Distributed Systems
236351
Tutorial 3
REST

Dolev Adas
(Revised by Yehonatan Buchnik)

Technion

November 8, 2018
Overview

- REST stands for **Representational State Transfer**
- A software architectural style that defines a set of constraints to be used for creating web services
- Allow the requesting systems to access and manipulate textual representations of web resources by using a uniform and predefined set of stateless operations
- Other kinds of remote actions, such as RPC, expose their own arbitrary sets of operations.
- What are the benefits?
Overview

So, what exactly is REST?

- Is it a protocol?
  - No!

- Is it a communication system?
  - No!

- Is it a service definition?
  - No!

- REST is an architecture style - an idea
  - Defines a set of constraints to be used for creating web services
  - Actually a spectrum - a system may be defined between *not a RESTful API* to *fully RESTful*
REST Constraints

- Client-server
  - The client application and server application must be able to evolve separately without any dependency on each other.

- Stateless
  - No client context being stored on the server between requests. Each request from any client contains all the information necessary to service the request.

- Cacheability
  - Clients can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable or not to prevent clients from getting stale or inappropriate data in response to further requests.

- Layered system
  - A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way.

- Code on demand (optional)
  - Servers can temporarily extend or customize the functionality of a client by transferring executable code. For example, JavaScript.
REST Constraints

Uniform interface

The uniform interface constraint is fundamental to the design of any REST system. It consist of four more constrains:

- Resource identification in requests
  - Resources are identified in requests, for example, using URIs. The resources themselves are conceptually separate from the representations that are returned to the client.

- Resource manipulation through representations
  - When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource.

- Self-descriptive messages
  - Each message includes enough information to describe how to process the message. For example, which parser to invoke can be specified by a media type.

- Hypermedia as the engine of application state
  - A client should then be able to use server-provided links dynamically to discover all the available actions and resources it needs.
HTTP and REST

He developed the REST architectural style in parallel with HTTP 1.1 of 1996 - 1999. Those, REST design was heavily inspired by HTTP concepts. Hence, the standard way to develop a REST system is over HTTP
Uniform interface

Resource identification in requests

- URI
  - A Uniform Resource Identifier (URI) is a string of characters that unambiguously identifies a particular resource in the web

```
http://my.store.com/fruits?category=citrus&limit=20
```

- The protocol identifies the transport scheme that will be used to process and respond to the request
- The host name identifies the server address of the resource
- The path and query string can be used to identify and customize the accessed resource
Uniform interface

Resource manipulation through representation

- When building APIs, we want our models to provide four basic types of functionality. The model must be able to Create, Read, Update, and Delete resources (CRUD).
- HTTP defines method on URLs (which are type of URIs). Hence, it is quite natural to use them with REST.
  - POST: Create a new resource
  - GET: Read a representation of resource
  - PUT: Update an existing resource
  - DELETE: Delete an existing resource
Uniform interface

Self-descriptive messages

- In the header of the request, the client sends the type of content that it is able to receive from the server. This is called the **Accept** field.
  - The **Accept** field is consist of **type** and **subtype** separated by a slash
  - For example: text/html

- In cases where the server is sending a data payload to the client, the server must include a content-type in the header of the response. This **content-type** header field alerts the client to the type of data it is sending in the response body.
  - As with **Accept**, **content-type** is consist of **type** and **subtype** separated by a slash
  - For example: image/png
Uniform interface

Examples

A GET HTTP request:

```
GET http://fashionboutique.com/customers
Accept: application/json
```

An HTTP response: ¹

```
Status Code: 200 (OK)
Content-type: application/json
```

Of course the header is followed by the body of the response that contains the json.

¹Find more about HTTP status codes on wikipedia
Uniform interface

Examples

A POST HTTP request:

POST http://fashionboutique.com/customers
Accept: application/json
Body:
{
    customer": {
        name" : "Scylla Buss",
        email" : "scylla.buss@codecademy.org"
    }
}

An HTTP response:

201 (CREATED)
Content-type: application/json
Body:
{
    customer": {
        "id" : 123,
        "name" : "Scylla Buss",
        "email" : "scylla.buss@codecademy.org"
    }
}
Uniform interface
Hypermedia as the engine of application state

The purpose of **HATEOAS** is to create a simple negotiation between the client and the server such that the client does not need to know the server API.
Implementing A RESTful Service Using Spring
Spring

Spring is a lightweight framework. It can be thought of as a framework of frameworks because it provides support to various frameworks such as Struts, Hibernate, Tapestry, EJB, JSF etc. The framework, in broader sense, can be defined as a structure where we find solution of the various technical problems.

Using Spring, we will implement a RESTful service for managing Employee in a company.
Implementation

First we have to design our system models.

- How to represent the data in the system?
- Which operations should we expose to the user?
- Which type of responses to return?
Implementation

Data Model

```java
package app.models;
import com.fasterxml.jackson.annotation.JsonProperty;
import lombok.Data;
@Data
public class Employee {
    private String name;
    private String role;
    public Employee(@JsonProperty(value = "name", required = true) String name,
                     @JsonProperty(value = "role", required = true) String role) {
        this.name = name;
        this.role = role;
    }
    public Employee setName(String name) {
        this.name = name;
        return this;
    }
    public Employee setRole(String role) {
        this.role = role;
        return this;
    }
    public String getName() { return name; }
    public String getRole() { return role; }
}
```
Implementation
Exposed Operations

```java
package app.controllers;
import app.models.Employee;
import org.springframework.hateoas.Resource;
import org.springframework.web.bind.annotation.*
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.concurrent.atomic.AtomicInteger;
import static org.springframework.hateoas.core.DummyInvocationUtils.methodOn;
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.linkTo;

@RestController
public class EmployeeController {
    private HashMap<Integer, Employee> employees = new HashMap<>();
    private AtomicInteger eid = new AtomicInteger(0);
    EmployeeController() {
        employees.put(eid.getAndIncrement(), new Employee("Sheldon", "Theoretical physicist"));
        employees.put(eid.getAndIncrement(), new Employee("Howard", "Engineer"));
    }

    ...
Implementation
Exposed Operations - GET

```java
...
@GetMapping("/employees")
List<Employee> all() {
    return new ArrayList<>(employees.values());
}

@GetMapping("/employees/{id}"），
Resource<Employee> one(@PathVariable int id) {
    return employees.get(id);
}

...
Implementation
Exposed Operations - POST

...  
@PostMapping("/employees")
Employee newEmployee(@RequestBody Employee newEmployee) {
   employees.put(eid.getAndIncrement(), newEmployee);
   return newEmployee;
}
}
... 

@PutMapping("/employees/{id}\")
Employee replaceEmployee(@RequestBody Employee newEmployee, @PathVariable int id) {
    return employees.get(id).setName(newEmployee.getName()).setRole(newEmployee.getRole());
}

@DeleteMapping("/employees/{id}\")
void deleteEmployee(@PathVariable int id) {
    employees.remove(id);
}

}
package app;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class Server {
    public static void main(String argv[]) {
        SpringApplication.run(Server.class, argv);
        //
    }
}
Conclusion

For more:

- RESTful Spring tutorial
- Example files