< **ALL GUIDES**

Building REST services with Spring

REST has quickly become the de-facto standard for building web services on the web because they’re easy to build and easy to consume.

There’s a much larger discussion to be had about how REST fits in the world of microservices, but — for this tutorial — let’s just look at building RESTful services.

Why REST? REST embraces the precepts of the web, including its architecture, benefits, and everything else. This is no surprise given its author, Roy Fielding, was involved in probably a dozen specs which govern how the web operates.

What benefits? The web and its core protocol, HTTP, provide a stack of features:

- Suitable actions (`GET`, `POST`, `PUT`, `DELETE`, …)
- Caching
- Redirection and forwarding
- Security (encryption and authentication)

These are all critical factors on building resilient services. But that is not all. The web is built out of lots of tiny specs, hence it’s been able to evolve easily, without getting bogged down in "standards wars".

Developers are able to draw upon 3rd party toolkits that implement these diverse specs and instantly have both client and server technology at their fingertips.

By building on top of HTTP, REST APIs provide the means to build:

- Backwards compatible APIs
- Evolvable APIs
- Scaleable services

- Securable services
- A spectrum of stateless to stateful services

What's important to realize is that REST, however ubiquitous, is not a standard, *per se*, but an approach, a style, a set of *constraints* on your architecture that can help you build web-scale systems. In this tutorial we will use the Spring portfolio to build a RESTful service while leveraging the stackless features of REST.

Getting Started

As we work through this tutorial, we'll use Spring Boot. Go to Spring Initializr and add the following dependencies to a project:

- Web
- JPA
- H2

Change the Name to "Payroll" and then choose "Generate Project". A `.zip` will download. Unzip it. Inside you'll find a simple, Maven-based project including a `pom.xml` build file (NOTE: You *can* use Gradle. The examples in this tutorial will be Maven-based.)

Spring Boot can work with any IDE. You can use Eclipse, IntelliJ IDEA, Netbeans, etc. The Spring Tool Suite is an open-source, Eclipse-based IDE distribution that provides a superset of the Java EE distribution of Eclipse. It includes features that make working with Spring applications even easier. It is, by no means, required. But consider it if you want that extra **oomph** for your keystrokes. Here's a video demonstrating how to get started with STS and Spring Boot. This is a general introduction to familiarize you with the tools.

The Story so Far...

Let’s start off with the simplest thing we can construct. In fact, to make it as simple as possible, we can even leave out the concepts of REST. (Later on, we’ll add REST to understand the difference.)

Big picture: We’re going to create a simple payroll service that manages the employees of a company. We’ll store employee objects in a (H2 in-memory) database, and access them (via something called JPA). Then we’ll wrap that with something that will allow access over the internet (called the Spring MVC layer).

The following code defines an Employee in our system.

nonrest/src/main/java/payroll/Employee.java

COPY

```java
package payroll;

import java.util.Objects;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
class Employee {

  private @Id @GeneratedValue Long id;
  private String name;
  private String role;

  Employee() {}

  Employee(String name, String role) {

    this.name = name;
    this.role = role;
  }

  public Long getId() {
    return this.id;
  }

  public String getName() {
    return this.name;
  }

  public String getRole() {
    return this.role;
  }
  public void setId(Long id) {
    this.id = id;
  }

  public void setName(String name) {
    this.name = name;
  }

  public void setRole(String role) {
    this.role = role;
  }

  @Override
  public boolean equals(Object o) {

    if (this == o)
      return true;
    if (!(o instanceof Employee))
      return false;
    Employee employee = (Employee) o;
    return Objects.equals(this.id, employee.id) && Objects.equals(this.name,
employee.name)
        && Objects.equals(this.role, employee.role);
  }

  @Override
  public int hashCode() {
    return Objects.hash(this.id, this.name, this.role);
  }

  @Override
  public String toString() {
    return "Employee{" + "id=" + this.id + ", name='" + this.name + '\'' + ",
role='" + this.role + '\'' + '}';
  }
}
```

Despite being small, this Java class contains much:

- `@Entity` is a JPA annotation to make this object ready for storage in a JPA-based data store.
- `id`, `name`, and `role` are attributes of our Employee domain object. `id` is marked with more JPA annotations to indicate it's the primary key and automatically populated by the JPA provider.
- a custom constructor is created when we need to create a new instance, but don't yet have an id.

With this domain object definition, we can now turn to Spring Data JPA to handle the tedious database interactions.

Spring Data JPA repositories are interfaces with methods supporting creating, reading, updating, and deleting records against a back end data store. Some repositories also support data paging, and sorting, where appropriate. Spring Data synthesizes implementations based on conventions found in the naming of the methods in the interface.

There are multiple repository implementations besides JPA. You can use Spring Data MongoDB, Spring Data GemFire, Spring Data Cassandra, etc. For this tutorial, we'll stick with JPA.

Spring makes accessing data easy. By simply declaring the following `EmployeeRepository` interface we automatically will be able to

- Create new Employees
- Update existing ones
- Delete Employees
- Find Employees (one, all, or search by simple or complex properties)

nonrest/src/main/java/payroll/EmployeeRepository.java

COPY

```
package payroll;

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

interface EmployeeRepository extends JpaRepository<Employee, Long> {

}
```

To get all this free functionality, all we had to do was declare an interface which extends Spring Data JPA's `JpaRepository`, specifying the domain type as `Employee` and the id type as `Long`.

Spring Data's repository solution makes it possible to sidestep data store specifics and instead solve a majority of problems using domain-specific terminology.

Believe it or not, this is enough to launch an application! A Spring Boot application is, at a minimum, a `public static void main` entry-point and the `@SpringBootApplication` annotation. This tells Spring Boot to help out, wherever possible.

nonrest/src/main/java/payroll/PayrollApplication.java

COPY

```
package payroll;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class PayrollApplication {

  public static void main(String... args) {
    SpringApplication.run(PayrollApplication.class, args);
  }
}
```

`@SpringBootApplication` is a meta-annotation that pulls in **component scanning**, **autoconfiguration**, and **property support**. We won't dive into the details of Spring Boot in this tutorial, but in essence, it will fire up a servlet container and serve up our service.

Nevertheless, an application with no data isn't very interesting, so let's preload it. The following class will get loaded automatically by Spring:

nonrest/src/main/java/payroll/LoadDatabase.java

COPY

```
package payroll;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
class LoadDatabase {

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

  @Bean
  CommandLineRunner initDatabase(EmployeeRepository repository) {

    return args -> {
      log.info("Preloading " + repository.save(new Employee("Bilbo Baggins",
"burglar")));
      log.info("Preloading " + repository.save(new Employee("Frodo Baggins",
"thief")));
    };
  }
}
```

What happens when it gets loaded?

- Spring Boot will run ALL `CommandLineRunner` beans once the application context is loaded.
- This runner will request a copy of the `EmployeeRepository` you just created.
- Using it, it will create two entities and store them.

Right-click and **Run** `PayRollApplication`, and this is what you get:

Fragment of console output showing preloading of data

```
...
2018-08-09 11:36:26.169  INFO 74611 --- [main] payroll.LoadDatabase : Preloading Emp
2018-08-09 11:36:26.174  INFO 74611 --- [main] payroll.LoadDatabase : Preloading Emp
...
```

This isn't the **whole** log, but just the key bits of preloading data. (Indeed, check out the whole console. It's glorious.)

HTTP is the Platform

To wrap your repository with a web layer, you must turn to Spring MVC. Thanks to Spring Boot, there is little in infrastructure to code. Instead, we can focus on actions:

nonrest/src/main/java/payroll/EmployeeController.java

```
package payroll;

import java.util.List;

import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
```

```
@RestController
class EmployeeController {

  private final EmployeeRepository repository;

  EmployeeController(EmployeeRepository repository) {
    this.repository = repository;
  }


  // Aggregate root
  // tag::get-aggregate-root[]
  @GetMapping("/employees")
  List<Employee> all() {
    return repository.findAll();
  }
  // end::get-aggregate-root[]

  @PostMapping("/employees")
  Employee newEmployee(@RequestBody Employee newEmployee) {
    return repository.save(newEmployee);
  }

  // Single item

  @GetMapping("/employees/{id}")
  Employee one(@PathVariable Long id) {

    return repository.findById(id)
      .orElseThrow(() -> new EmployeeNotFoundException(id));
  }

  @PutMapping("/employees/{id}")
  Employee replaceEmployee(@RequestBody Employee newEmployee, @PathVariable Long
id) {

    return repository.findById(id)
      .map(employee -> {
        employee.setName(newEmployee.getName());
        employee.setRole(newEmployee.getRole());
        return repository.save(employee);
      })
      .orElseGet(() -> {
        newEmployee.setId(id);
        return repository.save(newEmployee);
      });
  }

  @DeleteMapping("/employees/{id}")
  void deleteEmployee(@PathVariable Long id) {
    repository.deleteById(id);
  }
}
```

- `@RestController` indicates that the data returned by each method will be written straight into the response body instead of rendering a template.
- An `EmployeeRepository` is injected by constructor into the controller.
- We have routes for each operation (`@GetMapping`, `@PostMapping`, `@PutMapping` and `@DeleteMapping`, corresponding to HTTP `GET`, `POST`, `PUT`, and `DELETE` calls). (NOTE: It's useful to read each method and understand what they do.)
- `EmployeeNotFoundException` is an exception used to indicate when an employee is looked up but not found.

nonrest/src/main/java/payroll/EmployeeNotFoundException.java

COPY

```
package payroll;

class EmployeeNotFoundException extends RuntimeException {

  EmployeeNotFoundException(Long id) {
    super("Could not find employee " + id);
  }
}
```

When an `EmployeeNotFoundException` is thrown, this extra tidbit of Spring MVC configuration is used to render an **HTTP 404**:

nonrest/src/main/java/payroll/EmployeeNotFoundAdvice.java

COPY

```
package payroll;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;

@ControllerAdvice
class EmployeeNotFoundAdvice {

  @ResponseBody
  @ExceptionHandler(EmployeeNotFoundException.class)
  @ResponseStatus(HttpStatus.NOT_FOUND)
  String employeeNotFoundHandler(EmployeeNotFoundException ex) {
    return ex.getMessage();
  }
}
```

- `@ResponseBody` signals that this advice is rendered straight into the response body.
- `@ExceptionHandler` configures the advice to only respond if an `EmployeeNotFoundException` is thrown.
- `@ResponseStatus` says to issue an `HttpStatus.NOT_FOUND`, i.e. an **HTTP 404**.
- The body of the advice generates the content. In this case, it gives the message of the exception.

To launch the application, either right-click the `public static void main` in `PayRollApplication` and select **Run** from your IDE, or:

Spring Initializr uses maven wrapper so type this:

```
$ ./mvnw clean spring-boot:run
```

Alternatively using your installed maven version type this:

```
$ mvn clean spring-boot:run
```

When the app starts, we can immediately interrogate it.

```
$ curl -v localhost:8080/employees
```

This will yield:

```
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
> GET /employees HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 200
< Content-Type: application/json;charset=UTF-8
< Transfer-Encoding: chunked
< Date: Thu, 09 Aug 2018 17:58:00 GMT
<
* Connection #0 to host localhost left intact
[{"id":1,"name":"Bilbo Baggins","role":"burglar"},{"id":2,"name":"Frodo Baggins","ro
```

Here you can see the pre-loaded data, in a compacted format.

If you try and query a user that doesn't exist…

```
$ curl -v localhost:8080/employees/99
```

You get…

```
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
> GET /employees/99 HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 404
< Content-Type: text/plain;charset=UTF-8
< Content-Length: 26
< Date: Thu, 09 Aug 2018 18:00:56 GMT
<
* Connection #0 to host localhost left intact
Could not find employee 99
```

This message nicely shows an **HTTP 404** error with the custom message **Could not find employee 99**.

It's not hard to show the currently coded interactions…

If you are using Windows Command Prompt to issue cURL commands, chances are the below command won't work properly. You must either pick a terminal that support single quoted arguments, or use double quotes and then escape the ones inside the JSON.

To create a new `Employee` record we use the following command in a terminal—the `$` at the beginning signifies that what follows it is a terminal command:

```
$ curl -X POST localhost:8080/employees -H 'Content-type:application/json' -d '{"nam
```

Then it stores newly created employee and sends it back to us:

```
{"id":3,"name":"Samwise Gamgee","role":"gardener"}
```

You can update the user. Let’s change his role.

```
$ curl -X PUT localhost:8080/employees/3 -H 'Content-type:application/json' -d '{"na
```

And we can see the change reflected in the output.

```
{"id":3,"name":"Samwise Gamgee","role":"ring bearer"}
```

> The way you construct your service can have significant impacts. In this situation, we said **update**, but **replace** is a better description. For example, if the name was NOT provided, it would instead get nulled out.

Finally, you can delete users like this:

```
$ curl -X DELETE localhost:8080/employees/3

# Now if we look again, it's gone
$ curl localhost:8080/employees/3
Could not find employee 3
```

This is all well and good, but do we have a RESTful service yet? (If you didn’t catch the hint, the answer is no.)

What’s missing?

What makes something RESTful?

So far, you have a web-based service that handles the core operations involving employee data. But that’s not enough to make things "RESTful".

- Pretty URLs like `/employees/3` aren’t REST.
- Merely using `GET`, `POST`, etc. isn’t REST.