Spring Boot: Upload/Import Excel file data into MySQL Database

bezkoder.com/spring-boot-upload-excel-file-database

bezkoder Last modified: October 13, 2021

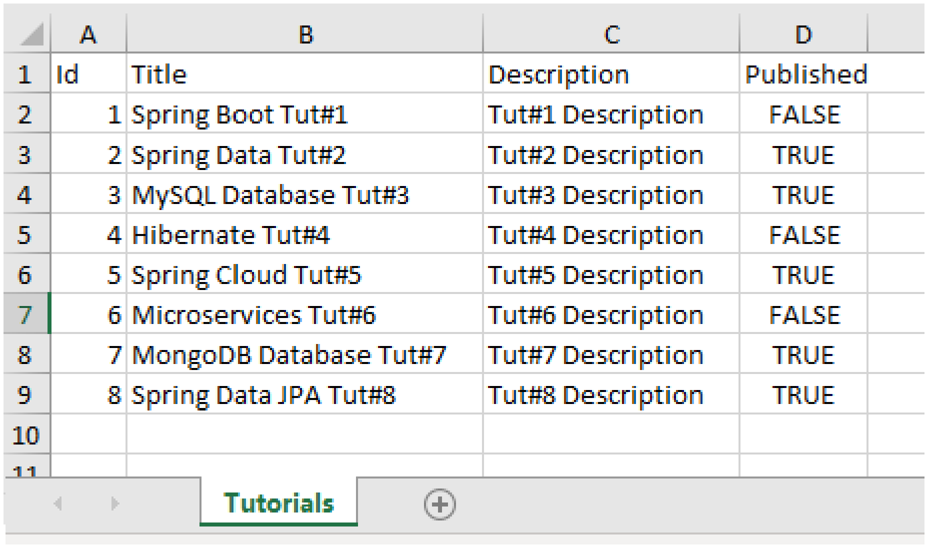
The Excel file is a spreadsheet file format created by Microsoft for use with Microsoft Excel. You can use the file to create, view, edit, analyse data, charts, budgets and more. In this tutorial, I will show you how to upload/import Excel file data into MySQL Database using Spring Boot & Apache POI, then export Rest API to return Excel file from database table.

Related Posts:
– Spring Boot Multipart File upload example
– How to upload multiple files in Java Spring Boot
– Upload/Import CSV file to MySQL Database in Spring Boot
– Spring Boot: Download Excel file from MySQL database table

Deployment:
– Deploy Spring Boot App on AWS – Elastic Beanstalk
– Docker Compose: Spring Boot and MySQL example

Spring Boot Rest APIs for uploading Excel Files

Assume that we have an **.xlsx** file that contains Tutorial data as following:



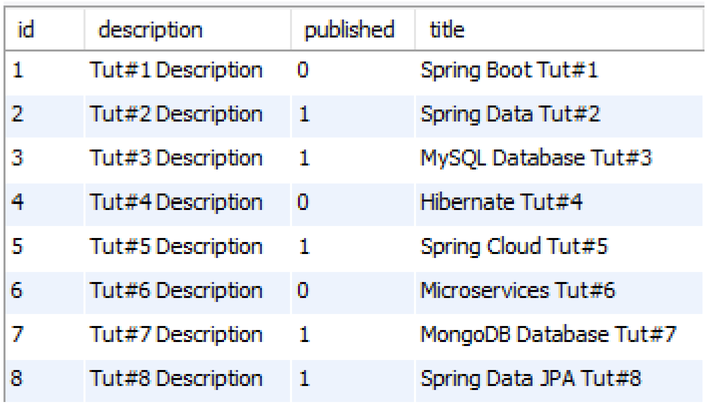
	A	B	C	D
1	Id	Title	Description	Published
2	1	Spring Boot Tut#1	Tut#1 Description	FALSE
3	2	Spring Data Tut#2	Tut#2 Description	TRUE
4	3	MySQL Database Tut#3	Tut#3 Description	TRUE
5	4	Hibernate Tut#4	Tut#4 Description	FALSE
6	5	Spring Cloud Tut#5	Tut#5 Description	TRUE
7	6	Microservices Tut#6	Tut#6 Description	FALSE
8	7	MongoDB Database Tut#7	Tut#7 Description	TRUE
9	8	Spring Data JPA Tut#8	Tut#8 Description	TRUE
10				

We're gonna create a Spring Boot Application that provides APIs for:

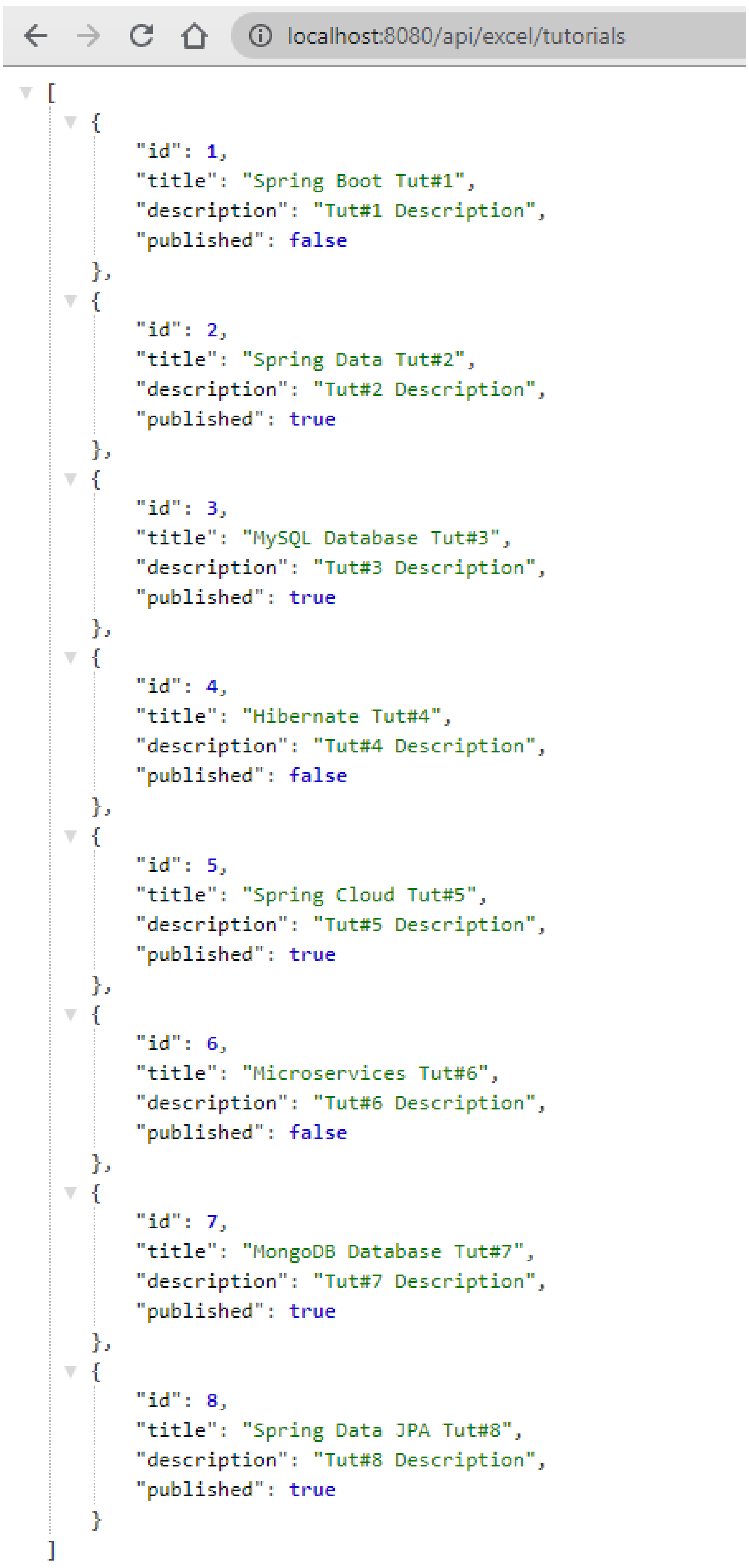
- uploading Excel File to the Spring Server & storing data in MySQL Database
- getting list of items from MySQL table
- downloading MySQL table data as Excel file

After the Excel file is uploaded successfully, tutorials table in MySQL database will look like this:

If we get list of Tutorials, the Spring Rest Apis will return:



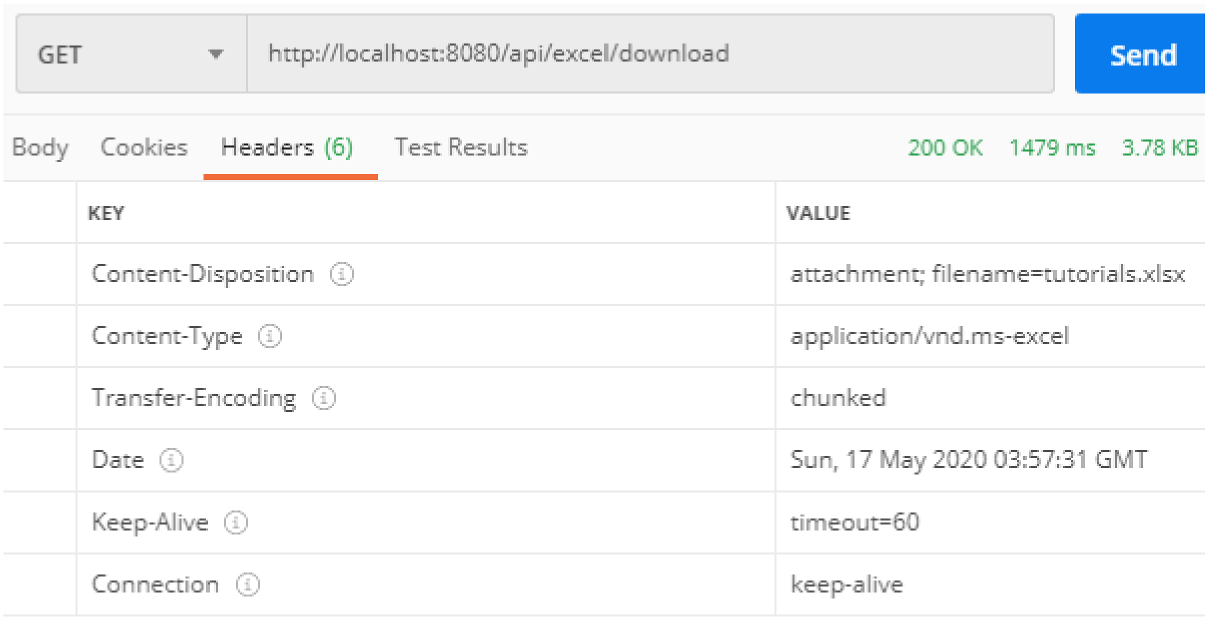
id	description	published	title
1	Tut#1 Description	0	Spring Boot Tut#1
2	Tut#2 Description	1	Spring Data Tut#2
3	Tut#3 Description	1	MySQL Database Tut#3
4	Tut#4 Description	0	Hibernate Tut#4
5	Tut#5 Description	1	Spring Cloud Tut#5
6	Tut#6 Description	0	Microservices Tut#6
7	Tut#7 Description	1	MongoDB Database Tut#7
8	Tut#8 Description	1	Spring Data JPA Tut#8



```
[
  {
      "id": 1,
      "title": "Spring Boot Tut#1",
      "description": "Tut#1 Description",
      "published": false
  },
  {
      "id": 2,
      "title": "Spring Data Tut#2",
      "description": "Tut#2 Description",
      "published": true
  },
  {
      "id": 3,
      "title": "MySQL Database Tut#3",
      "description": "Tut#3 Description",
      "published": true
  },
  {
      "id": 4,
      "title": "Hibernate Tut#4",
      "description": "Tut#4 Description",
      "published": false
  },
  {
      "id": 5,
      "title": "Spring Cloud Tut#5",
      "description": "Tut#5 Description",
      "published": true
  },
  {
      "id": 6,
      "title": "Microservices Tut#6",
      "description": "Tut#6 Description",
      "published": false
  },
  {
      "id": 7,
      "title": "MongoDB Database Tut#7",
      "description": "Tut#7 Description",
      "published": true
  },
  {
      "id": 8,
      "title": "Spring Data JPA Tut#8",
      "description": "Tut#8 Description",
      "published": true
  }
]
```

Spring Boot Rest API returns Excel File

If you send request to `/api/excel/download`, the server will return a response with an Excel file **tutorials.xlsx** that contains data in MySQL table:



KEY	VALUE
Content-Disposition	attachment; filename=tutorials.xlsx
Content-Type	application/vnd.ms-excel
Transfer-Encoding	chunked
Date	Sun, 17 May 2020 03:57:31 GMT
Keep-Alive	timeout=60
Connection	keep-alive

How to do this?
You need to set the HTTP header:

```
"Content-disposition" : "attachment; filename=[yourFileName]"
"Content-Type" : "application/vnd.ms-excel"
```

You can find step by step for downloading Excel file in the tutorial:
Spring Boot: Download Excel file from MySQL database table

These are APIs to be exported:

Methods	Urls	Actions
POST	/api/excel/upload	upload an Excel File
GET	/api/excel/tutorials	get List of items in db table
GET	/api/excel/download	download db data as Excel file

Technology

- Java 8
- Spring Boot 2 (with Spring Web MVC)
- Maven 3.6.1
- Apache POI 4.1.2

Project Structure

This is the project directory that we're gonna build:

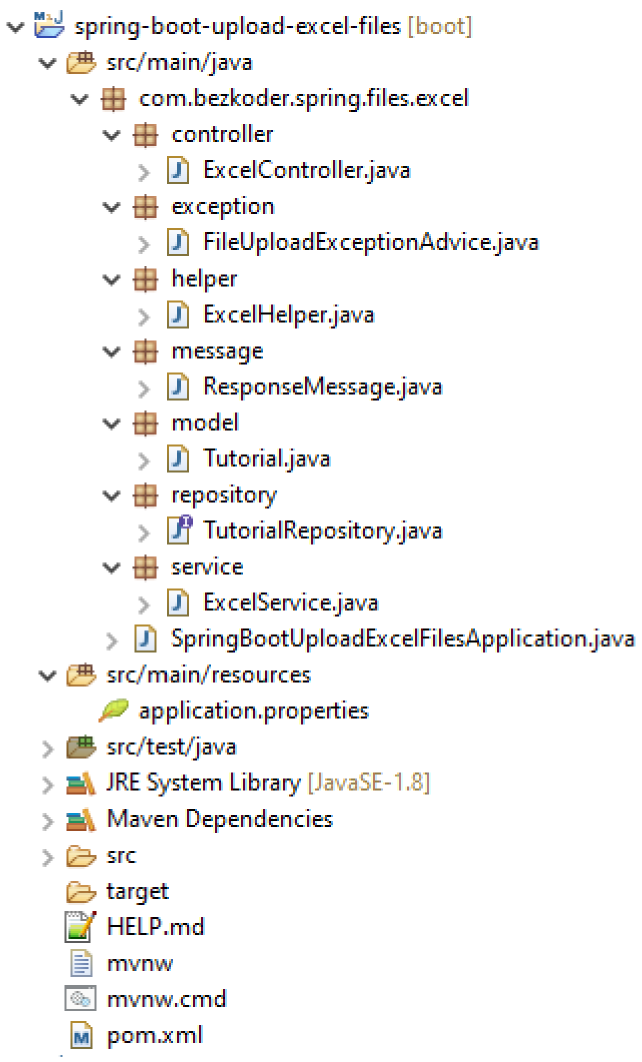
– `ExcelHelper` provides functions to read Excel file.

– `Tutorial` data model class corresponds to entity and table **tutorials**.
– `TutorialRepository` is an interface that extends `JpaRepository` for persisting data.

– `ExcelService` uses `ExcelHelper` and `TutorialRepository` methods to save Excel data to MySQL, load data to Excel file, or get all Tutorials from MySQL table.

– `ExcelController` calls `ExcelService` methods and export Rest APIs: upload Excel file, get data from MySQL database.

– `FileUploadExceptionAdvice` handles exception when the controller processes file upload.



```
spring-boot-upload-excel-files [boot]
  src/main/java
    com.bezkoder.spring.files.excel
      controller
        ExcelController.java
      exception
        FileUploadExceptionAdvice.java
      helper
        ExcelHelper.java
      message
        ResponseMessage.java
      model
        Tutorial.java
      repository
        TutorialRepository.java
      service
        ExcelService.java
      SpringBootUploadExcelFilesApplication.java
  src/main/resources
    application.properties
  src/test/java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

– *application.properties* contains configuration for Spring Data and Servlet Multipart file.
– *pom.xml* for Spring Boot, MySQL connector, Apache POI dependencies.

Setup Spring Boot Excel File Upload project

Use Spring web tool or your development tool (Spring Tool Suite, Eclipse, Intellij) to create a Spring Boot project.

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

```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
	<groupId>org.apache.poi</groupId>
	<artifactId>poi-ooxml</artifactId>
	<version>4.1.2</version>
</dependency>
<dependency>
	<groupId>mysql</groupId>
	<artifactId>mysql-connector-java</artifactId>
	<scope>runtime</scope>
</dependency>
```

Configure Spring Datasource, JPA, Hibernate

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

```
spring.datasource.url= jdbc:mysql://localhost:3306/testdb?useSSL=false
spring.datasource.username= root
spring.datasource.password= 123456
spring.jpa.properties.hibernate.dialect= org.hibernate.dialect.MySQL5InnoDBDialect
# Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto= update
```

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

Define Data Model

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

model/Tutorial.java

```
package com.bezkoder.spring.files.excel.model;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table(name = "tutorials")
public class Tutorial {
  @Id
  @Column(name = "id")
  private long id;
  @Column(name = "title")
  private String title;
  @Column(name = "description")
  private String description;
  @Column(name = "published")
  private boolean published;
  public Tutorial() {
  }
  public Tutorial(String title, String description, boolean published) {
    this.title = title;
    this.description = description;
    this.published = published;
  }
  public long getId() {
    return id;
  }
  public void setId(long id) {
    this.id = id;
  }
  public String getTitle() {
    return title;
  }
  public void setTitle(String title) {
    this.title = title;
  }
  public String getDescription() {
    return description;
  }
  public void setDescription(String description) {
    this.description = description;
  }
  public boolean isPublished() {
    return published;
  }
  public void setPublished(boolean isPublished) {
    this.published = isPublished;
  }
  @Override
  public String toString() {
    return "Tutorial [id=" + id + ", title=" + title + ", desc=" + description +
", published=" + published + "]";
  }
}
```

– `@Entity` annotation indicates that the class is a persistent Java class.
– `@Table` annotation provides the table that maps this entity.
– `@Id` annotation is for the primary key.

– `@Column` annotation is used to define the column in database that maps annotated field.

Create Data Repository for working with Database

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

repository/TutorialRepository.java

```
package com.bezkoder.spring.files.excel.repository;
import org.springframework.data.jpa.repository.JpaRepository;
import com.bezkoder.spring.files.excel.model.Tutorial;
public interface TutorialRepository extends JpaRepository<Tutorial, Long> {
}
```

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

The quantity of rows in Excel file (also **tutorials** table) could be large, so you may want to get only several at once by modifying this Repository to work with Pagination, the instruction can be found at:
Spring Boot Pagination & Filter example | Spring JPA, Pageable

You also find way to write Unit Test for this JPA Repository at:
Spring Boot Unit Test for JPA Repositiory with @DataJpaTest

Implement Read/Write Excel Helper Class

We're gonna use Apache POI classes such as: `Workbook`, `Sheet`, `Row`, `Cell`.
Let me summarize the steps for reading from Excel file:

- create `Workbook` from `InputStream`
- create `Sheet` using `Workbook.getSheet()` method
- iterate over `Row`s by `Iterator` with `Sheet.iterator()` and `Iterator.hasNext()`
- from each `Row`, iterate over `Cell`s
- with each `Cell`, use `getNumericCellValue()`, `getStringCellValue()`... methods to read and parse the content

```
Workbook workbook = new XSSFWorkbook(inputStream);
Sheet sheet = workbook.getSheet(SHEET);
Iterator<Row> rows = sheet.iterator();
while (rows.hasNext()) {
  Row currentRow = rows.next();
  Iterator<Cell> cellsInRow = currentRow.iterator();
  while (cellsInRow.hasNext()) {
     Cell currentCell = cellsInRow.next();
     // each cell case
     id = (long) currentCell.getNumericCellValue();
     title = currentCell.getStringCellValue();
     published = currentCell.getBooleanCellValue();
  }

workbook.close();
```

Under **helper** package, we create `ExcepHelper` class with 2 methods:

- `hasExcelFormat()` : check if a file has Excel format or not
- `excelToTutorials()` : read `InputStream` of a file, return a list of Tutorials

Here is full code of *helper/ExcelHelper.java*:

```
package com.bezkoder.spring.files.excel.helper;
import java.io.IOException;
import java.io.InputStream;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import org.apache.poi.ss.usermodel.Cell;
import org.apache.poi.ss.usermodel.Row;
import org.apache.poi.ss.usermodel.Sheet;
import org.apache.poi.ss.usermodel.Workbook;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;
import org.springframework.web.multipart.MultipartFile;
import com.bezkoder.spring.files.excel.model.Tutorial;
public class ExcelHelper {
  public static String TYPE = "application/vnd.openxmlformats-
officedocument.spreadsheetml.sheet";
  static String[] HEADERs = { "Id", "Title", "Description", "Published" };
  static String SHEET = "Tutorials";
  public static boolean hasExcelFormat(MultipartFile file) {
    if (!TYPE.equals(file.getContentType())) {
      return false;
    }
    return true;
  }
  public static List<Tutorial> excelToTutorials(InputStream is) {
    try {
      Workbook workbook = new XSSFWorkbook(is);
      Sheet sheet = workbook.getSheet(SHEET);
      Iterator<Row> rows = sheet.iterator();
      List<Tutorial> tutorials = new ArrayList<Tutorial>();
      int rowNumber = 0;
      while (rows.hasNext()) {
        Row currentRow = rows.next();
        // skip header
        if (rowNumber == 0) {
          rowNumber++;
          continue;
        }
        Iterator<Cell> cellsInRow = currentRow.iterator();
        Tutorial tutorial = new Tutorial();
        int cellIdx = 0;
        while (cellsInRow.hasNext()) {
          Cell currentCell = cellsInRow.next();
          switch (cellIdx) {
          case 0:
            tutorial.setId((long) currentCell.getNumericCellValue());
            break;
          case 1:
            tutorial.setTitle(currentCell.getStringCellValue());
            break;
          case 2:
            tutorial.setDescription(currentCell.getStringCellValue());
            break;
          case 3:
            tutorial.setPublished(currentCell.getBooleanCellValue());
            break;
          default:
            break;
          }
          cellIdx++;
        }
        tutorials.add(tutorial);
      }
      workbook.close();
      return tutorials;
    } catch (IOException e) {
      throw new RuntimeException("fail to parse Excel file: " + e.getMessage());
    }
  }
}
```

Don’t forget to change the sheet name to **Tutorials** (or any name you want). It’s because we create `Sheet` object from that name.

```
static String SHEET = "Tutorials";
...
Sheet sheet = workbook.getSheet(SHEET);
```

Create Excel File Service

`ExcelService` class will be annotated with `@Service` annotation, it uses `ExcelHelper` and `TutorialRepository` for 2 functions:

- `save(MultipartFile file)` : store Excel data to database
- `getAllTutorials` (): read data from database and return `List<Tutorial>`

service/*ExcelService.java*

```
package com.bezkoder.spring.files.excel.service;
import java.io.IOException;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.web.multipart.MultipartFile;
import com.bezkoder.spring.files.excel.helper.ExcelHelper;
import com.bezkoder.spring.files.excel.model.Tutorial;
import com.bezkoder.spring.files.excel.repository.TutorialRepository;
@Service
public class ExcelService {
  @Autowired
  TutorialRepository repository;
  public void save(MultipartFile file) {
    try {
      List<Tutorial> tutorials =
ExcelHelper.excelToTutorials(file.getInputStream());
      repository.saveAll(tutorials);
    } catch (IOException e) {
      throw new RuntimeException("fail to store excel data: " + e.getMessage());
    }
  }
  public List<Tutorial> getAllTutorials() {
    return repository.findAll();
  }
}
```

Define Response Message

The `ResponseMessage` is for message to client that we're gonna use in Rest Controller and Exception Handler.

message/*ResponseMessage.java*

```
package com.bezkoder.spring.files.excel.message;
public class ResponseMessage {
  private String message;
  public ResponseMessage(String message) {
    this.message = message;
  }
  public String getMessage() {
    return message;
  }
  public void setMessage(String message) {
    this.message = message;
  }
}
```

Create Controller for Upload Excel Files

In **controller** package, we create `ExcelController` class for Rest Apis.
– `@CrossOrigin` is for configuring allowed origins.
– `@Controller` annotation indicates that this is a controller.
– `@GetMapping` and `@PostMapping` annotation is for mapping HTTP GET & POST requests onto specific handler methods:

- POST /upload: `uploadFile()`
- GET /tutorials: `getAllTutorials()`

– We use `@Autowired` to inject implementation of `ExcelService` bean to local variable.

controller/*ExcelController.java*

```
package com.bezkoder.spring.files.excel.controller;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.multipart.MultipartFile;
import com.bezkoder.spring.files.excel.helper.ExcelHelper;
import com.bezkoder.spring.files.excel.message.ResponseMessage;
import com.bezkoder.spring.files.excel.model.Tutorial;
import com.bezkoder.spring.files.excel.service.ExcelService;
@CrossOrigin("http://localhost:8081")
@Controller
@RequestMapping("/api/excel")
public class ExcelController {
  @Autowired
  ExcelService fileService;
  @PostMapping("/upload")
  public ResponseEntity<ResponseMessage> uploadFile(@RequestParam("file")
MultipartFile file) {
    String message = "";
    if (ExcelHelper.hasExcelFormat(file)) {
      try {
        fileService.save(file);
        message = "Uploaded the file successfully: " + file.getOriginalFilename();
        return ResponseEntity.status(HttpStatus.OK).body(new
ResponseMessage(message));
      } catch (Exception e) {
        message = "Could not upload the file: " + file.getOriginalFilename() +
"!";
        return ResponseEntity.status(HttpStatus.EXPECTATION_FAILED).body(new
ResponseMessage(message));
      }
    }
    message = "Please upload an excel file!";
    return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(new
ResponseMessage(message));
  }
  @GetMapping("/tutorials")
  public ResponseEntity<List<Tutorial>> getAllTutorials() {
    try {
      List<Tutorial> tutorials = fileService.getAllTutorials();
      if (tutorials.isEmpty()) {
        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
      }
      return new ResponseEntity<>(tutorials, HttpStatus.OK);
    } catch (Exception e) {
      return new ResponseEntity<>(null, HttpStatus.INTERNAL_SERVER_ERROR);
    }
  }
}
```

Configure Multipart File for Servlet

Let’s define the maximum file size that can be uploaded in *application.properties* as following:

```
spring.servlet.multipart.max-file-size=2MB
spring.servlet.multipart.max-request-size=2MB
```

- `spring.servlet.multipart.max-file-size` : max file size for each request.
- `spring.servlet.multipart.max-request-size` : max request size for a multipart/form-data.

Handle File Upload Exception

This is where we handle the case in that a request exceeds Max Upload Size. The system will throw `MaxUploadSizeExceededException` and we’re gonna use `@ControllerAdvice` with `@ExceptionHandler` annotation for handling the exceptions.

exception/FileUploadExceptionAdvice.java

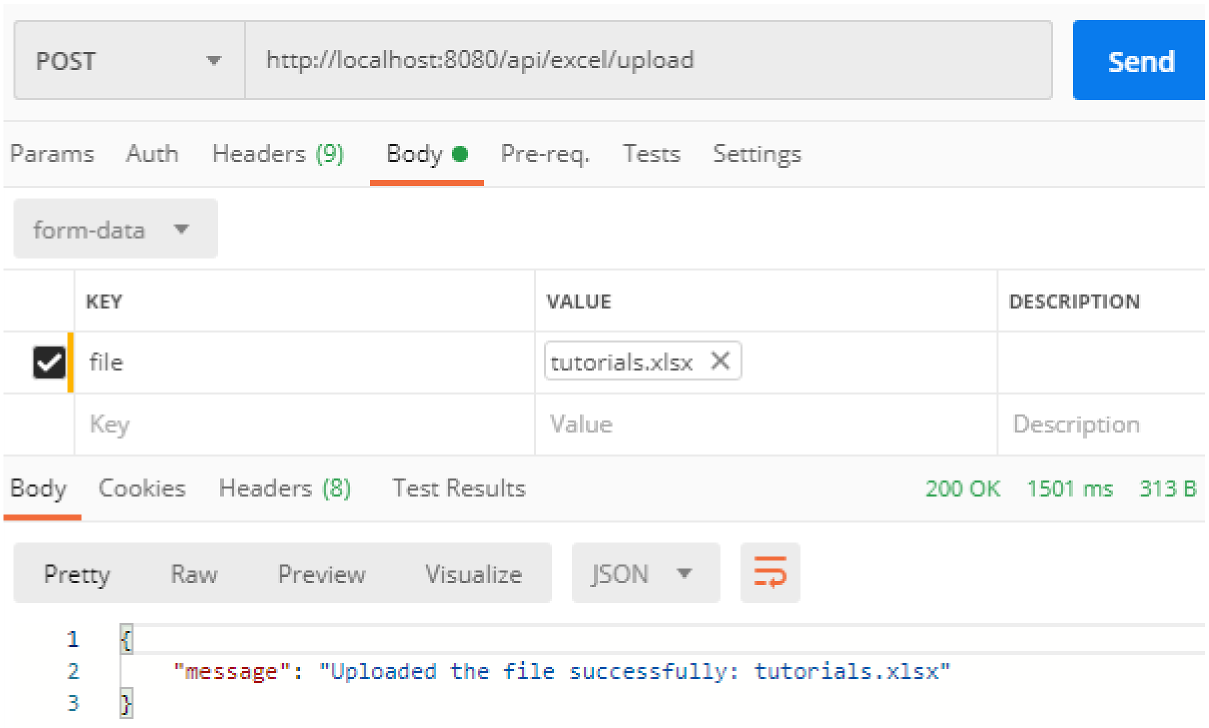
```
package com.bezkoder.spring.files.excel.exception;
import org.springframework.web.multipart.MaxUploadSizeExceededException;
import
org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandle

import com.bezkoder.spring.files.excel.message.ResponseMessage;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
@ControllerAdvice
public class FileUploadExceptionAdvice extends ResponseEntityExceptionHandler {
  @ExceptionHandler(MaxUploadSizeExceededException.class)
  public ResponseEntity<ResponseMessage>
handleMaxSizeException(MaxUploadSizeExceededException exc) {
    return ResponseEntity.status(HttpStatus.EXPECTATION_FAILED).body(new
ResponseMessage("File too large!"));
  }
}
```

Run & Check

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

Let’s use **Postman** to make some requests.



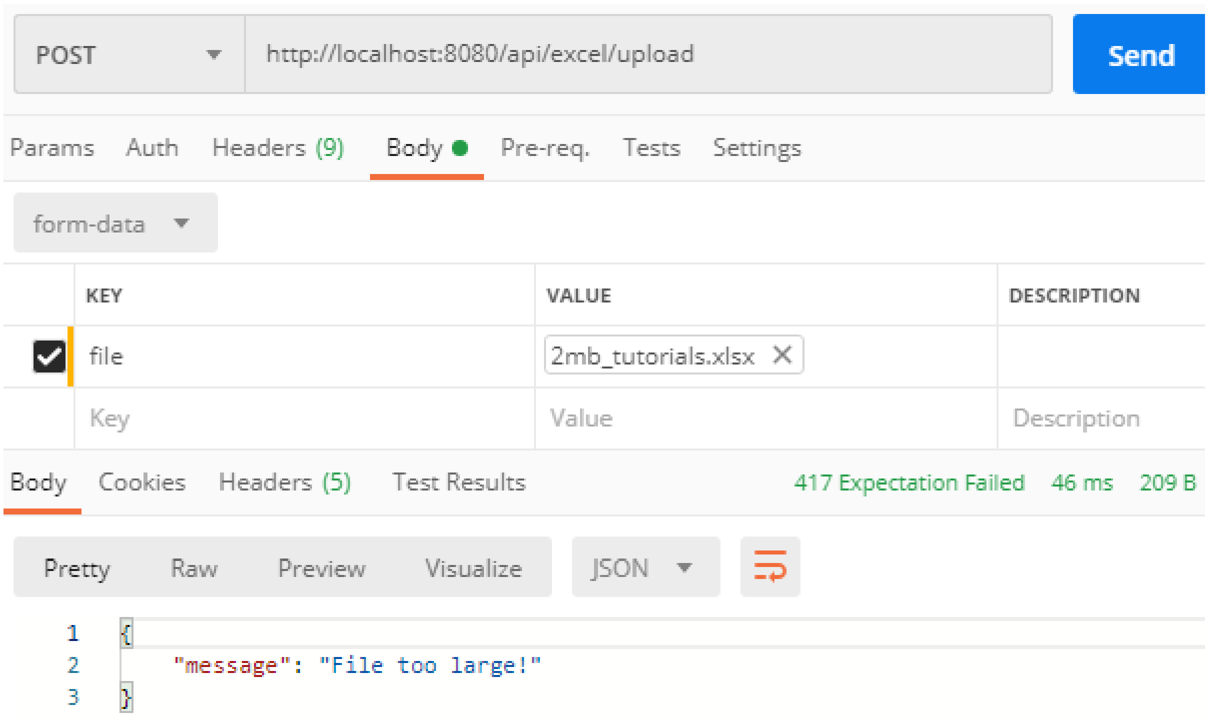
http://localhost:8080/api/excel/upload

KEY	VALUE	DESCRIPTION
file	tutorials.xlsx	
Key	Value	Description

Body Cookies Headers (8) Test Results 200 OK 1501 ms 313 B

```
{
    "message": "Uploaded the file successfully: tutorials.xlsx"
}
```

If you upload a file with size larger than max file size (2MB):



http://localhost:8080/api/excel/upload

KEY	VALUE	DESCRIPTION
file	2mb_tutorials.xlsx	
Key	Value	Description

Body Cookies Headers (5) Test Results 417 Expectation Failed 46 ms 209 B

```
{
    "message": "File too large!"
}
```

Conclusion

Today we've built a Rest CRUD API using Spring Boot to upload and import data from Excel file to Mysql database table.

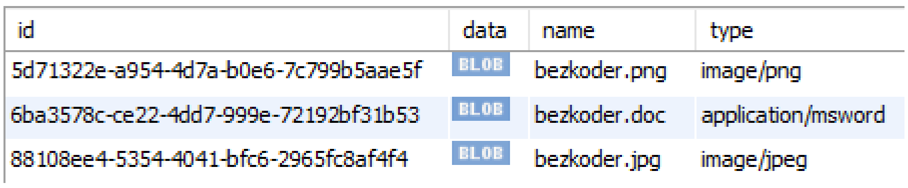
We also see how to use Apache POI to read data from Excel Sheet, `JpaRepository` to retrieve items in database table without need of boilerplate code.

If you want to add Pagination to this Spring project, you can find the instruction at:
Spring Boot Pagination & Filter example | Spring JPA, Pageable

For downloading Excel file:
Spring Boot: Download Excel file from MySQL database table

Upload CSV File instead:
Spring Boot: Upload CSV file Data into MySQL Database

Or upload files to database as BLOB:
Spring Boot Upload/Download File to/from Database example



id	data	name	type
5d71322e-a954-4d7a-b0e6-7c799b5aae5f	BLOB	bezkoder.png	image/png
6ba3578c-ce22-4dd7-999e-72192bf31b53	BLOB	bezkoder.doc	application/msword
88108ee4-5354-4041-bfc6-2965fc8af4f4	BLOB	bezkoder.jpg	image/jpeg

Happy learning! See you again.

Further Reading

Deployment:
– Deploy Spring Boot App on AWS – Elastic Beanstalk
– Docker Compose: Spring Boot and MySQL example

Source Code

You can find the complete source code for this tutorial on Github.