Web Services, esp. using Spring

1

Last time: Intro to Web Services

- REST web service requests are just HTTP requests with certain conventions holding
- Everything is a resource, specified by a URI/URL: topping 12 could be https://myserver:9002/pizzaservice/toppings/12, and order 201
- https://myserver:9002/pizzaservice/orders/201
- Here the server is at https://myserver:9002
- The service ("base URL") is at https://myserver:9002/pizzaservice
- And the individual resources handled by the service have the longer URLs.
- Since all the resource URLs all start with the base URL, we usually only quote the relative URL, like /toppings/12 here.

2

HTTP verbs can go a long way…

The resources have URLs, and "representations" in JSON or XML

We use HTTP verbs to act on the resources (CRUD actions)
- GET for reading data (no changes allowed in server!): get the representation
- POST for creating new data items: send in a new representation
- PUT for updating old data items (in whole): replace the rep.
- DELETE for deleting old data items
- Not all actions can be squeezed into these compartments of course.

- A request to get the resource at https://myserver/pizzaservice/toppings/12 would involve a TCP connection to myserver:9002, then GET /pizzaservice/toppings/12 HTTP/1.1 using that TCP connection.
- We usually abbreviate this to "GET /toppings/12" leaving off the base URL part and the HTTP version.

3

POSTing a new resource

How does the client find out the new id after a POST?

The HTTP Location header in the response gives this. We saw an example last time:
```
POST /edit/ HTTP/1.1
… new feed entry
Response:
    HTTP/1.1 201 Created
    Date: Fri, 7 Oct 2005 17:17:11 GMT
    Content-Length: nnn
    Content-Type: application/atom+xml;type=entry;charset="utf-8"
    Location: http://example.org/edit/first-post.atom
    ETag: "c180de84f991g8"
```

Often this information is also in the POST response body, but it's better to find it in this standardized spot.

4

REST in Java: server side

We have been using servlets to accept HTTP requests coming in, with and without the help of Spring controllers.

In a web app, usually we know the incoming request URLs pretty completely, for example:

GET /orderPizza.html?room=6&toppings=Onions&size=small

This is all handled by @RequestMapping("orderPizza.html")

But in REST, we might need to serve /products/guitars/10 and /product/basses/2 and … where new categories might be added over time

How is that possible? How can we get a servlet to handle more-or-less arbitrary incoming URLs?

Answer: use wildcards in the URL pattern—they can be used at start or end of the string.

5

REST in Java: server side

How can we get a servlet to handle more-or-less arbitrary incoming URLs?

Answer: use wildcards in the URL pattern—they can be used at start or end of the string.

Example: pizza3's SystTestServlet has
```
@WebServlet(
    name = "SysTestServlet",
    description = "Servlet to run SystemTest",
    urlPatterns = {"/servlet/SystemTestServlet"}
)
```

We could change this to /servlet/* and accept /servlet/abc and /servlet/xyz and …

Then in doGet/doPost/doPut/doDelete we can find out the actual incoming URL with request.getRequestURI(). It would return "/servlet/abc" or "/servlet/xyz" or …

6

REST in Java: client side

Note that all the GETs and POSTs we've been generating so far have been actually generated by the *browser* interpreting HTML that we have composed.

Now if we want to utilize a Web service from a Java program, we need to get our Java code to generate a HTTP request directly.

The JDK must have support for this…

A plain GET is pretty easy: (using import java.net.* to get the needed JDK classes)

```
URL url = new URL("http://example.com");
HttpURLConnection con = (HttpURLConnection) url.openConnection();
con.setRequestMethod("GET");
OutputStream out = connection.getOutputStream();
// then read response body using out
```

For details, see https://www.baeldung.com/java-http-request
Even just adding request parameters to this is complicated. That's included in Baeldung's doc, as are headers, cookies, and handling redirection. But not there: writing out the POST body.

7

More convenient ways to do this

We usually want JSON on the wire, so need to use JSON parser, JSON generation.

Surprisingly, the JDK doesn't support JSON parsing or generation, but it is in JEE libs and avail via Maven. Spring uses Jackson, another JSON serializer/deserializer.

For serious project, should use a framework, let it handle JSON for you.

JEE has spec JAX-RS for a server-side REST framework

Jersey provides an implementation of this, with client-side libraries as well. For the server side, it provides a servlet to use in any servlet container (tomcat or Glassfish or whatever) to handle incoming REST requests. That servlet uses a wildcard URL pattern, of course.
- Good tutorial on Jersey: firstRest at vogella.com. Note you don't need to use gradle: see https://howtodoinjava.com/jersey/jax-rs-jersey-hello-world-example/ for a Maven pom.xml.
- Jersey also has client-side support.

Spring Boot makes it easy to support REST web services.

8

Examples to look at

Server-side tutorial at Building REST services with Spring (first 12 pages)
- Example handles employees with GET, POST, PUT, and DELETE
- Code at github https://github.com/spring-guides/tut-rest

Client-side tutorial: Consuming a RESTful Web Service
- Client accesses a server of quotes about Spring on the Internet
- Code at github: https://github.com/spring-guides/gs-consuming-rest

Both of these use H2, so easy to run

We'll look at my mod of the client-side project to make it get data from server-side tutorial's server, i.e., employees.

9

REST service for employees

REST API using "URI templates": here {id} stands for a numeric id

GET /employees

GET /employees/{id}

POST /employees

PUT /employees/{id}

DELETE /employees/{id}

This is a CRUD API. To change someone's "role", we would GET that employee object, change its role property, and then PUT it back.

It is possible to have action-driven (change-role for ex.) calls in a REST API, but first look at CRUD…

10

The domain object: Employee

```
package payroll;
import …
@Entity          ←JPA entity
class Employee {
  private @Id @GeneratedValue Long id;  ← works for H2 (Oracle??)
  private String name;     ← two properties, name and role
  private String role;

  Employee() {}
  Employee(String name, String role) {
    this.name = name;
    this.role = role;
  }
// getters and setters for name and role
```

11

For this CRUD API, we only need a CRUD DAO

If using JPA and Spring, it's incredibly easy to come up with a CRUD DAO using Spring JPA Repositories

All we have to do is define an interface of the right Spring type, and Spring fills it out for us.

```
package payroll;
import org.springframework.data.jpa.repository.JpaRepository;     Domain object type

interface EmployeeRepository extends JpaRepository<Employee, Long> {

}
```

Bingo! We now have a working interface to use to call the implemented CRUD methods: findAll, findById, save, deleteById, and others. There's no updateById, so PUT needs deleteById followed by save. The concrete class that implements this interface will become a @Repository Spring bean.

12

This is a Spring Boot app, so main looks like this:

```
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);
  }
}
```

- We know that by default, run will start up tomcat for us (good, we need it here), and we can set the port by using application.properties in src/main/resources.
- We can gain control after beans are set up by having a CommandLineRunner type bean...

13

Loading the DB in a CommandLineRunner

```
package payroll;
import …

@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")));
    };
  }
}
```

@Bean annotation:
Says to Spring: "please call this method and install its return value as a Spring bean"

Easy CommandLineRunner: using a lambda function for its "run" method (old name). Needs Java 8.

@Configuration: a kind of @Component, so this class generates a bean, which in turn creates the CommandLineRunner, which is found by the Spring code (after bean-creation) and executed (with argument repository set to a ref to the JPA Repository bean).

14

Spring @RestController

```
package payroll;
import …
@RestController  ←return value of handler will spec. response body, not forward-to view
class EmployeeController {
  private final EmployeeRepository repository;  ←ref from presentation to DAO directly
  EmployeeController(EmployeeRepository repository) {  // called in bean-creation
                                          // phase, with ref to repository implementation
    this.repository = repository;
  }
  // Aggregate root
  // tag::get-aggregate-root[]
  @GetMapping("/employees")          ← same as @GetMapping for web app: handle GET to this URL
  List<Employee> all() {
    return repository.findAll();     ← collection of Employees, ready to gen JSON response body
  }
  …
```

So no service layer here. Also no explicit transactions. Running on auto-commit.

15

RestController, cont.

```
@GetMapping("/employees/{id}")      ←URI template, captures id from URL
Employee one(@PathVariable Long id) {
    return repository.findById(id)
        .orElseThrow(() -> new EmployeeNotFoundException(id));
}

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

16

Last part of RestController

```
@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);
        });
```

findById returns Optional<Employee>, and .map of that applies the given function, here a lambda function that updates name and role and saves the result.
The Optional also has method orElseGet that returns the value if there, or if not (this case), calls the given function, here a lambda function, that sets the id in the employee object that came with the request, and saves it.
This is asking for a transaction!

Note: Optional is new in Java 8, along with ->.

17

Testing the REST server

For GETs, we can use a browser:

18

3
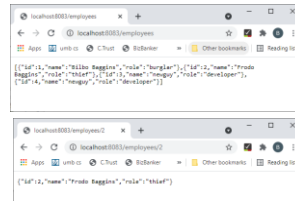
For GETs, POSTs, etc., we can use curl

From tutorial: $ curl -v localhost:8080/employees

The –v is for verbose, so you see headers, etc.

For my server, without –v, just see returned response:

F:\cs\cs636>curl localhost:8083/employees

[{"id":1,"name":"Bilbo Baggins","role":"burglar"},{"id":2,"name":"Frodo Baggins","role":"thief"}]

From tutorial: POST new employee:

$ curl -X POST localhost:8080/employees -H 'Content-type:application/json' -d '{"name": "Samwise Gamgee", "role": "gardener"}'   ←doesn't work on Windows CMD

Similarly curl –X DELETE localhost:8083/employees/3

19

POSTing, curl command on Windows CMD

Following tutorial's instructions: don't use ' in a CMD command on Windows, just use ", escaped if inside another " (and make sure it's a plain ASCII ", not "). Hard to do in Word or Powerpoint.

F:\cs\cs636>curl -X POST localhost:8083/employees -H "Content-type:application/json" -d "{\"name\": \"Samwise Gamgee\", \"role\": \"gardener\"} "

F:\cs\cs636>   ← no returned response body, so no display here

Can repeat GET /employees   to see the gardener now:

[{"id":1,"name":"Bilbo Baggins","role":"burglar"},{"id":2,"name":"Frodo Baggins","role":"thief"},{"id":3,"name":"Samwise Gamgee","role":"gardener"}]

20

Tutorial goes on to say…

The tutorial gets preachy at this point, claiming this isn't proper REST because it's so bare-bones. That's why the project is called nonrest.

The problem they are pointing to is that the data is not self-describing.

But most REST APIs today look like this and work great.

The next part of the tutorial adds in more identification information to make this minimally RESTful in their estimation. Here is the fancier output for the original two employees:

```
F:\cs\cs636>curl localhost:8080/employees
{"_embedded":{"employeeList":[{"id":1,"name":"Bilbo
Baggins","role":"burglar","_links":{"self":{"href":"http://localhost:8080/employe
es/1"},"employees":{"href":"http://localhost:8080/employees"}}},{"id":2,"name":"F
rodo
Baggins","role":"thief","_links":{"self":{"href":"http://localhost:8080/employees
/2"},"employees":{"href":"http://localhost:8080/employees"}}}]},"_links":{"self":
{"href":"http://localhost:8080/employees"}}}
```

21

What about a Java client?

We have seen we can test a REST service with curl, but that's only for developer's use.

Spring has Java REST client support in class RestTemplate

Its Javadoc says: NOTE: As of 5.0 this class is in maintenance mode, with only minor requests for changes and bugs to be accepted going forward. Please, consider using the org.springframework.web.reactive.client.WebClient which has a more modern API and supports sync, async, and streaming scenarios.

But WebClient is harder to use, so for simple classic REST (see, it's already aging) we'll look at RestTemplate, and we see that Spring still has it in use in their examples.

I took their example: Consuming a RESTful Web Service (Client accesses a server of quotes about Spring on the Internet) and converted it to a client for the employees service.

22

RestTemplate: helps generate REST requests

From that tutorial: A more useful way (than curl) to consume a REST web service is programmatically. To help you with that task, Spring provides a convenient template class called RestTemplate. RestTemplate makes interacting with most RESTful services a one-line incantation. And it can even bind that data to custom domain types.

The "domain type" we're using with the employees service is Employee. On the client side, we don't need the @Entity markup because we're not using the repository ourselves.

23

Client-side Spring Boot main class

```
@SpringBootApplication
public class ConsumingRestApplication {
    private static final Logger log =
            LoggerFactory.getLogger(ConsumingRestApplication.class);
    public static void main(String[] args) {
        SpringApplication.run(ConsumingRestApplication.class, args);
    }
    @Bean
    public RestTemplate restTemplate(RestTemplateBuilder builder) {
        return builder.build();
    }
    @Bean
    public CommandLineRunner run(RestTemplate restTemplate) throws Exception
    { …
```

See two @Beans right here in top-level @Component: they will provide a RestTemplate bean and a CommandLineRunner bean, which itself will be run after all the beans are created. Spring will find its own RestTemplateBuilder and create its bean first, then RestTemplate, then CommandLineRunner.

24

4

Client is also a server?

We know that Spring Boot defaults to running tomcat unless we explicitly turn it off in main.

So this client is itself a server, if coded this simple way.

That's not a real problem. It just ties up a port. We don't have to *use* it as a server.

My code turns off tomcat to make a pure Java client.

OK, that's a detail: where's the client code here, calling the REST service?

In the CommandLineRunner of course, just like SystemTest in pizza or music

The CommandLineRunner: first part

```
@Bean
public CommandLineRunner run(RestTemplate restTemplate) throws Exception {
    return args -> {
        String url = "http://localhost:8083/employees";
        @SuppressWarnings(value = { "unchecked" })
        Set<Employee> emps = restTemplate.getForObject(url, HashSet.class);
        System.out.println("did GET to /employees, got: "+ emps);
        …
```

Confusing names: here "run" is not the actual run-method of the CommandLineRunner, that's the lambda function. The "run" is the bean-creator method.

The call in bold does the GET using the url, receives the response, and fills up the HashSet<Employee> with the results. Pretty neat!

The CommandLineRunner: second part

```
Employee newEmp = new Employee("newguy", "developer");
HttpEntity<Employee> request = new HttpEntity<Employee>(newEmp, null);
Employee e = restTemplate.postForObject(url, request, Employee.class);
System.out.println("Did POST of new Employee, got back: " + e);
```

First the request is set up using the new Employee object, with no headers (the null).

Then call in bold does the POST using the url and wrapped up Employee object, receives the response, and interprets it as an Employee object and returns that.

Note that we may easily want more from the response than just the Employee object here: what about the Location header?
- OK, fine, don't use postForObject then, but the more powerful postForEntity…see the project.
- Turns out that the employee service doesn't fill in the Location header.

Running this client

Since this is a Spring Boot app, we can run it from eclipse by finding the main class and running it as a Java app.

Or use `mvn clean package` and then run the resulting jar:

```
java –jar target/consuming-rest-0.0.1-SNAPSHOT.jar
```

With my version, this will not hang, because I turned off tomcat in main.

Beyond CRUD

JPA Repositories can do more than just CRUD. So can REST APIs.

Example from pizza2S's PizzaOrderRepository: all implemented by Spring code-generation
```
// custom-made finders: do what their name says
List<PizzaOrder> findByRoomNumberAndDay(int room, int day);
List<PizzaOrder> findByDayBetween(int day1, int day2);
// use first on list for findFirstOrder
List<PizzaOrder> findByStatusOrderById(int status);
```
Of course this project has a service layer, so the relevant methods for web services are the service methods:
```
public List<PizzaOrderData> getOrdersByDay(int day)
public List<PizzaOrderData> getTodaysOrdersByStatus(int status)
public List<PizzaOrderData> getOrderStatus(int roomNumber)
```

Web services with parameters

```
public List<PizzaOrderData> getOrdersByDay(int day)
public List<PizzaOrderData> getTodaysOrdersByStatus(int status)
public List<PizzaOrderData> getOrderStatus(int roomNumber)
```

We could use a longer URL for orders: /orders/{room}/{id}
So the last one would involve GET /orders/7 for room 7

Or we could specify the room in a query string:
GET /orders?room=7
Similarly
GET /orders?status=preparing
We have seen how to get Spring to pick up query string parameters conveniently.
Spring will also parse the URI template parameters for us.

Summary on Spring REST support

- Easy REST controllers, much like the web app case
- Easy repositories, useful also in client-server case
- Easy development/debugging with embedded tomcat
- Fairly easy client-side support, though in transition due to Javascript invasion, no I mean need to support streaming and asynch i/o.

Things not covered here:
- Microservice support: Eureka server to manage service endpoint discovery, etc.
- CORS support, so Javascript code can access the REST services on "another" server
- Internationalization support: detect language of user using the service, handle it
- Cloud support: distributed messaging, etc.. Specializations for AWS, Google GCD, Azure, etc.

31