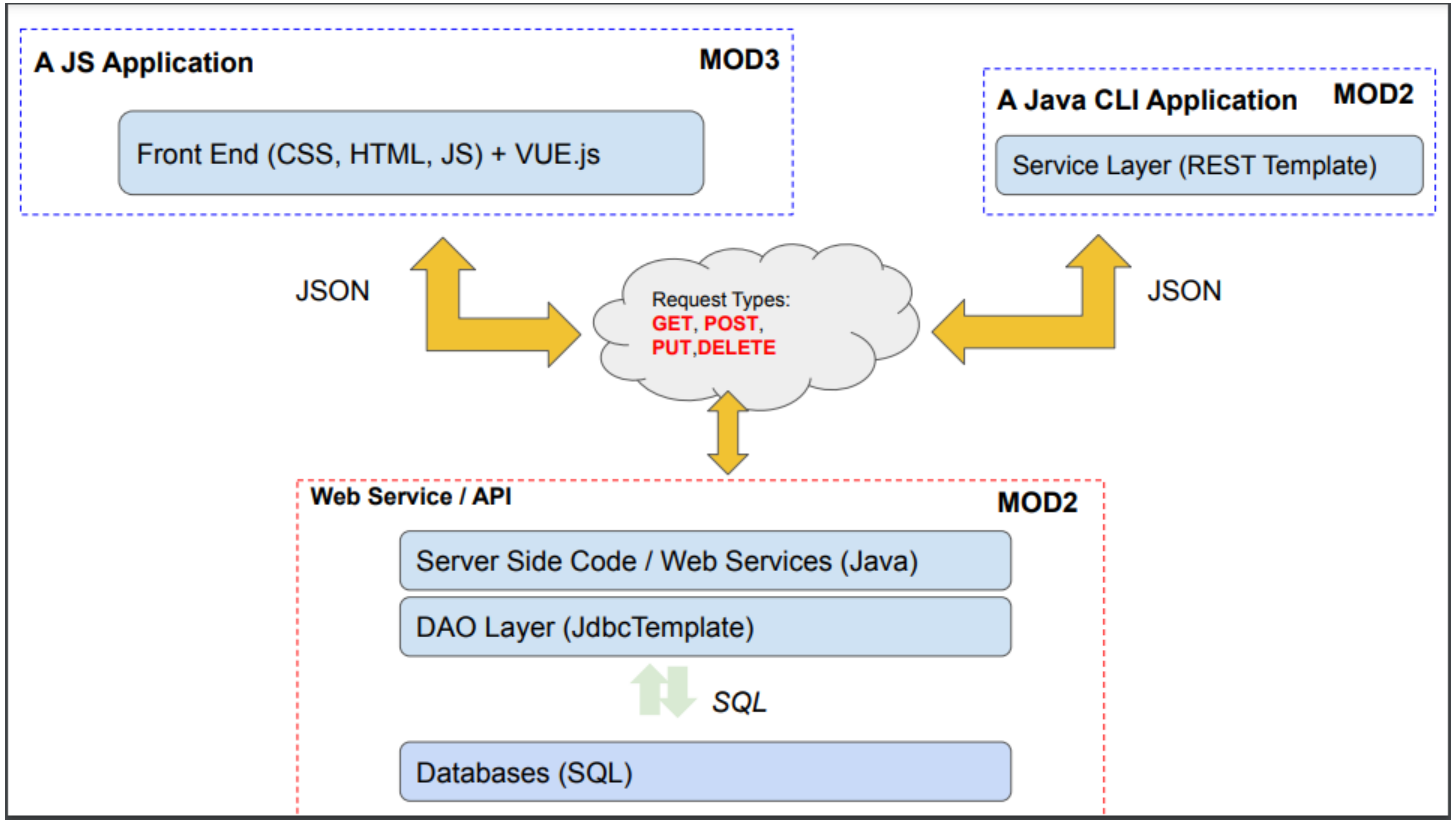
# HTTP Web Services POST

Module 2: 12

# Today’s Objectives

1. Services
2. HTTP Method Types
3. HTTP Method: GET
4. Object Serialization and Deserialization
5. Handling API Errors
6. HTTP Method: POST
7. HTTP Method: PUT
8. HTTP Method: DELETE


A JS Application
MOD3
Front End (CSS, HTML, JS) + VUE.js
A Java CLI Application
MOD2
Service Layer (REST Template)
JSON
Request Types:
GET, POST,
PUT,DELETE
JSON
Web Service / API
MOD2
Server Side Code / Web Services (Java)
DAO Layer (JdbcTemplate)
SQL
Databases (SQL)

# 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

# GET vs. POST

**GET**

- Can be cached
- Remain in browser history
- Can be bookmarked
- Should never be used for sensitive data
- Maximum length of 2048 characters
- Used to request data

**POST**

- Are never cached
- Do not remain in browser history
- Cannot be bookmarked
- Have no restrictions on length

# HTTP Methods - GET

## Usage

- HTTP GET is generally used to retrieve web pages to display
- It also is included for
  - images
  - documents
  - stylesheets, script files
- Search Pages
- HTTP GET requests are easily bookmarked because the parameters are in the url.

**HTTP GET** should never modify any data on the server. Get does not change the state on the server and has the same result each time it is repeated.

Parameters for a GET request travel as either Path Parameters or in the **Query String**.

**http://localhost:3000/hotels/2/reviews?stars=4**

# HTTP Methods - POST

## Usage

- HTTP POST is used when
  - Data must be secure  (credit card number, password, etc.)
  - Data is too large for the URL
  - When the request is asking to add something on the server
- HTTP POST requests cannot be bookmarked or sent with the browser directly.

.

**HTTP POST** indicates that the request will add new data to the the server.  Post modifies the server and leaves the server in a different state each time the same request is repeated.

POST transfers data in the message body instead of the URL.
While HTTPS encrypts the message body, it cannot encrypt the URL.

# HTTP Methods - PUT

## Usage

- HTTP PUT is used to update existing data
- HTTP PUT requests cannot be bookmarked or sent with the browser directly.
- HTTP PUT requests are meant to overwrite the entire record with the new values.

.

**HTTP PUT** indicates that the request will update existing data on the the server.  Put changes the state on the server but the state remains the same if the same request is repeated multiple times.

PUT transfers data in the message body instead of the URL.

While HTTPS encrypts the message body, it cannot encrypt the URL.

# HTTP Methods - DELETE

## Usage

- HTTP DELETE is used to remove existing data from the server
- HTTP DELETE requests cannot be bookmarked or sent with the browser directly.
- Usually only requires the entities id, so parameters are sent in the query string.

.

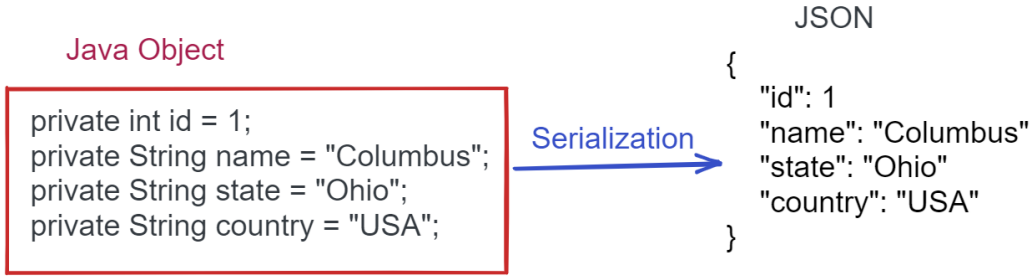
**HTTP DELETE** indicates that the request will remove existing data from the the server.

Delete changes the state of the server but the state remains the same if the request is repeated multiple times.

# Object Serializing and Deserializing

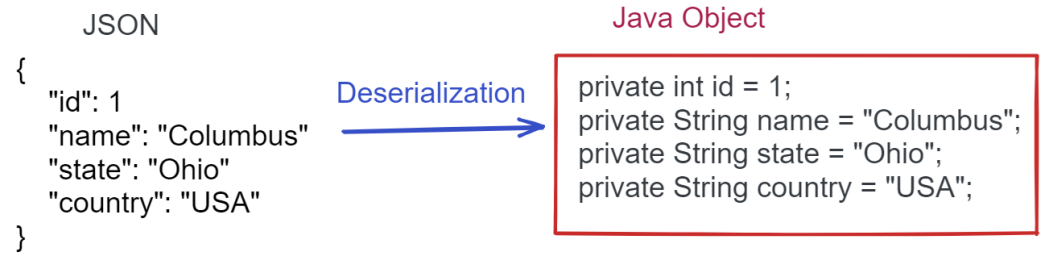
Transforming a Java Object to a string representation of the object, JSON, is called **Serialization.**

Java Object → JSON is **Serialization**




Transforming a string representation of an object, JSON, into an Java Object is called **Deserialization**.

JSON → Java Object is **Deserialization**



# Exceptions and Error Handling

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

- **RestClientResponseException** - for when a status code other than a 2XX is returned.

- **ResourceAccessException** - for when there was a network issue that prevented a successful call.

# Handling REST API Errors

`RestClientResponseException`

Catches common error status codes, like 401 (Unauthorized), 404 (Not Found) , or 500 (Server Exception).

`ResourceAccessException`

Catches connection errors when the server cannot be reached.

These can be handled using the Java Exception try...catch

```
try {
            hotels = restTemplate.getForObject(BASE_URL + "hotels", Hotel[].class);
catch (RestClientResponseException e) {
            // handle common errors
} catch (ResourceAccessException e) {
            // handle connection errors
}
```

# Successful status codes

Successful POST, PUT and Delete return successful status codes:

POST – returns 201

PUT – 200 or 204

DELETE – 202 or 204

# Implementing a POST

Suppose the documentation for the API specifies POST as well :

(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);
```

# Coding a POST

## Setting Headers

```
   HttpHeaders headers = new HttpHeaders();

   headers.setContentType(MediaType.APPLICATION_JSON);

   HttpEntity<Reservation> entity = new HttpEntity<>(reservation, headers);
```

## POST with postForObject()

```
restTemplate.postForObject(url, entity, Reservation.class);
```

The **HttpEntity** object contains the *Headers* and message *Body*.

Since we want the reservation to be in the message body instead of the Query String, we set it in the entity object and pass it to the restTemplate.

# Implementing an PUT

Suppose the API’s documentation states that there is a PUT endpoint:
*(PUT) http://localhost:3000/reservations/{id}*

Using a REST template we can implement the following:

```
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.put(BASE_URL + "reservations/" + reservation.getId(), entity);
```

# Implementing an PUT

Sometimes requests require that a body and a header be sent along as well. The HttpEntity object helps us captures these pieces of information:

```
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.put(BASE_URL + "reservations/" + reservation.getId(), entity);
```

- Here we have a header consisting of an instance of the HttpHeaders class.
- We also have a body, which will just be an instance of a Reservation class.

# Coding a PUT

## Setting Headers

```
   HttpHeaders headers = new HttpHeaders();

   headers.setContentType(MediaType.APPLICATION_JSON);

  HttpEntity<Reservation> entity = new HttpEntity<>(reservation, headers);
```

## POST with put()

```
restTemplate.put(url, entity);
```

# Implementing a DELETE

Assuming that the API's documentation states that there is a DELETE endpoint:
*(DELETE) http://localhost:3000/reservations/{id}*

… using a REST template we can implement the following:

```
String API_BASE_URL = "http://localhost:3000/"
RestTemplate restTemplate = new RestTemplate();

// Where id is an int:
restTemplate.delete(BASE_URL + "reservations/" + id);
```

Let’s code.