Contents

[Spring Boot REST Service – No database connectivity 2](#_Toc108687203)

[Steps 2](#_Toc108687204)

[Create a new Maven Project 2](#_Toc108687205)

[Add the Spring Boot dependencies to the project's *pom.xml*. 2](#_Toc108687206)

[Create a simple class with a *main* method and add the Spring Boot bits. 3](#_Toc108687207)

[Package / Source Layout 3](#_Toc108687208)

[Create an Object Class 4](#_Toc108687209)

[Create a DAO Class - ToDoDao.java 5](#_Toc108687210)

[Create a Persistence Class - ToDoInMemoryDao.java 6](#_Toc108687211)

[Create a Controller Class ToDoController 8](#_Toc108687212)

[Run the application using Postman 10](#_Toc108687213)

[Summary 15](#_Toc108687214)

[Spring Boot REST Service with JDBC Template 16](#_Toc108687215)

[pom.xml changes - Dependencies 16](#_Toc108687216)

[application.properties file 17](#_Toc108687217)

[Clean and Rebuild 17](#_Toc108687218)

[ToDoDatabaseDao 17](#_Toc108687219)

[Modify ToDoInMemoryDao 22](#_Toc108687220)

[Run and Test With Postman 22](#_Toc108687221)

[Spring Boot Profiles 22](#_Toc108687222)

[DAO Changes 23](#_Toc108687223)

[application.properties 23](#_Toc108687224)

[Common @Profile Uses 24](#_Toc108687225)

# Spring Boot REST Service – No database connectivity

# Steps

## Create a new Maven Project

Start with a new Maven project.

### Add the Spring Boot dependencies to the project's *pom.xml*.

<!-- snip -->

</properties>

<!-- (Don't include the above. Provided for context.) -->

<!--The Spring Boot starter parent. Controls <dependencies> below.-->

<parent>

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

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

<version>2.1.0.RELEASE</version>

<relativePath />

</parent>

<dependencies>

<!--Spring Boot starter children. No versions needed.-->

<dependency>

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

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

</dependency>

<dependency>

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

<artifactId>spring-boot-devtools</artifactId>

<scope>runtime</scope>

</dependency>

</dependencies>

<!-- snip -->

Clean and build.

## Create a simple class with a *main* method and add the Spring Boot bits.

package corbos.todoapi;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class App {

public static void main(String[] args) {

SpringApplication.run(App.class, args);

}

}

## Package / Source Layout

Our plan:

[base.package]

App.java <-- main method goes here

[base.package].controllers

ToDoController.java

[base.package].data

ToDoDao.java <-- interface

ToDoInMemoryDao.java <-- concrete implementation of ToDoDao

[base.package].models

ToDo.java <-- the class from our Jdbc projects

Work from back to front.

## Create an Object Class

This class is identical to the class used in our JDBC projects.

package corbos.todoapi.models;

public class ToDo {

private int id;

private String todo;

private String note;

private boolean finished;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getTodo() {

return todo;

}

public void setTodo(String todo) {

this.todo = todo;

}

public String getNote() {

return note;

}

public void setNote(String note) {

this.note = note;

}

public boolean isFinished() {

return finished;

}

public void setFinished(boolean finished) {

this.finished = finished;

}

}

### Create a DAO Interface / Class - ToDoDao.java

This interface defines our ToDo CRUD operations.

package corbos.todoapi.data;

import corbos.todoapi.models.ToDo;

import java.util.List;

public interface ToDoDao {

ToDo add(ToDo todo);

List<ToDo> getAll();

ToDo findById(int id);

// true if item exists and is updated

boolean update(ToDo todo);

// true if item exists and is deleted

boolean deleteById(int id);

}

Not much to discuss here. We choose to return booleans for *update* and *deleteById* to indicate both that the source exists and that the operation succeeded. *add* returns a ToDo with a generated id.

### Create a Persistence Class - ToDoInMemoryDao.java

This is a concrete implementation of *ToDoDao*. Data is stored in memory.

package corbos.todoapi.data;

import corbos.todoapi.models.ToDo;

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Collectors;

import org.springframework.stereotype.Repository;

@Repository

public class ToDoInMemoryDao implements ToDoDao {

private static final List<ToDo> todos = new ArrayList<>();

@Override

public ToDo add(ToDo todo) {

int nextId = todos.stream()

.mapToInt(i -> i.getId())

.max()

.orElse(0) + 1;

todo.setId(nextId);

todos.add(todo);

return todo;

}

@Override

public List<ToDo> getAll() {

return new ArrayList<>(todos);

}

@Override

public ToDo findById(int id) {

return todos.stream()

.filter(i -> i.getId() == id)

.findFirst()

.orElse(null);

}

@Override

public boolean update(ToDo todo) {

int index = 0;

while (index < todos.size()

&& todos.get(index).getId() != todo.getId()) {

index++;

}

if (index < todos.size()) {

todos.set(index, todo);

}

return index < todos.size();

}

@Override

public boolean deleteById(int id) {

return todos.removeIf(i -> i.getId() == id);

}

}

The code is standard collection manipulation, but it has one interesting bit: *@Repository*. The *@Repository* annotation is a class-level annotation that tells Spring this is an injectable bean. Functionally, it's the same as *@Component*, but *@Repository* carries a bit more semantic value. The **repository pattern** presents a collection-like interface for a data store. We add, update, find, get, or delete without needing to know exactly how the data is stored. Most DAOs are repositories (or close to it), so we prefer *@Repository* over *@Component*.

### Create a Controller Class ToDoController

All of our code so far is pretty standard. We've seen it before. It works well in command line applications, but it works just as well on the web! With Spring MVC and Spring Boot, there's very little code needed to expose our code to the web. It's not much, but the code is interesting, so let's get to it.

ToDoController.java -- Initial

package corbos.todoapi.controllers;

import corbos.todoapi.data.ToDoDao;

import corbos.todoapi.models.ToDo;

import java.util.List;

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

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

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

@RestController

@RequestMapping("/api/todo")

public class ToDoController {

private final ToDoDao dao;

public ToDoController(ToDoDao dao) {

this.dao = dao;

}

@GetMapping

public List<ToDo> all() {

return dao.getAll();

}

}

* The first thing to note is the constructor. The constructor takes a *ToDoDao* as a dependency. Spring satisfies the dependency with the *ToDoInMemoryDao* class annotated with *@Repository*. It's perfectly valid to annotate the constructor or private member with *@Autowired* or *@Inject* to make the dependency explicit, but it's not required. Spring makes assumptions about our expectations since there are no other constructors.
* *@RestController* does three things:
  + Makes our class injectable. It will be injected into Spring MVC core dependents.
  + Tells Spring MVC to scan for methods that can handle HTTP requests.
  + Tells Spring MVC to convert method results to JSON.
* *@RequestMapping("/api/todo")* requires the prefix "/api/todo" on all requests for ToDoController. It's common to require the name of the resource (*todo* in our case) in all HTTP requests for the resource. We include "api" as an indication that this is a remote API. It's a personal preference, not a requirement.
* The *all* method is annotated with *@GetMapping*. That means it will be activated for GET requests for the path "/api/todo". It's returned type, *List<ToDo>* will be serialized to JSON in the HTTP response body.

Run the application and use Postman to send a GET request to http://localhost:8080/api/todo. What's the result? If all goes well, you should see an empty JSON array. That makes sense: our ToDos are stored in memory and we haven't created one yet. We'll do that next.

create

@PostMapping

@ResponseStatus(HttpStatus.CREATED)

public ToDo create(@RequestBody ToDo todo) {

return dao.add(todo);

}

* *@PostMapping* enables our method to accept POST requests for the path "/api/todo".
* *@ResponseStatus(HttpStatus.CREATED)* sets the HTTP status code to *204 Created* for all responses generated from this method. That may not be exactly what we want. What happens if the ToDo is invalid or the operation fails? For now, we opt for simplicity, but we'll probably want to revisit this.
* Look closely at this: *@RequestBody ToDo todo*. Our *create* method takes a ToDo as a parameter. That may not seem amazing, but it is! Spring MVC interrogates the HTTP request and looks for data that it can transform into a ToDo. We don't have to do anything (other than ensuring the data is present, is named correctly, and is in the right place). The framework does it for us. This is sometimes called **model binding**.

*@RequestBody* tells Spring MVC to expect the data fully serialized in the HTTP request body. Spring MVC can also find data in the URL or form elements, but here we want JSON in the body.

Run the application using Postman

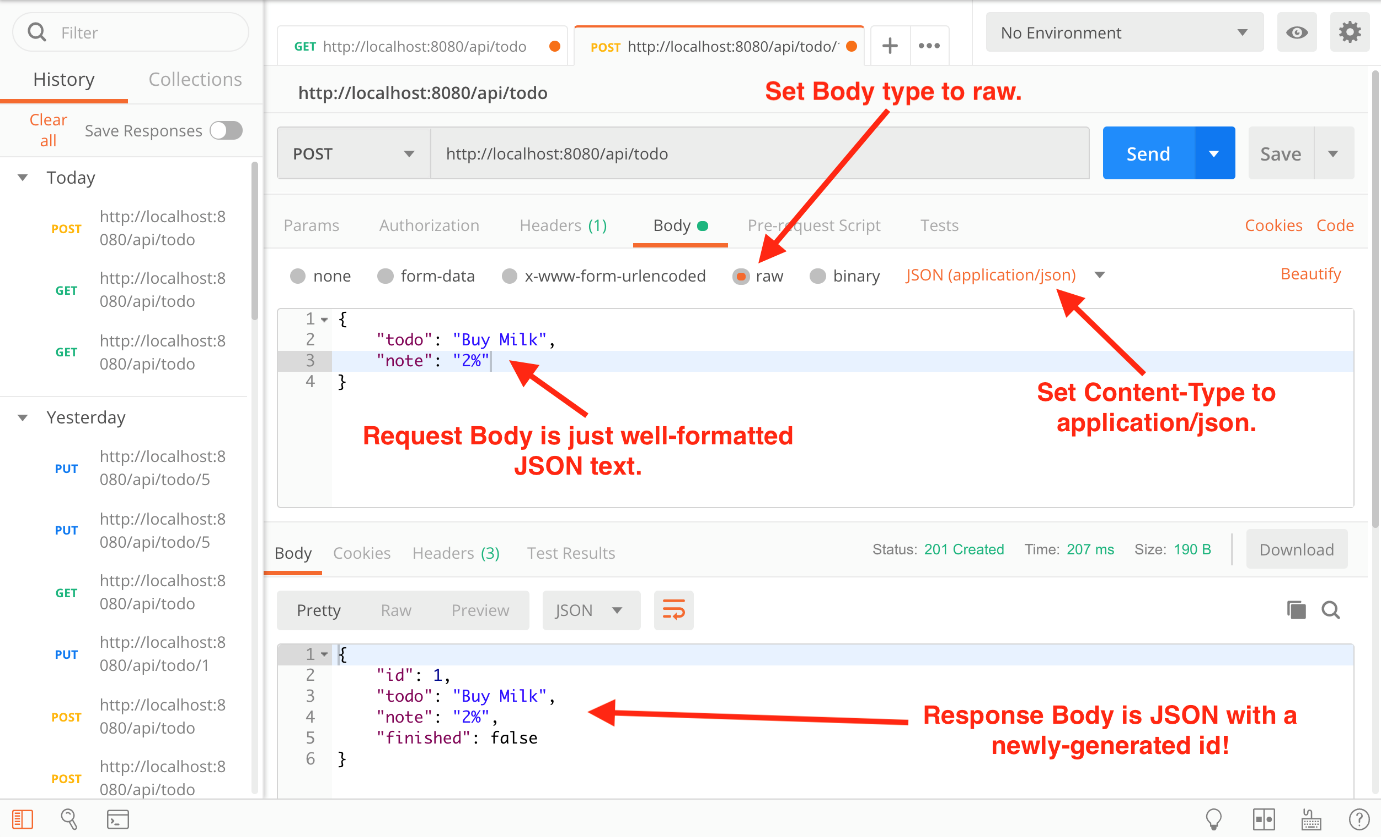
Then use Postman to send a POST request that activates our *create* method. There are a few extra steps in Postman.

* Select the request's *Body* tab.
* Select *raw*. This indicates we'll enter "raw" text versus adding key values. We write our JSON as text.
* Select *JSON (application/json)* in the Content-Type dropdown (it's not labeled).
* Enter a ToDo in JSON format:
* {
* "todo": "Buy Milk",
* "note": "2%"

}

We don't specify an *id* since it will be assigned or *finished* since we're okay with the default value.

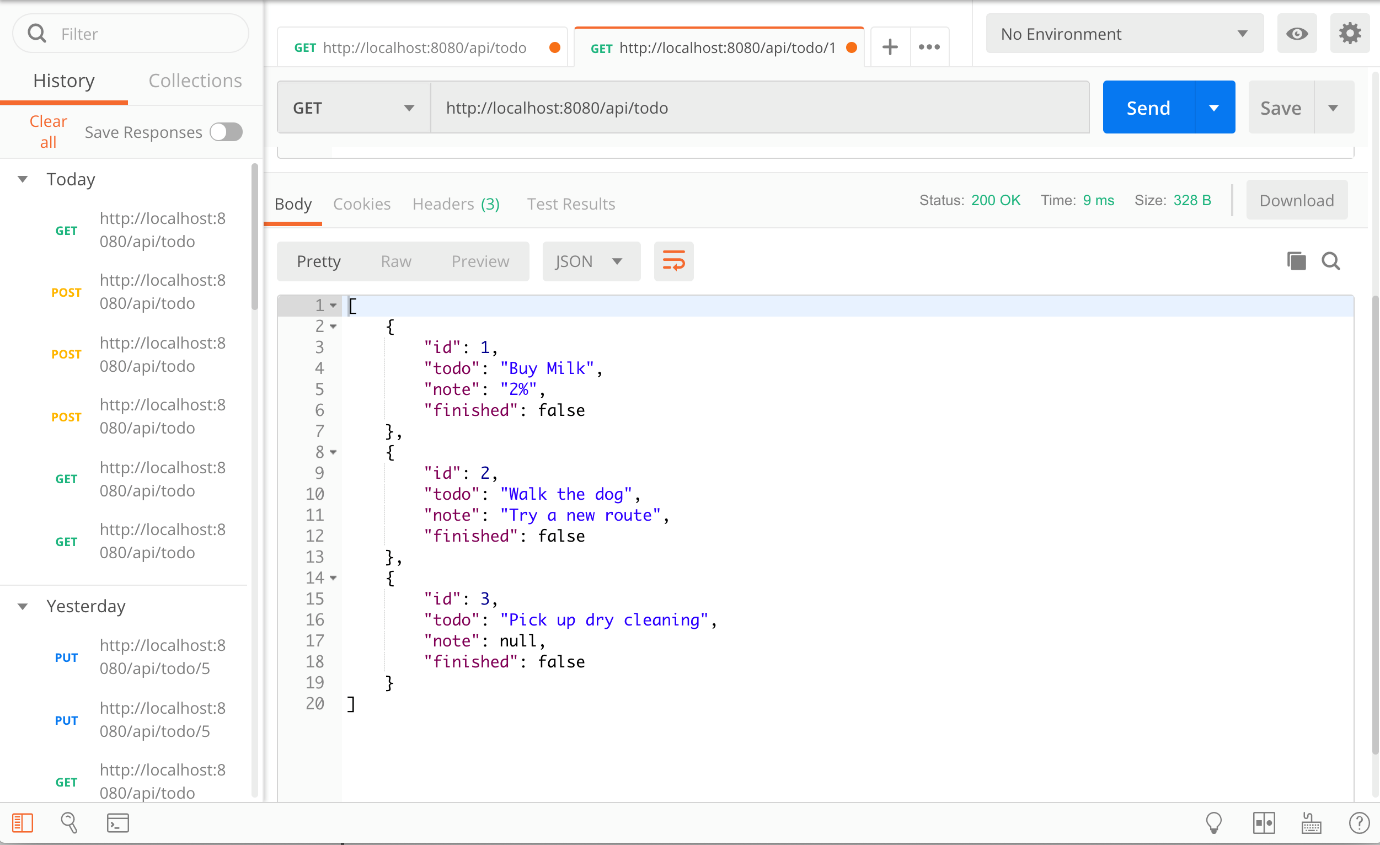
* Send!



If all is well, we should see a JSON ToDo in the HTTP response body. The *create* method returns a ToDo. That's where the JSON ToDo came from. It includes all fields. Its *id* is freshly generated and it is not *finished*.

Spring MVC's *@RequestBody* (provide a parameter from a serialized object in the request body) has a partner in *@ResponseBody* (serialize the return value to the response body). We don't need *@ResponseBody* in a *@RestController* because it's automatically applied to each method that returns a plain old object. It is required in a *@Controller*, introduced later.

Create a couple more ToDos. Then use Postman to GET them all.



findById

It's common to fetch a single item using its unique identifier. What happens if an item doesn't exist? It would be nice to return *200 OK* when we find what we're looking for and *404 Not Found* when we don't. Luckily, there's an easy way to do that.

@GetMapping("/{id}")

public ResponseEntity<ToDo> findById(@PathVariable int id) {

ToDo result = dao.findById(id);

if (result == null) {

return new ResponseEntity(null, HttpStatus.NOT\_FOUND);

}

return ResponseEntity.ok(result);

}

* The biggest new concept in this method is *ResponseEntity<T>*. *ResponseEntity<T>* is a Spring MVC type designed to add flexibility to generating HTTP responses. It can include a data payload as its generic type, accepts an explicit HTTP status code, and includes several static utility methods for quickly generating responses.

*ResponseEntity<ToDo>* indicates this method can include a JSON ToDo in its body, but it has the flexibility of sending different statuses. If the requested ToDo is not found, we return *null* in the body and a *404 Not Found* status. Otherwise, we use the utility method *ResponseEntity.ok* to serialize our ToDo and send *200 OK*.

* *@GetMapping("/{id}")* adds a named dynamic chunk to the base URL. *findById* is activated by the GET method on a URL ending with an id. e.g. "/api/todo/314"
* *@PathVariable* tells Spring MVC to expect a value, id, in the URL and convert it to an integer.

Use Postman to GET a single ToDo. Try to fetch some ToDos that exist and some that don't. Note the HTTP statuses.

update

@PutMapping("/{id}")

public ResponseEntity update(@PathVariable int id, @RequestBody ToDo todo) {

ResponseEntity response = new ResponseEntity(HttpStatus.NOT\_FOUND);

if (id != todo.getId()) {

response = new ResponseEntity(HttpStatus.UNPROCESSABLE\_ENTITY);

} else if (dao.update(todo)) {

response = new ResponseEntity(HttpStatus.NO\_CONTENT);

}

return response;

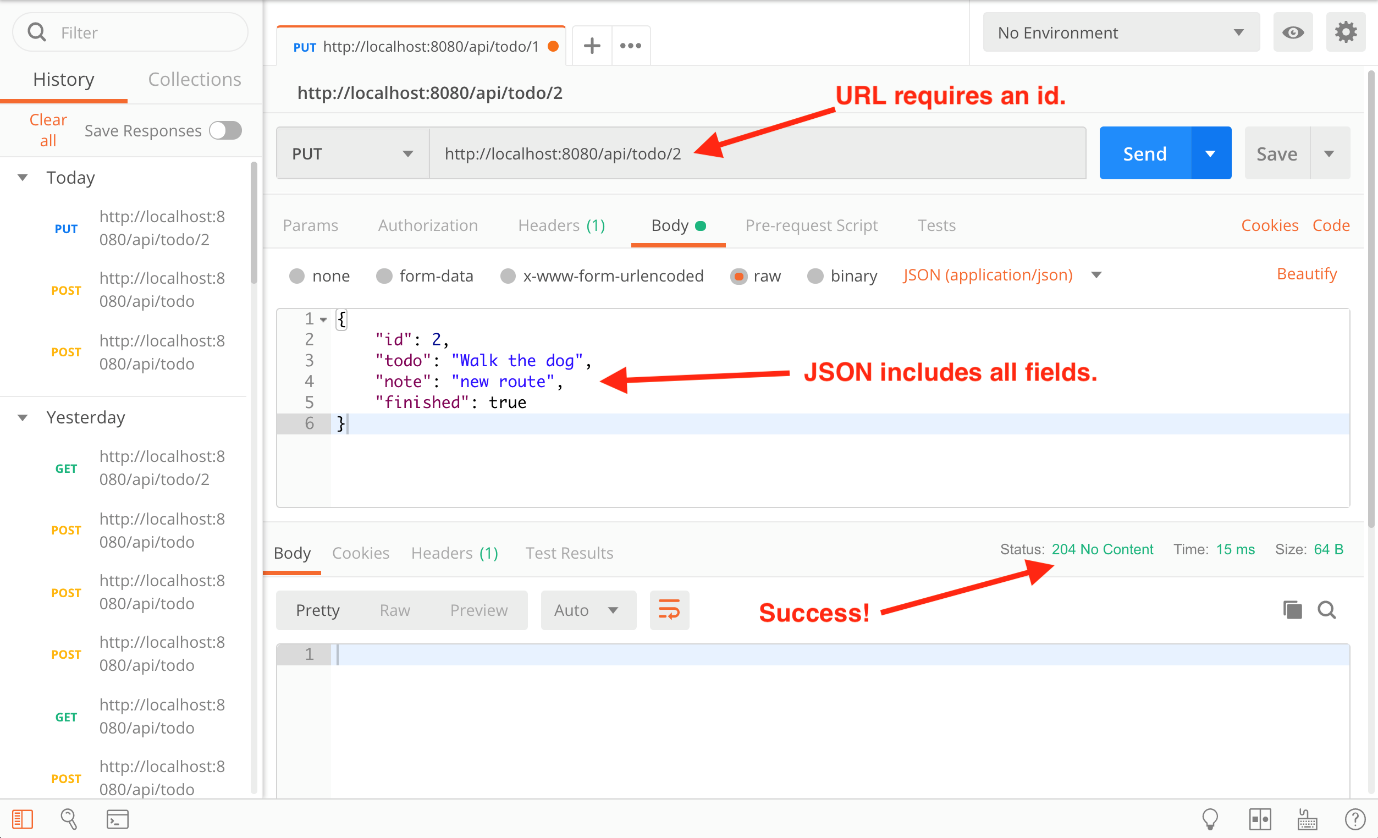
}

* *@PutMapping("/{id}")* - Our method responds to PUT on URLs like "/api/todo/123" and "/api/todo/81".
* *ResponseEntity* is type-less. When we update a ToDo, there's nothing to return, so a type-less *ResponseEntity* makes sense. We use *ResponseEntity* to return an appropriate HTTP status.
* *update* has two parameters. We read an id from the URL as a *@PathVariable* and we read a full ToDo from the HTTP body as *@RequestBody*. Controller methods can have as many parameters as we like as long as Spring MVC knows where to find them.

Maybe you're wondering why we would pass a ToDo's id in the URL *and* a full ToDo in the request body. If we have a full ToDo, is it necessary to pass the id a second time in the URL? It's a fair question. In this case, we follow the convention that every URL that operates on an existing resource uses the form "/api/todo/{id}". The particular operation is specified by the request method: GET, PUT, and DELETE. There's a nice consistency.

* There are two new HTTP status code. *422 Unprocessable Entity* indicates there's something wrong with the request so it can't complete successfully. In this case, the URL id and the body's ToDo id conflict so we can't proceed. Which one is right? *204 No Content* is a success status that politely tells the client not to expect content in the response body. Since there's nothing for *update* to return, this makes perfect sense.

Use Postman to send a PUT request that updates an existing ToDo's *finished* property to *true*. Be careful with your JSON. If you omit a property, it will revert to the default value. So if you forget to specify a ToDo name/title (called *todo* in our class), it will be set to *null*.



Send a few more PUT requests. What happens when you try to update a ToDo that doesn't exist? What happens when the ToDo id in the URL differs from the JSON ToDo id? What happens if you omit the JSON body altogether?

delete

Our final CRUD operation is [D]elete.

@DeleteMapping("/{id}")

public ResponseEntity delete(@PathVariable int id) {

if (dao.deleteById(id)) {

return new ResponseEntity(HttpStatus.NO\_CONTENT);

}

return new ResponseEntity(HttpStatus.NOT\_FOUND);

}

* *@DeleteMapping("/{id}")* - responds to DELETE on "/api/todo/{id}"
* Type-less *ResponseEntity*
* Returns *404 Not Found* when the requested id does not exist, otherwise *204 No Content*.

Use Postman to DELETE a ToDo.

Summary

A REST API uses HTTP requests and responses to read and write data. The request URL identifies the resource, and HTTP methods identify the operation. GET reads resource data, POST creates a resource, PUT updates a resource, and DELETE deletes a resource.

Spring MVC's *@RestController* takes a lot of the work out of creating a REST API. The DAO and service components we've used in the past are just as useful in a web service. The methods in a class annotated with *@RestController* are capable of accepting HTTP request data as parameters and returning a result as an HTTP response.

Spring MVC uses annotations to specify the URL and HTTP method that activates a controller method. Examples include: *@RequestMapping*, *@GetMapping*, and *@PutMapping*.

Spring MVC also uses annotations to specify where to find method parameters in the HTTP request and how to configure the HTTP response. Examples include: *@ResponseStatus*, *@PathVariable*, and *@RequestBody*.

The *ResponseEntity<T>* class is a flexible way to create HTTP responses

## Spring Boot REST Service with JDBC Template

### pom.xml changes - Dependencies

We no longer need specific versions for **spring-boot-starter-jdbc** and **mysql-connector-java**. The Spring Boot parent controls the versions.

<!--The Spring Boot starter parent. Controls <dependencies> below.-->

<parent>

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

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

<version>2.1.0.RELEASE</version>

<relativePath />

</parent>

<dependencies>

<!--Spring Boot starter children. No versions needed.-->

<!--Enables and configures Spring MVC. -->

<dependency>

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

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

</dependency>

<!--Database dependencies -->

<dependency>

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

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

</dependency>

<dependency>

<groupId>mysql</groupId>

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

</dependency>

<!-- Spring Boot Dev Tools -->

<dependency>

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

<artifactId>spring-boot-devtools</artifactId>

<scope>runtime</scope>

</dependency>

</dependencies>

This code block lacks a bit of context. See if you can put it where it belongs in *pom.xml*.

### application.properties file

If it doesn't exist (and it probably doesn't), add the file "/ToDoAPI/src/main/resources/application.properties" to our project. Then open it and put the database configuration inside.

spring.datasource.url=jdbc:mysql://localhost:3306/todoDB?serverTimezone=UTC&useSSL=false

spring.datasource.username=root

spring.datasource.password=your-root-password

The *application.properties* file is very important to Spring Boot. It's the central location for all configuration. We'll add to it as we add features to our applications.

Clean and Rebuild

This will update our Maven dependencies.

### ToDoDatabaseDao

Add a new class, *ToDoDatabaseDao*, to the **[base-package].data** package.

package corbos.todoapi.data;

import corbos.todoapi.models.ToDo;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

import java.util.List;

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

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.core.RowMapper;

import org.springframework.jdbc.support.GeneratedKeyHolder;

import org.springframework.stereotype.Repository;

@Repository

public class ToDoDatabaseDao implements ToDoDao {

private final JdbcTemplate jdbcTemplate;

@Autowired

public ToDoDatabaseDao(JdbcTemplate jdbcTemplate) {

this.jdbcTemplate = jdbcTemplate;

}

@Override

public ToDo add(ToDo todo) {

final String sql = "INSERT INTO todo(todo, note) VALUES(?,?);";

GeneratedKeyHolder keyHolder = new GeneratedKeyHolder();

jdbcTemplate.update((Connection conn) -> {

PreparedStatement statement = conn.prepareStatement(

sql,

Statement.RETURN\_GENERATED\_KEYS);

statement.setString(1, todo.getTodo());

statement.setString(2, todo.getNote());

return statement;

}, keyHolder);

todo.setId(keyHolder.getKey().intValue());

return todo;

}

@Override

public List<ToDo> getAll() {

final String sql = "SELECT id, todo, note, finished FROM todo;";

return jdbcTemplate.query(sql, new ToDoMapper());

}

@Override

public ToDo findById(int id) {

final String sql = "SELECT id, todo, note, finished "

+ "FROM todo WHERE id = ?;";

return jdbcTemplate.queryForObject(sql, new ToDoMapper(), id);

}

@Override

public boolean update(ToDo todo) {

final String sql = "UPDATE todo SET "

+ "todo = ?, "

+ "note = ?, "

+ "finished = ? "

+ "WHERE id = ?;";

return jdbcTemplate.update(sql,

todo.getTodo(),

todo.getNote(),

todo.isFinished(),

todo.getId()) > 0;

}

@Override

public boolean deleteById(int id) {

final String sql = "DELETE FROM todo WHERE id = ?;";

return jdbcTemplate.update(sql, id) > 0;

}

private static final class ToDoMapper implements RowMapper<ToDo> {

@Override

public ToDo mapRow(ResultSet rs, int index) throws SQLException {

ToDo td = new ToDo();

td.setId(rs.getInt("id"));

td.setTodo(rs.getString("todo"));

td.setNote(rs.getString("note"));

td.setFinished(rs.getBoolean("finished"));

return td;

}

}

}

That's a lot at once, but we can break it down.

* *ToDoDatabaseDao* implements *ToDoDao* so every public operation is identical to our in-memory DAO.
* *@Repository* makes our class an injectable dependency, just like *ToDoInMemoryDao*. How does Spring DI know which version to inject into *ToDoController*? Well, it doesn't. Spring DI will get confused. We need to fix that. We'll remove the *@Repository* annotation from *ToDoInMemoryDao* next.
* *@Autowired* on the constructor asks Spring DI for a *JdbcTemplate*. That seems a little magical. Where is *JdbcTemplate* registered? The **spring-boot-starter-jdbc** package can create one with nothing but a bit of configuration.
* All SQL and the *ToDoMapper* were stolen from the JDBC Template lesson.
* *final* local variables prevent us from modifying variables that shouldn't be modified and allow the compiler to optimize our class.

There is one bit that deserves a closer look. The *add* method uses a *JdbcTemplate.update* overload to execute a SQL insert and grab the generated keys.

// snip...

GeneratedKeyHolder keyHolder = new GeneratedKeyHolder();

jdbcTemplate.update((Connection conn) -> {

PreparedStatement statement = conn.prepareStatement(sql, Statement.RETURN\_GENERATED\_KEYS);

statement.setString(1, todo.getTodo());

statement.setString(2, todo.getNote());

return statement;

}, keyHolder);

todo.setId(keyHolder.getKey().intValue());

// snip...

* This *JdbcTemplate.update* takes two parameters: a *PreparedStatementCreator* and a *KeyHolder*. Both are Spring JDBC types.
* *PreparedStatementCreator* is a functional interface with one method signature that generates a *PreparedStatement* from a *Connection*. Both *PreparedStatement* and *Connection* are plain old JDBC classes. JDBC is shining through.

We opt to satisfy the *PreparedStatementCreator* parameter with a lambda. The lambda is an unnamed method that takes a *Connection* as a parameter and returns a completed *PreparedStatement*.

* A *KeyHolder* is an interface that defines something that holds keys. We opt for the concrete class *GeneratedKeyHolder*.
* After the SQL insert has been executed, we can grab the generated keys from the *GeneratedKeyHolder*.

Modify ToDoInMemoryDao

Remove the *@Repository* annotation. This prevents ambiguity when Spring tries to decide which *ToDoDao* to inject.

// @Repository

public class ToDoInMemoryDao implements ToDoDao {

// snip...

}

Run and Test With Postman

With our changes complete, we should have swapped our new database DAO for the old in-memory DAO. Run ToDoAPI. Use Postman to send HTTP requests for all operations: get all, find by id, insert, update, and delete.

If you encounter errors, double check your database connection string and credentials. Confirm they work with MySQL Workbench. Double check the *todoDB* database. Ensure it exists and that its schema matches the JDBC lesson.

Even if you don't encounter errors, try adding a breakpoint in Netbeans. Debug your application and send a Postman request that ultimately hits your breakpoint. Use breakpoints and debugging to troubleshoot errors.

Spring Boot Profiles

Adding and removing the *@Repository* annotation is a real pain. It's not practical. Imagine having to manually change hundreds of dependencies every time we build for a different environment. There's no way to do it reliably.

Spring Boot profiles solve that problem. If we annotate an injectable class with *@Profile("profile-name")*, we can use it conditionally, depending on configuration. In our case, we can use either *ToDoInMemoryDao* or *ToDoDatabaseDao* depending on a value in our configuration file.

DAO Changes

Annotate *both* classes as *@Repository* and add *@Profile*.

@Repository

@Profile("memory")

public class ToDoInMemoryDao implements ToDoDao {

// snip...

}

@Repository

@Profile("database")

public class ToDoDatabaseDao implements ToDoDao {

// snip ...

}

application.properties

Add **spring.profiles.active** to application.properties.

spring.datasource.url=jdbc:mysql://localhost:3306/todoDB?serverTimezone=UTC&useSSL=false

spring.datasource.username=root

spring.datasource.password=your-root-password

# Set the active dependency profile as "database".

# This selects ToDoDatabaseDao as the ToDoController's ToDoDao.

spring.profiles.active=database

Run the application and send it a few requests with Postman. We should see data from the MySQL database.

Now change the profile.

spring.datasource.url=jdbc:mysql://localhost:3306/todoDB?serverTimezone=UTC&useSSL=false

spring.datasource.username=root

spring.datasource.password=your-root-password

# This selects ToDoInMemoryDao as the ToDoController's ToDoDao.

spring.profiles.active=memory

Run the application and GET all ToDos. The ToDo response should be empty since we're using the in-memory DAO and haven't inserted a ToDo yet. POST a ToDo and try again.

Common @Profile Uses

It's common to use profiles to change behavior across environments. Choosing one of two components ("memory"/"database") isn't nearly as powerful as choosing dependencies for a whole test or production environment. If you annotate all test environment dependencies with *@Profile("test")* and all of their production partners with *Profile("prod")*, it's easy to configure the appropriate set of dependencies when it's needed.