to any custom query of Spring Data JPA. For **static ordering**, the simplest and fastest way is to use the JPQL's ORDER BY clause inside the query. It is very much similar to what you use for the native SQL query. Here is an example:

// sort notes in ascending order

@Query("SELECT n FROM Note n WHERE n.title = ?1 ORDER BY n.priority ASC")

List<Note> findByTitleOrderByPriorityAsc(String title);

// sort notes in descending order

@Query("SELECT n FROM Note n WHERE n.featured = ?1 ORDER BY n.created DESC")

List<Note> findByFeaturedOrderByCreatedDesc(boolean featured);

For **dynamic sorting**, Spring Data JPA allows you to add a special parameter of type Sort to the custom method definition. Spring Data JPA will then generate the ORDER BY clause automatically. This is similar to what we have used in a [derived query](https://attacomsian.com/blog/derived-query-methods-spring-data-jpa#sorting-derived-query-results):

@Query("SELECT n FROM Note n WHERE n.title = ?1")

List<Note> findByTitle(String title, Sort sort);

Now if you want to call the above sorting method, you need to create a Sort object to specify the entity attributes and their ordering:

// sort by title ascending

List<Note> startupNotes = noteRepository.findByTitle("startup", Sort.by("title").ascending());

// sort by priority descending

List<Note> techNotes = noteRepository.findByTitle("tech", Sort.by("priority").descending());

// sort by title's length in ascending order

List<Note> lengthyNotes = noteRepository.findByTitle("job", JpaSort.unsafe("LENGTH(title)"));

## Paginate Custom Query Results

In addition to sorting, Spring Data JPA also allows you to apply paging to your custom query results. When you use pagination, you only get a subset of the complete result as a Page object. A Page knows about the total number of elements and pages available.

To use pagination in a JPQL query, all you need to do is just pass the special parameter Pageable to your method definition. Spring Data JPA will generate the required code automatically to handle the pagination of the query result.

Here is an example:

@Query("SELECT n FROM Note n")

Page<Note> findAllNotesWithPagination(Pageable pageable);

When you call the findAllNotesWithPagination method, you need to pass an object that implements the Pageable interface:

Pageable pageable = PageRequest.of(0, 10);

Page<Note> notePage = noteRepository.findAllNotesWithPagination(pageable);

Sorting can also be added to the PageRequest instance:

Pageable pageable = PageRequest.of(0, 10, Sort.by("title").descending());

Page<Note> notePage = noteRepository.findAllNotesWithPagination(pageable);

## Update Queries with @Modifying Annotation

The @Query annotation is not just limited to defining queries that read data from the database. You can also use it to **define queries that insert, update, or delete records** in the database.

Since these queries change the state of the database, they are treated differently. You need to explicitly tell Spring Data JPA that your custom query changes the data by annotating the repository method with an additional @Modifying annotation. It will then execute the custom query as an update operation.

Here are some examples that demonstrate how to use the @Modifying annotation for update queries:

@Modifying

@Query("UPDATE Note n SET n.title = ?1 WHERE n.id = ?2")

int updateTitleById(String title, Long id);

@Modifying

@Query("DELETE FROM Note n WHERE n.title = ?1")

void deleteByTitle(String title);

@Modifying

@Query("UPDATE Note n SET n.title = ?1 WHERE n.id IN ?2")

int bulkUpdateTitle(String title, Set<Long> id);

@Modifying

@Query("DELETE FROM Note n WHERE n.featured = ?1 AND n.id IN ?2")

void bulkDeleteByFeatured(boolean featured, Set<Long> id);

## Dynamic Queries with SpEL Expressions

Spring Data JPA also allows you to use the Spring Expression Language (SpEL) expressions in custom queries that are defined using the @Query annotation. Upon query execution, these expressions are evaluated against a predefined set of variables.

### Generic Entity Names

In the above queries, we always referenced the entity by its name. However, you can replace the actual entity name with the entityName expression variable. This allows you to define queries for generic repositories. The entityName variable is especially useful if you decide to rename the entity in the future. Because you don't need to update all queries that reference the old entity name.

The following example query avoids referencing the Note entity by its name:

@Query("SELECT n from #{#entityName} n WHERE n.title = ?1")

List<Note> findByTitleGeneric(String title);

Spring Data JPA will replace the #{#entityName} SpEL expression with the actual entity name of the domain type of the repository. In the above example, Spring Data JPA replaces #{#entityName} with Note.

### Advanced LIKE Expressions

Another example of SpEL expressions is the definition of advanced LIKE expressions. You can, for example, append % to the beginning and end of a bind parameter and change the given parameter value to lower case:

@Query("SELECT n FROM Note n WHERE lower(n.title) LIKE %?#{[0].toLowerCase()}%")

List<Note> findByTitleIgnoreCaseSpEL(String title);

**Source Code:** Download the complete source code from [GitHub](https://github.com/attacomsian/code-examples/tree/master/spring-data-jpa/jpa-queries) available under MIT license.

## Conclusion

That's all peoples. In this article, you have learned about different ways of defining custom queries in Spring Data JPA by using the @Query annotation. Custom queries defined through the @Query annotation take precedence over all other query generation strategies including [derived queries](https://attacomsian.com/blog/derived-query-methods-spring-data-jpa).

The @Query annotation gives you full control over the executed query. You can choose between a JPQL or a native SQL query. By default, Spring Data JPA expects a JPQL query with the @Query annotation. If you want to run a native query instead, you have to set the nativeQuery parameter to true.

The @Query annotation can also be used to define modifying queries that insert, update, or remove records from the database. However, you need to annotate repository methods that execute modifying queries with the @Modifying annotation.

**Continue reading** the next part of this tutorial that explains how to [create and use named JPQL and native queries](https://attacomsian.com/blog/spring-data-jpa-named-queries).

# Derived Query Methods in Spring Data JPA

October 04, 2019 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

[Spring Data JPA](https://attacomsian.com/blog/getting-started-spring-data-jpa) allows you to execute different kinds of queries to retrieve data from the database. You can either use the method name to derive a query directly or manually define your own JPQL/native query using the [@Query annotation](https://attacomsian.com/blog/spring-data-jpa-query-annotation).

For simple use-cases, you can easily write a **derive query method by simply looking at the corresponding method name** in the entity class. Just define the query methods in a repository interface that extends one of the Spring Data's repositories such as CrudRepository. Spring Data JPA will create queries automatically by parsing these method names.

In this article, you'll learn how Spring Data JPA leverages the idea of method naming conventions for mapping JPA queries.

## Derived Query Methods Structure

A derived query **method name has two main components** separated by the first By keyword:

1. **The introducer** clause like find, read, query, count, or get which tells Spring Data JPA what you want to do with the method. This clause can contain further expressions, such as Distinct to set a distinct flag on the query to be created.
2. **The criteria** clause that starts after the first By keyword. The first By acts as a delimiter to indicate the start of the actual query criteria. The criteria clause is where you define conditions on entity properties and concatenate them with And and Or keywords.

Here are some examples:

# find users by last name

List<User> findByLastName(String lastName);

# find distinct users by email

List<User> findDistinctByEmail(String email);

# count users by profession

int countByProfession(String profession);

**Note:** You can also use readBy, getBy, and queryBy in place of findBy and Spring Data JPA would behave the same. For example, readByName(String name) is equivalent to findByName(String name).

## Create an Application

Let us first create a sample [Spring Boot application](https://attacomsian.com/blog/spring-data-jpa-h2-database) with Spring Data JPA and H2 database. If you prefer to use MySQL for the data store, follow this step-by-step [tutorial](https://attacomsian.com/blog/accessing-data-spring-data-jpa-mysql).

The sample application has just one User entity. Here is how it looks like:

**User.java**

package com.attacomsian.jpa.derived.domains;

import javax.persistence.\*;

import java.util.Date;

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private Long id;

private String name;

@Column(unique = true)

private String email;

private int age;

private Date birthDate;

private boolean active;

public User() {

}

public User(String name, String email, int age, Date birthDate, boolean active) {

this.name = name;

this.email = email;

this.age = age;

this.birthDate = birthDate;

this.active = active;

}

// getters and setters, equals(), toString(), ... (omitted for brevity)

}

Next, create the UserRepository interface which extends CrudRepository. We will use this repository interface for defining all our derived query methods. Here is how it looks like:

**UserRepository.java**

package com.attacomsian.jpa.derived.repositories;

import com.attacomsian.jpa.derived.domains.User;

import org.springframework.data.repository.CrudRepository;

public interface UserRepository extends CrudRepository<User, Long> {

// TODO: add derived methods

}

## Simple Derived Queries

Let us start with a simple example of a derived query that fetches User entities with a given name:

List<User> findByName(String name);

The above-derived query is pretty simple and self-explaining. The name of the method begins with the findBy keyword and then used the entity attribute name to specify the criteria. This method takes only one parameter that will be used to match the referenced entity attribute in the WHERE clause. The findByName() method returns a list of User entities as multiple users can have the same name.

Now if you run the application, the above-derived method will be translated to the following JPQL query:

SELECT u from User u WHERE u.name = ?1

Similarly, we can create simple queries for other User's entity attributes. The following query method returns a User instance matched with an email address:

Optional<User> findByEmail(String email);

## Derived Queries with Multiple Parameters

You can easily combine multiple condition expressions by using And and Or keywords. Let us combine the above two query methods to search for User entities with a given name or an email address:

List<User> findByNameOrEmail(String name, String email);

Another example of a derived query with multiple parameters that looks for User entities with a given name and age:

List<User> findByNameAndAge(String name, int age);

You can **concatenate as much expressions as you want** with And and Or keywords:

List<User> findByActiveAndBirthDateOrNameAndAge(boolean active,Date dob, String name, int age);

Spring Data JPA **does not explicitly limit** the number of expressions you can combine in a single derived query. However, you shouldn't go insane. Too long derived queries are hard to read and maintain. For complex use cases, you should rather use [custom queries](https://attacomsian.com/blog/spring-data-jpa-query-annotation).

## Equality Condition Keywords

For all derived queries we have written above, Spring Data JPA will generate a simple equals comparison. You can also specify other comparison operators by using equality keywords.

For an exact match, we only need to reference the entity attribute as we did above:

List<User> findByName(String name);

For more readability, you can add either Is or Equals:

List<User> findByNameIs(String name);

// OR

List<User> findByNameEquals(String name);

To express the inequality, you can use the IsNot keyword:

List<User> findByNameIsNot(String name);

// OR

List<User> findByNameNot(String name);

For the null value comparison, we cannot directly use the equal (=) operator. If you pass a null parameter value, Spring Data JPA won't through an exception. Instead, it will translate the query to an IS NULL SQL comparison.

To explicitly add the IS NULL or IS NOT NULL criteria to the query, you can use IsNull and IsNotNull respectively:

List<User> findByEmailIsNull();

List<User> findByEmailIsNotNull();

For boolean entity attributes, you can even go one step further and directly add True and False keywords for the equality conditions:

List<User> findByActiveTrue();

List<User> findByActiveFalse();

## Matching Condition Keywords

To check if the value of an entity attribute contains the given substring, we can use the matching condition keywords.

For example, we can find users whose names **start with a specific value** by using the StartingWith keyword:

List<User> findByNameStartingWith(String prefix);

// OR

List<User> findByNameIsStartingWith(String prefix);

// OR

List<User> findByNameStartsWith(String prefix);

Spring Data JPA will translate this to WHERE name LIKE 'prefix%' query.

For users whose names **end with a specific value**, you should use endingWith keyword that translates to WHERE name LIKE '%suffix':

List<User> findByNameEndingWith(String suffix);

Similarly, for users whose names **contain a specific value**, there exists a Containing keyword which is equivalent to WHERE name LIKE '%infix%':

List<User> findByNameContaining(String infix);

All the above three condition keywords automatically append the % operator to the parameter value. It is good enough for simple use cases.

For complex matching operation which includes several % operators, you should use the Like keyword instead:

List<User> findByEmailLike(String pattern);

The Like (or NotLike) keyword does not append the % operator to the argument. You have to explicitly define the matching pattern like below:

String pattern = "%atta%@gmail%";

List<User> users = userRepository.findByEmailLike(pattern);

## Comparison Condition Keywords

Spring Data JPA also provides keywords for comparison operations. For example, we can use LessThan and LessThanEqual keywords to compare the value of a specific property with the given value:

List<User> findByAgeLessThan(int age);

List<User> findByAgeLessThanEqual(int age);

For > and >= comparison operations, you should use GreaterThan and GreaterThanEqual keywords:

List<User> findByAgeGreaterThan(int age);

List<User> findByAgeGreaterThanEqual(int age);

To check if the value of an entity attribute is between two given values, Spring Data JPA provides Between keyword:

List<User> findByAgeBetween(int start, int end);

Two more interesting keywords, supported by the Spring Data JPA, are Before and After. We can use these keywords to find users who were born before or after a given date:

List<User> findByBirthDateBefore(Date before);

List<User> findByBirthDateAfter(Date after);

## Distinct and IgnoreCase Keywords

The Distinct keyword is used to enables the distinct flag for the query:

List<User> findDistinctByEmail(String email);

List<User> findDistinctPeopleByNameOrEmail(String name, String email);

The IgnoreCase keyword is used to enable case-insensitive search for a specific property:

List<User> findByNameIgnoreCase(String name);

To enable case-insensitive search for all suitable properties, you should use the AllIgnoreCase keyword:

List<User> findByNameOrEmailAllIgnoreCase(String name, String email);

## Sorting Derived Query Results

Spring Data JPA also allows us to enable **static ordering** by appending an OrderBy clause to the query method that references a property and by providing a sorting direction (Asc or Desc).

The following example uses static ordering to retrieve all User entities whose name contains a given value in the ascending order:

List<User> findByNameContainingOrderByName(String name);

// OR

List<User> findByNameContainingOrderByNameAsc(String name);

By default, the OrderBy clause sorts the results in the ascending order. But you can add Desc to reverse the sorting direction:

List<User> findByNameContainingOrderByNameDesc(String name);

If you need **dynamic ordering**, you can add a Sort parameter to your query method. This is one of the special parameters supported by Spring Data JPA. Sort triggers the generation of an ORDER BY clause. Here is an example:

List<User> findByNameContaining(String name, Sort sort);

To call the above method, you need to create a Sort object to specify the entity attributes and their ordering:

// sort users in ascending order

List<User> users = userRepository.findByNameContaining("john", Sort.by("name").descending());

// sort users in descending order

List<User> users = userRepository.findByNameContaining("john", Sort.by("name").descending());

// multiple sort parameters

List<User> users = userRepository.findByNameContaining("john", Sort.by("name", "age").descending());

## Limiting Derived Query Results

The results of derived query methods can be limited by using the First or Top keywords interchangeably. An optional numeric value can be added to Top or First to specify the maximum number of results to be returned. If the number is not provided, a result size of 1 is used.

The following examples show how to limit the derived query result size:

User findFirstByOrderByName();

User findTopByOrderByAgeDesc();

List<User> findFirst5ByEmail(String email);

List<User> findDistinctTop3ByAgeLessThan(int age);

## Paginate Derived Query Results

Spring Data JPA provides another special parameter Pageable to apply pagination to query results. You just need to add this special parameter to your query method definition and change the return type to Page<User>:

Page<User> findByActive(boolean active, Pageable pageable);

The Pageable interface makes it very easy to dynamically add paging to your statically defined query. A Page knows about the total number of elements and pages available.

You just define the page number you want to retrieve and how many records should be on a page. That’s it. Spring Data JPA will automatically create an appropriate paging query:

Pageable pageable = PageRequest.of(0, 10);

Page<User> userPage = userRepository.findByActive(true, pageable)

You can even add dynamic sorting to derived query through the Pageable instance:

Pageable pageable = PageRequest.of(0, 10, Sort.by("name").descending());

Page<User> userPage = userRepository.findByActive(true, pageable);

## Derived Delete Queries

Spring Data JPA also supports derived delete queries:

// delete all users by name

void deleteByName(String name);

// delete all active or inactive users

void deleteAllByActive(boolean active);

**Source code:** Download the complete source code from [GitHub](https://github.com/attacomsian/code-examples/tree/master/spring-data-jpa/jpa-queries) available under MIT license.

## Conclusion

That's all folks for creating and using derived queries in Spring Data JPA. You have learned the structure of derived query methods, JPA supported keywords, sorting, paginating, limiting the result size, and much more.

Spring Data JPA is a powerful tool that **provides an extra layer of abstraction** on top of an existing JPA provider like Hibernate. The derived query feature, which I just explain in this tutorial, is one of the most loved features of Spring Data JPA.

You just need to follow conventions while writing derived methods. As long as your query method name starts with find…By, read…By, query…By, count…By, or get…By and follows the right criteria pattern, Spring Data translates it to create the required JPQL query.

Check out the [official documentation](https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods) to learn more about all the available options.

**Continue reading** the next part of this tutorial that describes how to [create and use custom JPQL and native queries](https://attacomsian.com/blog/spring-data-jpa-query-annotation) with the @Query annotation.

# Accessing Data with Spring Data JPA and MySQL

October 03, 2019 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

[Spring Data JPA](https://attacomsian.com/blog/getting-started-spring-data-jpa) is a powerful tool for building Spring-based applications that use different data access technologies, relational and non-relational databases, map-reduce frameworks, and cloud-based data services.

In an [earlier article](https://attacomsian.com/blog/spring-data-jpa-h2-database), I talked about using Spring Data JPA with the H2 database in a Spring Boot application for **storing and accessing data from the in-memory store**.

In this article, you'll learn **how to use MySQL with Spring Data JPA in a Spring Boot application** for data persistence. We'll be building a simple note-taking application that uses Spring Data JPA to store and retrieve data in a relational database — MySQL.

## Dependencies

You only need spring-data-starter-data-jpa and mysql-connector-java dependencies to use MySQL database with Spring Data JPA.

If you are using Gradle, add the following dependencies to your build.gradle file:

implementation 'org.springframework.boot:spring-boot-starter-data-jpa'

runtimeOnly 'mysql:mysql-connector-java'

For a Maven project, include the following dependencies to the pom.xml file:

<dependency>

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

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

</dependency>

<dependency>

<groupId>mysql</groupId>

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

<scope>runtime</scope>

</dependency>

If you want to create a new Spring Boot project from scratch, just use [Spring Initializr](https://start.spring.io/) web tool to bootstrap a new application with the above dependencies.

## Configure MySQL Database

By default, Spring Boot **automatically configures the Hibernate as a JPA provider** and also tries to configure the data source object based on the dependencies available in the classpath and properties declared in the application.properties or application.yml file.

The classpath dependencies work well for in-memory databases like [H2](https://attacomsian.com/blog/spring-data-jpa-h2-database#configure-h2-database). But for MySQL, we need to explicitly define the database connection credentials in a properties file.

Just open the application.properties file and add the following properties into it:

# MySQL connection properties

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.username=root

spring.datasource.password=mypass

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

# Log JPA queries

# Comment this in production

spring.jpa.show-sql=true

# Drop and create new tables (create, create-drop, validate, update)

# Only for testing purpose - comment this in production

spring.jpa.hibernate.ddl-auto=create

# Hibernate SQL dialect

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

Spring Boot will read the above properties and will auto-configure a DataSource bean for you.

Alternatively, you can define the DataSource bean in a @Configuration annotated Java class like below:

@Bean

public DataSource dataSource() {

DriverManagerDataSource dataSource = new DriverManagerDataSource();

dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");

dataSource.setUsername("root");

dataSource.setPassword("mypass");

dataSource.setUrl("jdbc:mysql://localhost:3306/testdb?createDatabaseIfNotExist=true&useSSL=false");

return dataSource;

}

The above configuration will automatically create a MySQL database named **testdb**. Make sure to change the username and password properties to match your installation.

## Create an Entity

The next step is to create the Note entity for storing notes. This class has the following properties:

* id — Unique primary key with auto-increment enabled.
* title — The title of the Note entity.
* content — Note's body. Must not be null.
* created — The timestamp at which the Note object was created.

Now, let's add the Note entity with the following content:

**Note.java**

package com.attacomsian.jpa.domains;

import javax.persistence.\*;

import java.util.Date;

@Entity

@Table(name = "notes")

public class Note {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String title;

private String content;

private Date created;

public Note() {

}

public Note(String title, String content, Date created) {

this.title = title;

this.content = content;

this.created = created;

}

// getters and setters, equals(), toString() .... (omitted for brevity)

}

All Spring Data JPA models must be annotation with the @Entity annotation to indicate that they are JPA entities.

This time, we also annotated the Note class with the @Table annotation to specify the name of the database table that this entity will be mapped to.

The id attribute is annotated with both @Id and @GeneratedValue annotations. The former indicates that it is a primary key of the entity. The latter defines the primary key generation strategy. In the above case, we have declared the primary key as an AUTO INCREMENT field.

The other three properties, title, content and created, are left unannotated. This means they are mapped to database columns that share the same names as the properties themselves.

## Create a Repository

Let us create the NoteRepository interface to store and access data from MySQL database:

**NoteRepository.java**

package com.attacomsian.jpa.repositories;

import com.attacomsian.jpa.domains.Note;

import org.springframework.data.repository.CrudRepository;

import java.util.List;

public interface NoteRepository extends CrudRepository<Note, Long> {

List<Note> findByTitleContaining(String title);

}

NoteRepository extends the CrudRepository interface. The type of entity and ID that it needs, Note and Long, are specified in the generic parameters on CrudRepository.

We also defined an additional method findByTitleContaining() to get all notes that contain a specific keyword in their title value.

That's it. You can now use generic CRUD methods like save(), delete(), count(), findById(), and findAll() on NoteRepository.

The best thing is you **don't need to provide an implementation for these methods**. Spring Data JPA will automatically create an implementation when you run the application. That's what makes Spring Data JPA so powerful too to use for data access layers.

## Create an Application Class

The next step is to create the main application class that will act as an entry point for our [Spring Boot console application](https://attacomsian.com/blog/spring-boot-console-application):

**Application.java**

package com.attacomsian.jpa;

import com.attacomsian.jpa.domains.Note;

import com.attacomsian.jpa.repositories.NoteRepository;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.Bean;

import java.util.Date;

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

@Bean

public CommandLineRunner noteDemo(NoteRepository noteRepository) {

return (args) -> {

// create notes

noteRepository.save(new Note("Welcome to Spring Boot", "Basic Introduction", new Date()));

noteRepository.save(new Note("Learn Spring Data JPA", "Java Persistence Layer", new Date()));

noteRepository.save(new Note("Learn Spring Security", "Build Secure Web Apps", new Date()));

// fetch all notes

System.out.println("Notes found with findAll():");

System.out.println("---------------------------");

for (Note note : noteRepository.findAll()) {

System.out.println(note.toString());

}

System.out.println();

// fetch note by id

Note note = noteRepository.findById(1L).get();

System.out.println("Note found with findById(1L):");

System.out.println("-----------------------------");

System.out.println(note.toString());

System.out.println();

// fetch all notes that contain keyword `learn`

System.out.println("Notes that contain keyword 'learn':");

System.out.println("-----------------------------------");

for (Note n : noteRepository.findByTitleContaining("learn")) {

System.out.println(n.toString());

}

System.out.println();

// update note title

Note noteUpdate = noteRepository.findById(2L).get();

noteUpdate.setTitle("Understanding Spring Data JPA");

noteRepository.save(noteUpdate);

System.out.println("Update note title:");

System.out.println("------------------");

System.out.println(noteUpdate.toString());

System.out.println();

// total notes in DB

System.out.println("Total notes in DB:");

System.out.println("------------------");

System.out.println(noteRepository.count());

System.out.println();

// delete all notes

noteRepository.deleteAll();

};

}

}

The above code is very much self-explanatory. The Application class is annotated with @SpringBootApplication to enable the auto configurations. The main() method calls Spring Boot’s SpringApplication.run() method to start the application.

We are auto wiring NoteRepository to the noteDemo() method. Inside this method, we first save a few Note objects through the save() method. Next, we call the fetchAll() method to get all Note objects from the MySQL database.

Later, we get a single Note object from the database by its ID using findById(). Finally, we call the findByTitleContaining() method to fetch all notes which contain the keyword learn in their titles.

## Run the Application

Let us run the application to see the output. If you are using Gradle, execute the following command to start the application:

$ ./gradlew bootRun

For Maven, type the following command to launch the application:

$ ./mvnw spring-boot:run

You should see the following output printed on the console:

Notes found with findAll():

---------------------------

Note{id=1, title='Welcome to Spring Boot', content='Basic Introduction', created=2019-10-03 13:48:52.0}

Note{id=2, title='Learn Spring Data JPA', content='Java Persistence Layer', created=2019-10-03 13:48:52.0}

Note{id=3, title='Learn Spring Security', content='Build Secure Web Apps', created=2019-10-03 13:48:52.0}

Note found with findById(1L):

-----------------------------

Note{id=1, title='Welcome to Spring Boot', content='Basic Introduction', created=2019-10-03 13:48:52.0}

Notes that contain keyword 'learn':

-----------------------------------

Note{id=2, title='Learn Spring Data JPA', content='Java Persistence Layer', created=2019-10-03 13:48:52.0}

Note{id=3, title='Learn Spring Security', content='Build Secure Web Apps', created=2019-10-03 13:48:52.0}

Update note title:

------------------

Note{id=2, title='Understanding Spring Data JPA', content='Java Persistence Layer', created=2019-10-03 13:48:52.0}

Total notes in DB:

------------------

3

**Source code:** Download the complete source code from [GitHub](https://github.com/attacomsian/code-examples/tree/master/spring-data-jpa/jpa-mysql) available under MIT license.

## Conclusion

Congratulations! You have successfully created a simple Spring Boot application that uses Spring Data JPA with MySQL database to store, access, delete and update data. In short, it performs all CRUD operations.

Check out the [getting started with Spring Data JPA](https://attacomsian.com/blog/getting-started-spring-data-jpa) guide to learn more about the core features and configuration instructions for different data sources.

# How to Schedule Tasks in Spring Boot

August 20, 2019 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

Task scheduling is frequently used in web applications to execute different jobs at certain times without any manual input. Examples include backing up the database, sending newsletter emails, deleting log files, moving files from one server to another server, just to name a few.

Spring Boot provides multiple ways to schedule tasks. You can either use @Scheduled annotation or use a custom thread pool to run your tasks at specific times.

In this article, we will learn how to schedule tasks in a Spring Boot application using @Scheduled annotation. We will also take a look at using a custom thread pool to execute scheduled tasks.

## Dependencies

We only need spring-boot-starter dependency to use the @Scheduled annotation. Add the following to your build.gradle file:

implementation 'org.springframework.boot:spring-boot-starter'

If you are using Maven, add the following dependency to your pom.xml file:

<dependency>

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

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

</dependency>

## Enable Scheduling

We can enable the scheduling functionality by adding @EnableScheduling annotation to the main application class or any other configuration class like below:

**Application.java**

package com.attacomsian.scheduling;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.scheduling.annotation.EnableScheduling;

@SpringBootApplication

@EnableScheduling

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

@EnableScheduling annotation ensures that a background task executor is created with a single thread.

## Scheduling Tasks

Scheduling a task is easy. Simply add the @Scheduled annotation to any method you want to schedule and set the time it should execute. However, all such methods must meet the following two conditions:

1. The method return type should be void (void)
2. The method should not accept any parameter

That being said, let us create a Java class that will act as a container to hold all our scheduled tasks:

**ScheduledTasks.java**

package com.attacomsian.scheduling;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.stereotype.Component;

import java.time.format.DateTimeFormatter;

@Component

public class ScheduledTasks {

private static final Logger logger = LoggerFactory.getLogger(ScheduledTasks.class);

private static final DateTimeFormatter formatter = DateTimeFormatter.ofPattern("HH:mm:ss");

// add scheduled methods here

}

## Schedule Task with Fixed Rate

Let us schedule our first task that executes at a fixed interval of time by using fixedRate property in the @Scheduled annotation:

@Scheduled(fixedRate = 2500)

public void scheduleTaskWithFixedRate() {

logger.info("Fixed Rate Task: Current Time - {}", formatter.format(LocalDateTime.now()));

}

The above task is scheduled to execute every 3 seconds and outputs the following on the console:

Fixed Rate Task: Current Time - 13:36:53

Fixed Rate Task: Current Time - 13:36:56

Fixed Rate Task: Current Time - 13:36:58

Fixed Rate Task: Current Time - 13:37:01

...

The fixedRate task is invoked for every specified interval. The specified interval between method invocations is measured from the start time of each invocation. In other words, the task is executed again even if the previous invocation of the task is not completed.

This option is **best suited when each execution of the task is independent**. The fixedRate property executes the task at every n milliseconds. It does not wait for any previous execution to finish.

## Schedule Task with Fixed Delay

Very similar to fixedRate, the fixedDelay task is invoked for every specified interval but the time is measured from the completion time of each preceding invocation.

In short, the time between the end of the last execution and the start of the next execution is constant. The task always waits until the previous one is completed.

Consider the following example:

@Scheduled(fixedDelay = 2000)

public void scheduleTaskWithFixedDelay() throws InterruptedException {

logger.info("Fixed Delay Task: Start Time - {}", formatter.format(LocalDateTime.now()));

// add some virtual processing time

TimeUnit.SECONDS.sleep(3);

logger.info("Fixed Delay Task: End Time - {}", formatter.format(LocalDateTime.now()));

}

The task is scheduled to execute every 2 seconds and prints the start and finish times of the execution. Since we have added 3 seconds of virtual processing time, it will take at least 3 seconds to complete. There will be a delay of 5 seconds between successive invocations:

Fixed Delay Task: Start Time - 14:02:28

Fixed Delay Task: End Time - 14:02:31

Fixed Delay Task: Start Time - 14:02:33

Fixed Delay Task: End Time - 14:02:36

Fixed Delay Task: Start Time - 14:02:38

Fixed Delay Task: End Time - 14:02:41

Fixed Delay Task: Start Time - 14:02:43

Fixed Delay Task: End Time - 14:02:46

...

This option is **best suited when the previous execution of the task must be completed** before executing it again. The fixedDelay property makes sure that is always a delay of n milliseconds between consecutive invocations of a task. For dependent tasks, it is pretty helpful.

## Schedule Task with Initial Delay

You can also specify the initial time to wait (in milliseconds) before the first execution of the task begins by using the initialDelay property. It works with both fixedRate and fixedDelay properties.

In the following example, the task is executed the first time after a wait of 5 seconds and then it executes normally after every 2 seconds:

@Scheduled(fixedRate = 2000, initialDelay = 5000)

public void scheduleTaskWithFixedRateAndInitialDelay() {

logger.info("Fixed Rate Task with Initial Delay: Current Time - {}", formatter.format(LocalDateTime.now()));

}

The initialDelay property delays the first execution of the task for the specified milliseconds. After the first execution, the task starts executing normally:

# Server started at 14:42:20

Fixed Rate Task with Initial Delay: Current Time - 14:42:25

Fixed Rate Task with Initial Delay: Current Time - 14:42:27

Fixed Rate Task with Initial Delay: Current Time - 14:42:29

Fixed Rate Task with Initial Delay: Current Time - 14:42:31

...

The initialDelay property helps **delay the first execution of the task until the data required to execute the tasks is provided** by some other services.

## Schedule Task using Cron Expression

Sometimes the fixed-rate and fixed-delay are not enough to fulfill our needs. We want more flexibility to control the schedule of our tasks. For example, we might want to send a newsletter every Thursday or back up our database every week. This is something that cannot be done with the above properties.

That is where the [cron expressions](https://en.wikipedia.org/wiki/Cron" \t "_blank" \o "Open in new window) come handy. They provide complete flexibility to schedule the tasks, whatever the way you want to choose.

Here is an example task that uses cron expression to execute every minute:

@Scheduled(cron = "0 \* \* \* \* ?")

public void scheduleTaskWithCronExpression() {

logger.info("Cron Task: Current Time - {}", formatter.format(LocalDateTime.now()));

}

Cron Task: Current Time - 15:17:00

Cron Task: Current Time - 15:18:00

Cron Task: Current Time - 15:19:00

Cron Task: Current Time - 15:20:00

...

Let us have a few more examples of cron expressions. The following tasks are scheduled to be executed at 4:45 AM on the 10th day of every month:

@Scheduled(cron = "0 45 4 10 \* ?")

public void scheduleTaskWithCronExpression2() {

logger.info("Cron Task: Current Time - {}", formatter.format(LocalDateTime.now()));

}

Cron expressions are somehow complex to write and understand. [crontab.guru](https://crontab.guru/" \t "_blank" \o "Open in new window) is a nice little tool that makes it easy to generate cron schedule expressions:

# At 12:00 on Sunday

0 0 12 \* \* Sun

# At 14:15 in every 2nd month

0 15 14 \* \*/2 \*

# At 08:00 on every day-of-week from Monday through Friday

0 0 8 \* \* Mon-Fri

# At 12:30 on every 15th day-of-month if it's on Wednesday

0 30 12 \*/15 \* Wed

## Parameterizing the Schedule

In the above examples, we have hard-coded the time intervals and cron expressions. Now if you want to change the execution time of any task, we have to recompile and redeploy the entire application. This is certainly not flexible.

Fortunately, we can make use of Spring Expression Language (SpPL) and store the configuration of the tasks in a properties file:

@Scheduled(fixedRateString = "${fixed-rate.in.milliseconds}")

public void scheduleDynamicTaskWithFixedRate() {

logger.info("Fixed Rate Dynamic Task: Current Time - {}", formatter.format(LocalDateTime.now()));

}

@Scheduled(fixedDelayString = "${fixed-delay.in.milliseconds}")

public void scheduleDynamicTaskWithFixedDelay() {

logger.info("Fixed Delay Dynamic Task: Current Time - {}", formatter.format(LocalDateTime.now()));

}

@Scheduled(cron = "${cron.expression}")

public void scheduleDynamicTaskWithCronExpression() {

logger.info("Cron Dynamic Task: Current Time - {}", formatter.format(LocalDateTime.now()));

}

And our application.properties file will look like this:

**application.properties**

fixed-rate.in.milliseconds=5000

fixed-delay.in.milliseconds=4000

cron.expression=0 15 5 \* \* FRI

## Custom Thread Pool Configuration

By default, @EnableScheduling annotation creates a thread pool with only one thread. The invocation of all @Scheduled tasks is queued and executed by an only thread. So if you have multiple scheduled tasks in your application, you might see weird behavior of invocation (since the tasks are queued).

But the good thing is you can create your own custom thread pool with multiple threads and configure the application to use that for executing all scheduled tasks:

**SchedulerConfig.java**

package com.attacomsian.scheduling;

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

import org.springframework.scheduling.annotation.SchedulingConfigurer;

import org.springframework.scheduling.concurrent.ThreadPoolTaskScheduler;

import org.springframework.scheduling.config.ScheduledTaskRegistrar;

public class SchedulerConfig implements SchedulingConfigurer {

@Value("${thread.pool.size}")

private int POOL\_SIZE;

@Override

public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {

ThreadPoolTaskScheduler scheduler = new ThreadPoolTaskScheduler();

scheduler.setPoolSize(POOL\_SIZE);

scheduler.setThreadNamePrefix("my-scheduled-task-pool-");

scheduler.initialize();

taskRegistrar.setTaskScheduler(scheduler);

}

}

Add the following property to your application.properties file:

thread.pool.size=10

Spring will now create a custom thread pool with 10 threads to execute the tasks. You can also find the thread name in log that was invoked to execute the task:

[ My-Scheduler-4] : Fixed Delay Dynamic Task: Current Time - 17:20:03

[ My-Scheduler-8] : Fixed Rate Dynamic Task: Current Time - 17:20:04

[ My-Scheduler-1] : Fixed Delay Dynamic Task: Current Time - 17:20:07

[ My-Scheduler-7] : Fixed Rate Dynamic Task: Current Time - 17:20:09

[ My-Scheduler-2] : Fixed Delay Dynamic Task: Current Time - 17:20:11

...

**Source code:** Download the complete source code from [GitHub](https://github.com/attacomsian/code-examples/tree/master/spring-boot/scheduling) available under MIT license.

## Conclusion

That's all folks for task scheduling in Spring Boot. We have learned how to schedule tasks in Spring Boot using @Scheduled annotation. We also learned to create and use a custom thread pool for executing these tasks.

Task scheduling is much valuable to automate complex tasks that would have taken a lot of time doing manually. It saves time that can be spent on other productive things instead of worrying about routine tasks.

# Uploading and downloading files using Spring Boot REST API

March 21, 2019 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

Handling uploading and downloading files are very common jobs in most of the web applications. Spring Boot provides the MultipartFile interface to handle HTTP multi-part requests for uploading files.

In this tutorial, we will learn the following:

* Create a Spring Boot web application that allows file uploads
* Upload single and multiple files using RESTful web services
* Download file using RESTful web service
* List all files uploaded on the server
* A simple [Thymeleaf](https://attacomsian.com/blog/spring-boot-thymeleaf-example) & HTML web interface to upload file(s) from browser

Tools you need to complete this tutorial:

* [Java 8+](https://attacomsian.com/blog/install-java-on-ubuntu)
* JDK 1.8+
* [Spring Boot](https://attacomsian.com/blog/scaffolding-spring-boot-application)
* Thymeleaf
* [Gradle 4+](https://attacomsian.com/blog/install-gradle-on-ubuntu)
* Postman (optional for testing RESTful APIs)

**Note:** This article uses RESTful web services to upload and download files in Spring Boot. If you are using Thymeleaf and want to upload a file, check out [this guide](https://attacomsian.com/blog/spring-boot-thymeleaf-file-upload).

## Project Dependencies

We only need spring-boot-starter-web and spring-boot-starter-thymeleaf [starter dependencies](https://attacomsian.com/blog/spring-boot-starters) for our example Spring Boot project. We do not need any extra dependency for file upload. Here is how our build.gradle file looks like:

**build.gradle**

plugins {

id 'org.springframework.boot' version '2.1.3.RELEASE'

id 'java'

}

apply plugin: 'io.spring.dependency-management'

group = 'com.attacomsian'

version = '0.0.1-SNAPSHOT'

sourceCompatibility = '1.8'

repositories {

mavenCentral()

}

dependencies {

implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'

implementation 'org.springframework.boot:spring-boot-starter-web'

}

I used [Spring Initializr](https://start.spring.io/) to generate the above Gradle configuration file. It is an easier and quicker way to create a Spring Boot application.

## Configure Properties

Before we start the actual work, let's first configure the location on the server where all the uploaded files will be stored. We'll also configure the maximum file size that can be uploaded in a single HTTP multi-part request. Spring Boot automatically enables multipart/form-data requests, so we do not need to do anything.

**application.properties**

# max file size

spring.servlet.multipart.max-file-size=10MB

# max request size

spring.servlet.multipart.max-request-size=10MB

# files storage location (stores all files uploaded via REST API)

storage.location=./uploads

In above properties file, we have two multi-part settings:

* spring.servlet.multipart.max-file-size is set to 10MB, which means total files size cannot exceed 10MB.
* spring.servlet.multipart.max-request-size sets the maximum multipart/form-data request size to 10MB.

In simple words, we cannot upload files greater than 10MB in size given the above configuration.

## Enable Configuration Properties

In our application.properties file, we define the storage location. Now let's create a POJO class called StorageProperties and annotate it with @ConfigurationProperties to automatically bind the properties defined in application.properties file.

**StorageProperties.java**

package com.attacomsian.uploadfiles.storage;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties(prefix = "storage")

public class StorageProperties {

private String location;

public String getLocation() {

return location;

}

public void setLocation(String location) {

this.location = location;

}

}

Notice the prefix= "storage" attribute in the above annotation. It instructs @ConfigurationProperties to bind all the properties that start with storage prefix to their corresponding attributes of POJO class when the application is started.

The next step is to enable the ConfigurationProperties feature by adding @EnableConfigurationProperties annotation to our main configuration class.

**Application.java**

package com.attacomsian.uploadfiles;

import com.attacomsian.uploadfiles.storage.StorageProperties;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.boot.context.properties.EnableConfigurationProperties;

@SpringBootApplication

@EnableConfigurationProperties(StorageProperties.class)

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

## Files Upload Controller

Let's now create a controller class called FileController for handling uploading and downloading files via RESTful web services. It also defines a route to list all the uploaded files.

**FileController.java**

package com.attacomsian.uploadfiles.controllers;

import com.attacomsian.uploadfiles.commons.FileResponse;

import com.attacomsian.uploadfiles.storage.StorageService;

import org.springframework.core.io.Resource;

import org.springframework.http.HttpHeaders;

import org.springframework.http.ResponseEntity;

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

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

import org.springframework.web.multipart.MultipartFile;

import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

@Controller

public class FileController {

private StorageService storageService;

public FileController(StorageService storageService) {

this.storageService = storageService;

}

@GetMapping("/")

public String listAllFiles(Model model) {

model.addAttribute("files", storageService.loadAll().map(

path -> ServletUriComponentsBuilder.fromCurrentContextPath()

.path("/download/")

.path(path.getFileName().toString())

.toUriString())

.collect(Collectors.toList()));

return "listFiles";

}

@GetMapping("/download/{filename:.+}")

@ResponseBody

public ResponseEntity<Resource> downloadFile(@PathVariable String filename) {

Resource resource = storageService.loadAsResource(filename);

return ResponseEntity.ok()

.header(HttpHeaders.CONTENT\_DISPOSITION,

"attachment; filename=\"" + resource.getFilename() + "\"")

.body(resource);

}

@PostMapping("/upload-file")

@ResponseBody

public FileResponse uploadFile(@RequestParam("file") MultipartFile file) {

String name = storageService.store(file);

String uri = ServletUriComponentsBuilder.fromCurrentContextPath()

.path("/download/")

.path(name)

.toUriString();

return new FileResponse(name, uri, file.getContentType(), file.getSize());

}

@PostMapping("/upload-multiple-files")

@ResponseBody

public List<FileResponse> uploadMultipleFiles(@RequestParam("files") MultipartFile[] files) {

return Arrays.stream(files)

.map(file -> uploadFile(file))

.collect(Collectors.toList());

}

}

As always, our controller class is annotated with @Controller to let the Spring MVC pick it up for routes. Each method is decorated with @GetMapping or @PostMapping to bind the path and the HTTP action with that particular method.

* GET / loads the current list of uploaded files and renders it into a Thymeleaf template called listFiles.html.
* POST /download/{filename} resolves the resource if it exists, and sends it to the browser for download. HttpHeaders.CONTENT\_DISPOSITION adds the "Content-Disposition" response header to indicate file attachment.
* POST /upload-file & /upload-multiple-files routes handle HTTP multi-part requests and use StorageService for saving files on the server. Both these methods return an object of FileResponse after the upload is finished.

The FileResponse class is used to return a [JSON response](https://attacomsian.com/blog/processing-json-spring-boot) for RESTful web services.

**FileResponse.java**

package com.attacomsian.uploadfiles.commons;

public class FileResponse {

private String name;

private String uri;

private String type;

private long size;

public FileResponse(String name, String uri, String type, long size) {

this.name = name;

this.uri = uri;

this.type = type;

this.size = size;

}

// getters and setters removed for the sake of brevity

}

The FileController class uses the StorageService interface for storing and resolving files in the file system. It is the most important class for handling files in our example. We'll define these classes in the next section.

In production, it's not advised to store the uploaded files in your application file system. You might lose all files if your application server is damaged. It also makes very difficult to move the application from one server to another. Therefore, it is a good practice to use external storage like AWS S3 for storing all the uploaded files. I'll write about this topic in the future.

## Storage Service

Finally, it is time to create a storage service called StorageService for our controller to connect with a storage layer (e.g. file system in our case). This task involves several classes. We'll define these classes one-by-one.

The first step is to define an interface called StorageService as shown below:

**StorageService.java**

package com.attacomsian.uploadfiles.storage;

import org.springframework.core.io.Resource;

import org.springframework.web.multipart.MultipartFile;

import java.nio.file.Path;

import java.util.stream.Stream;

public interface StorageService {

void init();

String store(MultipartFile file);

Stream<Path> loadAll();

Path load(String filename);

Resource loadAsResource(String filename);

void deleteAll();

}

The above interface declares several abstract methods for initializing, storing, removing and retrieving files. It only lists possible storage operations without their implementation. Now, it is up to you to decide how you want to implement them. In this example, we will use our file system for handling files. It can also be implemented to store the files on any external location.

Let's create a concrete class FileSystemStorageService that implements the StorageService interface.

**FileSystemStorageService.java**

package com.attacomsian.uploadfiles.storage;

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

import org.springframework.core.io.Resource;

import org.springframework.core.io.UrlResource;

import org.springframework.stereotype.Service;

import org.springframework.util.FileSystemUtils;

import org.springframework.util.StringUtils;

import org.springframework.web.multipart.MultipartFile;

import javax.annotation.PostConstruct;

import java.io.IOException;

import java.io.InputStream;

import java.net.MalformedURLException;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.Paths;

import java.nio.file.StandardCopyOption;

import java.util.stream.Stream;

@Service

public class FileSystemStorageService implements StorageService {

private final Path rootLocation;

@Autowired

public FileSystemStorageService(StorageProperties properties) {

this.rootLocation = Paths.get(properties.getLocation());

}

@Override

@PostConstruct

public void init() {

try {

Files.createDirectories(rootLocation);

} catch (IOException e) {

throw new StorageException("Could not initialize storage location", e);

}

}

@Override

public String store(MultipartFile file) {

String filename = StringUtils.cleanPath(file.getOriginalFilename());

try {

if (file.isEmpty()) {

throw new StorageException("Failed to store empty file " + filename);

}

if (filename.contains("..")) {

// This is a security check

throw new StorageException(

"Cannot store file with relative path outside current directory "

+ filename);

}

try (InputStream inputStream = file.getInputStream()) {

Files.copy(inputStream, this.rootLocation.resolve(filename),

StandardCopyOption.REPLACE\_EXISTING);

}

}

catch (IOException e) {

throw new StorageException("Failed to store file " + filename, e);

}

return filename;

}

@Override

public Stream<Path> loadAll() {

try {

return Files.walk(this.rootLocation, 1)

.filter(path -> !path.equals(this.rootLocation))

.map(this.rootLocation::relativize);

}

catch (IOException e) {

throw new StorageException("Failed to read stored files", e);

}

}

@Override

public Path load(String filename) {

return rootLocation.resolve(filename);

}

@Override

public Resource loadAsResource(String filename) {

try {

Path file = load(filename);

Resource resource = new UrlResource(file.toUri());

if (resource.exists() || resource.isReadable()) {

return resource;

}

else {

throw new FileNotFoundException(

"Could not read file: " + filename);

}

}

catch (MalformedURLException e) {

throw new FileNotFoundException("Could not read file: " + filename, e);

}

}

@Override

public void deleteAll() {

FileSystemUtils.deleteRecursively(rootLocation.toFile());

}

}

The above implementation class is taken from Spring Boot [official files uploading example](https://github.com/spring-guides/gs-uploading-files) with few modifications done by me. The important change I made is the addition of @PostConstruct annotation on the init() method. It guarantees that the init() method is only called once the bean is fully initialized with all the dependencies injected.

The FileSystemStorageService class throws exceptions in case of unexpected scenarios, for example, the file requested by the user might not exist.

The first exception is StorageException which is thrown when we are unable to create the storage directory or the uploaded file is empty etc.

**StorageException.java**

package com.attacomsian.uploadfiles.storage;

public class StorageException extends RuntimeException {

public StorageException(String message) {

super(message);

}

public StorageException(String message, Throwable cause) {

super(message, cause);

}

}

The FileNotFoundException exception is thrown when a file is requested by the user but it does not exist on the server.

**FileNotFoundException.java**

package com.attacomsian.uploadfiles.storage;

import org.springframework.http.HttpStatus;

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

@ResponseStatus(HttpStatus.NOT\_FOUND)

public class FileNotFoundException extends StorageException {

public FileNotFoundException(String message) {

super(message);

}

public FileNotFoundException(String message, Throwable cause) {

super(message, cause);

}

}

Notice the @ResponseStatus(HttpStatus.NOT\_FOUND) annotation above. This annotation ensures that Spring Boot responds with a 404 (Not Found) HTTP status instead of 501 (Internal Server Error) when the exception is thrown.

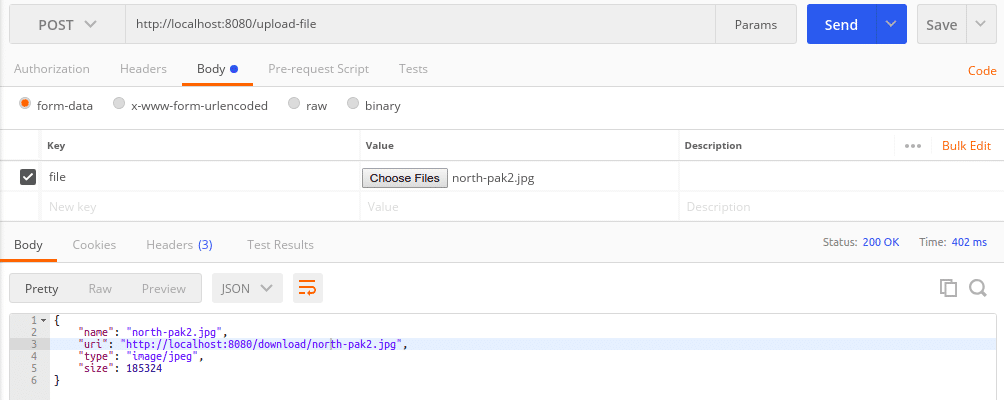
## Running & Testing the Application

We are almost done with our backend development. Since we created RESTful APIs for uploading and downloading files, we can test them via Postman. Let's run the application by typing the following command in your terminal from the root directory of the project:

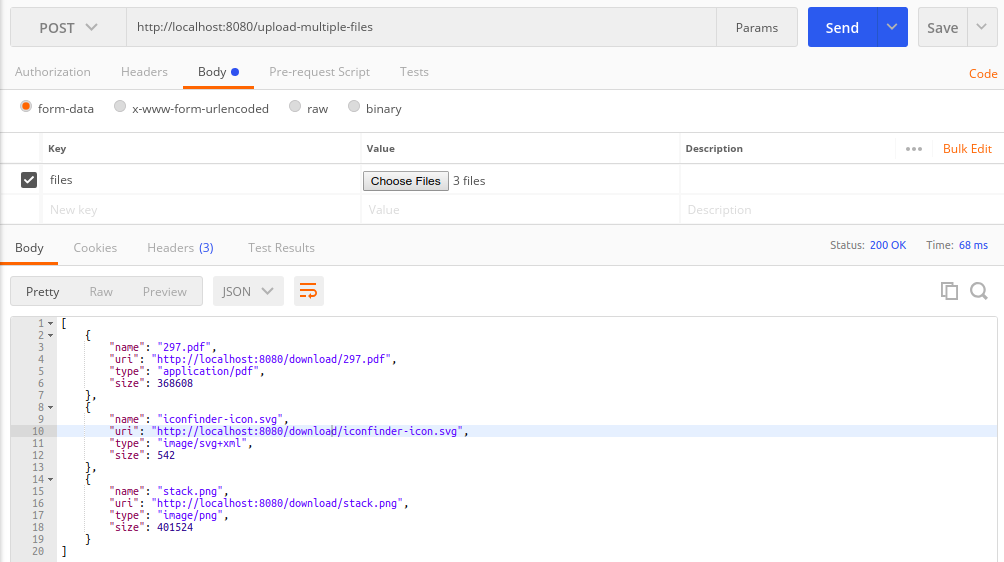
$ ./gradlew bootRun

Once the application is started, you can access it at [http://localhost:8080](http://localhost:8080/).

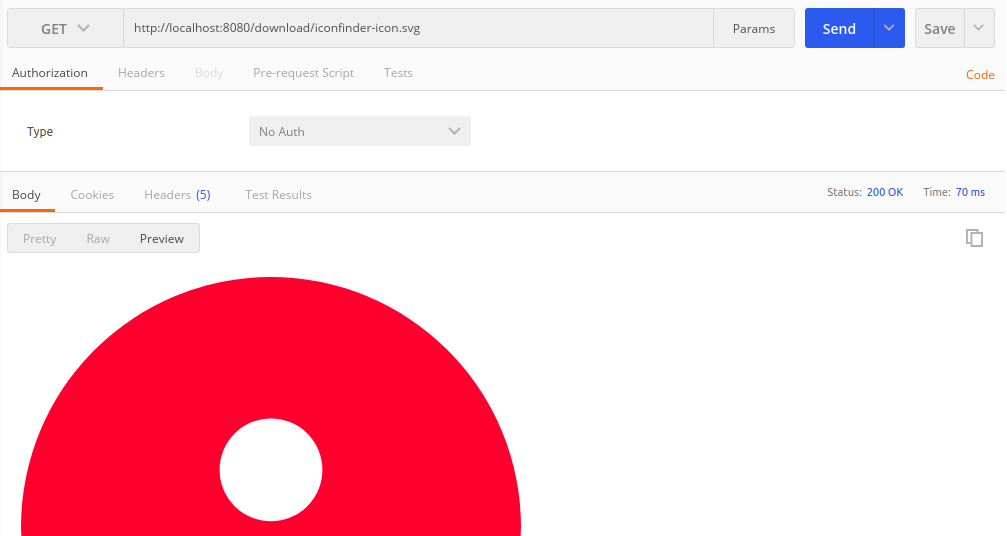
### 1. Upload Single File



### 2. Upload Multiple Files



### 3. Download File



## HTML Web Form

We have tested our RESTful APIs and they are working fine. Now it is time to create a simple front-end interface using [HTML & Thymeleaf](https://attacomsian.com/blog/spring-boot-thymeleaf-form-handling) that lists all the files uploaded so far. It will also allow users to upload files directly from the browser.

**listFiles.html**

<!doctype html>

<html lang="en" xmlns:th="http://www.thymeleaf.org">

<body>

<h1>Spring Boot File Upload Example</h1>

<hr/>

<h4>Upload Single File:</h4>

<form method="POST" enctype="multipart/form-data" th:action="@{/upload-file}">

<input type="file" name="file"> <br/><br/>

<button type="submit">Submit</button>

</form>

<hr/>

<h4>Upload Multiple Files:</h4>

<form method="POST" enctype="multipart/form-data" th:action="@{/upload-multiple-files}">

<input type="file" name="files" multiple> <br/><br/>

<button type="submit">Submit</button>

</form>

<hr/>

<h2>All Uploaded Files:</h2>

<ul>

<li th:each="file : ${files}">

<a th:href="${file}" target="\_blank" th:text="${file}"></a>

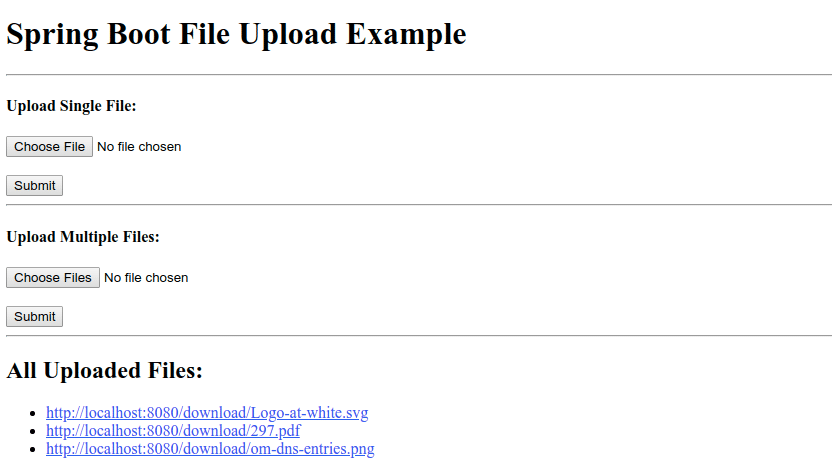
</li>

</ul>

</body>

</html>

The above template has two forms that enable users to upload a single file as well as multiple files. At the bottom, it also shows a list of currently uploaded files on the server. Here is how it looks like:



**Source code:** Download the complete source code from [GitHub](https://github.com/attacomsian/code-examples/tree/master/spring-boot/upload-files) available under MIT license.

## Conclusion

That's all folks for uploading and downloading files in Spring Boot. We discussed strategies for handling single as well as multiple files via RESTful web services. We tested our REST APIs via Postman to confirm that they are working as expected. Finally, we created the simplest web interface in HTML and Thymeleaf for showing a list of all the uploaded files.

In the end, I really appreciate that you read this article and hope that you'd have learned how to handle files in Spring Boot today. If you have any questions or feedback, please feel free to send me a [tweet](https://twitter.com/attacomsian).

# Export & Download Data as CSV File in Spring Boot

March 27, 2019 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

A Comma-Separated Values (CSV) file is just a simple plain text file that uses a comma as a delimiter to separate values. It stores data in a tabular format where each row consists of one or more fields and each column represents a specific field. These fields are separated by commas, one of the reasons why these files are called comma-separated values.

CSV is a widely used data exchange format in the industry due to its simplicity and better integration with existing applications. These files are usually used for exporting and importing large data sets.

In this tutorial, we will learn how to export and download the data as a CSV file in a Spring Boot project. Data export (JSON, CSV, PDF, etc.) is a very common feature implemented in many Java enterprise applications.

## Project Dependencies

Since Java does not provide native support for creating and parsing CSV files, we will use [OpenCSV](http://opencsv.sourceforge.net/" \t "_blank" \o "Open in new window) 3rd-party library for this purpose. Here is how our build.gradle file looks like:

**build.gradle**

plugins {

id 'org.springframework.boot' version '2.1.3.RELEASE'

id 'java'

}

apply plugin: 'io.spring.dependency-management'

group = 'com.attacomsian'

version = '0.0.1-SNAPSHOT'

sourceCompatibility = '1.8'

repositories {

mavenCentral()

}

dependencies {

implementation 'org.springframework.boot:spring-boot-starter-web'

implementation 'com.opencsv:opencsv:4.5'

}

If you are working with a maven project, make sure you include the following maven dependency to the project's pom.xml file:

<dependency>

<groupId>com.opencsv</groupId>

<artifactId>opencsv</artifactId>

<version>4.5</version>

</dependency>

## User Model

Here is our User model class that will be used to write to a CSV file.

**User.java**

package com.attacomsian.exportcsv.data;

import com.opencsv.bean.CsvBindByName;

import com.opencsv.bean.CsvBindByPosition;

public class User {

private long id;

private String name;

private String email;

private String country;

private int age;

public User(long id, String name, String email, String country, int age) {

this.id = id;

this.name = name;

this.email = email;

this.country = country;

this.age = age;

}

// getters and setters removed for the sake of brevity

}

Since we want to generate a CSV file from a list of users and then return it back to the client for downloading, let's create a dummy service that acts as a data source and returns a list of users.

**UserService.java**

package com.attacomsian.exportcsv.data;

import org.springframework.stereotype.Service;

import java.util.ArrayList;

import java.util.List;

@Service

public class UserService {

public List<User> listUsers() {

List<User> users = new ArrayList<>();

//create dummy users

users.add(new User(1, "Jack Lee", "jack@example.com", "Germany", 35));

users.add(new User(2, "Jovan Srovoki", "jovan@srovoki.me", "Russia", 21));

users.add(new User(3, "Atta", "atta@gmail.com", "Pakistan", 29));

return users;

}

}

The UserService above is just for demo purposes. You may want to populate a list of users from the database or from any other source.

## Generate & Download CSV File

Finally, below is a Spring MVC controller class that handles the export and download of data as a CSV file.

**UserController.java**

package com.attacomsian.exportcsv.controllers;

import com.attacomsian.exportcsv.data.User;

import com.attacomsian.exportcsv.data.UserService;

import com.opencsv.CSVWriter;

import com.opencsv.bean.ColumnPositionMappingStrategy;

import com.opencsv.bean.StatefulBeanToCsv;

import com.opencsv.bean.StatefulBeanToCsvBuilder;

import org.springframework.http.HttpHeaders;

import org.springframework.stereotype.Controller;

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

import javax.servlet.http.HttpServletResponse;

@Controller

public class UserController {

private UserService userService;

public UserController(UserService userService) {

this.userService = userService;

}

@GetMapping("/export-users")

public void exportCSV(HttpServletResponse response) throws Exception {

//set file name and content type

String filename = "users.csv";

response.setContentType("text/csv");

response.setHeader(HttpHeaders.CONTENT\_DISPOSITION,

"attachment; filename=\"" + filename + "\"");

//create a csv writer

StatefulBeanToCsv<User> writer = new StatefulBeanToCsvBuilder<User>(response.getWriter())

.withQuotechar(CSVWriter.NO\_QUOTE\_CHARACTER)

.withSeparator(CSVWriter.DEFAULT\_SEPARATOR)

.withOrderedResults(false)

.build();

//write all users to csv file

writer.write(userService.listUsers());

}

}

The above UserController class contains an exportCSV() method that is mapped to /export-users HTTP route and returns a CSV file as attachment for browser to download. This method does the following:

* Set the response's content type to text/csv.
* Use HttpHeaders.CONTENT\_DISPOSITION to add "Content-Disposition" response header that indicates file attachment to the browser. It also sets the attachment file name to users.csv
* Use response writer (response.writer() returns an object of type PrintWriter) to build an instance of StatefulBeanToCsv.
* Use write() method of StatefulBeanToCsv instance to write a list of users to CSV file.
* Afterwards, the data is pushed to the client where the browser downloads the attached users.csv file.

## Running the Application

Below is the main application class used for running Spring Boot project:

**Application.java**

package com.attacomsian.exportcsv;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

Let's run the application by typing the following command in your terminal from the root directory of the project:

$ ./gradlew bootRun

After Spring Boot application is started, open <http://localhost:8080/export-users> link in your favorite browser to generate and download users.csv file. Here is content of the generated CSV file:

**users.csv**

age,country,email,id,name

35,Germany,jack@example.com,1,Jack Lee

21,Russia,jovan@srovoki.me,2,Jovan Srovoki

29,Pakistan,atta@gmail.com,3,Atta

Notice the first line. OpenCSV automatically generated column headers using User class members. Another important thing to note is the order of the columns in the CSV file. OpenCSV sorts the column names in ascending order before writing into the CSV file.

## OpenCSV Columns Ordering

There is no built-in functionality in OpenCSV that allows writing bean to CSV with custom column names and ordering. However, using @CsvBindByPosition annotation, you can control the column positions in the generated CSV file. But the downside of this annotation is that it removes column headers from the generated CSV file.

public class User {

@CsvBindByPosition(position = 0)

private long id;

@CsvBindByPosition(position = 1)

private String name;

@CsvBindByPosition(position = 2)

private String email;

@CsvBindByPosition(position = 3)

private String country;

@CsvBindByPosition(position = 4)

private int age;

//contructor, getting and settings

}

@CsvBindByPosition specifies a binding between a column number of the CSV file and a field in a bean. This column number is zero-based (means position starts from 0).

**Source code:** Download the complete source code from [GitHub](https://github.com/attacomsian/code-examples/tree/master/spring-boot/export-data-csv) available under MIT license.

## Conclusion

That's all for explaining the usage of the OpenCSV library to generate and download a CSV file in Spring Boot. If you are not comfortable with using a 3rd-party library, you can write your own CSV writer. Writing a CSV file is quite similar to writing a text file with few exceptions.

If you have any questions or feedback, please feel free to send me a [tweet](https://twitter.com/attacomsian).

Happy learning Spring Boot 😍

# Displaying a Custom Error Page in Spring Boot

February 16, 2020 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

By default, Spring Boot shows a white-label error page if it encounters an error during the execution. However, you can easily create your own custom error page and configure the Spring Boot application to display it.

In this short article, you'll **learn how to disable the default white-label error page** and **display a customized error page** in a Spring Boot & Thymeleaf web application. If you need more information on how to use Thymeleaf with Spring Boot, take a look at [this guide](https://attacomsian.com/blog/spring-boot-thymeleaf-example).

## Disabling White-label Error Page

To completely disable the default white-label error page, just add the following property to the application.properties file:

server.error.whitelabel.enabled=false

The above property will disable the white-label error page shown by the Spring Boot application. Instead, the default error page from the underlying application server (like Tomcat) will be displayed.

Another way to disable the white-label error page is by **excluding** the ErrorMvcAutoConfiguration bean by either adding the following property to application.properties file:

spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration

Or the following annotation to the main application class:

@SpringBootApplication(exclude = {ErrorMvcAutoConfiguration.class})

public class Application {

// ...

}

## Displaying Custom Error Page

To display a custom error page in a Spring Boot and Thymeleaf application, just create a new Thymeleaf template called error.html inside the src/main/resources/templates/ folder. Spring Boot default BasicErrorController will automatically pick up this template and display it to the user whenever there is an error or exception.

Here is an example of a generic error.html page:

<!DOCTYPE html>

<html lang="en" xmlns:th="http://www.thymeleaf.org">

<head>

<meta charset="UTF-8">

<title>Page Not Found</title>

</head>

<body>

<h1>There's no page here!</h1>

<p>

We're sorry, the page you are looking for does not exist!

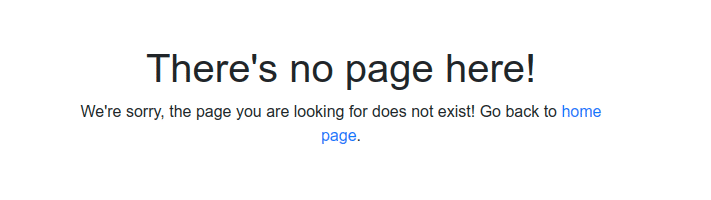
Go back to <a href="/">home page</a>.

</p>

</body>

</html>

This is all you need to do to display a custom error page. After adding some [Bootstrap 4](https://getbootstrap.com/) styling, you'll have a much nicer error page ready to serve the users:



You can be even **more specific** by putting the corresponding error page for each status code (such as 404.html in case of 404 error) inside the src/main/resources/templates/error/ directory.

Spring Boot will look for specific error pages (404.html or 500.html) before displaying the generic error.html page whenever your application encounters an error or exception.

### Custom Error Controller

The above error page templates are enough for most of the applications. But, sometimes, you want more control over how the error should be handled and a message displayed to the user. A common example is sending the error log to a 3rd-party application or storing it in the database for later review.

To achieve this, you need to create a custom error controller that **implements the ErrorController interface** and overrides its getErrorPath() method to return the path of the custom error page as shown in the following example:

**MyErrorController.java**

@Controller

public class MyErrorController implements ErrorController {

@RequestMapping("/error")

public String handleError(HttpServletRequest request) {

// get error status

Object status = request.getAttribute(RequestDispatcher.ERROR\_STATUS\_CODE);

// TODO: log error details here

if (status != null) {

int statusCode = Integer.parseInt(status.toString());

// display specific error page

if (statusCode == HttpStatus.NOT\_FOUND.value()) {

return "404";

} else if (statusCode == HttpStatus.INTERNAL\_SERVER\_ERROR.value()) {

return "500";

} else if (statusCode == HttpStatus.FORBIDDEN.value()) {

return "403";

}

}

// display generic error

return "error";

}

}

As you can see above, we have created a custom error controller that handles error requests on the /error endpoint. It tries to find the HTTP status code and returns a corresponding error page if found. Otherwise, it just displays the generic error page.

## Conclusion

That's all folks for creating and displaying a custom error page in a Spring Boot and Thymeleaf web application. In the first part of this tutorial, we looked at different ways to disable the default white-label error page in Spring Boot.

In the remaining half, we discussed multiple ways to handle the Spring Boot application errors and exceptions. For simple use cases, it is enough to create corresponding error page templates inside the templates directory and let the default BasicErrorController does the rest of the work.

# Spring Data JPA Auditing using Spring Boot and MySQL

October 20, 2019 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

Auditing helps us in **tracking and logging the persistence layer changes** made by the user in an application. By using auditing, we can easily determine who created or updated the entity record or when it happened. Basically, we keep track of every action that changes the state of the entity like insert, update, and delete operations.

[Spring Data JPA](https://attacomsian.com/blog/getting-started-spring-data-jpa) provides excellent support to transparently keep track of who created or changed an entity and the point in time this happened.

To enable the auditing feature in Spring Boot, we can make use of Spring Data JPA's @CreateDate, @CreatedBy, @LastModifiedDate, and @LastModifiedBy annotations. You can add these annotations either directly to your entity classes or by extending an abstract class that defines annotated audit fields.

Since we need auditing feature for most of the entities, in this article, we will create a generic Auditable abstract class with audit fields. Any entity can later extend this abstract class to enable the auditing functionality.

## Dependencies

To use [Spring Data JPA with the MySQL database](https://attacomsian.com/blog/accessing-data-spring-data-jpa-mysql) in a Spring Boot application, you only need spring-data-starter-data-jpa and mysql-connector-java dependencies.

Add the following dependencies to your Gradle project's build.gradle file:

implementation 'org.springframework.boot:spring-boot-starter-data-jpa'

runtimeOnly 'mysql:mysql-connector-java'

For Maven, include the following dependencies to your pom.xml file:

<dependency>

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

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

</dependency>

<dependency>

<groupId>mysql</groupId>

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

<scope>runtime</scope>

</dependency>

If you are starting a new Spring Boot project, just use [Spring Initializr](https://start.spring.io/) web tool to bootstrap a new application with the above-mentioned dependencies.

## Configure MySQL Database

Spring Boot automatically configures the DataSource bean for in-memory databases like [H2 database](https://attacomsian.com/blog/spring-data-jpa-h2-database#configure-h2-database), HSQLDB, and Apache Derby. For the MySQL database, you have to explicitly specify the database connection details in a properties file.

Open the application.properties file and copy and paste the following properties:

# MySQL connection properties

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.username=root

spring.datasource.password=mypass

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

# Log JPA queries

# Comment this in production

spring.jpa.show-sql=true

# Drop and create new tables (create, create-drop, validate, update)

# Only for testing purpose - comment this in production

spring.jpa.hibernate.ddl-auto=create

# Hibernate SQL dialect

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

Spring Boot will automatically configure DataSource based on the above properties. Make sure you change the spring.datasource.username and spring.datasource.password properties as per your MySQL database installation.

The hibernate property spring.jpa.hibernate.ddl-auto = create will automatically create database tables based on the entity classes when the application starts.

The createDatabaseIfNotExist=true configuration property, included in spring.datasource.url, makes sure that the database schema is automatically created if it doesn't already exist.

## Create Auditable Abstract Class

Let us now create an abstract Auditable class with the createdBy, createdDate, lastModifiedBy, and lastModifiedDate properties. This generic class acts as a base class with all the common auditing fields for the child entities.

To let the Spring Boot knows about these audit fields, you have to annotate the fields with @CreatedBy and @LastModifiedBy to track the user who created or updated the entity as well as @CreatedDate and @LastModifiedDate to log the time when these changes were made.

Here is how our Auditable abstract class looks like:

**Auditable.java**

package com.attacomsian.jpa.domains;

import org.springframework.data.annotation.\*;

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

import javax.persistence.\*;

import java.util.Date;

@MappedSuperclass

@EntityListeners(AuditingEntityListener.class)

public abstract class Auditable<T> {

@CreatedBy

protected T createdBy;

@Temporal(TemporalType.TIMESTAMP)

@CreatedDate

protected Date createdDate;

@LastModifiedBy

protected T lastModifiedBy;

@Temporal(TemporalType.TIMESTAMP)

@LastModifiedDate

protected Date lastModifiedDate;

public T getCreatedBy() {

return createdBy;

}

public void setCreatedBy(T createdBy) {

this.createdBy = createdBy;

}

public Date getCreatedDate() {

return createdDate;

}

public void setCreatedDate(Date createdDate) {

this.createdDate = createdDate;

}

public T getLastModifiedBy() {

return lastModifiedBy;

}

public void setLastModifiedBy(T lastModifiedBy) {

this.lastModifiedBy = lastModifiedBy;

}

public Date getLastModifiedDate() {

return lastModifiedDate;

}

public void setLastModifiedDate(Date lastModifiedDate) {

this.lastModifiedDate = lastModifiedDate;

}

}

As you can see above, the Auditable class is also annotated with @MappedSuperclass and @EntityListeners annotations. The @MappedSuperclass annotation indicates that the Auditable class is only a superclass and is not a JPA entity.

The @EntityListeners annotation is used to configure a JPA entity listener AuditingEntityListener to capture auditing information on persisting and updating entities. This entity listener class contains callback methods (annotated with @PrePersist and @PreUpdate) to persist and update audit fields when there is any create or update activity on the entity.

Now any entity that extends the Auditable abstract class will benefit from the JPA auditing feature. Spring Data JPA will automatically manage CreatedBy, CreatedDate, LastModifiedBy, LastModifiedDate columns when the entity is persisted or updated.

## Create an Entity

The next step is to create a Todo entity class and then extend it from the Auditable abstract class to add the auditing functionality:

**Todo.java**

package com.attacomsian.jpa.domains;

import javax.persistence.\*;

import java.io.Serializable;

@Entity

@Table(name = "todos")

public class Todo extends Auditable<String> implements Serializable {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String title;

private boolean completed;

public Todo() {

}

public Todo(String title, boolean completed) {

this.title = title;

this.completed = completed;

}

// getters and setters, equals(), toString() .... (omitted for brevity)

}

The Todo class is annotated with the Entity annotation to indicate that it is a JPA entity. The @Table annotation is used to specify the name of the database table that should be mapped to this entity.

The id attribute is annotated with both @Id and @GeneratedValue annotations. The former indicates that it is a primary key of the entity. The latter defines the primary key generation strategy. In the above case, we have declared the primary key as an AUTO INCREMENT field.

## Auditing Author with AuditorAware

We use the @CreatedDate and @LastModifiedDate annotations for tracking created and last modified dates respectively. Spring Data JPA automatically updates these fields by taking the current system time.

But how to tell the auditing infrastructure about the author who made these changes? It somehow needs to know this information since we have defined the @CreatedBy and @LastModifiedBy annotations in our Auditable abstract class.

To tell the auditing infrastructure about the current logged in user, we have to provide the implementation of AuditorAware and override its getCurrentAuditor method, as show below:

**EntityAuditorAware.java**

package com.attacomsian.jpa.domains;

import org.springframework.data.domain.AuditorAware;

import java.util.Optional;

public class EntityAuditorAware implements AuditorAware<String> {

@Override

public Optional<String> getCurrentAuditor() {

return Optional.of("Atta");

}

}

**Note:** For the sake of simplicity, I am returning a hard-coded auditor name in the getCurrentAuditor() method. For real-world applications with Spring Security, you need to [find the current logged-in user](https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#auditing.interfaces) and return it back.

## Create a Repository

Let us now create the TodoRepository interface to save and retrieve Todo entities from the database as follows:

**TodoRepository.java**

package com.attacomsian.jpa.repositories;

import com.attacomsian.jpa.domains.Todo;

import org.springframework.data.repository.CrudRepository;

public interface TodoRepository extends CrudRepository<Todo, Long> {

// TODO: add queries

}

We're extending TodoRepository from Spring Data JPA's [CrudRepository](https://attacomsian.com/blog/spring-data-jpa-repositories#crudrepository-interface) interface to inherit the standard CRUD methods for creating, reading, updating, and deleting Todo entities.

## Enable JPA Auditing

Finally, we need to enable the JPA auditing feature by specifying @EnableJpaAuditing on one of our configuration classes. We also need to define a bean of type AuditorAware and return an instance of the EntityAuditorAware class.

You can either create a separate configuration class or use the main application class to define these configurations. Let us create the AuditConfiguration class to let the Spring Data JPA knows we want to enable auditing:

**AuditConfiguration.java**

package com.attacomsian.jpa.config;

import com.attacomsian.jpa.domains.EntityAuditorAware;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.data.domain.AuditorAware;

import org.springframework.data.jpa.repository.config.EnableJpaAuditing;

@Configuration

@EnableJpaAuditing

public class AuditConfiguration {

@Bean

public AuditorAware<String> auditorAware() {

return new EntityAuditorAware();

}

}

That's all you need to do to enable Spring Data JPA auditing functionality in your Spring Boot and MySQL application. Let us now create the main application class to test our implementation.

## Testing the Application

Now, it is time to create the main application class for our Spring Boot project. This class also exposes a bean of type CommandLineRunner that defines a run() method which is invoked by Spring Boot after the application context has been loaded.

**Application.java**

package com.attacomsian.jpa;

import com.attacomsian.jpa.domains.Todo;

import com.attacomsian.jpa.repositories.TodoRepository;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.Bean;

import java.util.Arrays;

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

@Bean

public CommandLineRunner auditingDemo(TodoRepository todoRepository) {

return args -> {

// create new todos

todoRepository.saveAll(Arrays.asList(

new Todo("Buy milk", false),

new Todo("Email John", false),

new Todo("Visit Emma", false),

new Todo("Call dad", true),

new Todo("Weekend walk", true),

new Todo("Write Auditing Tutorial", true)

));

// retrieve all todos

Iterable<Todo> todos = todoRepository.findAll();

// print all todos

todos.forEach(System.out::println);

};

}

}

As you can see above, we have first saved several todos and then retrieved them by using the findAll() method. The last statement prints all todos on the console.

To see the actual output, you need to run the application. If you are using Gradle, execute the following command to start the application:

$ ./gradlew bootRun

For Maven, type the following command to launch the application:

$ ./mvnw spring-boot:run

Once the application is started, you should see the following output:

Todo{id=1, title='Buy milk', completed=false, createdBy=Atta, createdDate=2019-10-21 03:01:25.0, lastModifiedBy=Atta, lastModifiedDate=2019-10-21 03:01:25.0}

Todo{id=2, title='Email John', completed=false, createdBy=Atta, createdDate=2019-10-21 03:01:25.0, lastModifiedBy=Atta, lastModifiedDate=2019-10-21 03:01:25.0}

Todo{id=3, title='Visit Emma', completed=false, createdBy=Atta, createdDate=2019-10-21 03:01:25.0, lastModifiedBy=Atta, lastModifiedDate=2019-10-21 03:01:25.0}

Todo{id=4, title='Call dad', completed=true, createdBy=Atta, createdDate=2019-10-21 03:01:25.0, lastModifiedBy=Atta, lastModifiedDate=2019-10-21 03:01:25.0}

Todo{id=5, title='Weekend walk', completed=true, createdBy=Atta, createdDate=2019-10-21 03:01:25.0, lastModifiedBy=Atta, lastModifiedDate=2019-10-21 03:01:25.0}

Todo{id=6, title='Write Auditing Tutorial', completed=true, createdBy=Atta, createdDate=2019-10-21 03:01:25.0, lastModifiedBy=Atta, lastModifiedDate=2019-10-21 03:01:25.0}

**Source Code:** Download the complete source code from [GitHub](https://github.com/attacomsian/code-examples/tree/master/spring-data-jpa/jpa-auditing) available under MIT license.

## Conclusion

That's all folks. In this article, you have learned what is Spring Data JPA auditing and how to enable it in a Spring Boot and MySQL application.

In short, all you need to do is the following to enable JPA auditing feature:

1. Define the audit fields by using the @CreatedDate, @CreatedBy, @LastModifiedDate, and @LastModfiiedBy annotations. The best way to do so is by creating a generic abstract class and then extending the entities which need the auditing functionality.
2. Implement the AuditorAware interface to let Spring Data JPA auditing infrastructure knows about the currently logged-in user who is making the changes.
3. Add the @EnableJpaAuditing annotation to any configuration class to enable JPA auditing.
4. Expose a bean of type AuditorAware (only required if you need auditing author).

That's it. This article also pretty much sums up the Spring Data JPA tutorial series. If you've not read the earlier articles yet, check the links below in the further reading section.

# Reading and Writing XML in Java

January 02, 2020 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

XML — short for e**X**tensible **M**arkup **L**anguage — is a popular format for exchanging data between web services, computers, and front-ends after [JSON](https://attacomsian.com/blog/what-is-json). It was defined by W3C in 1998 and has a markup structure similar to HTML. Despite having a markup like HTML, XML is commonly used for storing and transporting data.

In my previous articles, I have already covered how to [read and write JSON](https://attacomsian.com/blog/java-read-write-json-files) in Java as well as in [Spring Boot](https://attacomsian.com/blog/parsing-xml-response-spring-boot). In this article, you'll learn **how to read and write XML** using different Java APIs.

Let us first look at what is an XML document and how it is structured.

## XML Document

An XML document consists of **elements** (also known as tags) similar to HTML. Each element has an opening and a closing tag along with content. Every XML must have exactly one **root element** — one tag that wraps the remaining tags. Tag names are can-sensitive which means XML differentiates between capital and non-capital letters. Each element can have any number of nested child elements.

Unlike HTML, XML **doesn't** have a **pre-defined set of tags**. This gives complete freedom to developers to define any tag they want to use in the document. A valid XML file is well-formed and must contain a link to an XML schema.

Let us look at the below XML document that contains a user information:

**user.xml**

<?xml version="1.0" encoding="UTF-8" ?>

<user id="1">

<name>John Doe</name>

<email>john.doe@example.com</email>

<roles>

<role>Member</role>

<role>Admin</role>

</roles>

<admin>true</admin>

</user>

As you can see above, user.xml file starts with <?xml> known as **XML prolog**. Another important thing to notice is that each element is wrapped in its own tag e.g. <name>John Deo</name>. Since roles is an array, we have to specify each array element using the nested role tag.

## Read and Write XML with JAXB

JAXB stands for **J**ava **A**rchitecture for **X**ML **B**inding which provides a convenient way for manipulating XML in Java. It is **Java standard** that defines an API for **reading and writing Java objects** to and from XML documents.

Starting from Java 6, JAXB is a part of the Java Development Kit (JDK). So there is no need to include any 3rd-party dependency to use JAXB in projects that are using Java 6 and higher.

In the following sections, you'll learn how to use JAXB to do the following:

1. [Marshalling](https://en.wikipedia.org/wiki/Marshalling_(computer_science)) — Convert a Java Object into XML.
2. [Unmarshalling](https://en.wikipedia.org/wiki/Unmarshalling) — Convert XML content into a Java Object.

Before we discuss marshalling and unmarshalling in detail, let us first create a simple Java class named User.java that represents the user described in the above user.xml file:

**User.java**

@XmlRootElement

public class User {

private int id;

private String name;

private String email;

private String[] roles;

private boolean admin;

public User() {

}

public User(int id, String name, String email, String[] roles, boolean admin) {

this.id = id;

this.name = name;

this.email = email;

this.roles = roles;

this.admin = admin;

}

public int getId() {

return id;

}

@XmlAttribute

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

@XmlElement

public void setName(String name) {

this.name = name;

}

public String getEmail() {

return email;

}

@XmlElement

public void setEmail(String email) {

this.email = email;

}

public String[] getRoles() {

return roles;

}

@XmlElementWrapper(name = "roles")

@XmlElement(name = "role")

public void setRoles(String[] roles) {

this.roles = roles;

}

public boolean isAdmin() {

return admin;

}

@XmlElement

public void setAdmin(boolean admin) {

this.admin = admin;

}

@Override

public String toString() {

return "User{" +

"id=" + id +

", name='" + name + '\'' +

", email='" + email + '\'' +

", roles=" + Arrays.toString(roles) +

", admin=" + admin +

'}';

}

}

As you can see above, we have annotated the class attributes with different JAXB annotations. These annotations serve a specific purpose while converting a Java object to and from XML.

* @XmlRootElement — This annotation is used to specify the root element of the XML document. It maps a class or an enum type to an XML element. By default, it uses the name of the class or enum as the name of the root element. However, you can customize the name by explicitly setting the name attribute i.e. @XmlRootElement(name = "person").
* @XmlAttribute — This annotation maps a Java object property to an XML element derived from the property name. To specify a different XML property name, you can pass the name parameter to the annotation declaration.
* @XmlElement — This annotation maps a Java object property to an XML element derived from the property name. The name of the XML element being mapped can be customized by using the name parameter.
* @XmlElementWrapper — This annotation generates a wrapper element around the XML representation, an array of String in our case. You must explicitly specify elements of the collection by using the @XmlElement annotation.

### Marshalling — Convert Java Object to XML

Marshalling in JAXB refers to the process of converting a Java object to an XML document. JAXB provides the Marshaller class for this purpose.

All you need to do is just create a new instance of JAXBContext by calling the newInstance() static method with a reference to the User class. You can then call the createUnmarshaller() method to create an instance of Marshaller. The Marshaller class provides several marshal() overloaded methods to turn a Java object into a file, an output stream, or output directly to the console.

Here is an example that demonstrates how to convert a User object into an XML document called user2.xml:

try {

// create XML file

File file = new File("user2.xml");

// create an instance of `JAXBContext`

JAXBContext context = JAXBContext.newInstance(User.class);

// create an instance of `Marshaller`

Marshaller marshaller = context.createMarshaller();

// enable pretty-print XML output

marshaller.setProperty(Marshaller.JAXB\_FORMATTED\_OUTPUT, true);

// create user object

User user = new User(2, "Tom Deo", "tom.doe@example.com",

new String[]{"Member", "Moderator"}, false);

// convert user object to XML file

marshaller.marshal(user, file);

} catch (JAXBException ex) {

ex.printStackTrace();

}

Now if you run the above code, you should see an XML file called user2.xml created in the root directory with the following contents:

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<user id="2">

<admin>false</admin>

<email>tom.doe@example.com</email>

<name>Tom Deo</name>

<roles>

<role>Member</role>

<role>Moderator</role>

</roles>

</user>

The Marshall class also provides an overloaded method to **output the generated XML document on the console** as shown below:

// print XML to console

marshaller.marshal(user, System.out);

### Unmarshalling — Convert XML to Java Object

Unmarshalling is very much similar to the marshalling process we discussed above. Except that, this time, we will be using the Unmarshaller class to convert an XML document to a Java object.

The following example demonstrates the JAXB's ability to read the above user.xml XML file to create a User object:

try {

// XML file path

File file = new File("user.xml");

// create an instance of `JAXBContext`

JAXBContext context = JAXBContext.newInstance(User.class);

// create an instance of `Unmarshaller`

Unmarshaller unmarshaller = context.createUnmarshaller();

// convert XML file to user object

User user = (User) unmarshaller.unmarshal(file);

// print user object

System.out.println(user);

} catch (JAXBException ex) {

ex.printStackTrace();

}

The above code will output the following:

User{id=1, name='John Doe', email='john.doe@example.com', roles=[Member, Admin], admin=true}

By default, the unmarshal() method returns an object. So we have to explicit typecast it to the correct type (User in our case). There are several other unmarshal() overloaded methods provided by Unmarshaller that you can use to read an XML document from different sources like a URL, a reader, or a writer.

## Read and Write XML using DOM Parser

DOM (**D**ocument **O**bject **M**odel) XML parser is another ways of reading and writing XML in Java. It is an older API that defines an **interface for accessing and updating the style, structure, and contents** of XML documents. XML parsers that support DOM implement this interface.

The DOM parser parses the XML document to create a **tree-like structure**. Everything in the DOM of an XML document is a **node**. So you have to traverse node by node to retrieve the required values.

The DOM defines several Java interfaces to represent an XML document. Here are the most commonly used interfaces:

* Node — The base datatype of the DOM.
* Element — Represents an individual element in the DOM.
* Attr — Represents an attribute of an element.
* Text — The actual content of an Element or Attr.
* Document — Represents the entire XML document. A Document object is often referred to as a DOM tree.

### Write XML to File using DOM Parser

To create an XML file using the DOM parser, you have to first create an instance of Document class using DocumentBuilder. Then define all the XML content — elements, attributes, values — with Element and Attr classes. In the end, use the Transformer class to output the entire XML document to an output stream, usually a file or a string.

Here is an example that creates a simple XML file using the DOM parser:

try {

// create new `Document`

DocumentBuilder builder = DocumentBuilderFactory.newInstance()

.newDocumentBuilder();

Document dom = builder.newDocument();

// first create root element

Element root = dom.createElement("user");

dom.appendChild(root);

// set `id` attribute to root element

Attr attr = dom.createAttribute("id");

attr.setValue("1");

root.setAttributeNode(attr);

// now create child elements (name, email, phone)

Element name = dom.createElement("name");

name.setTextContent("John Deo");

Element email = dom.createElement("email");

email.setTextContent("john.doe@example.com");

Element phone = dom.createElement("phone");

phone.setTextContent("800 456-4578");

// add child nodes to root node

root.appendChild(name);

root.appendChild(email);

root.appendChild(phone);

// write DOM to XML file

Transformer tr = TransformerFactory.newInstance().newTransformer();

tr.setOutputProperty(OutputKeys.INDENT, "yes");

tr.transform(new DOMSource(dom), new StreamResult(new File("file.xml")));

} catch (Exception ex) {

ex.printStackTrace();

}

Now if you execute the above code, you'd see the following file.xml file created with default UTF-8 encoded:

<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<user id="1">

<name>John Deo</name>

<email>john.doe@example.com</email>

<phone>800 456-4578</phone>

</user>

If you want to **output the XML document to the console**, just pass StreamResult with System.out as an argument as shown below:

// output XML document to console

tr.transform(new DOMSource(dom), new StreamResult(System.out));

### Read XML from File using DOM Parser

DOM parser can also be used to read and parse an XML file in Java. By default, DOM parser **reads the entire XML file into memory**; then parses it to create a tree structure for easy traversal or manipulation.

Let us look at the below **example that reads and parses the XML file**, we have just created above, using DOM XML parser:

try {

// parse XML file to build DOM

DocumentBuilder builder = DocumentBuilderFactory.newInstance()

.newDocumentBuilder();

Document dom = builder.parse(new File("file.xml"));

// normalize XML structure

dom.normalizeDocument();

// get root element

Element root = dom.getDocumentElement();

// print attributes

System.out.println("ID: " + root.getAttribute("id"));

// print elements

System.out.println("Name: " + root.getElementsByTagName("name").item(0).getTextContent());

System.out.println("Email: " + root.getElementsByTagName("email").item(0).getTextContent());

System.out.println("Phone: " + root.getElementsByTagName("phone").item(0).getTextContent());

} catch (Exception ex) {

ex.printStackTrace();

}

Here is the output of the above program:

ID: 1

Name: John Deo

Email: john.doe@example.com

Phone: 800 456-4578

**Note:** DOM Parser is good for reading and parsing small XML files as it loads the whole file into the memory. For larger XML files that contain a lot of data, you should consider using the **SAX** (Simple API for XML) parser. SAX doesn't load the entire file into memory that makes it faster than the DOM parser.

## Summary

Although XML is not widely used as a data exchange format in modern systems, it is still used by a lot of old services on the web as a primary source of data exchange. This is also true for many file formats that store data in XML-formatted files.

Java provides multiple ways to read and write XML files. In this article, we looked at JAXB and DOM parser for reading and writing XML data to and from a file.

JAXB is a modern replacement for old XML parsers like DOM and SAX. It provides methods to read and write Java objects to and from a file. By using JAXB annotations, we can easily define the relationship between XML elements and object attributes.

In case if you want to read and write JSON files, check out [how to read and write JSON in Java](https://attacomsian.com/blog/java-read-write-json-files) guide for JSON file read and write examples.

# Parsing XML Response in Spring Boot

March 17, 2019 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

While [JSON](https://attacomsian.com/blog/what-is-json) has become a popular choice for asynchronous communication, XML is still used by a large number of companies for data exchange in SOAP web services. As a Spring Boot developer, you might come across a requirement to parse the XML response of a web service in your career. It is not uncommon.

In this tutorial, we will learn how to parse an XML response of a web service using [DOM XML parser](https://attacomsian.com/blog/java-read-write-xml#read-and-write-xml-using-dom-parser) in Spring Boot. DOM XML parser loads the entire data into memory and parses it into an XML document. The parsed XML document can be used to traverse different elements and nodes in any order.

DOM XML parser is relatively slow and is only good for parsing small XML documents as it loads a complete XML document into memory. This makes it a bad choice for parsing large XML documents. For large XML files, you should use SAX parser. I will explain how SAX parser works and how it is better than DOM parser in the future.

## Dependencies

DOM XML parser is a part of standard Java extensions (javax) and does not require any 3rd party dependency. We only need spring-boot-starter dependency for Spring Boot project. Here is how our build.gradle file looks like:

**build.gradle**

plugins {

id 'org.springframework.boot' version '2.1.3.RELEASE'

id 'java'

}

apply plugin: 'io.spring.dependency-management'

group = 'com.attacomsian.xml'

version = '0.0.1-SNAPSHOT'

sourceCompatibility = '1.8'

repositories {

mavenCentral()

}

dependencies {

implementation 'org.springframework.boot:spring-boot-starter'

}

## XML Response

For this tutorial, we will use a [fake API](http://www.mocky.io/v2/5c8bdd5c360000cd198f831e) that returns an XML response. Here is how our XML response looks like:

<?xml version="1.0" encoding="UTF-8" ?>

<course>

<id>1</id>

<title>Introduction to Spring Boot &amp; Thymeleaf</title>

<price>49.99</price>

<created>2019-03-15</created>

<student>

<id>1</id>

<first\_name>George</first\_name>

<last\_name>Bluth</last\_name>

<avatar>https://s3.amazonaws.com/uifaces/faces/twitter/calebogden/128.jpg</avatar>

</student>

<student>

<id>2</id>

<first\_name>Janet</first\_name>

<last\_name>Weaver</last\_name>

<avatar>https://s3.amazonaws.com/uifaces/faces/twitter/josephstein/128.jpg</avatar>

</student>

<student>

<id>3</id>

<first\_name>Emma</first\_name>

<last\_name>Wong</last\_name>

<avatar>https://s3.amazonaws.com/uifaces/faces/twitter/olegpogodaev/128.jpg</avatar>

</student>

</course>

We will parse the above XML response to create a Course object that will also contain a list of Student.

## Spring Classes

Following are the two model classes (Course & Student) we need for this example:

**Student.java**

package com.attacomsian.xml.models;

public class Student {

private int id;

private String firstName;

private String lastName;

private String avatar;

public Student(int id, String firstName, String lastName, String avatar) {

this.id = id;

this.firstName = firstName;

this.lastName = lastName;

this.avatar = avatar;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

public String getAvatar() {

return avatar;

}

public void setAvatar(String avatar) {

this.avatar = avatar;

}

@Override

public String toString() {

return "Student{" +

"id=" + id +

", firstName='" + firstName + '\'' +

", lastName='" + lastName + '\'' +

", avatar='" + avatar + '\'' +

'}';

}

}

**Course.java**

package com.attacomsian.xml.models;

import java.util.Date;

import java.util.List;

public class Course {

private int id;

private String title;

private double price;

private Date created;

private List<Student> students;

public Course(int id, String title, double price, Date created) {

this.id = id;

this.title = title;

this.price = price;

this.created = created;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getTitle() {

return title;

}

public void setTitle(String title) {

this.title = title;

}

public double getPrice() {

return price;

}

public void setPrice(double price) {

this.price = price;

}

public Date getCreated() {

return created;

}

public void setCreated(Date created) {

this.created = created;

}

public List<Student> getStudents() {

return students;

}

public void setStudents(List<Student> students) {

this.students = students;

}

@Override

public String toString() {

return "Course{" +

"id=" + id +

", title='" + title + '\'' +

", price=" + price +

", created=" + created +

", students=" + students +

'}';

}

}

Notice that we have overridden the toString() method in both classes to print the details.

## DOM XML Parser

We will create a Spring service named XMLService for calling our fake API. This service uses Java DOM parser to read the XML from a remote URL and parse it into a Course object.

**XMLService.java**

package com.attacomsian.xml.services;

import com.attacomsian.xml.models.Course;

import com.attacomsian.xml.models.Student;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.stereotype.Service;

import org.w3c.dom.Document;

import org.w3c.dom.Element;

import org.w3c.dom.Node;

import org.w3c.dom.NodeList;

import javax.xml.parsers.DocumentBuilder;

import javax.xml.parsers.DocumentBuilderFactory;

import java.text.SimpleDateFormat;

import java.util.ArrayList;

import java.util.List;

@Service

public class XMLService {

private final Logger logger = LoggerFactory.getLogger(XMLService.class);

public Course parseCourse() {

Course course = null;

try {

// fake end point that returns XML response

String URL = "http://www.mocky.io/v2/5c8bdd5c360000cd198f831e";

DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

DocumentBuilder builder = factory.newDocumentBuilder();

Document doc = builder.parse(URL);

// normalize XML response

doc.getDocumentElement().normalize();

//read course details first

course = new Course(Integer.parseInt(doc.getElementsByTagName("id").item(0).getTextContent()),

doc.getElementsByTagName("title").item(0).getTextContent(),

Double.parseDouble(doc.getElementsByTagName("price").item(0).getTextContent()),

new SimpleDateFormat("yyyy-MM-dd").parse(doc.getElementsByTagName("created").item(0).getTextContent())

);

//read students list

NodeList nodeList = doc.getElementsByTagName("student");

//create an empty list for students

List<Student> students = new ArrayList<>();

//loop all available student nodes

for (int i = 0; i < nodeList.getLength(); i++) {

Node node = nodeList.item(i);

if(node.getNodeType() == Node.ELEMENT\_NODE) {

Element elem = (Element) node;

Student student = new Student(

Integer.parseInt(elem.getElementsByTagName("id").item(0).getTextContent()),

elem.getElementsByTagName("first\_name").item(0).getTextContent(),

elem.getElementsByTagName("last\_name").item(0).getTextContent(),

elem.getElementsByTagName("avatar").item(0).getTextContent()

);

students.add(student);

}

}

//set students in course

course.setStudents(students);

} catch (Exception ex) {

logger.error(ex.getMessage());

}

return course;

}

}

In parseCourse() method above, we build an instance of DocumentBuilder by using DocumentBuilderFactory. This instance parse() method is then used to send an HTTP request to our fake API to get the XML and parse it into a Document object. After that, we traverse this document element and nodes to create a Course object.

## Executing Application

Now let's create the main application class that implements the CommandLineRunner interface to run the Spring Boot project as a [console application](https://attacomsian.com/blog/spring-boot-console-application).

**XmlParserApplication.java**

package com.attacomsian.xml;

import com.attacomsian.xml.models.Course;

import com.attacomsian.xml.services.XMLService;

import org.springframework.boot.Banner;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class XmlParserApplication implements CommandLineRunner {

private XMLService xmlService;

public XmlParserApplication(XMLService xmlService) {

this.xmlService = xmlService;

}

public static void main(String[] args) {

SpringApplication app = new SpringApplication(XmlParserApplication.class);

// disable spring banner

app.setBannerMode(Banner.Mode.OFF);

app.run(args);

}

@Override

public void run(String... args) throws Exception {

// load course from XMLService

Course course = xmlService.parseCourse();

// print course details

System.out.println(course);

}

}

We you execute the Spring Boot application, you will see the following output printed on the terminal:

Course{id=1, title='Introduction to Spring Boot & Thymeleaf', price=49.99, created=Fri Mar 15 00:00:00 PKT 2019, students=[Student{id=1, firstName='George', lastName='Bluth', avatar='https://s3.amazonaws.com/uifaces/faces/twitter/calebogden/128.jpg'}, Student{id=2, firstName='Janet', lastName='Weaver', avatar='https://s3.amazonaws.com/uifaces/faces/twitter/josephstein/128.jpg'}, Student{id=3, firstName='Emma', lastName='Wong', avatar='https://s3.amazonaws.com/uifaces/faces/twitter/olegpogodaev/128.jpg'}]}

**Source code:** Download the complete source code from [GitHub](https://github.com/attacomsian/code-examples/tree/master/spring-boot/xml-response-parser) available under MIT license.

## Conclusion

In this article, I explain how to use DOM XML parser to parse an XML response in Spring Boot. This XML response can be returned by a SOAP web service or a remote XML file.

If you have any questions, please feel free to send me a [tweet](https://twitter.com/attacomsian) anytime. I would appreciate it if you share this tutorial with your friends and followers. You may want to join my weekly newsletter below to receive a curated list of top Spring Boot tutorials and jobs from my blog and around the internet.

# Spring Boot Console Application

March 10, 2019 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

Sometimes, we do not need the web component of Spring Boot in our application. To create such a non-web application, the main class must implement CommandLineRunner interface and override therun() method.

## Dependencies

We only need spring-boot-starter dependency for console application. Here is how our build.gradle file looks like:

**build.gradle**

plugins {

id 'org.springframework.boot' version '2.1.3.RELEASE'

id 'java'

}

apply plugin: 'io.spring.dependency-management'

group = 'com.attacomsian'

version = '0.0.1-SNAPSHOT'

sourceCompatibility = '1.8'

repositories {

mavenCentral()

}

dependencies {

implementation 'org.springframework.boot:spring-boot-starter'

}

I am using the Gradle plugin version 5.2.1 for this example. We used implementation keyword in the dependencies list instead of compile. compile configuration was depreciated starting from Gradle version 3.0.

If you are using Maven, please add the following dependency to your pom.xml file:

<dependency>

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

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

</dependency>

## Spring Classes

Let's create a simple Service class that returns a message. It has an overloaded getMessage() method that returns the default message if no argument is passed. The default message is an external property declared in application.properties file.

**HelloService.java**

package com.attacomsian.console.services;

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

import org.springframework.stereotype.Service;

@Service

public class HelloService {

@Value("${message.default}")

private String message;

public String getMessage() {

return message;

}

public String getMessage(String message) {

return "Hey, " + message;

}

}

**application.properties**

message.default=attacomsian.com

Now create the main class for our Spring Boot console application which implements the CommandLineRunner interface. This interface provides a simple run() method that is automatically invoked by Spring Boot after the application context has been loaded.

**ConsoleApplication.java**

package com.attacomsian.console;

import com.attacomsian.console.services.HelloService;

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

import org.springframework.boot.Banner;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class ConsoleApplication implements CommandLineRunner {

private HelloService helloService;

public ConsoleApplication(HelloService helloService) {

this.helloService = helloService;

}

public static void main(String[] args) {

SpringApplication app = new SpringApplication(ConsoleApplication.class);

// disable spring banner

app.setBannerMode(Banner.Mode.OFF);

app.run(args);

}

@Override

public void run(String... args) throws Exception {

//check if user passes any argument

if (args.length > 0) {

System.out.println(helloService.getMessage(args[0]));

} else {

//print the default message

System.out.println(helloService.getMessage());

}

}

}

In the code above, we are using the @SpringBootApplication annotation on our main class to enable the auto-configuration. The run() method is the entry point of our application. Inside this method, we first check if the user passes any command-line argument or not. Based on the argument availability, we call the corresponding HelloService method.

Another way to explicitly inform Spring Boot that this is not a web application is by using external properties. For example, we can specify the following properties in our application.properties file to disable the Spring Boot banner on startup. It will also not start the application as an embedded web server.

spring.main.web-application-type=none

spring.main.banner-mode=off

## Execution

Execute the following command in your terminal to run the Gradle project:

$ cd go/to/your/project/directory

$ ./gradlew bootRun

...

attacomsian.com

If you want to pass a command-line argument, use --args flag.

$ ./gradlew bootRun --args "Mike"

...

Hey, Mike

**Source code:** Download the complete source code from [GitHub](https://github.com/attacomsian/code-examples/tree/master/spring-boot/console-app) available under MIT license.

## Conclusion

That's all for creating a console (non-web) application in Spring Boot. We discussed several options available to explicitly inform Spring Boot that this is not a web application.

Most of the time, console applications are simple and have only one class that implements CommandLineRunner. But if you want to implement the CommandLineRunner interface more than once, you should use @Order annotation to specify their execution sequence.

# Uploading and Parsing CSV File using Spring Boot

January 17, 2020 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

CSV — short for **Comma Separated Values** — is a popular data exchange format that is frequently used for importing and exporting data between different servers and applications. A CSV file is just a **plain-text file** that stores data in a tabular format where each row consists of one or more fields and each column represents a specific field. These fields are separated by a delimiter usually a comma or a tab.

In an earlier article, I wrote about [exporting and downloading data as a CSV file](https://attacomsian.com/blog/export-download-data-csv-file-spring-boot) in Spring Boot. In this article, you'll **learn how to upload and parse a CSV file** using Spring Boot & Thymeleaf.

**Note:** To read and parse a CSV file in core Java, check out [reading and parsing a CSV file in Java](https://attacomsian.com/blog/java-read-parse-csv-file) tutorial.

## Dependencies

To upload parse a CSV file in Spring Boot, you only need spring-boot-starter-web and opencsv dependencies. Additionally, we also need spring-boot-starter-thymeleaf for serving Thymeleaf templates. The [OpenCSV](http://opencsv.sourceforge.net/" \t "_blank" \o "Open in new window) 3rd-party library will be used for parsing the uploaded file.

Add the following dependencies to your Gradle project's build.gradle file:

implementation 'org.springframework.boot:spring-boot-starter-web'

implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'

implementation 'com.opencsv:opencsv:5.0'

For Maven, include the following dependencies to your pom.xml file:

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

<groupId>com.opencsv</groupId>

<artifactId>opencsv</artifactId>

<version>5.0</version>

</dependency>

To create a new Spring Boot project from scratch, you can either use [Spring Initializr](https://attacomsian.com/blog/scaffolding-spring-boot-application#spring-initializr) or [Spring Boot CLI](https://attacomsian.com/blog/scaffolding-spring-boot-application#spring-boot-cli) to bootstrap a new application with the above-mentioned dependencies.

## Model Class

OpenCSV allows us to directly map the [CSV record fields to a Java object](https://attacomsian.com/blog/read-write-csv-files-opencsv#reading-a-csv-file-with-each-record-as-a-java-object-using-annotations). Let us create a simple model class named User.java that will be used to populate data from the CSV file:

**User.java**

public class User {

@CsvBindByName

private long id;

@CsvBindByName

private String name;

@CsvBindByName

private String email;

@CsvBindByName(column = "country")

private String countryCode;

@CsvBindByName

private int age;

public User(long id, String name, String email, String countryCode, int age) {

this.id = id;

this.name = name;

this.email = email;

this.countryCode = countryCode;

this.age = age;

}

// getters and setters removed for the sake of brevity

}

As you can see above, we have annotated the User class attributes with @CsvBindByName annotation. This annotation is provided by OpenCSV to specify a binding between a column name of the CSV input and a field in a bean.

You can only use the @CsvBindByName annotation if the CSV file has a header. It accepts up to five parameters like column, required, and locale. All parameters are options except column which is also only required if the header column name in the CSV file is different from the bean field name.

## Spring Boot Controller

Next, create a Spring Boot controller class named UploadController.java that handles the uploading and parsing of a CSV file:

**UploadController.java**

@Controller

public class UploadController {

@GetMapping("/")

public String index() {

return "index";

}

@PostMapping("/upload-csv-file")

public String uploadCSVFile(@RequestParam("file") MultipartFile file, Model model) {

// validate file

if (file.isEmpty()) {

model.addAttribute("message", "Please select a CSV file to upload.");

model.addAttribute("status", false);

} else {

// parse CSV file to create a list of `User` objects

try (Reader reader = new BufferedReader(new InputStreamReader(file.getInputStream()))) {

// create csv bean reader

CsvToBean<User> csvToBean = new CsvToBeanBuilder(reader)

.withType(User.class)

.withIgnoreLeadingWhiteSpace(true)

.build();

// convert `CsvToBean` object to list of users

List<User> users = csvToBean.parse();

// TODO: save users in DB?

// save users list on model

model.addAttribute("users", users);

model.addAttribute("status", true);

} catch (Exception ex) {

model.addAttribute("message", "An error occurred while processing the CSV file.");

model.addAttribute("status", false);

}

}

return "file-upload-status";

}

}

As you can see above, we have annotated the UploadController class with @Controller to indicate that the annotated class is a "Controller" (e.g. a web controller). Each method is decorated with @GetMapping or @PostMapping to bind the path and the HTTP action with that particular method:

* GET / route simply renders an HTML form to allow the user to upload a CSV file.
* POST /upload-csv-file route handles HTTP multipart/form-data requests and accepts a MultipartFile object as a route parameter. This is where we actually parse the uploaded CSV file into a list of User objects by using the CsvToBean class. This method returns an HTML response to either display a list of users or an error message.

## Thymeleaf Templates

The next step is to create Thymeleaf templates to allow users to upload a CSV file and display results. To nicely display the HTML form, we will use [Bootstrap 4](https://getbootstrap.com/) default styles.

### HTML Form for File Upload

Here is a simple HTML form that enables users to select a CSV file for upload:

**index.html**

<form method="POST" th:action="@{/upload-csv-file}" enctype="multipart/form-data">

<div class="form-group mt-3">

<label for="file">Select a CSV file</label>

<input type="file" name="file" class="form-control-file" id="file" accept=".csv">

</div>

<button type="submit" class="btn btn-primary">Import Users</button>

</form>

### Display Upload Results

Let us create another Thymeleaf template to display the upload results:

**file-upload-status.html**

<div class="container py-5">

<div class="row">

<div class="col-10 mx-auto">

<h1>File Upload Status</h1>

<!--display error if any-->

<div class="alert alert-danger" role="alert" th:if="${!status}">

<strong>Error:</strong>

<span th:text="${message}"></span>

</div>

<!--display users list-->

<table class="table table-striped" th:if="${status}">

<thead>

<tr>

<th scope="col">#</th>

<th scope="col">ID</th>

<th scope="col">Name</th>

<th scope="col">Email</th>

<th scope="col">Country</th>

<th scope="col">Age</th>

</tr>

</thead>

<tbody>

<tr th:each="user, i : ${users}">

<th scope="row" th:text="${i.index + 1}"></th>

<td th:text="${user.id}"></td>

<td th:text="${user.name}"></td>

<td th:text="${user.email}"></td>

<td th:text="${user.countryCode}"></td>

<td th:text="${user.age}"></td>

</tr>

</tbody>

</table>

</div>

</div>

</div>

## Running & Testing the Application

First of all, you need to create the main application class for Spring Boot application as shown below:

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

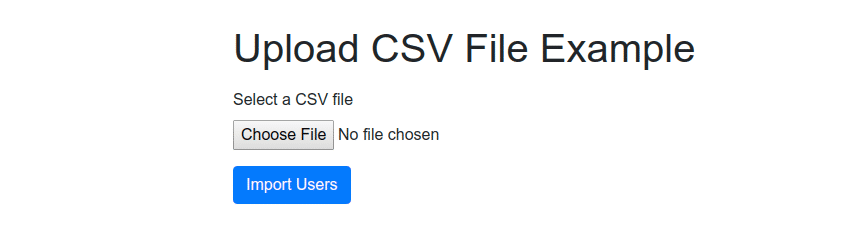
}

}

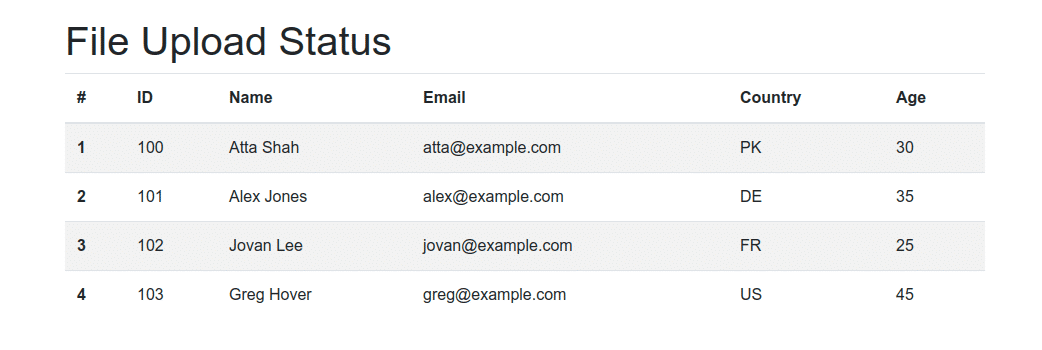
Let's run the application by typing the following command in your terminal from the root directory of the project:

$ ./gradlew bootRun

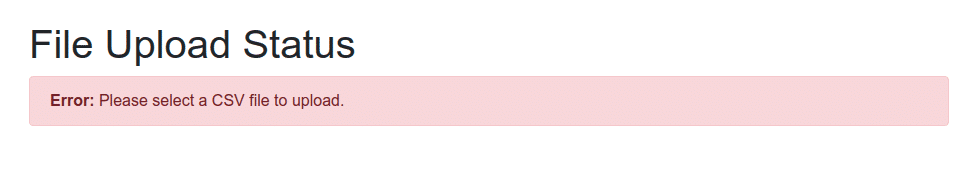
Once the Spring Boot application is started, open [http://localhost:8080](http://localhost:8080/) in your favorite web browser to see the upload form. Here is how it looks like:



As shown in the above image, select a CSV file and then click on **Import Users** button to kickstart file upload operation. If everything goes right, you should see a **list of users displayed** as shown in the following screenshot:



If you forget to select a CSV file or the CSV file is not valid, you should see the following **error message displayed** on the screen:



Finally, here is the sample CSV file I just uploaded in the above example:

**users.csv**

id,name,email,country,age

100,Atta Shah,atta@example.com,PK,30

101,Alex Jones,alex@example.com,DE,35

102,Jovan Lee,jovan@example.com,FR,25

103,Greg Hover,greg@example.com,US,45

**Source Code:** Download the complete source code from [GitHub](https://github.com/attacomsian/code-examples/tree/master/spring-boot/parse-data-csv) available under MIT license.

## Conclusion

That's all folks! In this article, you have learned how to upload and parse a CSV file using Spring Boot and Thymeleaf web application. We used [OpenCSV](https://attacomsian.com/blog/read-write-csv-files-opencsv), a popular open-source library, for parsing the uploaded file into a list of Java objects.

# Uploading and downloading files using Spring Boot REST API

March 21, 2019 • [Atta](https://twitter.com/attacomsian)

TABLE OF CONTENTS ⛱

Handling uploading and downloading files are very common jobs in most of the web applications. Spring Boot provides the MultipartFile interface to handle HTTP multi-part requests for uploading files.

In this tutorial, we will learn the following:

* Create a Spring Boot web application that allows file uploads
* Upload single and multiple files using RESTful web services
* Download file using RESTful web service
* List all files uploaded on the server
* A simple [Thymeleaf](https://attacomsian.com/blog/spring-boot-thymeleaf-example) & HTML web interface to upload file(s) from browser

Tools you need to complete this tutorial:

* [Java 8+](https://attacomsian.com/blog/install-java-on-ubuntu)
* JDK 1.8+
* [Spring Boot](https://attacomsian.com/blog/scaffolding-spring-boot-application)
* Thymeleaf
* [Gradle 4+](https://attacomsian.com/blog/install-gradle-on-ubuntu)
* Postman (optional for testing RESTful APIs)

**Note:** This article uses RESTful web services to upload and download files in Spring Boot. If you are using Thymeleaf and want to upload a file, check out [this guide](https://attacomsian.com/blog/spring-boot-thymeleaf-file-upload).

## Project Dependencies

We only need spring-boot-starter-web and spring-boot-starter-thymeleaf [starter dependencies](https://attacomsian.com/blog/spring-boot-starters) for our example Spring Boot project. We do not need any extra dependency for file upload. Here is how our build.gradle file looks like:

**build.gradle**

plugins {

id 'org.springframework.boot' version '2.1.3.RELEASE'

id 'java'

}

apply plugin: 'io.spring.dependency-management'

group = 'com.attacomsian'

version = '0.0.1-SNAPSHOT'

sourceCompatibility = '1.8'

repositories {

mavenCentral()

}

dependencies {

implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'

implementation 'org.springframework.boot:spring-boot-starter-web'

}

I used [Spring Initializr](https://start.spring.io/) to generate the above Gradle configuration file. It is an easier and quicker way to create a Spring Boot application.

## Configure Properties

Before we start the actual work, let's first configure the location on the server where all the uploaded files will be stored. We'll also configure the maximum file size that can be uploaded in a single HTTP multi-part request. Spring Boot automatically enables multipart/form-data requests, so we do not need to do anything.

**application.properties**

# max file size

spring.servlet.multipart.max-file-size=10MB

# max request size

spring.servlet.multipart.max-request-size=10MB

# files storage location (stores all files uploaded via REST API)

storage.location=./uploads

In above properties file, we have two multi-part settings:

* spring.servlet.multipart.max-file-size is set to 10MB, which means total files size cannot exceed 10MB.
* spring.servlet.multipart.max-request-size sets the maximum multipart/form-data request size to 10MB.

In simple words, we cannot upload files greater than 10MB in size given the above configuration.

## Enable Configuration Properties

In our application.properties file, we define the storage location. Now let's create a POJO class called StorageProperties and annotate it with @ConfigurationProperties to automatically bind the properties defined in application.properties file.

**StorageProperties.java**

package com.attacomsian.uploadfiles.storage;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties(prefix = "storage")

public class StorageProperties {

private String location;

public String getLocation() {

return location;

}

public void setLocation(String location) {

this.location = location;

}

}

Notice the prefix= "storage" attribute in the above annotation. It instructs @ConfigurationProperties to bind all the properties that start with storage prefix to their corresponding attributes of POJO class when the application is started.

The next step is to enable the ConfigurationProperties feature by adding @EnableConfigurationProperties annotation to our main configuration class.

**Application.java**

package com.attacomsian.uploadfiles;

import com.attacomsian.uploadfiles.storage.StorageProperties;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.boot.context.properties.EnableConfigurationProperties;

@SpringBootApplication

@EnableConfigurationProperties(StorageProperties.class)

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

## Files Upload Controller

Let's now create a controller class called FileController for handling uploading and downloading files via RESTful web services. It also defines a route to list all the uploaded files.

**FileController.java**

package com.attacomsian.uploadfiles.controllers;

import com.attacomsian.uploadfiles.commons.FileResponse;

import com.attacomsian.uploadfiles.storage.StorageService;

import org.springframework.core.io.Resource;

import org.springframework.http.HttpHeaders;

import org.springframework.http.ResponseEntity;

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

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

import org.springframework.web.multipart.MultipartFile;

import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

@Controller

public class FileController {

private StorageService storageService;

public FileController(StorageService storageService) {

this.storageService = storageService;

}

@GetMapping("/")

public String listAllFiles(Model model) {

model.addAttribute("files", storageService.loadAll().map(

path -> ServletUriComponentsBuilder.fromCurrentContextPath()

.path("/download/")

.path(path.getFileName().toString())

.toUriString())

.collect(Collectors.toList()));

return "listFiles";

}

@GetMapping("/download/{filename:.+}")

@ResponseBody

public ResponseEntity<Resource> downloadFile(@PathVariable String filename) {

Resource resource = storageService.loadAsResource(filename);

return ResponseEntity.ok()

.header(HttpHeaders.CONTENT\_DISPOSITION,

"attachment; filename=\"" + resource.getFilename() + "\"")

.body(resource);

}

@PostMapping("/upload-file")

@ResponseBody

public FileResponse uploadFile(@RequestParam("file") MultipartFile file) {

String name = storageService.store(file);

String uri = ServletUriComponentsBuilder.fromCurrentContextPath()

.path("/download/")

.path(name)

.toUriString();

return new FileResponse(name, uri, file.getContentType(), file.getSize());

}

@PostMapping("/upload-multiple-files")

@ResponseBody

public List<FileResponse> uploadMultipleFiles(@RequestParam("files") MultipartFile[] files) {

return Arrays.stream(files)

.map(file -> uploadFile(file))

.collect(Collectors.toList());

}

}

As always, our controller class is annotated with @Controller to let the Spring MVC pick it up for routes. Each method is decorated with @GetMapping or @PostMapping to bind the path and the HTTP action with that particular method.

* GET / loads the current list of uploaded files and renders it into a Thymeleaf template called listFiles.html.
* POST /download/{filename} resolves the resource if it exists, and sends it to the browser for download. HttpHeaders.CONTENT\_DISPOSITION adds the "Content-Disposition" response header to indicate file attachment.
* POST /upload-file & /upload-multiple-files routes handle HTTP multi-part requests and use StorageService for saving files on the server. Both these methods return an object of FileResponse after the upload is finished.

The FileResponse class is used to return a [JSON response](https://attacomsian.com/blog/processing-json-spring-boot) for RESTful web services.

**FileResponse.java**

package com.attacomsian.uploadfiles.commons;

public class FileResponse {

private String name;

private String uri;

private String type;

private long size;

public FileResponse(String name, String uri, String type, long size) {

this.name = name;

this.uri = uri;

this.type = type;

this.size = size;

}

// getters and setters removed for the sake of brevity

}

The FileController class uses the StorageService interface for storing and resolving files in the file system. It is the most important class for handling files in our example. We'll define these classes in the next section.

In production, it's not advised to store the uploaded files in your application file system. You might lose all files if your application server is damaged. It also makes very difficult to move the application from one server to another. Therefore, it is a good practice to use external storage like AWS S3 for storing all the uploaded files. I'll write about this topic in the future.

## Storage Service

Finally, it is time to create a storage service called StorageService for our controller to connect with a storage layer (e.g. file system in our case). This task involves several classes. We'll define these classes one-by-one.

The first step is to define an interface called StorageService as shown below:

**StorageService.java**

package com.attacomsian.uploadfiles.storage;

import org.springframework.core.io.Resource;

import org.springframework.web.multipart.MultipartFile;

import java.nio.file.Path;

import java.util.stream.Stream;

public interface StorageService {

void init();

String store(MultipartFile file);

Stream<Path> loadAll();

Path load(String filename);

Resource loadAsResource(String filename);

void deleteAll();

}

The above interface declares several abstract methods for initializing, storing, removing and retrieving files. It only lists possible storage operations without their implementation. Now, it is up to you to decide how you want to implement them. In this example, we will use our file system for handling files. It can also be implemented to store the files on any external location.

Let's create a concrete class FileSystemStorageService that implements the StorageService interface.

**FileSystemStorageService.java**

package com.attacomsian.uploadfiles.storage;

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

import org.springframework.core.io.Resource;

import org.springframework.core.io.UrlResource;

import org.springframework.stereotype.Service;

import org.springframework.util.FileSystemUtils;

import org.springframework.util.StringUtils;

import org.springframework.web.multipart.MultipartFile;

import javax.annotation.PostConstruct;

import java.io.IOException;

import java.io.InputStream;

import java.net.MalformedURLException;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.Paths;

import java.nio.file.StandardCopyOption;

import java.util.stream.Stream;

@Service

public class FileSystemStorageService implements StorageService {

private final Path rootLocation;

@Autowired

public FileSystemStorageService(StorageProperties properties) {

this.rootLocation = Paths.get(properties.getLocation());

}

@Override

@PostConstruct

public void init() {

try {

Files.createDirectories(rootLocation);

} catch (IOException e) {

throw new StorageException("Could not initialize storage location", e);

}

}

@Override

public String store(MultipartFile file) {

String filename = StringUtils.cleanPath(file.getOriginalFilename());

try {

if (file.isEmpty()) {

throw new StorageException("Failed to store empty file " + filename);

}

if (filename.contains("..")) {

// This is a security check

throw new StorageException(

"Cannot store file with relative path outside current directory "

+ filename);

}

try (InputStream inputStream = file.getInputStream()) {

Files.copy(inputStream, this.rootLocation.resolve(filename),

StandardCopyOption.REPLACE\_EXISTING);

}

}

catch (IOException e) {

throw new StorageException("Failed to store file " + filename, e);

}

return filename;

}

@Override

public Stream<Path> loadAll() {

try {

return Files.walk(this.rootLocation, 1)

.filter(path -> !path.equals(this.rootLocation))

.map(this.rootLocation::relativize);

}

catch (IOException e) {

throw new StorageException("Failed to read stored files", e);

}

}

@Override

public Path load(String filename) {

return rootLocation.resolve(filename);

}

@Override

public Resource loadAsResource(String filename) {

try {

Path file = load(filename);

Resource resource = new UrlResource(file.toUri());

if (resource.exists() || resource.isReadable()) {

return resource;

}

else {

throw new FileNotFoundException(

"Could not read file: " + filename);

}

}

catch (MalformedURLException e) {

throw new FileNotFoundException("Could not read file: " + filename, e);

}

}

@Override

public void deleteAll() {

FileSystemUtils.deleteRecursively(rootLocation.toFile());

}

}

The above implementation class is taken from Spring Boot [official files uploading example](https://github.com/spring-guides/gs-uploading-files) with few modifications done by me. The important change I made is the addition of @PostConstruct annotation on the init() method. It guarantees that the init() method is only called once the bean is fully initialized with all the dependencies injected.

The FileSystemStorageService class throws exceptions in case of unexpected scenarios, for example, the file requested by the user might not exist.

The first exception is StorageException which is thrown when we are unable to create the storage directory or the uploaded file is empty etc.

**StorageException.java**

package com.attacomsian.uploadfiles.storage;

public class StorageException extends RuntimeException {

public StorageException(String message) {

super(message);

}

public StorageException(String message, Throwable cause) {

super(message, cause);

}

}

The FileNotFoundException exception is thrown when a file is requested by the user but it does not exist on the server.

**FileNotFoundException.java**

package com.attacomsian.uploadfiles.storage;

import org.springframework.http.HttpStatus;

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

@ResponseStatus(HttpStatus.NOT\_FOUND)

public class FileNotFoundException extends StorageException {

public FileNotFoundException(String message) {

super(message);

}

public FileNotFoundException(String message, Throwable cause) {

super(message, cause);

}

}

Notice the @ResponseStatus(HttpStatus.NOT\_FOUND) annotation above. This annotation ensures that Spring Boot responds with a 404 (Not Found) HTTP status instead of 501 (Internal Server Error) when the exception is thrown.

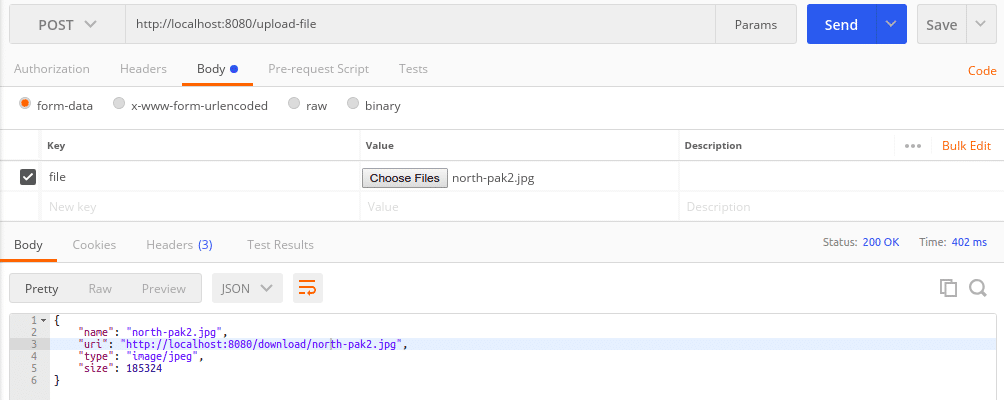
## Running & Testing the Application

We are almost done with our backend development. Since we created RESTful APIs for uploading and downloading files, we can test them via Postman. Let's run the application by typing the following command in your terminal from the root directory of the project:

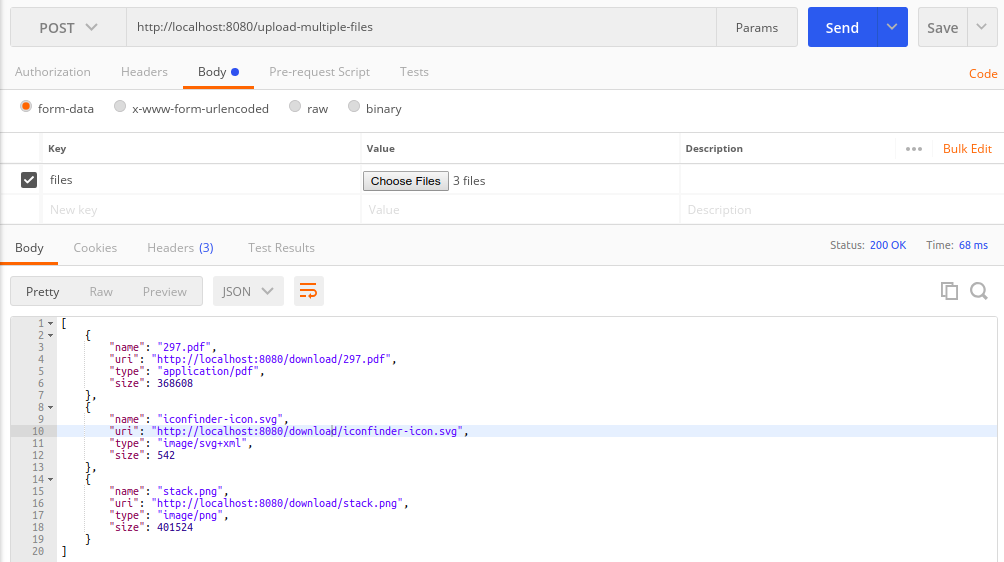
$ ./gradlew bootRun

Once the application is started, you can access it at [http://localhost:8080](http://localhost:8080/).

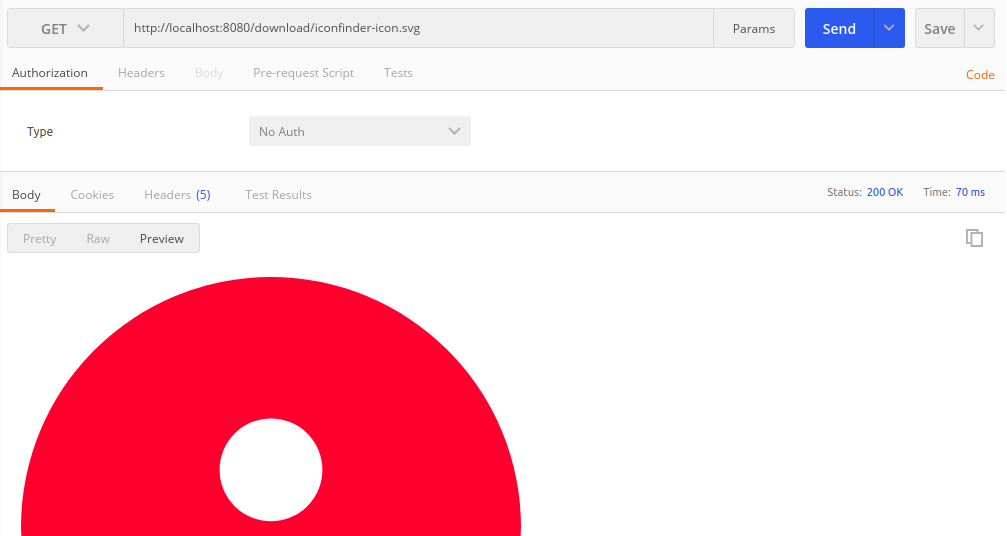
### 1. Upload Single File



### 2. Upload Multiple Files



### 3. Download File



## HTML Web Form

We have tested our RESTful APIs and they are working fine. Now it is time to create a simple front-end interface using [HTML & Thymeleaf](https://attacomsian.com/blog/spring-boot-thymeleaf-form-handling) that lists all the files uploaded so far. It will also allow users to upload files directly from the browser.

**listFiles.html**

<!doctype html>

<html lang="en" xmlns:th="http://www.thymeleaf.org">

<body>

<h1>Spring Boot File Upload Example</h1>

<hr/>

<h4>Upload Single File:</h4>

<form method="POST" enctype="multipart/form-data" th:action="@{/upload-file}">

<input type="file" name="file"> <br/><br/>

<button type="submit">Submit</button>

</form>

<hr/>

<h4>Upload Multiple Files:</h4>

<form method="POST" enctype="multipart/form-data" th:action="@{/upload-multiple-files}">

<input type="file" name="files" multiple> <br/><br/>

<button type="submit">Submit</button>

</form>

<hr/>

<h2>All Uploaded Files:</h2>

<ul>

<li th:each="file : ${files}">

<a th:href="${file}" target="\_blank" th:text="${file}"></a>

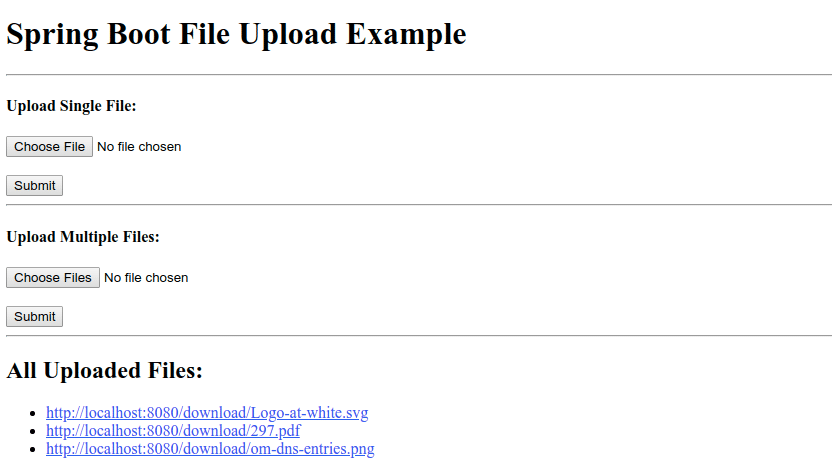
</li>

</ul>

</body>

</html>

The above template has two forms that enable users to upload a single file as well as multiple files. At the bottom, it also shows a list of currently uploaded files on the server. Here is how it looks like:



**Source code:** Download the complete source code from [GitHub](https://github.com/attacomsian/code-examples/tree/master/spring-boot/upload-files) available under MIT license.

## Conclusion

That's all folks for uploading and downloading files in Spring Boot. We discussed strategies for handling single as well as multiple files via RESTful web services. We tested our REST APIs via Postman to confirm that they are working as expected. Finally, we created the simplest web interface in HTML and Thymeleaf for showing a list of all the uploaded files.

In the end, I really appreciate that you read this article and hope that you'd have learned how to handle files in Spring Boot today. If you have any questions or feedback, please feel free to send me a [tweet](https://twitter.com/attacomsian).