REST Basics Activity

{ JSON }

http://

SWEN-261
Introduction to Software Engineering

Department of Software Engineering
Rochester Institute of Technology

Overview

- You have been provided with starter project for a REST API for a Heroes service.
  - *You'll be completing the "Tour of Heroes" User Interface (UI) that will connect to this REST API service in a later unit.*
- The project is built on Spring Boot, which provides the scaffolding necessary to support a REST API and an embedded Tomcat server.
  - *See spring.io more information on Spring Boot.*
- The Tomcat server makes our REST API available to respond to HTTP Requests.
  - *See tomcat.apache.org for more information on Tomcat.*

Annotations

- Annotations, a form of metadata, provide data about a program that is not part of the program itself
- Annotations have no direct effect on the operation of the code they annotate.
- Annotations have a number of uses, among them:
  - ***Information for the compiler — Annotations can be used by the compiler to detect errors or suppress warnings.***
  - ***Compile-time and deployment-time processing — Software tools can process annotation information to generate code, XML files, and so forth.***
  - ***Runtime processing — Some annotations are available to be examined at runtime.***
- Annotations were introduced in Java in version 1.5

Source: https://docs.oracle.com/javase/tutorial/java/annotations/

Spring Annotations

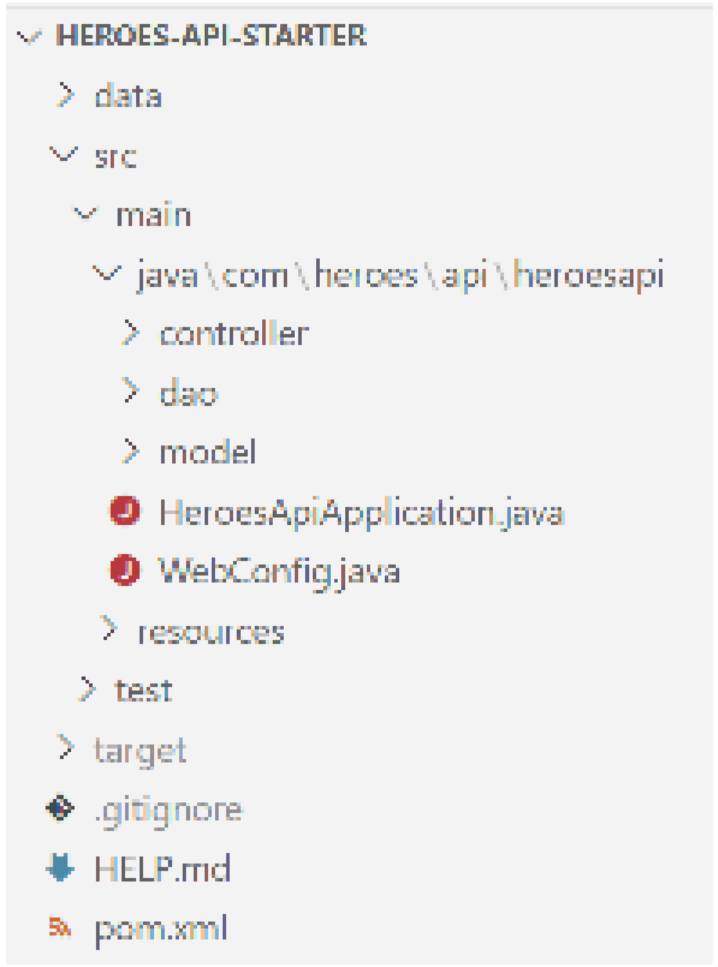
- The Spring Framework provides several annotations that we will leverage to configure Spring behavior
- By using annotations with classes, methods, fields, and parameters, Spring does a lot of work for us
  - ***Class Annotations***
    - `@Component` – instantiates a single instance of the class and injects it into other class constructors as needed
    - `@RestController` - identifies the class as a REST API method handler
    - `@RequestMapping` – maps the part of the URI that identifies the resource to the class that will handle requests for that resource
  - ***Method Annotations***
    - `@GetMapping` – identifies a controller method that will respond to HTTP GET requests
    - `@PostMapping` – identifies a controller method that will respond to HTTP POST requests
    - `@PutMapping` – identifies a controller method that will respond to HTTP PUT requests
    - `@DeleteMapping` – identifies a controller method that will respond to HTTP DELETE requests
  - ***Parameter Annotations***
    - `@Value` – associates a method parameter with a value from the application.properties file
    - `@PathVariable` – associates a method parameter with a URI variable from the Maping annotation
    - `@RequestBody` – converts (deserializes) the JSON body of a request to a Java object
- This activity will explain the usage of these annotations in the provided Heroes API code

Serialization, Deserialization, and the Jackson Library

- In the lecture, we learned about serialization and deserialization – the process of mapping text to an application objects
- Jackson is a popular Java library that provides the functionality to map JSON objects to Java objects
  - ***This library is included with the Spring framework***
- In the Heroes API, and your term project API, there are at least 2 places where serialization and deserialization will be needed
  - ***Handling of HTTP requests and responses***
    - HTTP Request JSON objects will be deserialized into REST API Java objects
    - REST API Java objects will be serialized into HTTP Response JSON objects
    - The Spring framework will do this for us automatically when we use the `@RequestBody` annotation with Controller classes and the `@JsonProperty` annotation with our Model classes
  - ***Persisting data to and from files***
    - REST API Model Java objects will be serialized into JSON objects and written as text to files
    - JSON objects as text read from files and deserialized into REST API Model Java objects
    - The Jackson library provides functionality for this when we use the `@JsonProperty` annotation with our Model classes and the `ObjectMapper` class with our Persistence classes
- Don't worry if all of this doesn't make complete sense right now, explanations and examples are forthcoming…

Project Structure

- Take a moment to look at the project structure under the `src\main\java\com\heroes\api\heroesapi` directory.
- There are 3 tiers, which are typical of a REST API Service
  - **`Controller` *– responds to the API requests and generally orchestrates CRUD operations***
  - **`Model` *– implements the core domain entities and logic***
  - ***`Persistence` – provides access to and stores the application data***
- We'll spend more time on a tiered architectures in a later lesson



```
HEROES-API-STARTER
  data
  src
    main
      java\com\heroes\api\heroesapi
        controller
        dao
        model
        HeroesApiApplication.java
        WebConfig.java
      resources
    test
  target
  .gitignore
  HELP.md
  pom.xml
```

Model Tier

- We'll start with the `Model` Tier because classes in the this tier generally do not have dependencies on classes in other Tiers
- Classes in the `Model` Tier generally represent entities from the Domain Model and implement domain logic
- For our Heroes API, we have a single simple class representing a `Hero`
  - ***There are two attributes: an id and a name***
  - ***The usual constructors, getters, setters, and toString***
- As `Hero` objects are the primary data transmitted on HTTP requests and responses, as well storage in files, we use Jackson annotations with the class field declarations and the constructor parameters to map the JSON object fields to Java object fields to support serialization and deserialization

```
@JsonProperty("id") private int id;
@JsonProperty("name") private String name;

public Hero(@JsonProperty("id") int id, @JsonProperty("name") String name) {…}
```

- Take a moment to look at the `Hero` class in
  - **`src/main/java/com/heroes/api/heroesapi/model/Hero.java`**

Persistence Tier

- This tier provides an interface to the application data
- The primary purpose is to abstract the underlying data storage mechanism from the rest of the application
- Data Access Objects (DAO) provide an interface that accepts and returns objects rather than raw data from the underlying storage mechanism
- This abstraction allows the backend to be swapped out with little or no effect on the rest of the application
  - ***For example, we'll use a standard file to store our heroes in JSON format. Because we are abstracting this implementation and using dependency injection, we could swap in a database with few changes to the rest of the application.***

Persistence Tier – HeroDAO Interface

- To support the abstraction, we have a HeroDAO interface that provides CRUD operations for Hero objects

src/main/java/com/heroes/api/heroesapi/persistence/HeroDAO.java

```java
public interface HeroDAO {
    Hero[] getHeroes() throws IOException;
    Hero[] findHeroes(String containsText) throws IOException;
    Hero getHero(int id) throws IOException;
    Hero createHero(Hero hero) throws IOException;
    Hero updateHero(Hero hero) throws IOException;
    boolean deleteHero(int id) throws IOException;
}
```

- You'll notice that the interface is written for the Hero class and does not expose any details of the underlying storage mechanism
  - ***Any class that implements this interface will need to translate between the encapsulated*** Hero ***attributes to the format used for storage***

Persistence Tier – HeroFileDAO Class Summary

- Take some time to look over the `HeroFileDAO` class in the `src/main/java/com/heroes/heroesapi/persistence` folder
- Our underlying storage mechanism will be a file storing JSON objects representing an array of `Hero` objects
- To accomplish this, we have a class, `HeroFileDAO`, that implements the `HeroDAO` interface
- Our Hero data is stored in a text file as an array of Hero JSON objects
  - ***A file with 10 heroes has been provided in*** **`data/heroes.json`**

```
[{"id": 11, "name": "Mr. Nice"},{"id": 12,"name": "Narco"}…]
```

Persistence Tier – HeroFileDAO Class Summary

- The constructor will load the file and each time a change is made, i.e. an addition, update, or delete, the data will be written to the file
- We'll use the Jackson `ObjectMapper` class to handle the conversion of JSON objects from/to the data file to Hero Java objects
- The Hero Java Objects are stored in a Map for efficient lookup by id
- You'll notice that there is quite a bit of usage of synchronized blocks
  - ***REST APIs are intended to receive multiple requests in parallel, so we need to ensure that there isn't simultaneous reading and writing of the local data structure and file***
- Let's go over some of these more interesting implementation details…

Persistence Tier – Hero Class Declaration

- The first thing to notice is the `@Component` Spring annotation before the class is declared
- Using the `@Component` annotation allows Spring to do a few things for us:
    - ***Instantiates a single instance of this class***
    - ***Injects that instance as a dependency into other classes (we'll see where this happens when we get to the `Controller` Tier)***

src/main/java/com/heroes/api/heroesapi/persistence/HeroFileDAO.java

```
@Component
public class HeroFileDAO implements HeroDAO {…}
```

Persistence Tier – Hero Class Constructor

- The `HeroFileDAO` constructor accepts two parameters:
  - **`Filename`** – ***name of the file that stores the Hero JSON objects***
  - **`ObjectMapper`** – ***facilitates conversion of***
    - JSON objects read from the data file to Java objects (deserialization)
    - Java objects to JSON objects written to the data file (serialization)

src/main/java/com/heroes/api/heroesapi/persistence/HeroFileDAO.java

```
public HeroFileDAO(@Value("${heroes.file}") String filename,ObjectMapper objectMapper)
throws IOException {…}
```

- The `filename` parameter is prefixed with the `@Value` Spring annotation
  - ***The `heroes.file` property is read from the application properties file and injected into the constructor when Spring creates the instance***

src/application.properties

```
heroes.file=data/heroes.json
```

Persistence Tier – Deserialization and Serialization

- To deserialize the array of JSON Hero objects into an array of Java Hero objects, we use the Jackson `ObjectMapper.readValue` method, by passing in
  - ***A File object constructed with the path and name of our*** `heroes.json` ***file***
  - ***An Hero Array class literal so that the*** `ObjectMapper` ***knows what class to convert the JSON objects into***

src/main/java/com/heroes/api/heroesapi/persistence/HeroFileDAO.java

```
Hero[] heroArray = objectMapper.readValue(new File(filename),Hero[].class);
```

- To serialize an array of Java Hero objects into an array of JSON Hero objects and write to a file, we use the Jackson `ObjectMapper.writeValue` method by passing in
  - ***A file object constructed with the path and name of our heroes.json file***
  - ***An array of Java Hero objects***

src/main/java/com/heroes/api/heroesapi/persistence/HeroFileDAO.java

```
objectMapper.writeValue(new File(filename),heroArray);
```

- Recall that in the Hero Model class, we annotated fields with the Jackson `@JsonProperty` annotation to provide the `ObjectMapper` a mapping from JSON Object fields to Java Object fields

Controller Tier

- The `Controller` tier is responsible for handling REST API requests
- There should be one controller class for each resource
  - ***For example, consider a REST API for University management***
    - There should be a controller class for managing students, i.e. “/student”, and a separate controller class for managing courses, i.e. “/course”
- Each controller class method should respond to one REST API request (HTTP method type and resource)
- If there is repeated application logic and code between controller class methods, service classes should be implemented so as to not violate the DRY principle (Don't Repeat Yourself)
  - ***For example, many controller methods might check that a user is logged in and need some basic information about the user → Create a `UserService` class***

Controller Tier – Hero Controller

- Take a moment to look a the `HeroController` class in `src/main/java/com/heroes/api/heroesapi/HeroController.java`
- Just before the class declaration, we use the `@RestController` annotation to let the Spring framework know that this class will respond to REST API (HTTP) requests
- We then use the @RequestMapping annotation to identify the resource requests that this class will handle
  - ***That is, any requests that start with*** **`http://server:port/heroes`** ***will be routed to our*** **`HeroController`** ***class***

src/main/java/com/heroes/api/heroesapi/controller/HeroController.java

```
@RestController
@RequestMapping("heroes")
public class HeroController {…}
```

Controller Tier – Hero Controller Constructor

- The `HeroController` constructor looks like a typical constructor – it accepts a parameter representing a HeroDAO object and assigns it to an attribute

src/main/java/com/heroes/api/heroesapi/controller/HeroController.java

```
public HeroController(HeroDAO heroDao) {
   this.heroDao = heroDao;
}
```

- Recall that before the declaration of the `HeroFileDAO` class, we used the Spring `@Component` annotation
  - *When the Spring framework starts up, it creates a single instance of* `HeroFileDAO`
  - *Also when starting, the Spring framework creates an instance of the* `HeroController` *class to respond to REST API (HTTP) requests*
  - *The Spring framework notices that the* `HeroController` *constructor requires an* `HeroDAO` *object and injects the* `HeroFileDAO` *object that it created*

Controller Tier – Hero Controller getHero Method

- One complete REST API request handler method has been provided
  - ***Stubs have been provided for the handler methods you will complete as a Spike at the end of this activity***
- The `getHero` method responds to a REST API request to get a `Hero` object given an `id`
  - ***If a Hero with the `id` does not exist, the caller receives an HTTP Status of `Not Found`***
- Let's take a closer look at this method

src/main/java/com/heroes/api/heroesapi/controller/HeroController.java

```java
@GetMapping("/{id}")
public ResponseEntity<Hero> getHero(@PathVariable int id) {
    LOG.info("GET /heroes/" + id);
    try {
        Hero hero = heroDao.getHero(id);
        if (hero != null)
            return new ResponseEntity<Hero>(hero,HttpStatus.OK);
        else
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    catch(Exception e) {
        LOG.log(Level.SEVERE,e.getLocalizedMessage());
        return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

Controller Tier – HTTP Method Mapping

- Before our method is declared, we use the Spring `@GetMapping` annotation
- This annotation tells Spring that this method should respond to HTTP GET requests for the heroes resource with a given `id`

src/main/java/com/heroes/api/heroesapi/controller/HeroController.java

```
@GetMapping("/{id}")
```

  - *The value in the `@GetMapping` annotation is appended to the URI identified in the class `@RequestMapping` annotation, so the full request URI looks like* **`http://server:port/heroes/{id}`**
    - `{id}` will be explained on the next slide

- There are also annotations for the other type of HTTP requests
  - **`@PostMapping(…)`**
  - **`@PutMapping(…)`**
  - **`@DeleteMapping(…)`**

Controller Tier – Path Variables

- The URI in the `@GetMapping` URI contains a path variable
  - ***This is a variable enclosed with "`{}`" and is passed to the request handler method as a parameter of the same name***
  - ***The method parameter is also preceded by the Spring `@PathVariable` annotation***
    - The `@PathVariable` annotation tells Spring that the id method parameter should receive the value from the URI that is in the same position as `{id}`

src/main/java/com/heroes/api/heroesapi/controller/HeroController.java

```
@GetMapping("/{id}")
public ResponseEntity<Hero> getHero(@PathVariable int id) {…}
```

Controller Tier – ResponseEntity

- The logic of the `getHero` method is straightforward
  - *Call the* `HeroDAO.getHero` *method with the* `id`
  - *If a* `Hero` *object with that* `id` *is found, return it along with an HTTP Status of* `OK`
  - *Otherwise, we let the client know we could not find a Hero object with the given* `id`

src/main/java/com/heroes/api/heroesapi/controller/HeroController.java

```
try {
    Hero hero = heroDao.getHero(id);
    if (hero != null)
        return new ResponseEntity<Hero>(hero,HttpStatus.OK);
    else
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
}
```

- The Spring `ResponseEntity` class represents the entire HTTP Response: status code, headers, and body
  - *If a Hero object with the given* `id` *is found, we are passing back a* `Hero` *object as a body and a HTTP Status of OK, but no headers are needed*
  - *If a* `Hero` *object with the given* `id` *is not found, we pass back only a HTTP Status of* `NOT FOUND`

Controller Tier – ResponseEntity

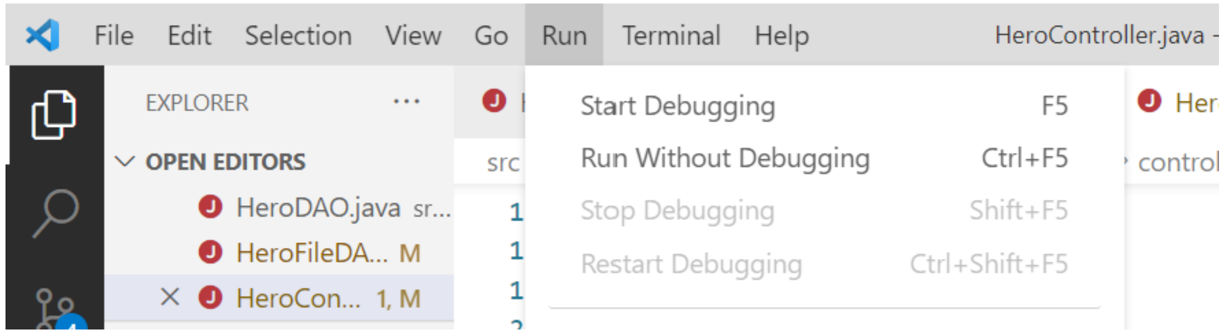
- The final bit of logic in the getHero method is to handle any exception that may be thrown
  - *In this case,* `heroDAO.getHero` *may throw an* `IOException`
    - As an `IOException` would be caused by the underlying storage mechanism, there isn't anything the client can do resolve this issue, so we just let the client know that the REST API is experiencing a general server issue
    - The `IOException` is logged by the REST API so that it can be detected and investigated

src/main/java/com/heroes/api/heroesapi/controller/HeroController.java

```java
catch(IOException e) {
    LOG.log(Level.SEVERE,e.getLocalizedMessage());
    return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
}
```

Starting the REST API Service

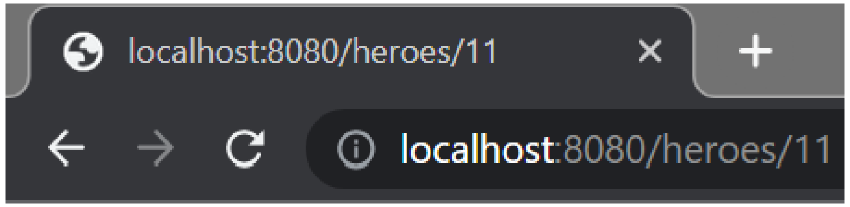
- Now that we've had a tour of the REST API service from bottom to top, we can give it a test
- You can run the REST API service in one of two ways:
  - ***From VSCode, choose "`Run Without Debugging`" from the "`Run`" menu***



  - ***From the command line, execute "`mvn compile exec:java`"***

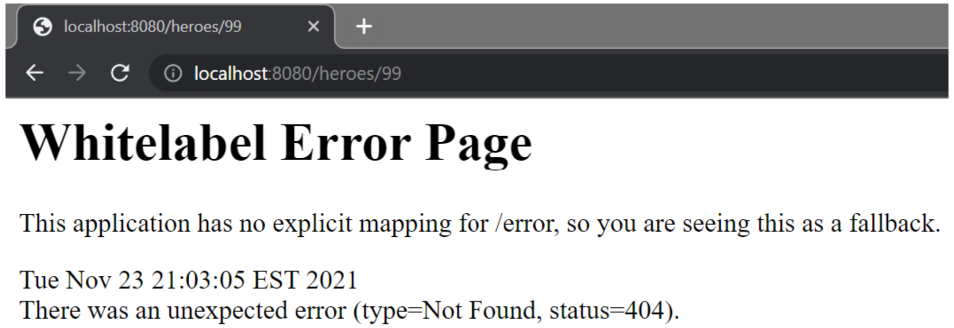
Testing the REST API Service

- Open a browser and enter http://localhost:8080/heroes/11 as the URL and you should get the following response



```
{"id":11,"name":"Mr. Nice"}
```

- Now let's try with a Hero id that does not exist
- Enter http://localhost:8080/heroes/99 as the URL and you should get the following response



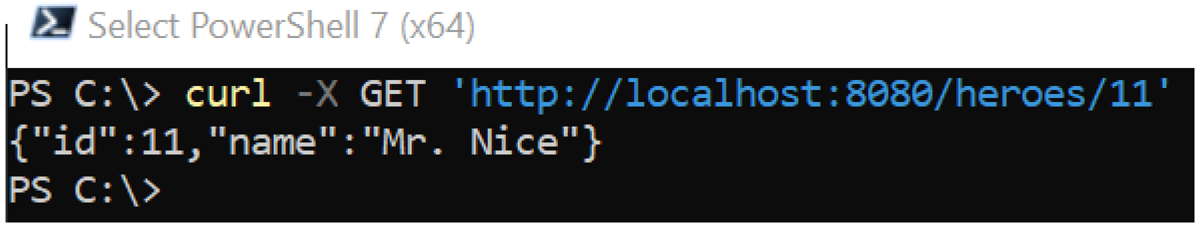
Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Tue Nov 23 21:03:05 EST 2021
There was an unexpected error (type=Not Found, status=404).

Testing With cURL

- The previous test worked because Web Browsers use the GET HTTP Method to request information
- As you implement the POST, PUT, DELETE methods, you will need to use a tool, such as cURL, that allows you to specify the HTTP Method, Headers, and Body
- cURL is a command line tool provided by most operating systems
- From a command line, enter
  `curl -X GET 'http://localhost:8080/heroes/11'`
- You should get the same hero data returned

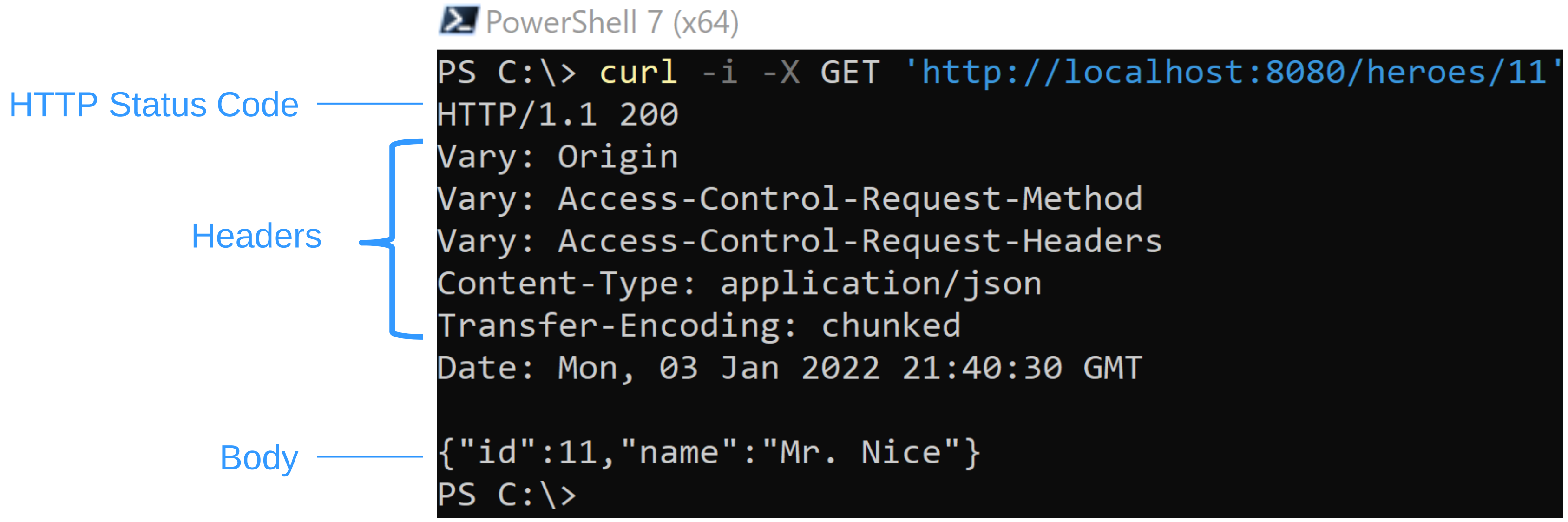


```
PS C:\> curl -X GET 'http://localhost:8080/heroes/11'
{"id":11,"name":"Mr. Nice"}
PS C:\>
```

Note that when using Windows PowerShell you will need to type the command `curl.exe` (with the `.exe` extension) for the commands shown in these examples to work.

More on cURL

- By default, cURL will only return the Body of the response
  - ***Which means if the request doesn't return a body or an error HTTP Status is returned, you won't get any output***
  - ***To also receive the HTTP Status and any headers use the `-i` option***



```
PS C:\> curl -i -X GET 'http://localhost:8080/heroes/11'
HTTP/1.1 200
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
Content-Type: application/json
Transfer-Encoding: chunked
Date: Mon, 03 Jan 2022 21:40:30 GMT

{"id":11,"name":"Mr. Nice"}
PS C:\>
```

Note that when using Windows PowerShell you will need to type the command `curl.exe` (with the `.exe` extension) for the commands shown in these examples to work.

Even More on cURL

- HTTP GET is a basic case as it only requires the URI
- Creating or Updating resources requires a bit more information
  - ***An HTTP header that specifies the format of the data***
  - ***The data used to create or update the resource***
- Here is an example of a request to create a hero
  - ***Windows***

```
Curl -i -X POST 'http://localhost:8080/heroes' -H 'Content-Type: application/json' -d '{\"name\": \"Bolt\"}'
```

HTTP Method

URI

Tells the API that the body is JSON format -H = *header*

Hero JSON Object -d = *data*

  - ***Mac/Linux***

```
curl -X POST 'http://localhost:8080/heroes' -H 'Content-Type: application/json' -d '{"name": “Bolt"}'
```

Note that on Windows, you need to escape the quotes in the JSON object

- If you try this now, you will receive an HTTP Status of 501 – Not Implemented

Sprint 0 Spike

Method stubs have been provided in the HeroController class for the other REST API methods you will need. Complete these methods in accordance with the table below.

Purpose	HTTP Method	URI	Headers	JSON Body	Response
Create a hero	POST	/heroes	Content-Type: application/json	`{ "name":“Bolt“ }`	Create and return the hero object with a status of CREATED
Update a hero	PUT	/heroes	Content-Type: application/json	`{ "id": 3, "name":“Zoom“ }`	If hero with id exists, update the name and return the updated hero object with a status of OK Otherwise, return a status of NOT FOUND
Delete a hero	DELETE	/heroes/{id}	N/A	N/A	If hero with id exists, delete the hero and return a status of OK Otherwise, return a status of NOT FOUND
Get all heroes	GET	/heroes	N/A	N/A	Return an array of all heroes (may be empty) and a status of OK
Search for heroes	GET	/heroes/?name=text	N/A	N/A	Return an array of heroes whose name contains text (may be empty) and a status of OK

Example Output Part 1

```
PS C:\> curl -X GET 'http://localhost:8080/heroes/11'
{"id":11,"name":"Mr. Nice"}

PS C:\> curl -X GET 'http://localhost:8080/heroes'
[{"id":11,"name":"Mr.
Nice"},{"id":12,"name":"Narco"},{"id":13,"name":"Bombasto"},{"id":14,"name":"Celeritas"},{"id":15,"name":"Magneta"
},{"id":16,"name":"RubberMan"},{"id":17,"name":"Dynama"},{"id":18,"name":"Dr
IQ"},{"id":19,"name":"Magma"},{"id":20,"name":"Tornado"},{"id":22,"name":"Mr Wonderful"}]

PS C:\> curl -X GET 'http://localhost:8080/heroes/?name=ag'
[{"id":15,"name":"Magneta"},{"id":19,"name":"Magma"}]

PS C:\> curl -X POST -H 'Content-Type:application/json' 'http://localhost:8080/heroes' -d '{\"name\": \"Mr
Wonderful\"}'
{"id":23,"name":"Mr Wonderful"}
PS C:\> curl -X GET 'http://localhost:8080/heroes'
[{"id":11,"name":"Mr.
Nice"},{"id":12,"name":"Narco"},{"id":13,"name":"Bombasto"},{"id":14,"name":"Celeritas"},{"id":15,"name":"Magneta"
},{"id":16,"name":"RubberMan"},{"id":17,"name":"Dynama"},{"id":18,"name":"Dr
IQ"},{"id":19,"name":"Magma"},{"id":20,"name":"Tornado"},{"id":22,"name":"Mr Wonderful"},{"id":23,"name":"Mr
Wonderful"}]
```

Example Output Part 2

```
PS C:\> curl –i -X PUT -H 'Content-Type:application/json' 'http://localhost:8080/heroes' -d '{\"id\": 21,
\"name\": \"Mr Awful\"}'
HTTP/1.1 404
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
Content-Length: 0
Date: Mon, 03 Jan 2022 21:53:14 GMT
PS C:\> curl -X GET 'http://localhost:8080/heroes’
HTTP/1.1 200
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
Content-Type: application/json
Transfer-Encoding: chunked
Date: Mon, 03 Jan 2022 21:58:50 GMT


[{"id":11,"name":"Mr.
Nice"},{"id":12,"name":"Narco"},{"id":13,"name":"Bombasto"},{"id":14,"name":"Celeritas"},{"id":15,"name":"Magneta"
},{"id":16,"name":"RubberMan"},{"id":17,"name":"Dynama"},{"id":18,"name":"Dr
IQ"},{"id":19,"name":"Magma"},{"id":20,"name":"Tornado"},{"id":22,"name":"Mr Wonderful"},{"id":23,"name":"Mr
Wonderful"}]
```

Example Output Part 3

```
PS C:\> curl –i -X DELETE 'http://localhost:8080/heroes/21'
HTTP/1.1 404
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
Content-Length: 0
Date: Mon, 03 Jan 2022 21:55:40 GMT
PS C:\> curl -X GET 'http://localhost:8080/heroes'
[{"id":11,"name":"Mr.
Nice"},{"id":12,"name":"Narco"},{"id":13,"name":"Bombasto"},{"id":14,"name":"Celeritas"},{"id":15,"name":"Magneta"
},{"id":16,"name":"RubberMan"},{"id":17,"name":"Dynama"},{"id":18,"name":"Dr
IQ"},{"id":19,"name":"Magma"},{"id":20,"name":"Tornado"},{"id":22,"name":"Mr Wonderful"},{"id":23,"name":"Mr
Wonderful"}]

PS C:\> curl –i -X GET 'http://localhost:8080/heroes/99'
HTTP/1.1 404
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
Content-Length: 0
Date: Mon, 03 Jan 2022 21:56:50 GMT
```

Example Output Part 4

```
PS C:\> curl –i -X DELETE 'http://localhost:8080/heroes/99'
HTTP/1.1 404
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
Content-Length: 0
Date: Mon, 03 Jan 2022 21:57:57 GMT
```

Additional Notes and Tips

- The Heroes API performs CRUD operations for Hero objects and as such does not contain much business logic
- Your term project, however, will require business logic, use of annotations and data structures not covered by this activity, and additional coding complexity, so here are some notes and tips
  - ***You can have more than one `@PathVariable` in a URI***

```
@GetMapping("/student/{studentId}/{courseId}/grade")
public ResponseEntity<Float> getCourseGrade(@PathVariable int studentId, @PathVariable int
courseId) {…}
```

  - ***`@PathVariable` can be used in conjunction with `@RequestBody`***

```
@PostMapping("/student/{studentId}/course/")
public ResponseEntity<Student> addCourse(@PathVariable String studentId, @RequestBody Course
course) {…}
```

Additional Notes and Tips II

- ***Sometimes returning an HTTP Status Code is not enough***
  - Consider the previous Student and Course example – what if a Student with `studentId` is not found? What if a Course with the `courseId` is not found? Returning a HTTP Status of `404` will not allow the client to distinguish which object could not found
  - You can use custom HTTP headers in the response to inform the client of which `id` could not be found

```
private static String STUDENT_NOT_FOUND = "1";
private static String COURSE_NOT_FOUND = "2";
HttpHeaders headers = new HttpHeaders();
headers.add(“api-status-code",STUDENT_NOT_FOUND);
return new ResponseEntity<>(headers,HttpStatus.NOT_FOUND);
```

  - The name of the header, e.g. `api-status-code`, and values, e.g. `"1"` or `"2"`, are up to the developers[1], but must be agreed upon between the REST API and the client, e.g. UI
  - Make sure you import import `org.springframework.http.HttpHeaders` and not `java.net.http.HTTPHeaders`

[1]As long as the name of the header does not conflict with a standard header, e.g. `Content-Type`

Additional Notes and Tips III

- When you have multiple controllers, you may find that more than one controller needs access to the same data
- Recall that if you prefix a class with the `@Component` annotation, Spring will create a single instance of the class upon application startup
- Also recall that when a controller class declares a parameter of the class that is prefixed with the `@Component` annotation, Spring will inject that single instance that it previously created