CS5220 Advanced Topics in Web Programming

REST API with Spring Boot

Chengyu Sun
California State University, Los Angeles

RESTful Web Service

- A.K.A.
  - REST Web Service
  - RESTful/REST Web API
  - RESTful/REST API
  - Web API

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}
```

$\rightarrow$ Request Line

$\}$ Headers

$\}$ Body (Optional)

HTTP Response Example

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: …
Date: Sun, 03 Oct 2017 18:26:57 GMT
Server: Apache-Coyote/1.1

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

Status Line

Headers

Body (Optional)

REST API Example

- Product Management
  - List
  - Get
  - Add
  - Update
  - Delete

Resource Representation

- 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

API Design Conventions (1)

- Operation: get a product
- URL
  - `/products/{id}` or
  - `/products/get?id={id}`

Path variables are preferred over request parameters.

"Action" is expressed in request method instead of URL.

API Design Conventions (2) …

- Map HTTP Request Methods to CRUD operations

`POST`	$\longleftrightarrow$	Create
`GET`	$\longleftrightarrow$	Retrieve
`PUT` (or `PATCH`)	$\longleftrightarrow$	Update
`DELETE`	$\longleftrightarrow$	Delete

… API Design Conventions (2)

- `PUT` vs `PATCH`
  - Use `PUT` when the full object is provided
  - Use `PATCH` when only some of the properties are provided

API Design Conventions (3)

- Choose which data format to use, a.k.a. *content negotiation*
- Solution:
  - Check the `Accept` request header
  - `/products/{id}.{format}`

Spring Boot

- Build Spring web applications as stand-alone Java applications
  - Embedded application server simplifies deployment
- Greatly simplifies configuration
  - Single configuration file
  - Default configurations
- Additional production-ready features, e.g. monitoring and metrics
- Seems to have become the preferred way to use Spring

Create A Spring Boot Application

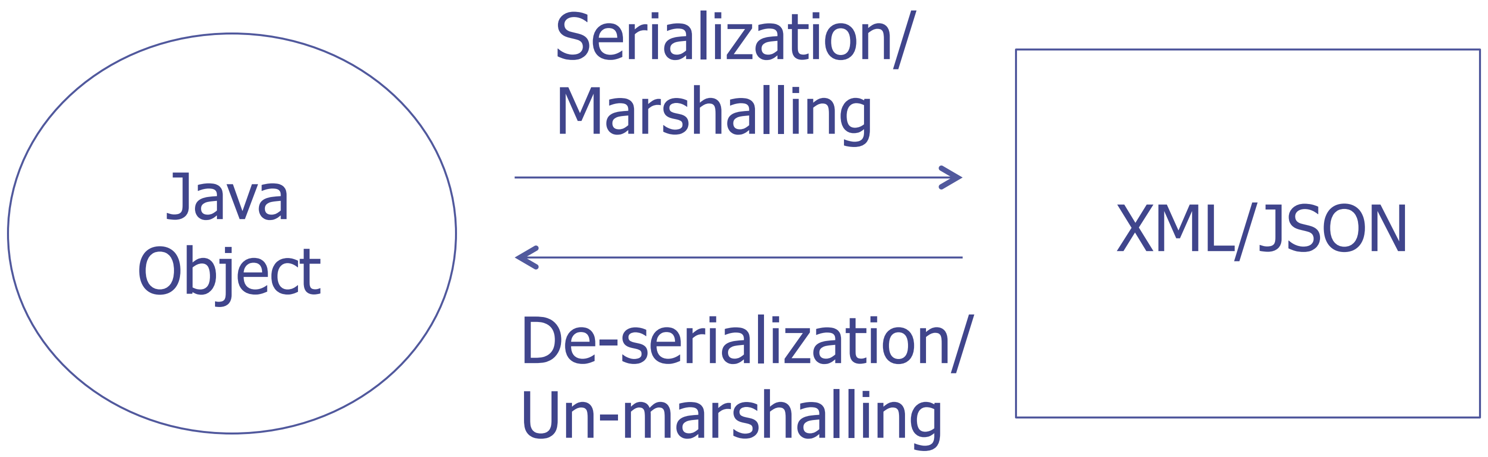
- https://csns.calstatela.edu/wiki/content/cysun/course_materials/cs5220/spring-boot-rest/

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 Products

- It's still Spring – beans, dependency injections, annotations …
- Java Objects ⇔ JSON



Example: Add A Product

- `@RequestBody`
- Use Postman
- Set response status with @ResponseStatus and HttpStatus, e.g. `HttpStatus.CREATED`

Example: Update A Product …

- `PUT`: replace the whole object
- Return `void`
- Potential problems
  - Use more bandwidth than necessary
  - Require a recent `GET`

… Example: Update A Product

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

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*

How to Send Back Error Information?

```
@PutMapping("/{id}")
public void update(@PathVariable Integer id,
        @RequestBody Product product)
{
    if( product.getName() == null ||
        product.getPrice() < 0 ||
        product.getQuantity() < 0 ) {
        ??
    }
    product = productDao.saveProduct(product); // Exception??
}
```

Problem with Java Exceptions

- Too many *checked* exceptions
  - Checked vs. Runtime exceptions
- Require lots of boilerplate exception handling code

Spring’s Solution to the Exception Problem

- Use primarily runtime exceptions
- Separate exception handling code into *exception handlers* using AOP

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

Example: Get A Product

- Throw a `ResponseStatusException` if the product is not found

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: Control Serialization/ Deserialization

```
class Product {

    Integer id;
    String name;
    int quantity;
    double price;

    Category category;
}
```

```
class Category {

    Integer id;
    String name;

    List<Product> products;
}
```

Create a "View Model"

- Domain models like `Product` may not be the right return type for a web API

```
class Product {

    Integer id;
    String name;
    int quantity;
    double price;

    Category category;
}
```

➡

```
class ProductViewModel {

    Integer id;
    String name;
    int quantity;
    double price;

    Integer categoryId;
}
```

A Product "View Model"

```
class Product {

    Integer id;
    String name;
    int quantity;
    double price;

    @JsonIgnore
    Category category;

    public Integer getCategoryId() {…}
    public void setCategoryId(Integer categoryId) {…}
}
```

Ignore this property during serialization/deserialization

A `categoryId` property used For serialization/deserialization

Depending on where you put JPA annotations (i.e. on fields or getters), you may or may not need to add a @Transient annotation on the categoryId property to tell ORM to ignore it.

Some Jackson Annotations

- Use @JsonIgnore to omit a property
- Use @JsonManagedReference and @JsonBackReference for bidirectional association
- Use @JsonIdentityInfo and @JsonIdentityReference to serialize objects to id's

Serialize Object to Id

```
class Product {

    Integer id;
    String name;
    int quantity;
    double price;

    @JsonIdentityInfo(generator=ObjectIdGenerators.PropertyGenerator.class,
        property="id")
    @JsonIdentityReference(alwaysAsId=true)
    Category category;
}
```

Serialize the first copy of an object, then use the id property for all subsequent references to the same object

Always use id, even for the first object

Example: Add Product Revisited

- How do we add a product with a category?
  - Create a "Binding Model"
  - Include a Category object in JSON
    - For existing categories, just an id would do
  - Use @JsonCreator (it doesn't seem to work with @JsonIdentityInfo though)

About Serialization/ Deserialization

- First determine JSON properties based on client code requirements
- Then work on Object ⇔ JSON
  - Object references are tricky, but
  - The view/binding model approach (a.k.a. Data Transfer Object or DTO) is always there
  - Jackson annotations can make things easier