COMP41720 Distributed Systems Practical

Lab 5: REST-based Distribution

Work individually. Submit your code on Brightspace by the deadline given. Please submit a single ZIP file (name is <student-num>.zip) containing your source code and any text files required for the solution.

You should use the basic version of LifeCo as a starting point for this lab. As a reminder, you can download the LifeCo source code from https://gitlab.com/ucd-cs-rem/distributed-systems/lifeco-basic.

The broad objective of this practical is to adapt the LifeCo system to use Representational State Transfer (REST) for interaction between each of the three quotation services and the broker and between the broker and the client. We will use the Spring Boot framework to implement the system. In the final version, each of these components should be deployable as a separate Docker image, and you should provide a docker-compose file that can be used to deploy the images.

As with the previous labs, I have broken the problem up into a set of tasks. My advice is to create a separate project for your solution and for you to copy code from the original project as needed.

Each task should be completed in turn, and **your submission for each task must be stored in a separate folder** called "tasks," where X is the task number. For example, your answer for Task 1 should be stored in a folder called "task1", while your answer for Task 2 should be stored in a folder called "task2". When moving on to the next task, duplicate and rename the previous task folder to represent the following task (i.e., after completing Task 1, copy the folder "task1" and rename the copy "task2". Modify the code in "task2" as required to complete the task.

Task 1: Setting up the Project **Grade: E**

As in the previous lab the first task is to break the original codebase up into a set of projects: one of each of the web services we are going to create; a project for the client and a shared core project that contains the code that is common across the web services. Again, we will also set up the core project.

a) You should create a folder "lifeco-rest" and inside that create a folder called "task1" which will be the root of the project file structure. As before, this project should contain the following sub-folders:
   - **core**: contains the common code (distributed object interfaces, abstract base classes & any data classes)
   - **auldfellas**: The Auldfella's Quotation Service
   - **dodgygeezers**: The Dodgy Geezers Quotation Service
   - **girlsallowed**: The Girls Allowed Quotation Service
   - **broker**: The broker service
   - **client**: The client service

   You should create a multi-module pom.xml file in the root folder of the project that should reference each subfolder. A standard "jar" packaging pom.xml file should be added to each subfolder. The artifactId for each sub-project should be the same as the name of the folder containing the subproject (e.g. the "core" projects artifactId should be "core" and the "girlsallowed" project artifactId should be "girlsallowed".) The artifactId for the main project should be based on the task (e.g. for this task it should be "task1" while for the next task it should be "task2"). The groupId should be "lifeco-rest".

b) Next, copy the "service.core" package from the basic project into "src/main/java" folder of the core folder. Remember that you should replicate the package structure. Perform the following modifications:

   - Delete the `Constants` class.
   - Remove all references to the `ServiceRegistry` and `Service` classes.
   - Delete the `QuotationService` and `BrokerService` interfaces.

- Make sure that the `Quotation` and `ClientInfo` classes implement the default constructor.
- Compile and install the "core" project.

Task 2: Designing the Quotation Services & Implementing Auldfellas **Grade: D**

This task involves creating a REST API for the quotation services. Remember, REST is data-oriented, about state transfer rather than remote procedure calls. It is about exposing representations of resources as endpoints and interacting with those resources using HTTP methods.

From a data perspective, a quotation service can be modeled as a collection of quotations. In REST terminology, these quotations are resources created by the quotation service. For completeness, the quotation service also needs to model the idea of a collection of resources. A URL pattern must be defined to allow each resource to be referenced. For this lab, the following endpoints will be determined:

- `/quotations`: the collection of quotation resources
- `/quotations/{id}`: a specific quotation resource identified by the given id {id}.

Intuitively, a quotation resource is created by interacting with the `/quotations` endpoint and a `/quotations/{id}` endpoint will be created for each quotation. Following best practice (and the HTTP standards[1]), we model the creation of a resource by submitting a POST request to the `/quotations` endpoint.

Each HTTP request requires a corresponding HTTP response with an appropriate response code. In the case of a POST request, the HTTP/1.1 specification states that: *"the origin server should respond with a 201 (Created) response with a Location header that provides an identifier (URL) of the primary resource created."* The specification also states that the server *"should also return a representation that describes the status of the request."*

There is some debate around the latter part of the specification regarding what "a representation that describes the status of the request" means. Generally, this can be interpreted as some description of the state of the process of creating the resource rather than the state of the resource itself. So, a good response could be, "A quotation with reference number ##### was created successfully." That resource's location (URL) can then be retrieved from the `Location` header. In cases where the body of a 201 (Created) response includes the state of the resource, the response should also include a `Content-Location` header. ***The identifier associated with this second header may well be the same as the identifier of the `Location` header however semantically, they mean different things: you cannot assume that a Location header refers to any representation of the representation returned in the body of the HTTP response.***

To be clear, HTTP POST requests can also return 200 (OK) responses. Such responses generally include a representation of the state of the resource created. This can be useful in situations where the client benefits from caching the resource representations. That said, according to the standards, the former 201 response is the preferred response to a POST request.

Retrieving representations of the state of a resource is achieved via an HTTP GET request. The expected response of a GET request is a 200 (OK) response where the body represents the state of the request. To be clear, the server should return the representation in a format requested by the client. Clients define the format they wish to receive through the definition of the `Accept` header in the request. This header contains an ordered comma-delimited list of acceptable media types (e.g., `text/plain`, `text/html`, `application/json`, …). The ordering represents preference, so in the example in the previous sentence, `text/plain` format is preferred to `text/html` or `application/json`. Matching the requested format against the available formats is known as content negotiation. If a response format cannot be agreed upon, then a 406 (Not Acceptable) response must be returned.

So, to summarise the design of a quotation service, it is expected that an HTTP POST request will be submitted to the `/quotations` endpoint to trigger the creation of a quotation. This request (if valid) will create a 201 (Created)

[1] https://www.rfc-editor.org/rfc/rfc7231

response where the `Location` header contains the URL of the created quotation resource. This URL will adhere to the `/quotations/{id}` format. The response will also include a representation of the quotation. In the implementation, both the `/quotations` and the `/quotations/{id}` endpoints are expected to accept HTTP GET requests. The former should return a representation containing a list of the URLs of the quotations created, and the latter should return a representation of a specific quotation.

NOTE: Choosing how to represent things like lists is an active research area. Various approaches have been proposed, but recent work is tending towards the use of semantic web technologies for specifying representations. We will talk about this later in the module, but for now, we will focus on ad hoc representations.

In this part, we focus on the Auldfellas service.

a) First, remember to duplicate the "task1" folder, creating a new folder called "task2". All the changes discussed here should be made to "task2".

b) Copy the Auldfellas service from the "lifeco-basic" project. Remember to maintain the "service.auldfellas" package structure.

c) To support the use of Spring Boot, add the following entries to the Auldfellas pom.xml file:

```
<parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>2.2.4.RELEASE</version>
   <relativePath/>
</parent>

<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<build>
   <plugins>
      <plugin>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
   </plugins>
</build>
```

Note: the dependency should be added to the existing `<dependencies>` block. Also, you can replace the existing exec-maven-plugin plugin as we will not be using that in this project.

d) Add an `application.properties` file to the /src/main/resources folder. This file should contain the following:

```
server.port=8080
```

To change the port that a Spring Boot instance runs on, simply change this property.

e) Create a class called Application in the "service" package with the following code:

```
package service;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

f) Next, we can implement the controller part of the Spring Boot application. This is another Java class that maps endpoints (and HTTP methods) to Java methods that implement the associated behaviour. Let's start with the code for modelling a collection of quotations and implementing support for the GET request. To do this, start by creating a package called "controllers" and then create a `QuotationController` class. The code for this class is below:

```
package service.controllers;

import java.util.Collection;
import java.util.Map;
import java.util.TreeMap;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import service.core.Quotation;

@RestController
public class QuotationController {
    private Map<String, Quotation> quotations = new TreeMap<>();
    private AFQService service = new AFQService();

    @GetMapping(value="/quotations", produces="application/json")
    public ResponseEntity<Collection<Quotation>> getQuotations() {
        return ResponseEntity
                    .status(HttpStatus.OK)
                    .body(quotations.values());
    }
}
```

So what does the above code do? First, the `@RestController` annotation declares that this class implements a controller. This means you can annotate methods as handlers for specific HTTP requests on particular endpoints.

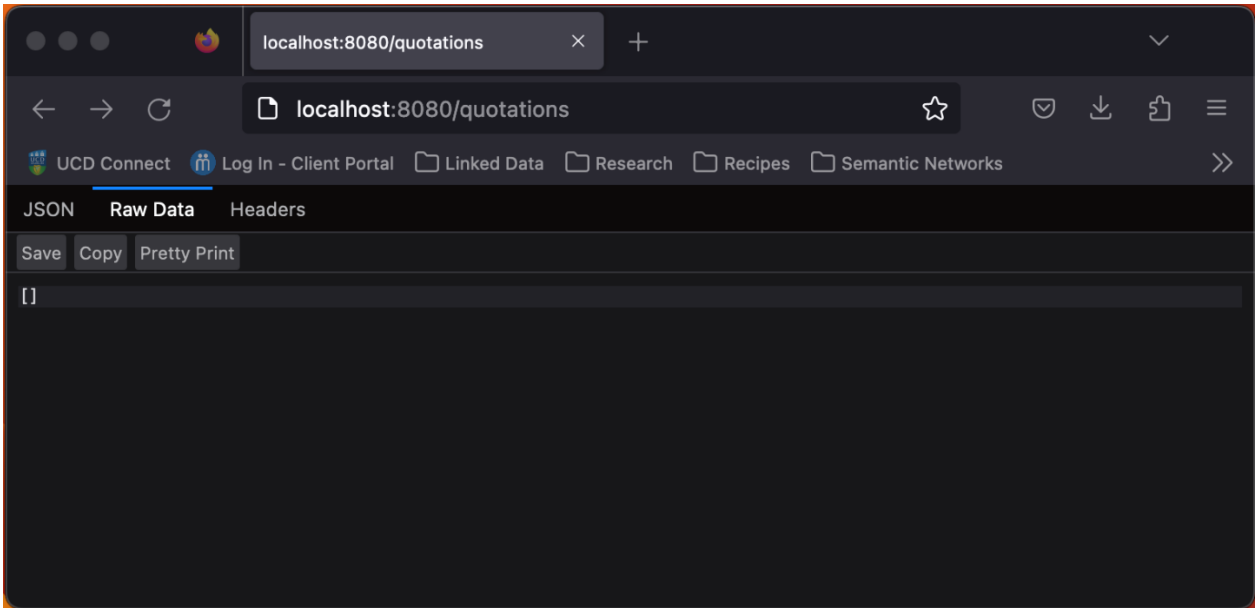
The `getQuotations()` method is mapped to an HTTP GET request to the `/quotations` URL for "application/json" content by the `@GetMapping` annotation. This method returns an object of type `ResponseEntity`. This is a Spring Boot class that can be used to send back a variety of HTTP responses. In the example code above, we return a response with a 200 (OK) status and pass a collection of `Quotation` objects as the body. Spring Boot transforms this into JSON behind the scenes using Jackson, a flexible serialization API that can transform objects into JSON, XML, and many other data formats.

The quotations map has been introduced here to store Quotations when they are created. We use a map because it will allow quick access to specific quotations when we implement the `/quotations/{id}` endpoint. Here, the {id} will be the key to retrieving the quotation.

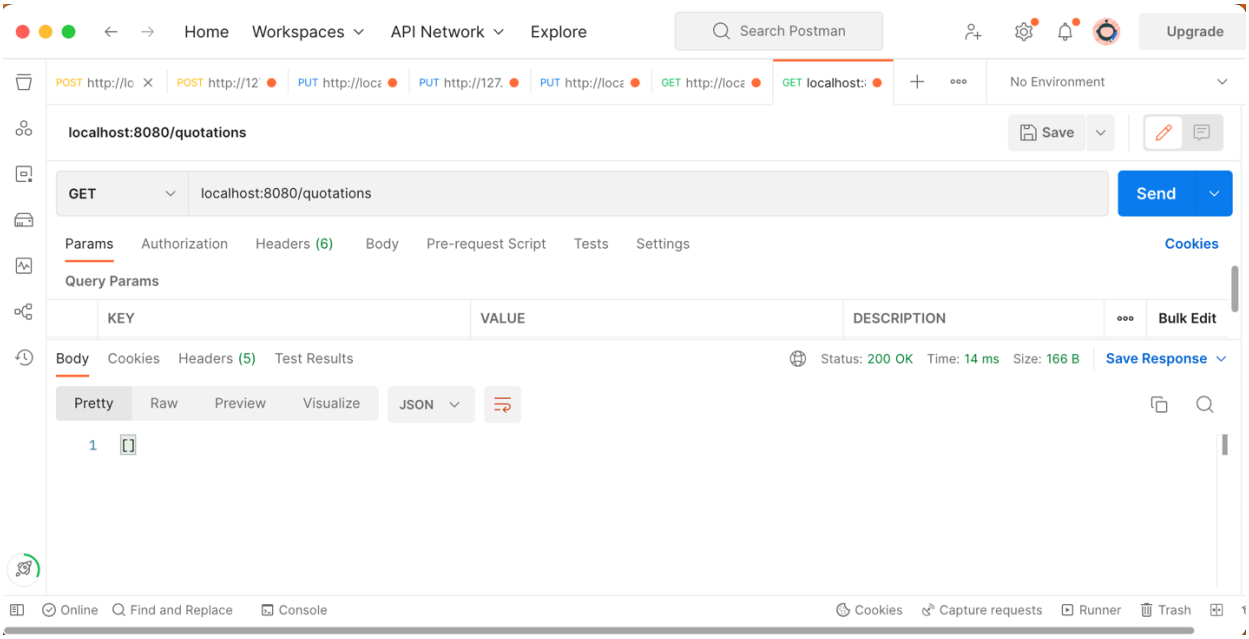
You can test this code by running the project:

```
$ mvn compile spring-boot:run -pl auldfellas
```

You can then use a browser to access the resource:



NOTE: Some browsers do not show JSON, so you may need to use a different browser. As an alternative, you can also use Postman:



g) Next we can implement the POST request that will allow us to create new `Quotation` objects. As in previous labs, we must pass a `ClientInfo` object to the service and then use this to create a `Quotation` object. Our solution will be to return a representation of the quotation object in the response. The broker service will aggregate the quotations into a single application.

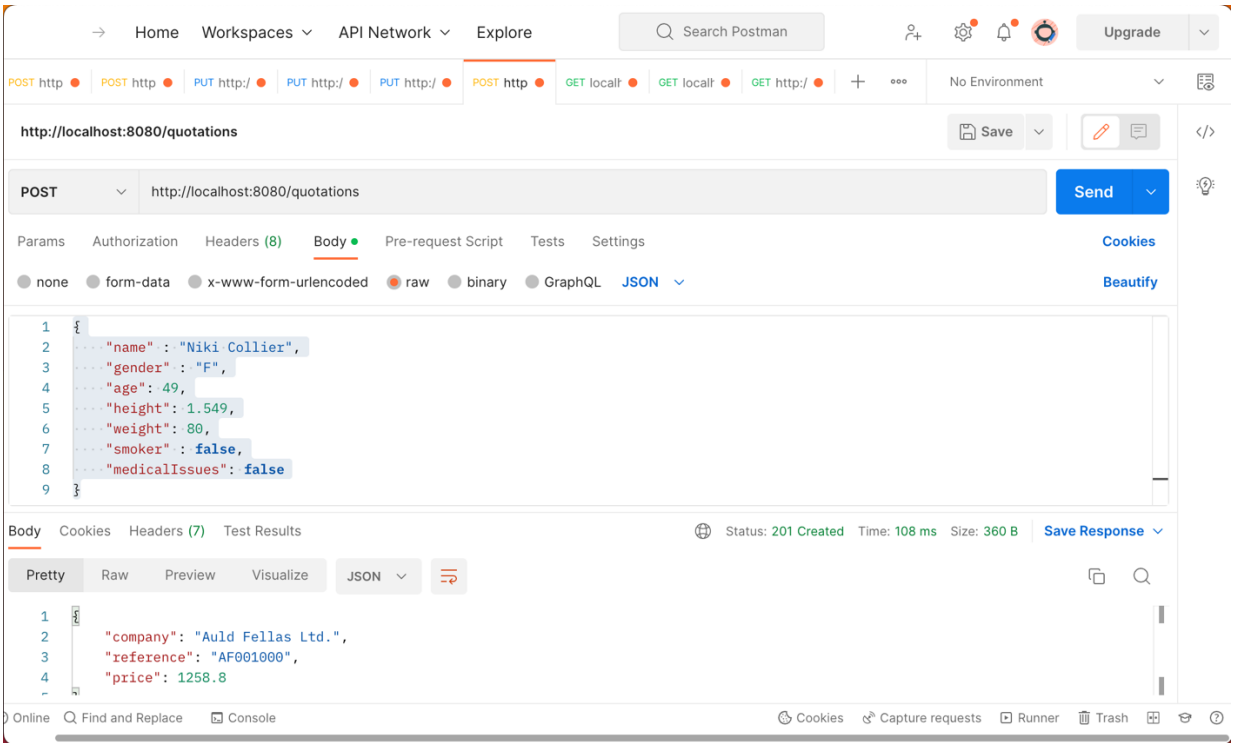
```
@PostMapping(value="/quotations", consumes="application/json")
public ResponseEntity<Quotation> createQuotation(
                                        @RequestBody ClientInfo info) {
    Quotation quotation = service.generateQuotation(info);
    quotations.put(quotation.reference, quotation);

    String url = "http://"+getHost()+"/quotations/"
                        + quotation.reference;
    return ResponseEntity
        .status(HttpStatus.CREATED)
        .header("Location", url)
        .header("Content-Location", url)
        .body(quotation);
}
```

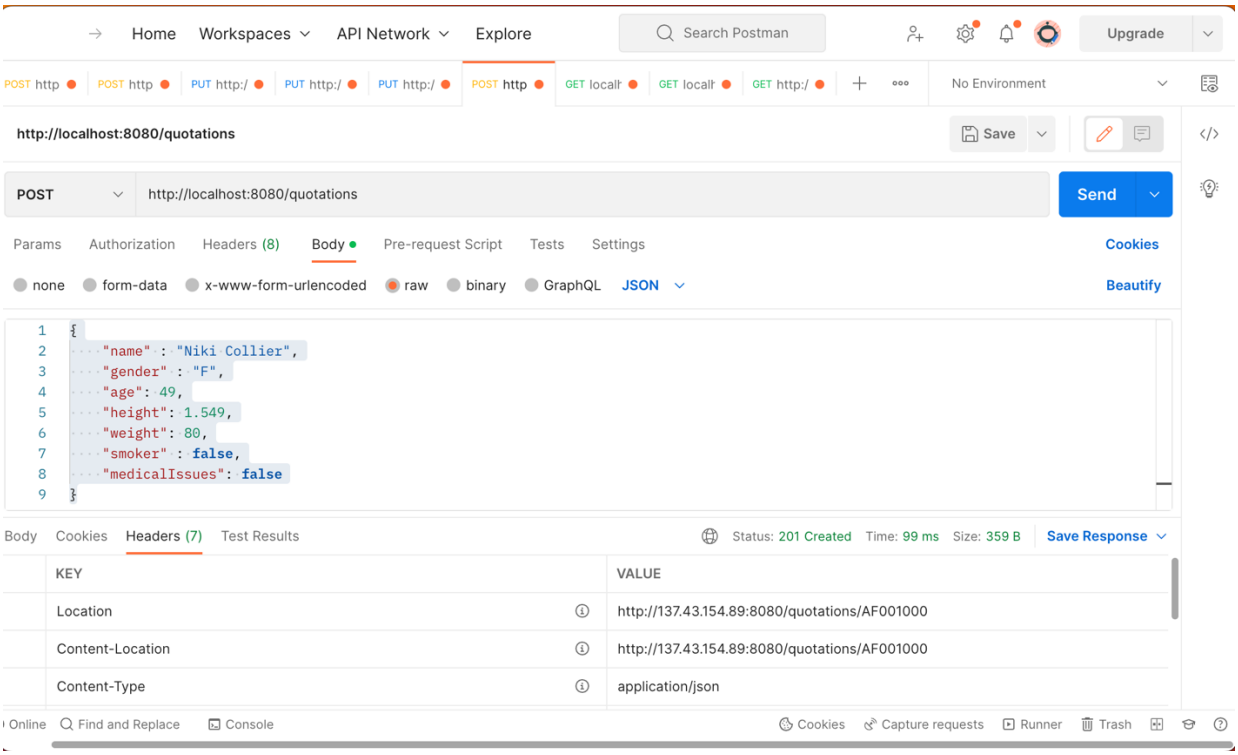
Again, Spring Boot does some heavy lifting for us. Behind the scenes, it can convert JSON objects into `ClientInfo` objects. Notice using the `@RequestBody` to map the body of the incoming HTTP Request to a `ClientInfo` object. To make this work, you need to use the following JSON structure. For example, the equivalent of the "Niki Collier" client is:

```
{
    "name" : "Niki Collier",
    "gender" : "F",
    "age": 49,
    "height": 1.549,
    "weight": 80,
    "smoker" : false,
    "medicalIssues": false
}
```

You can submit this as a post request using Postman:

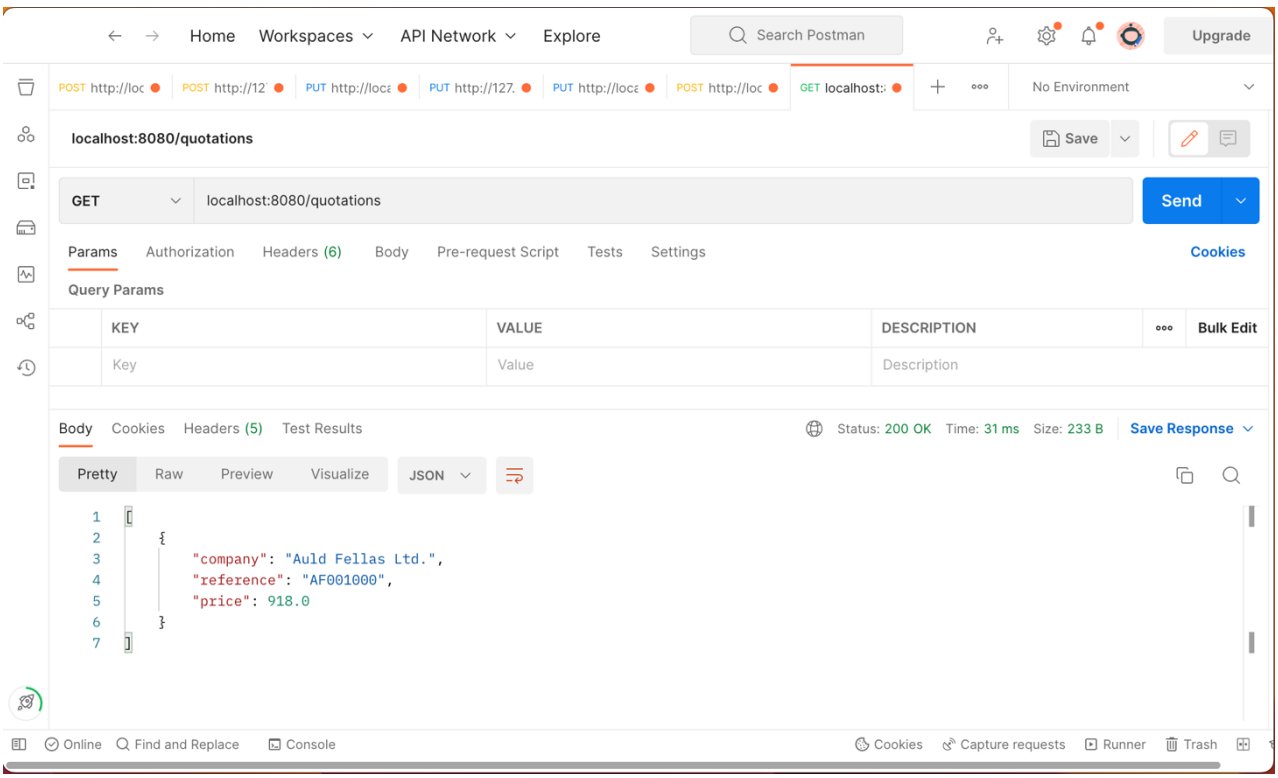


If you look at the Headers returned by the response, then you should see the Location and Content-Location headers:



Later, we will explore how to programmatically make HTTP requests and how to extract the Location header from the response, allowing the client to keep a record of the URL of any newly created resources.

Let's now explore what the `/quotations` endpoint returns in response to a GET request. If you submit the GET request, you should now see:



This is not really what we wanted for out specification – we wanted was a list of the URLs of the quotations created, not a list of the quotations. We can fix this by modifying the `getQuotations()` method:

```
@GetMapping(value="/quotations", produces="application/json")
public ResponseEntity<ArrayList<String>> getQuotations() {
    ArrayList<String> list = new ArrayList<>();
    for (Quotation quotation : quotations.values()) {
        list.add("http:" + getHost()
                + "/quotations/"+quotation.reference);
    }
    return ResponseEntity.status(HttpStatus.OK).body(list);
}
```

This method requires a supporting method called `getHost()`:

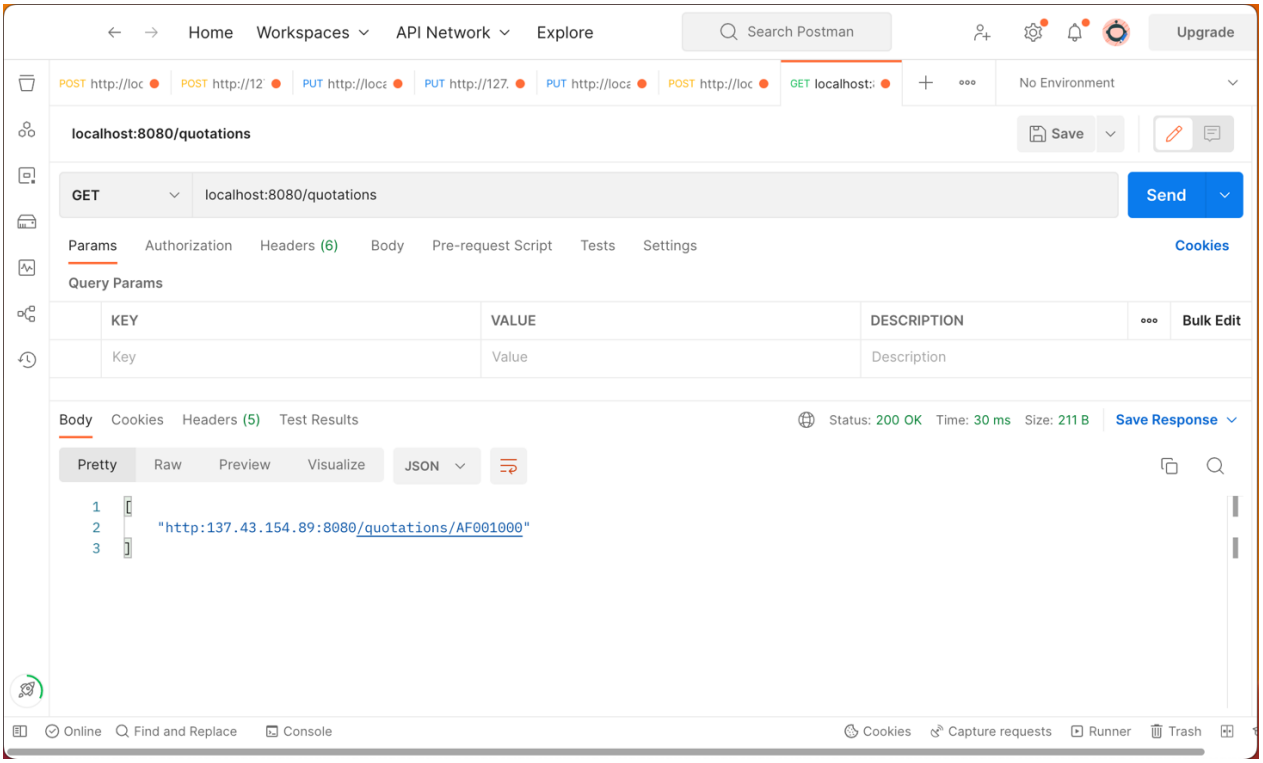
```
private String getHost() {
    try {
        return InetAddress.getLocalHost().getHostAddress() + ":" + port;
    } catch (UnknownHostException e) {
        return "localhost:" + port;
    }
}
```

This in turn needs the declaration of the `port` field:

```
@Value("${server.port}")
private int port;
```

The `@Value` annotation initialises the field by reading the specified property from the `application.properties` file.

The impact of this code can be seen in the Postman screenshot below:



What makes this a better solution is that we now have the ability to discover quotations by querying the `/quotations` endpoint. This gives us a list of URLs, each of which is a quotation. While this example is

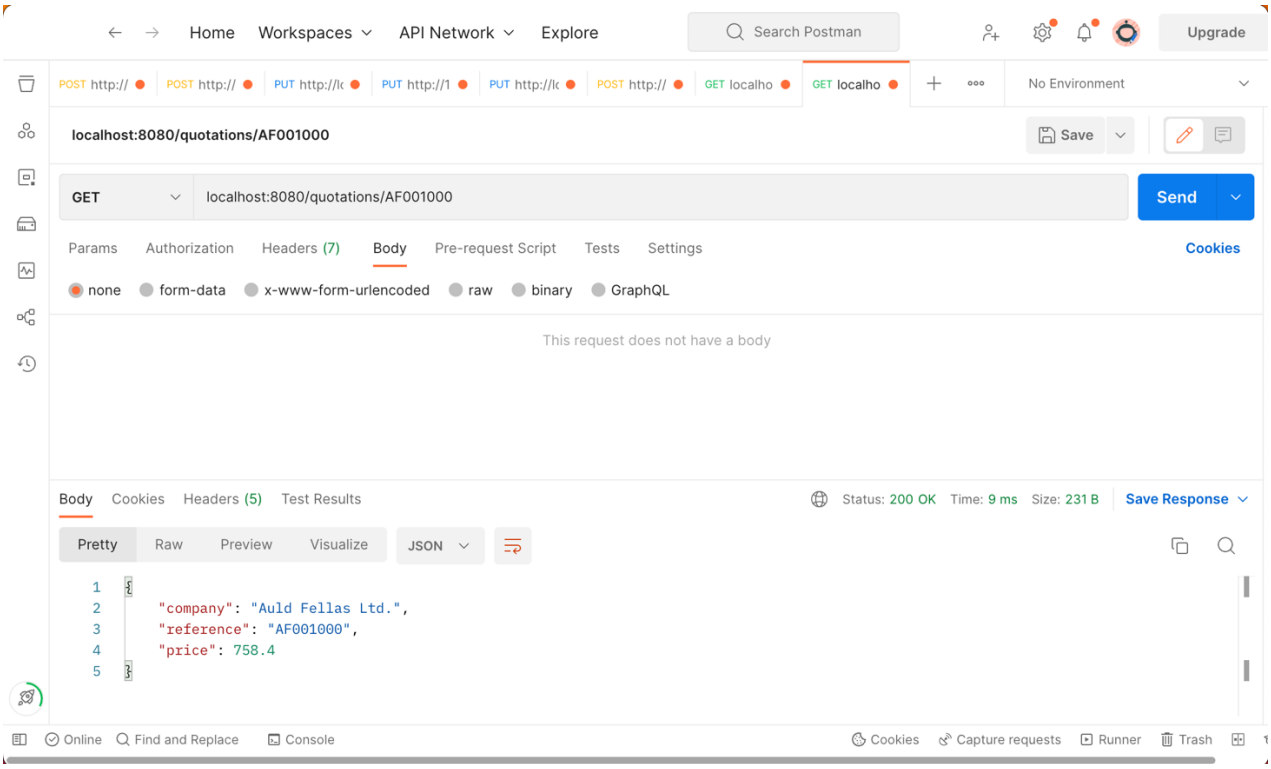
rather simplistic, it is the principle on which Hypermedia As The Engine Of Application State (HATEOAS) is built.

h) The final step is implementing support to GET a representation of an individual quotation. This can be realised by the following code:

```
@GetMapping(value="/quotations/{id}", produces={"application/json"})
public ResponseEntity<Quotation> getQuotation(@PathVariable String id) {
    Quotation quotation = quotations.get(id);
    if (quotation == null) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
    }

    return ResponseEntity.status(HttpStatus.OK).body(quotation);
}
```

To test this, start the server, use a POST request to create a quotation; perform a GET request on the /quotations endpoint; click on the created quotation (this creates a new GET request in Postman) and then click "Send" to see the response:



Task 3: Building the GirlsAllowed and DodgyGeezers Services **Grade: C**

Repeat the solution presented in task 2 but for the GirlsAllowed and DodgyGeezers services. GirlsAllowed should be made available on port 8081 and DodgyGeezers should be made available on 8082.

Task 4: Implementing the Broker and Client **Grade: B**

Now that we have working quotation services, the next task is to create the broker and the client. By far, the hardest part of this task is creating the broker, which you should do first. The client code is relatively simple and involves creating a REST client that can interact with the broker service. We start with a brief overview of the broker service.

The broker service is also a REST service. This means it is also responsible for creating resources exposed on the web. The specific resources created by this service can be viewed as `Application` resources. In this lab, an application is considered as a combination of some `ClientInfo` together with a list of URLs to quotations based on that client

info. Similarly to the Quotation Services, existing applications should be viewable by performing a GET request on a `/applications` endpoint. Performing a POST request should create an application. Individual applications should be identified by a URL matching the pattern: `/applications/{id}` where {id} is some unique number generated internally by the broker. Again, similarly to the quotations services , performing a GET request on a specific application URL should return a representation of that application.

a) Start by adding the following `Application` class to the core project:

```
package service.core;

import java.util.ArrayList;

public class Application {
    private static int COUNTER = 1000;
    public int id;
    public ClientInfo info;
    public ArrayList<Quotation> quotations;

    public Application(ClientInfo info) {
        this.id = COUNTER++;
        this.info = info;
        this.quotations = new ArrayList<>();
    }

    public Application() {}
}
```

The COUNTER static field assigns each application a unique number, which is then used to create the application-specific URL.

b) Create the `/applications` endpoint. You should include methods for both GET and POST. The POST method is the most interesting as it should contact each quotation service. For this, we can state by borrowing the idea of the list of URLs from the Web Services lab. Later, we will replace this with a registration service similar to the RMI lab's approach.

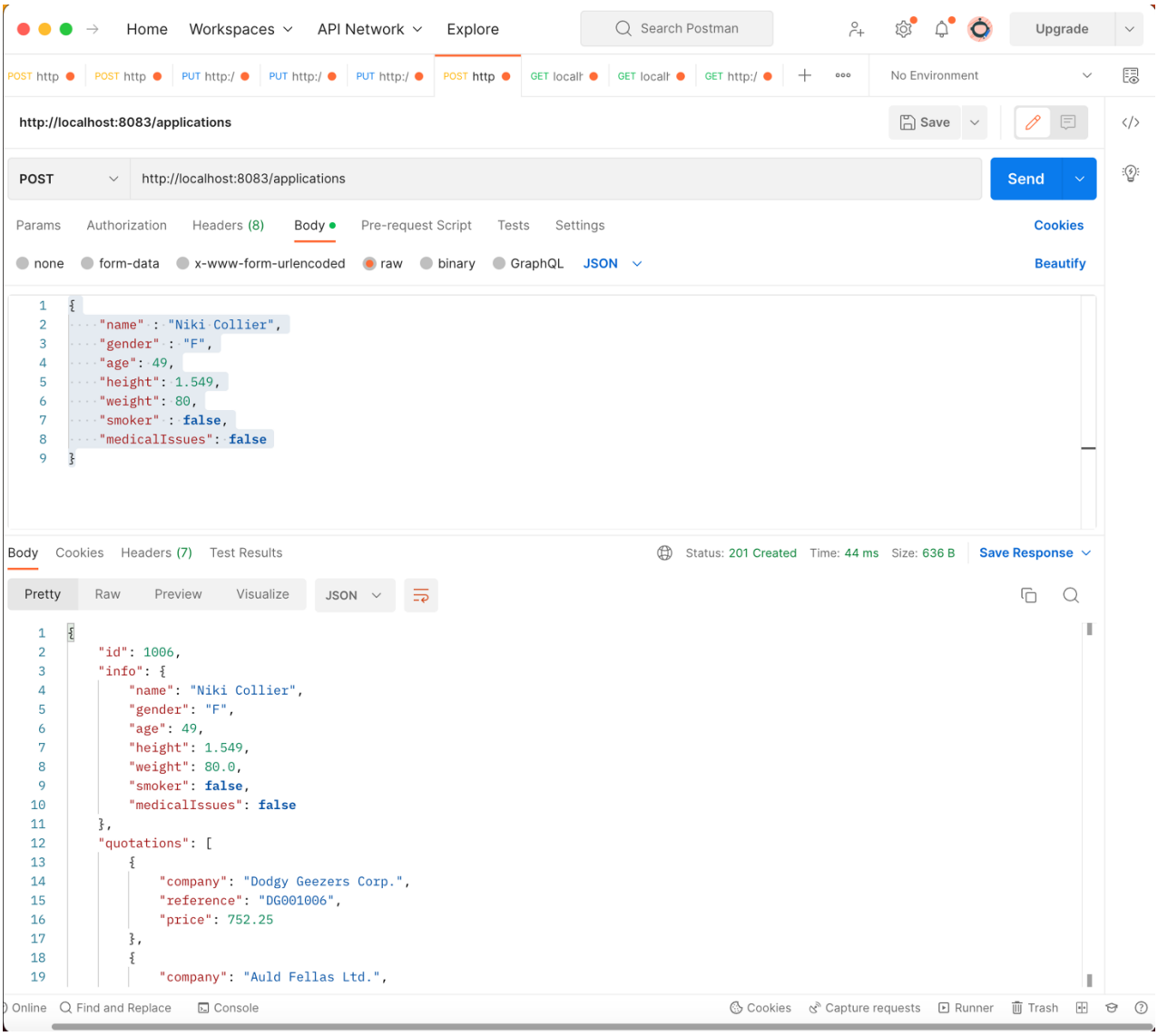
Given a URL, url, you can contact a REST Service using the RestTemplate class. An example of making a post request using this code is given below:

```
RestTemplate template = new RestTemplate();

ResponseEntity<String> response =
            template.postForEntity("{url}", info, String.class);
if (response.getStatusCode().equals(HttpStatus.CREATED)) {
    System.out.println("Location of resource: "
            + response.getHeaders().getLocation().toString());
}
```

NOTE: This service should run on port 8083.

When completed, you should be able to perform the following POST request and get the same response:



Note that, based on the above screenshot, the POST request to the `/applications` endpoint should return a representation of the application object that was created by the

c) Complete the first version of the broker code by implementing support for GETting the state of an application resource matching the pattern: `/applications/{id}`.

d) Now, implement the client. It would be best if you used the way the POST request works in part (b). The client should not be a spring boot application. Instead, you should explore one of the APIs listed at this URL:

https://reflectoring.io/comparison-of-java-http-clients/

I developed a solution using Apache HttpComponents, but the blog recommends OkHttpClient. I used synchronous `HttpPost` and `HttpGet` operations. To use the `ObjectMapper` class (this is part of the Jackson Serialisation API), you will also need the following dependency:

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.14.2</version>
</dependency>
```

As can be seen in the example code provided at the above blog, Jackson supports the transformation of an object to JSON (represented in Java as a String) through the following method:

```
String json = mapper.writeValueAsString(info);
```

Task 5: Containerisation **Grade: A**

This task involves containerizing the LifeCo-REST system. Again, the goal is to dockerize all the services but not the client. A docker-compose.yml file should be provided. The client should be run using Maven. Remember to make the folder "task5."

Note: The Spring Boot maven plugin already comes with a dependency packaging mechanism. You do not need to use the maven-shade plugin to do this.

To get an A+ in this task, you should integrate a registration service into the broker. This service should expose the list of the broker's URLs to contact quotation services. You must create a new endpoint for the broker: `/services to do this`. A POST request with a string body should add the string to the list of available quotation services. You do not necessarily need to create "quotation service" resources. Instead, a GET request to the `/services` endpoint that returns the list of the services registered with the broker should suffice. Look at the following page to see how to write `CommandLineRunners` that allow you to specify code executed at start-up in Spring Boot.

https://howtodoinjava.com/spring-boot/command-line-runner-interface-example/