mvn spring-boot:run

mvn compile spring-boot:run -pl auldfellas

<http://localhost:8080/greeting>

### Overview of Task 2: Designing Quotation Services & Implementing Auldfellas

This task requires creating a REST API specifically for **quotation services**, focusing on the **Auldfellas** service. REST APIs work by exposing resources as **endpoints** and interacting with these resources via HTTP methods (GET, POST, etc.). Here, quotations are the primary resources.

### Key Requirements

**Defining Endpoints for Quotations**:

* 1. **/quotations**: Represents the collection of all quotation resources.
  2. **/quotations/{id}**: Represents a specific quotation, identified by its id.

This setup allows clients to create new quotations at /quotations and retrieve individual quotations by their id at /quotations/{id}.

**HTTP Methods for Each Endpoint**:

* 1. **POST on /quotations**: Used to create a new quotation.
  2. **GET on /quotations**: Used to retrieve a list of all quotations.
  3. **GET on /quotations/{id}**: Used to retrieve details of a specific quotation by id.

**Response Codes**:

**201 Created** for a successful POST request, indicating a new resource has been created. The response should include:

* + 1. A **Location header**: Contains the URL of the created quotation.
    2. A **Content-Location header**: Optional but often the same as the Location header. This clarifies that the URL points to the newly created resource.

**200 OK** for a successful GET request, where the response body contains the data requested.

**406 Not Acceptable** if the client requests a format that the server cannot provide.

An **API** (Application Programming Interface) is a set of rules and protocols that allows different software applications to communicate with each other. Think of it as a bridge that enables one program to interact with another, often over the internet.

Example: **REST API**: Common for web services; uses HTTP methods like GET and POST.

**REST (Representational State Transfer)**: A web-based architecture for distributed systems, focusing on resources identified by URIs, and interactions defined by HTTP methods (GET, POST, PUT, PATCH, DELETE). REST follows a data-oriented approach, using HTTP verbs to map CRUD (Create, Read, Update, Delete) operations. Resources, represented in various formats (JSON, XML, etc.), are manipulated through stateless interactions.

resources and interactions are central concepts that enable a distributed system (like a web service) to communicate with clients (e.g., web browsers, apps) through simple and consistent methods. Here’s what each term means:

### ****Resources Identified by URIs****

“The key abstraction of information in REST is a resource. Any information that can be named can be a resource: a document or image, a temporal service (e.g., "today's weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g., a person), and so on. In other words, any concept that might be the target of an author's hypertext reference must fit within the definition of a resource. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time.”

* **Resource**: In REST, a "resource" refers to any item of interest in the system that you want to work with or retrieve. This can be data entities like a user profile, an image, a document, or even a service. EVERYTHING CAN BE SEEN AS A RESOURCE
* **URI (Uniform Resource Identifier)**: Each resource is uniquely identified by a URI, which is a web address pointing to that specific resource. For example:
  + /users/123 could be the URI for a specific user with an ID of 123.
  + /products/567 could be the URI for a specific product with an ID of 567.

URIs allow clients to access and interact with resources over the web.

### 2. ****Interactions Defined by HTTP Methods****

* REST defines a set of standard operations for interacting with these resources, typically using **HTTP methods** (verbs) that describe the type of action to perform on a resource. Here’s a quick rundown of the main HTTP methods used in REST and how they map to typical actions:
  + **GET**: Retrieves a representation of a resource. For example, GET /users/123 fetches the user with ID 123.
  + **POST**: Creates a new resource. For example, POST /users with data in the body might create a new user.
  + **PUT**: Updates or replaces a resource at the specified URI. For example, PUT /users/123 might update all details of the user with ID 123.
  + **PATCH**: Partially updates a resource. For example, PATCH /users/123 could update just the user's email.
  + **DELETE**: Removes a resource. For example, DELETE /users/123 deletes the user with ID 123.

### 3. ****Response Codes****

* HTTP response codes are standardized status codes that tell the client what happened with the request. Here are some commonly used codes in REST APIs:
  + **200 OK**: The request was successful, and the server is returning the requested data.
  + **201 Created**: A new resource was successfully created, typically in response to a POST request.
  + **204 No Content**: The request was successful, but there’s no content to return (often used with DELETE).
  + **400 Bad Request**: The server could not process the request due to client-side errors, like malformed request syntax.
  + **404 Not Found**: The requested resource was not found on the server.
  + **500 Internal Server Error**: The server encountered an error and couldn’t complete the request.

Response codes give clients immediate feedback about the success or failure of their requests, helping them understand if and why something went wrong.

### Example of Resource Identification and Interaction

Consider an API for managing a **library system**. Resources might include books, authors, and patrons. Here’s how URIs and HTTP methods are used together:

* **URI**: /books/42 identifies a specific book (Book ID 42).
* **GET**: GET /books/42 retrieves details of Book 42.
* **POST**: POST /books with book data in the body adds a new book.
* **PUT**: PUT /books/42 with new data in the body replaces the details of Book 42.
* **DELETE**: DELETE /books/42 removes Book 42 from the system.
* · If GET /books/789 is successful, the API returns 200 OK with the book data.
* · If the book with ID 789 doesn’t exist, it returns 404 Not Found.
* · After successfully creating a book with POST /books, it returns 201 Created.

### REST Stack Overview:

The REST stack is built around HTTP as the communication protocol between the client and the server. In a RESTful system, there is no need for additional complex layers such as stubs and skeletons (found in RPC/WS stacks), making it simpler and more lightweight. Here’s a breakdown of each component in the REST stack:

**HTTP as the Core Protocol**:

* 1. HTTP is used directly for client-server communication in REST.
  2. RESTful services use HTTP methods (GET, POST, PUT, DELETE, etc.) to interact with resources.

**Client**:

* 1. The client in a REST stack sends HTTP requests to access or manipulate resources on the server.
  2. Requests use a URL (Uniform Resource Locator) to identify the resource and an HTTP method to define the operation (e.g., retrieve data with GET, create data with POST).

**Service (Server)**:

* 1. The server hosts resources and handles client requests by providing the requested resource representation (e.g., JSON, XML) or performing the specified action.
  2. The server response is standardized, typically using HTTP response codes (like 200 for success, 404 for not found).

### Comparison with RPC/WS Stack:

* **RPC/WS Stack** includes:
  + **SOAP (Simple Object Access Protocol)**: A protocol for message exchange that requires a specific message structure, making it more complex.
  + **Stub and Skeleton**: Extra layers required to transform calls between the client and server into network calls, making RPC/WS stacks more complicated than REST.
  + REST avoids these layers, making it more efficient and straightforward.

In summary, the REST stack emphasizes simplicity and direct HTTP interactions, while RPC/WS adds extra layers and protocols, often leading to more complexity.

Sending Data Through REST

### 1. ****Data Through the URL Path****

* **Purpose**: Used to identify the primary resource being accessed or acted upon.
* **Example**: http://www.ucd.ie/modules/COMP30220
* **Usage**: Here, COMP30220 is part of the path and directly identifies a specific module. This is commonly used for data essential to locating or identifying a particular resource.
* **Typical Use Case**: Retrieving or modifying a specific item, such as a record in a database (like a specific student, book, or module).

### 2. ****Data Through the URL Query String****

* **Purpose**: Used for optional parameters, filters, or additional information that refines the request without changing the primary resource being accessed.
* **Example**: http://www.ucd.ie/modules?stage=3
* **Usage**: Here, stage=3 is a query parameter that filters the results to show only stage 3 modules. Query strings are often used when filtering, sorting, or otherwise refining a larger collection of resources.
* **Typical Use Case**: Searching or filtering resources, such as retrieving a list of students by year or a list of books by genre.

### 3. ****Data Through the HTTP Payload (Request Body)****

* **Purpose**: Used to send the actual data content to create or update a resource. This data is included in the body of the request rather than the URL.
* **Example**:
  + URL: http://www.ucd.ie/student/13123212
  + Payload (body): { "telephone": "012345678" }
* **Usage**: In this example, the URL identifies the student with ID 13123212, while the payload provides additional data (the updated telephone number).
* **Typical Use Case**: Sending detailed data for creating or updating resources, such as when submitting a form with user data or updating an existing record.

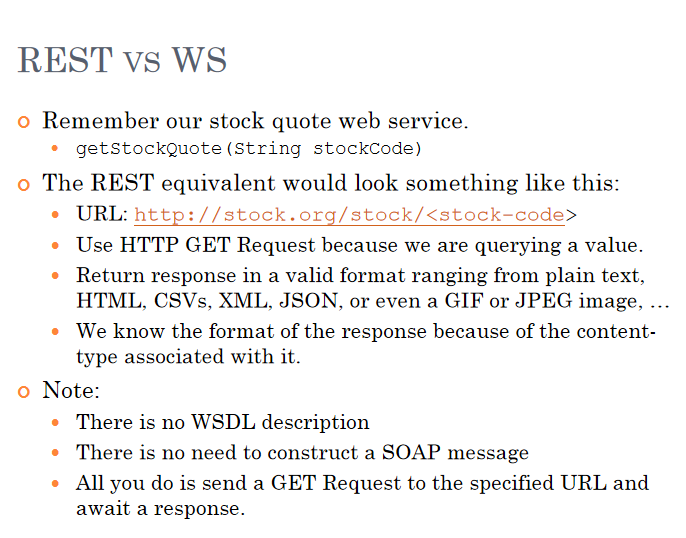
### Summary

* **URL Path**: Key data needed to identify the specific resource (e.g., /modules/COMP30220).
* **Query String**: Optional parameters for refining or filtering the request (e.g., ?stage=3).
* **HTTP Payload**: The main data content for create/update operations, sent in the request body (e.g., { "telephone": "012345678" } for updating a phone number).

### ****Specifying the Representation****

REST allows clients and servers to specify the **format of data** sent and received. This is crucial because REST APIs can exchange data in various formats like JSON, XML, or plain text. The two main HTTP headers used for this are:

1. **Content-Type**: Specifies the data format in the HTTP request body or response. For example:
   1. When sending JSON data, you would set Content-Type: application/json.
   2. For HTML, it would be Content-Type: text/html.
2. **Accept**: Indicates the data format the client prefers to receive from the server. It can be a prioritized list (comma-separated), like:
   1. Accept: application/json, text/html
   2. If the server cannot provide any of the listed types, it responds with a 406 Not Acceptable status.



**Introduction to Spring Boot**:

* · **Spring Framework**: A Java framework for enterprise applications, providing dependency injection and supporting architectural styles like MVC and WS.
* **Dependency Injection**: A design pattern where dependencies are injected into a class, making the code more modular and testable.
* **Spring Boot**: A Spring extension that simplifies application setup by auto-configuring dependencies and eliminating XML configurations. It includes @SpringBootApplication for bootstrapping and @Controller annotations for request handling.

· **Creating and Running Spring Boot Applications**:

* · Using Spring Initializr to set up projects with predefined dependencies.
* The use of @Controller to define routes and responses, and spring-boot-maven-plugin for running applications.

Spring Boot simplifies and speeds up the development of REST APIs by providing default configurations, easy dependency management, and an embedded server, eliminating complex setup. It works well with REST principles through annotations like @RestController, automatic JSON conversion, and structured exception handling. Key benefits include:

* **Fast Development**: Preconfigured settings and Spring Initializr make setup quick.
* **Easy REST Endpoint Creation**: Annotations like @GetMapping and @PostMapping make defining endpoints straightforward.
* **Data and Security Support**: Spring Boot integrates well with databases and Spring Security for secure and efficient data handling.
* **Easy Deployment**: Can be run as a standalone JAR or deployed to cloud platforms.

In short, Spring Boot is ideal for building efficient, scalable REST APIs.