Web Services
Part 2:
POST, PUT, DELETE

More Request Types

In the last lecture we saw GET's, which simply read the data. Today we will deal with request types that might potentially change the application's data permanently:

- **POST**: Ideally suited for inserting new data into the data source.
- **PUT**: Ideally suited for updating an existing record within a data source.
- **DELETE**: Ideally suited for removing an existing record from the data source.

For the POST & PUT requests we are converting an object to data

Let See POST/PUT/DELETE in Action...

Implementing POST Requests

***POST*: http://localhost:3000/hotels/{id}/reservations**

```
String API_BASE_URL = "http://localhost:3000/"
RestTemplate restTemplate = new RestTemplate();
HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.APPLICATION_JSON);

// Where reservation is an object of type Reservation.
HttpEntity<Reservation> entity = new HttpEntity<>(reservation, headers);
restTemplate.postForObject(BASE_URL + "hotels/" + reservation.getHotelID() +
"/reservations", entity, Reservation.class);
```

Implementing POST Requests

***POST*: http://localhost:3000/hotels/{id}/reservations**

Create HTTP Headers for POST

```
String API_BASE_URL = "http://localhost:3000/"
RestTemplate restTemplate = new RestTemplate();
HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.APPLICATION_JSON);

// Where reservation is an object of type Reservation.
HttpEntity<Reservation> entity = new HttpEntity<>(reservation, headers);
restTemplate.postForObject(BASE_URL + "hotels/" + reservation.getHotelID() +
"/reservations", entity, Reservation.class);
```

Implementing POST Requests

***POST*: http://localhost:3000/hotels/{id}/reservations**

Create HTTP Headers for POST

Set the **content-type** for JSON

```
String API_BASE_URL = "http://localhost:3000/"
RestTemplate restTemplate = new RestTemplate();
HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.APPLICATION_JSON);

// Where reservation is an object of type Reservation.
HttpEntity<Reservation> entity = new HttpEntity<>(reservation, headers);
restTemplate.postForObject(BASE_URL + "hotels/" + reservation.getHotelID() +
"/reservations", entity, Reservation.class);
```

Implementing POST Requests

***POST*: http://localhost:3000/hotels/{id}/reservations**

Create HTTP Headers for POST

Set the **content-type** for JSON

Create an **HttpEntity**, which allows us to combine headers and body

```
String API_BASE_URL = "http://localhost:3000/"
RestTemplate restTemplate = new RestTemplate();
HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.APPLICATION_JSON);

// Where reservation is an object of type Reservation.
HttpEntity<Reservation> entity = new HttpEntity<>(reservation, headers);
restTemplate.postForObject(BASE_URL + "hotels/" + reservation.getHotelID() +
"/reservations", entity, Reservation.class);
```

Implementing POST Requests

***POST*: http://localhost:3000/hotels/{id}/reservations**

Create HTTP Headers for POST

Set the **content-type** for JSON

Create an **HttpEntity**, which allows us to combine headers and body

Call **postForObject** with the **HttpEntity** and class to post for (Reservation)

```
String API_BASE_URL = "http://localhost:3000/"
RestTemplate restTemplate = new RestTemplate();
HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.APPLICATION_JSON);

// Where reservation is an object of type Reservation.
HttpEntity<Reservation> entity = new HttpEntity<>(reservation, headers);
restTemplate.postForObject(BASE_URL + "hotels/" + reservation.getHotelID() +
"/reservations", entity, Reservation.class);
```

Implementing PUT Requests

- PUT requests are similar to POST requests in that they usually have both headers and a payload contained in the message body.
    - We can write code for a PUT request much like our POST code but using the put method rather than postForObject method.

```
// Create instance of RestTemplate
RestTemplate restTemplate = new RestTemplate();
// Create instance of HttpHeaders and set Content-Type to application/json
HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.APPLICATION_JSON);
// Combine headers with existing user object to form HttpEntity
HttpEntity<User> entity = new HttpEntity<>(newUser, headers);

// Put (update) the existing user using the entity.
restTemplate.put(API_URL + "users/23", entity);
```

Implementing DELETE Requests

- DELETE requests are similar to GET requests In that they usually have only headers and not a payload contained in the message body.
    - We can write code for a DELETE request much like our GET code but using the delete method rather than getForObject method.

```
RestTemplate restTemplate = new RestTemplate();
restTemplate.delete(API_URL + "users/23");
```

LET'S TRY WRITING THE POST CODE!

Exceptions and Error Handling

There are 2 exceptions to be aware of when dealing with APIs:

- **`RestClientResponseException`** - is thrown when a status code other than a 2XX is returned.
  - Can check status code via this Exception's **`getRawStatusCode()`** method
  - Can get text description of the status code (i.e. Not Found for 404) from this Exception's **`getStatusText()`** method
- **`ResourceAccessException`** - is thrown when there was a network issue that prevented a successful call.

Closing Resources

It's important to clean up your resources when you are done using them. We've mentioned that you shouldn't close the **`System.in`** stream until your program exits because it can't be used in your code once closed.

It ***IS*** important to close **`Scanner`** objects and other resources that are still open when your program exits though. Let's look at a best practice for doing this by looking at the **`ConsoleService`** class used in yesterday's lecture:

```
public ConsoleService() {
    scanner = new Scanner(System.in);
 }
```

The **`ConsoleService`** creates a scanner object in its constructor. We should close it before exiting.

Closing Resources

When we use resources that stay open for the life of the program, a best practice is to clean them up using an `exit()` method. This is accomplished by creating a public `exit()` method in any classes that need to be cleaned up on exit. Here's what that would look like for our `ConsoleService`:

```
public void exit() {
    scanner.close();
    System.exit(0);
}
```

The exit method in the `ConsoleService` closes the `Scanner` object and then exits. The main code would call this method when it is ready to exit. If there were multiple objects that had `exit()` methods to be called, the `System.exit(0);` would be in the last `exit()` called.