CS5220 Advanced Topics in Web Programming

REST API with Spring Boot

Chengyu Sun
California State University, Los Angeles

JSON (JavaScript Object Notation)

- Used as a data exchange format
- Based on a *subset* of JavaScript syntax
  - Strings are double quoted
  - Property keys are strings

```
{
   "make": "Honda",
   "model": "Civic",
   "year":  2001,
   "owner": {
      "name": "Chengyu"
   }
}
```

HTTP Request Example

```
POST /products HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 ...
Accept: application/xml
Accept-Encoding: gzip,deflate
Accept-Charset: utf-8
Content-Type: application/json
Content-Length: ...

{"name": "Milk",
 "price": 3.99,
 "quantity": 10}
```

Request Line

Headers

Body
(Optional)

Request Components Commonly Used in REST API

- Request Method for representing operations
- Request URI for representing resources
- Request Body for sending data to the web API
- `Accept` header for preferred response format
- `Content-Type` header for the format of the data in request body

HTTP Response Example

HTTP/1.1 200 OK → Status Line

Content-Type: `application/json`
Content-Length: …
Date: Sun, 03 Oct 2017 18:26:57 GMT
Server: Apache-Coyote/1.1

} Headers

```
{"id": 100,
 "name": "Milk",
 "price": 3.99,
 "quantity": 10}
```

} Body (Optional)

Response Components Commonly Used in REST API

- Status Code to indicate the completion status of the operation
- Response Body for the "return value" of the API call
- `Content-Type` header for the format of the data in response body

Common Status Code

- Success codes
  - 200 OK
  - 201 Created
  - 204 No Content
- Error codes
  - 400 Bad Request
  - 401 Unauthorized
  - 403 Forbidden
  - 404 Not Found
  - 500 Internal Server Error

Use them in appropriate situations

REST API Example

- Employee Management
  - List
  - Get
  - Add
  - Update
  - Delete

A REST API Endpoint

Request

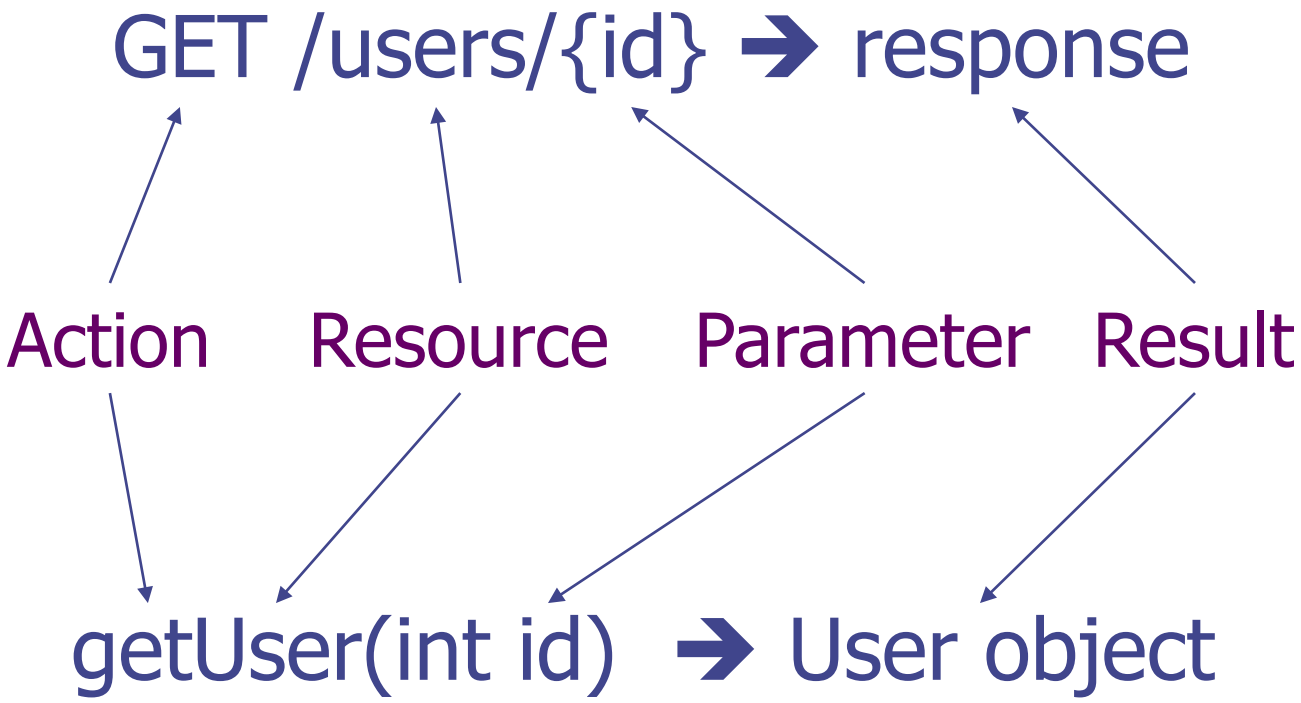
Get user with id: `/users/{id}`

XML Response or JSON Response

```
<user>
    <id>1</id>
    <firstName>John</firstName>
    <lastName>Doe</lastName>
    <email>jdoe1@localhost</email>
</user>
```

```
{
  "id": 1,
  "firstName": "John",
  "lastName": "Doe",
  "email": "jdoe1@localhost"
}
```

Design A REST API Endpoint



Object Format

- Data format should be easily "understandable" by all programming languages

- XML
  - Already widely in use as a platform independent data exchange format
  - XML parsers are readily available in many languages

- JSON
  - Much more concise than XML
  - Can be used directly in JavaScript

A REST API Endpoint Example

- Operation: get an employee
- URL
  - `/users/{id}` ✓
  - `/getUser?id={id}` ✗

REST API Design Conventions

- Use URL to represent resource
- Use request method to represent action
- Use request headers for content type and content negotiation
- Use response status code for result status

Use URL to Represent Resource ...

- And use URL segments to represent object structure
- For example:

```
class Employee {
    Integer id;
    String name;
    Employee supervisor;
    List<Employee> subordinates;
}
```

… Use URL to Represent Resources

URL	Represent
/employees	All employees
/employees/{id}	An employee with {id}
/employees/{id}/name	Name of the employee with {id}
/employees/{id}/supervisor	Supervisor of the employee with {id}
/employees/{id}/subordinates	Subordinates of the employee with {id}
/employees/{id1}/subordinates/{id2}	The subordinate with {id2} of the employee with {id1}

Use Request Method to Represent Action

- Mapping of HTTP Request Methods to CRUD operations

`POST`	⟷	Create
`GET`	⟷	Retrieve
`PUT` (or `PATCH`)	⟷	Update
`DELETE`	⟷	Delete

PUT vs PATCH ...

- Use `PUT` when the full object is provided (i.e. "put the provided object at the URL")

```
PUT /users/1 HTTP 1.1
{ "id": 1,
  "firstName": "Jane",
  "lastName": "Doe",
  "email": "jdoe@localhost"}
```

... PUT vs. PATCH

- Use `PATCH` when only part of the object is provided (i.e. "patch the object at the URL with what's provided")

```
PATCH /users/1 HTTP 1.1
{
  "firstName":"Jane"
}
```

Not All Operations Can Be Represented by Req Methods

- For example: search
  - Search can be quite complex, e.g. search by fields, logical operators in search
  - No Request Method for search
- Well ... that's why those are *conventions*
- I'd design a search endpoint with GET method and query object in request body

Spring Boot

- The preferred way to use Spring
- Greatly simplified configuration
- Build and run Spring web applications as stand-alone Java applications
- Additional production-ready features, e.g. monitoring and metrics

Create A Spring Boot Application

- https://csns.calstatela.edu/wiki/content/cysun/course_materials/cs5220/spring-boot-rest/

Run A Spring Boot Application

- In Eclipse, `Run As → Java Application`
- Use the Maven Wrapper (i.e. standalone Maven)
  - On Windows: `mvnw.cmd spring-boot:run`
  - On Linux/MacOS: `mvnw spring-boot:run`
- Package the application in a jar file and run it with `java -jar`

Example: List Employees

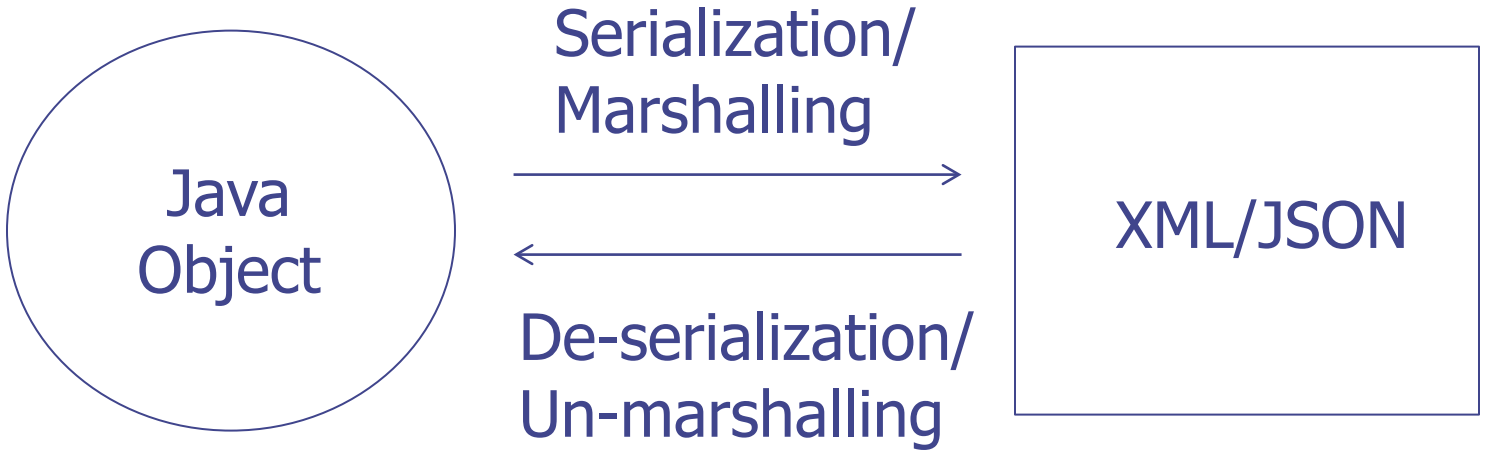
- We can reuse all the model and DAO code from Spring Web MVC Example
- Controller returns objects instead of a view
  - `@GetMapping` (remember request method is important)
  - `@RestController` = `@Controller` + `@ResponseBody`

It's Still Spring

- Beans, annotations, wiring ...
- Configuration is greatly simplified
  - Single configuration file
  - Convention over configuration

Serialization/De-serialization

- Java Objects ⇔ XML/JSON



Problems of Returning Data Model Objects

- Data models are designed for keeping information, especially for persistent storage
  - The information may not be in the best form to be used by different components of the system
  - May contain information not needed by clients, e.g. supervisor's supervisor
  - May contain information not supposed to be accessed by clients, e.g. password or hash in a User object

Data Transfer Object (DTO)

- An object that carries data from one part of a system to another
- Suitable as objects returned by web API
- There are many libraries that automatically map between data models and DTOs, e.g. MapStruct

Example: Get An Employee

- Basic implementation is easy
- But what if an employee doesn't exist?
  - Return `null` is not a good solution (in some cases `null` may be a valid value, which is different from Not Found)
  - How do we return 404?

Error Handling

- Expected errors, e.g. login failure, missing required fields, … ➔ need to inform client to correct the error
- Unexpected errors, i.e. exceptions ➔ need to log problems for analysis and fix
- *Error pages and redirects are not suitable for REST API*

Handle Errors in REST API

- Use ResponseStatusException for expected errors
- Use @ControllerAdvice to handle exceptions that you want handle
- And let Spring Boot's default exception handler to handle the rest

Spring Exception and Exception Handling

- Problems of Java exceptions
  - Too many *checked exceptions*
  - Require lots of boilerplate exception handling code
- Spring uses primarily runtime exceptions
- Separate exception handling code into *exception handlers* using AOP

Global Exception Handling Using @ControllerAdvice

```
@ControllerAdvice
public class SomeControllerAdvice {

    @ExceptionHandler(SomeException.class)
    public ResponseEntity<T>
    handleSomeException( SomeException ex ) { … }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<T>
    handleOtherExceptions( Exception ex ) { … }
}
```

T is the type of the object to be serialized into response body.

Example: Add An Employee

- Use `@RequestMapping` on the controller class
- `@RequestBody`
- Use Postman
- Set response status with @ResponseStatus and HttpStatus, e.g. `HttpStatus.CREATED`

Example: Update An Employee …

- `PUT`: replace the whole object
- Return `void` ➔ 204 No Content
- Potential problems
  - Use more bandwidth than necessary
  - Require a recent `GET`

… Example: Update An Employee

- Partial update
  - Approach 1: update individual property, e.g. `PUT /employees/1/name`
    - Will need an endpoint for each property
  - Approach 2: send only properties to be updated in a `PATCH` request, and bind them to a `Map<String,Object>`

Example: Delete An Employee

- Use `DELETE` request method
- *Hard Delete*: delete data from database
  - Difficult to recover data
  - May have unintended consequences, e.g. cascading delete, orphaned data
- *Soft Delete*: set a `deleted` flag
  - Faster, safer, easier to recover
  - Preferrable over hard delete except for certain conditions (e.g. required by law, limited storage space)