API Server Side

Part 2

Objectives

- Understand the high-level architectural elements of the REST architecture
- Explain how HTTP methods are used to interact with RESTful resources
- Conform to RESTful conventions when using HTTP status codes
- Define CRUD and how it relates to HTTP methods and APIs
- Handle errors from backend code and alert the frontend that something has gone wrong
- Perform validation on client supplied request data
- Implement a RESTful web service that provides full CRUD functionality

Addressing Resources

- Build a URL to address your resource
- http://localhost:8080/hotels/*reservationId*
- http://localhost:8080/hotels/*reservationId*/reservations

Actions on Resources

- CRUD!
  - Create => POST
  - Read => GET
  - Update => PUT
  - Delete => DELETE

HTTP Status codes

CATEGORY	DESCRIPTION
1xx: Informational	Communicates transfer protocol-level information.
2xx: Success	Indicates that the client's request was accepted successfully.
3xx: Redirection	Indicates that the client must take some additional action in order to complete their request.
4xx: Client Error	This category of error status codes points the finger at clients.
5xx: Server Error	The server takes responsibility for these error status codes.

Creating REST APIs

```java
/**
 * Updates a reservation
 *
 * @param reservation
 * @param id
 * @return the updated Reservation
 * @throws ReservationNotFoundException
 */
@RequestMapping(path = "/reservations/{id}", method = RequestMethod.PUT)
public Reservation update(@RequestBody Reservation reservation,
                @PathVariable int id) throws ReservationNotFoundException {

    return reservationDAO.update(reservation, id);
}

/**
 * Delete a reservation by id
 *
 * @param id
 * @throws ReservationNotFoundException
 */
@RequestMapping(path = "/reservations/{id}", method = RequestMethod.DELETE)
public void delete(@PathVariable int id) throws ReservationNotFoundException {

    reservationDAO.delete(id);
}
```

Validation in Spring Boot
The use of the @Valid annotation...

Creating REST APIs

```java
/**
 * Updates a reservation
 *
 * @param reservation
 * @param id
 * @return the updated Reserva
 * @throws ReservationNotFound        ion
 */
@RequestMapping(path = "/reservations/{id}", method = RequestMethod.PUT)
public Reservation update(@Valid @RequestBody Reservation reservation,
                @PathVariable int id) throws ReservationNotFoundException {

    return reservationDAO.update(reservation, id);
}

/**
 * Delete a reservation by id
 *
 * @param id
 * @throws ReservationNotFoundException
 */
@ResponseStatus(HttpStatus.NO_CONTENT)
@RequestMapping(path = "/reservations/{id}", method = RequestMethod.DELETE)
public void delete(@PathVariable int id) throws ReservationNotFoundException {

    reservationDAO.delete(id);
}
```

Server Side Validation

We can add special bits of annotation code to our classes to ensure that the data is consistent and free of errors. Some examples of these:

- In an Automobile class, an attribute measuring fuel tank capacity must be between 0 and 10 liters.

- For a Hotel reservation class, a begin date or end date must be provided.

- For a Customer class, a value must be provided for the customer name.

The goal is to implement these validation rules right on the classes that contain the fields that require validation.

Validation Annotations in Spring

Here is a list of common Spring validation annotations:

- **@NotBlank("message"):** Will check if a field is blank or not.

- **@Email("message"):** Verify in an input conforms to an email format.

- **@Min(value=<<x>>, message = "message"):** A form must have a minimum input value, where <<x>> is that number.

- **@Max(value=<<y>>, message = "message"):** A form must have a maximum input value, where <<y>> is that number.

Server Side Validation Example

Consider the following code:

```
public class Reservation {
    private int id;
    @Min( value = 1, message = "The field 'hotelID' is required.")
    @Max( value = 8, message = "Hotel IDs are between 1 and 8.")
    // a value must be provided for a hotel id:
    private int hotelID;
    @NotBlank( message = "The field 'fullName' is required.")
    // a value must be provided for the guest name
    private String fullName;
    @NotBlank( message = "The field 'checkinDate' is required.")
    private String checkinDate;
    @NotBlank( message = "The field 'checkoutDate' is required.")
    private String checkoutDate;
… }
```

PUT Requests

- A PUT request is used to update existing data.

- Like a POST, PUT requires a body containing the updated JSON Object:

- The validation techniques we learned with POST still apply!

- Consider the following example:

```
@RequestMapping(path = "/reservations/{id}", method = RequestMethod.PUT)
public Reservation update(@Valid @RequestBody Reservation reservation,
            @PathVariable int id)throws ReservationNotFoundException    {
    return reservationDAO.update(reservation, id);
  }
```

DELETE Requests

- A DELETE request is used to remove data.

```
@RequestMapping(path = "/reservations/{id}", method = RequestMethod.DELETE)
public void delete(@PathVariable int id,
        throws ReservationNotFoundException   {
    reservationDAO.delete(id);
}
```

Summary of Request Types

We have now covered the four basic persistent data storage operations. This is commonly referred to as CRUD (**C**reate, **R**ead, **U**pdate, and **D**elete).

Modulating the Response Code

Finally, Spring gives us the ability to tweak the response code a user receives. To review, recall the status code ranges:

- **2XX** : Everything is fine.

- **4XX** : There is a client side problem, something is wrong with your request.

- **5XX**: There is a server side problem

Common examples of each:

- **200**: Yep, everything's fine.

- **401**: Your request contains bad credentials.

- **500**: Internal Server Error

Modulating the Response Code

We can provide slightly more descriptive codes by using the @ResponseStatus annotations:

```
@ResponseStatus(HTTPStatus.CREATED)
@RequestMapping(path = "/hotels/{id}/reservations",
                            method = RequestMethod.POST)
public Reservation addReservation (…) {
    …
}
```

Provided the request completed without issue, the response back to the API user will now be 201 instead of 200.

Dependency Injection

Notice the constructor does not specify the class. We allow Spring to inject the correct class for us.

```java
private HotelDao hotelDao;
private ReservationDao reservationDao;

public HotelController(HotelDao hotelDao, ReservationDao reservationDao) {
    this.hotelDao = hotelDao;
    this.reservationDao = reservationDao;
}
```

Dependency Injection

Spring knows what class to inject
by the @Component annotation

```java
@Component
public class MemoryHotelDao implements HotelDao {
```

Dependency Injection

Spring will take this one step further for us, by Autowiring the component

```java
@Component
public class MemoryHotelDao implements HotelDao {
```

```java
private HotelDao hotelDao;
private ReservationDao reservationDao;

public HotelController(HotelDao hotelDao, ReservationDao reservationDao) {
    this.hotelDao = hotelDao;
    this.reservationDao = reservationDao;
}
```
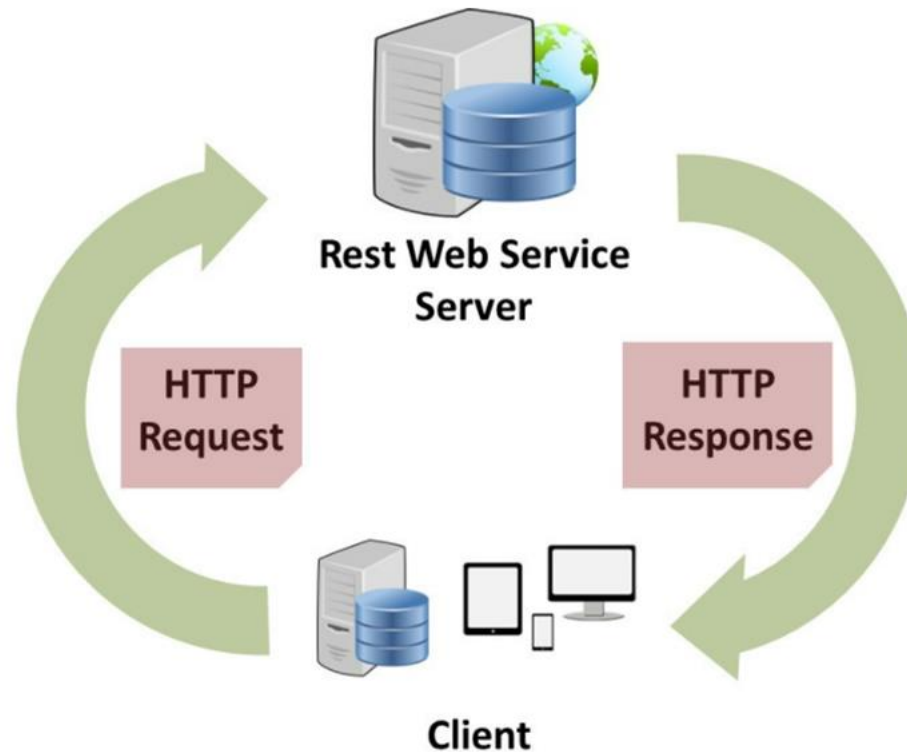
```java
@Autowired
private HotelDao hotelDao;
@Autowired
private ReservationDao reservationDao;
```
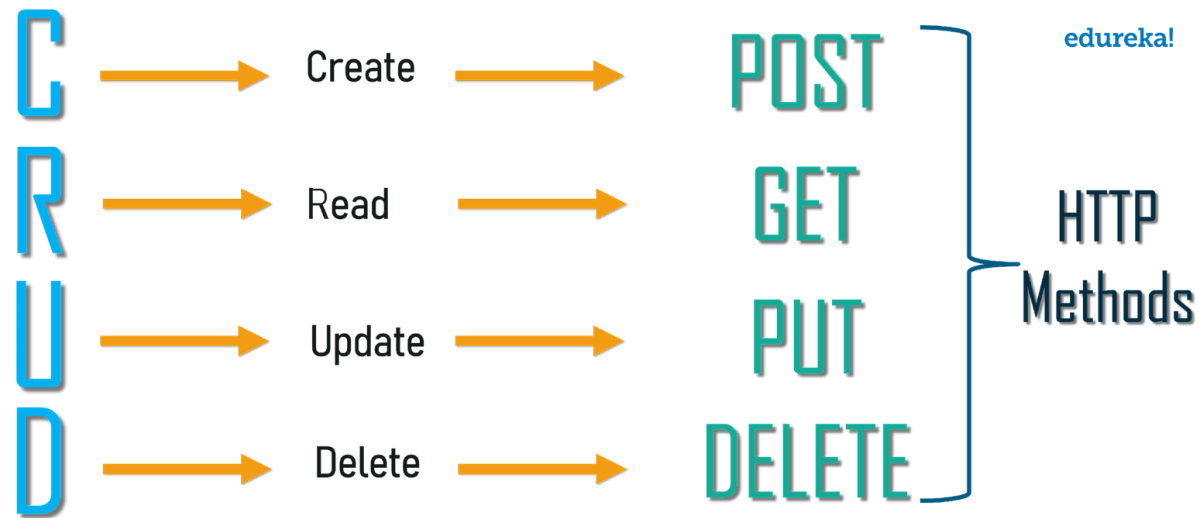
Objectives

- Understand the high-level architectural elements of the REST architecture

Rest Web Service Server

HTTP Request

HTTP Response

Client

REST is a simple way to organize interactions between independent systems.

Objectives

- Understand the high-level architectural elements of the REST architecture
- Define CRUD and how it relates to HTTP methods and APIs

Objectives

- Understand the high-level architectural elements of the REST architecture

- Define CRUD and how it relates to HTTP methods and APIs

- Handle errors from backend code and alert the frontend that something has gone wrong

- Perform validation on client supplied request data

```java
public class Reservation {

    private int id;
    @Min( value = 1, message = "The field 'hotelID' is required.")
    private int hotelID;
    @NotBlank( message = "The field 'fullName' is required.")
    private String fullName;
    @NotBlank( message = "The field 'checkinDate' is required.")
    private String checkinDate;
    @NotBlank( message = "The field 'checkoutDate' is required.")
    private String checkoutDate;
    @Min( value = 1, message = "The minimum number of guests is 1")
    @Max( value = 5, message = "The maximum number of guests is 5")
    private int guests;
```

Objectives

- Understand the high-level architectural elements of the REST architecture

- Define CRUD and how it relates to HTTP methods and APIs

- Handle errors from backend code and alert the frontend that something has gone wrong

- Perform validation on client supplied request data

- Implement a RESTful web service that provides full CRUD functionality

CRUD!

Let's Code!