Data Validation using Bean Validation API

Introduction to Bean Validation API

Bean Validation API

This API simplifies the validation of input field. It provides annotation based constraints such as @NotNull and @Email which can be used on bean attributes to specify validation criteria. Some of the annotations provided by this API are given below.

Annotation	Description
@NotNull	Checks that the annotated property is not null
@Null	Checks that the annotated property is null
@Min	Checks that the annotated property has value greater than or equal to provided value
@Max	Checks that the annotated property has value less than or equal to provided value
@Email	Checks that the annotated property is a valid email address
@NotEmpty	Checks that the annotated property is not null or empty for given collection
@NotBlank	Checks that the annotated property is not null or white space
@Pattern	Checks that the annotated property follows the provided as its regex attribute
@Past	Checks that a date value is in the past
@PastOrPresent	Checks that a date value is in the past including the present date
@Future	Checks that a date value is in the future
@FutureOrPresent	Checks that a date value is in the future including the present date
@Size	Checks that size of annotated property is between its min and max attributes. It can be applied to String, Collection, Map, and array properties.

All these annotations have message attribute which is the message that will be displayed when the validation fails.

To use Bean Validation API, you have to add following **spring-boot-starter-validation** dependency in pom.xml:

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

</dependency>
```

Not let us see how to use these annotations. Consider the following CustomerDTO class:

```
public class CustomerDTO {

  private Integer customerId;
  //It should be valid email id and should not be null
  private String emailId;
  //It should not be null and should contain only alphabets and space
  private String name;
    //It should be past or present date
    private LocalDate dateOfBirth;
    /getters and setters

}
```

Now suppose you have to apply following validations on attributes of this class:

- emailId should be valid email address
- name should not be null

This can be done by using constraint validation annotations of this API on attributes of Customer bean as follows:

```
public class CustomerDTO {

  private Integer customerId;
  @Email(message = "Please provide valid email address")
    @NotNull(message = "Please provide email address")
  private String emailId;
  @NotNull(message = "Please provide customer name")
    @Pattern(regexp="[A-Za-z]+( [A-Za-z]+)*", message="Name should contain only alphabets and space")
  private String name;
    @PastOrPresent(message = "Date of birth should be past or present date")
  private LocalDate dateOfBirth;

    //getter and setters

}
```

In above code, message attribute of constraint validation annotations specifies the error message which is sent to the client if validation fails. For example, if emailId validation fails "Please provide valid email address" message will be sent to the client. You can also write these validation message in external properties file and then you can use them in your validation annotations. To achieve this, create ValidationMessages.properties file in src/resources folder of your application and add the messages as shown below:

```
customer.emailid.absent=Please provide email address
customer.emailid.invalid=Please provide valid email address
customer.name.absent=Please provide customer name
customer.name.invalid=Name should contain only alphabets and space

customer.dob.invalid=Date of birth should be past or present date
```

Now you can read these messages directly in validation annotations as follows:

```
public class CustomerDTO {
    private Integer customerId;

    @Email(message = "{customer.emailid.invalid}")
    @NotNull(message = "{customer.emailid.absent}")
    private String emailId;

    @NotNull(message = "{customer.name.absent}")
    @Pattern(regexp="[A-Za-z]+( [A-Za-z]+)*", message="{customer.name.invalid}")
  private String name;

    @PastOrPresent(message = "customer.dob.invalid")
  private LocalDate dateOfBirth;
    //getter and setter

}
```

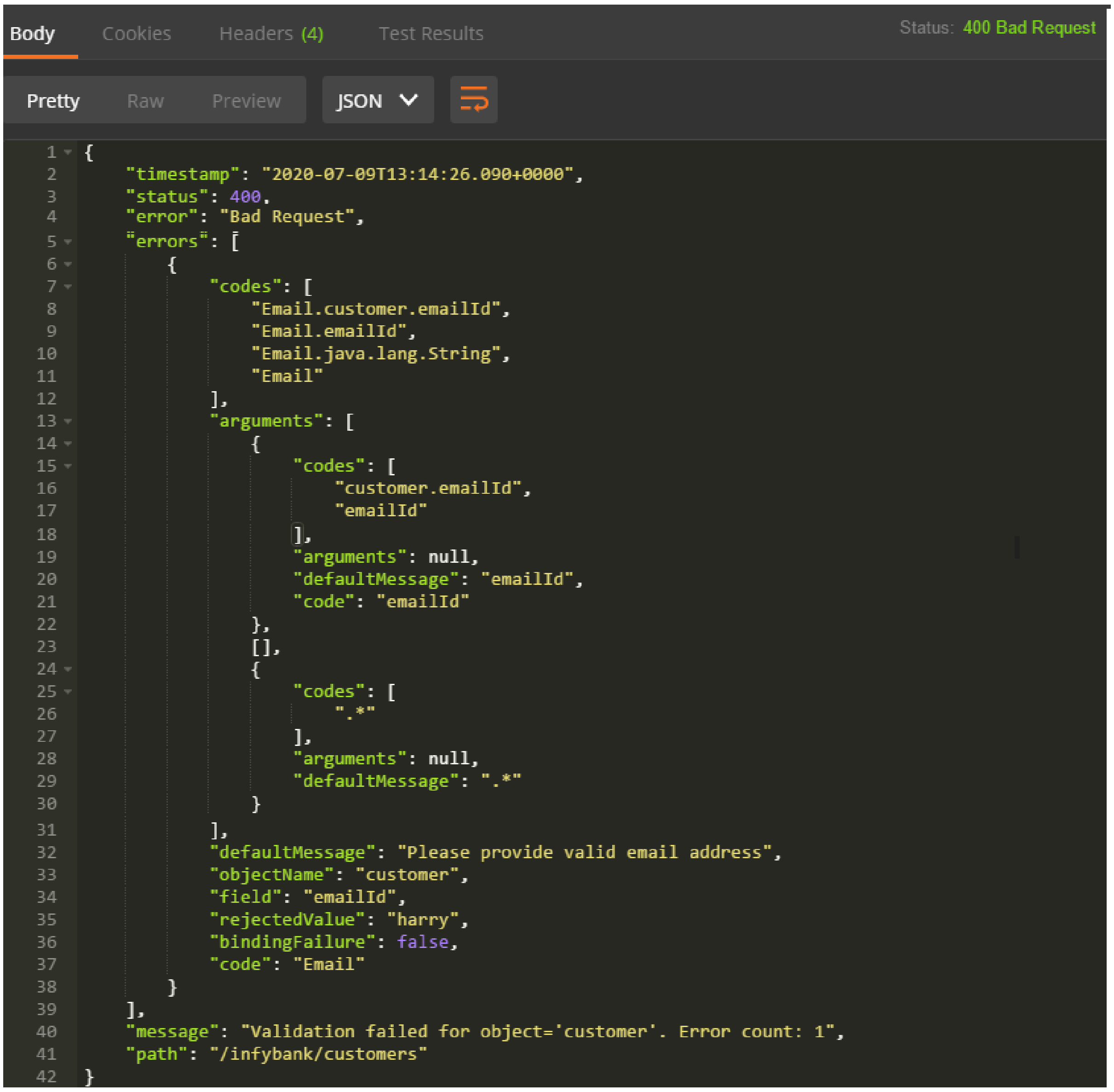
Adding the constraints on the bean attributes will not carry out validation. You also have to mention that validation is required. For this annotate customerDTO parameter of addCustomer() method of REST controller with **@Valid** annotation as shown below:

```
@PostMapping(value = "/customers")
public ResponseEntity<String> addCustomer(@Valid @RequestBody CustomerDTO customerDTO) throws CgBankException  {
    //rest of the code
}
```

This annotation tells Spring to perform data validation after binding input data to customerDTO. If the validation fails **MethodArgumentNotValidException** will be thrown and Spring will translate this exception to a response with HTTP status 400 (Bad Request). Now if you send POST request with following data:

```
{
    "emailId": "harry",
    "name": "Harry",
    "dateOfBirth": "2020-07-08"
}
```

You will get following response from which you can see that emailId validation has failed with HTTP status code 400.:



```
{
    "timestamp": "2020-07-09T13:14:26.090+0000",
    "status": 400,
    "error": "Bad Request",
    "errors": [
        {
            "codes": [
                "Email.customer.emailId",
                "Email.emailId",
                "Email.java.lang.String",
                "Email"
            ],
            "arguments": [
                {
                    "codes": [
                        "customer.emailId",
                        "emailId"
                    ],
                    "arguments": null,
                    "defaultMessage": "emailId",
                    "code": "emailId"
                },
                [],
                {
                    "codes": [
                        ".*"
                    ],
                    "arguments": null,
                    "defaultMessage": ".*"
                }
            ],
            "defaultMessage": "Please provide valid email address",
            "objectName": "customer",
            "field": "emailId",
            "rejectedValue": "harry",
            "bindingFailure": false,
            "code": "Email"
        }
    ],
    "message": "Validation failed for object='customer'. Error count: 1",
    "path": "/infybank/customers"
}
```

This error message is not user friendly. You can also customize this. Let us see how to do this

Customizing validation errors

Customizing validation errors

You have seen how error messages are displayed to client when validation fails. These error messages can also be customized to give more descriptive information to the client about the errors so that they can easily understand the errors. You have already learned that to customize the error messages a custom error class has to be created. For example, the following ErrroInfo class gives information about error message, error code and date and time of the exception:

```
public class ErrorInfo  {
  private String errorMessage;
  private Integer errorCode;
  private LocalDateTime timestamp;
  // getter and setter

}
```

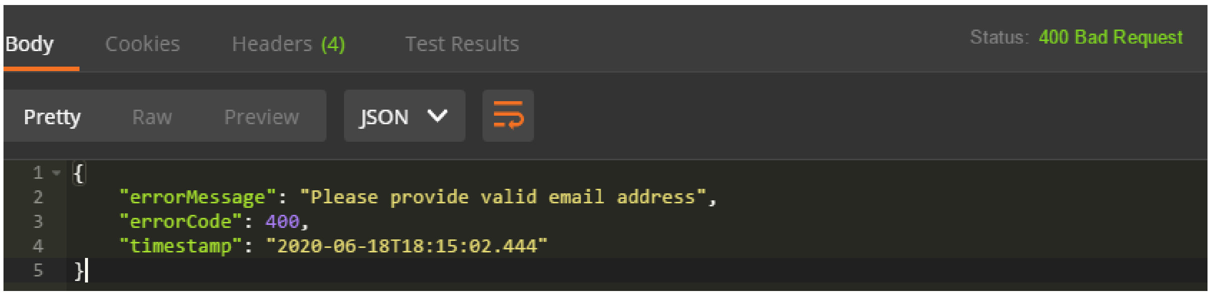
The object of this class is created and populated with values in exception handler method. The MethodArgumentNotValidException is thrown when bean attribute validation fails so to handle this exception a method has to be added in exception handler class as shown below:

```
@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity<ErrorInfo>
exceptionHandler(MethodArgumentNotValidException exception) {

          ErrorInfo errorInfo = new ErrorInfo();
          errorInfo.setErrorCode(HttpStatus.BAD_REQUEST.value());
      errorInfo.setTimestamp(LocalDateTime.now());

          String errorMsg =
exception.getBindingResult().getAllErrors().stream().map(x ->
x.getDefaultMessage())
                             .collect(Collectors.joining(", "));

          errorInfo.setErrorMessage(errorMsg);

          return new ResponseEntity<>(errorInfo, HttpStatus.BAD_REQUEST);
}

```

In above code, first an object of ErrorInfo class is created and populated with value of HTTP status code and timestamp of exception. Then received exception object is used to get all the errors and associated messages. All these messages are concatenated into single message separated by comma (,). Then this error message is set as message of ErrorInfo object. Then object of ResponseENtity is returned with ErrorInfo object. Now send the POST request to your controller with following data:

```
{
    "emailId": "harry",
    "name": "Harry",
    "dateOfBirth": "2020-07-08"

}
```

You will the get response which is more user friendly and simple as follows:



```
{
    "errorMessage": "Please provide valid email address",
    "errorCode": 400,
    "timestamp": "2020-06-18T18:15:02.444"
}
```

Cascading Validation

Cascading Validation

Using Bean Validation API, you can not only validate single object but also object having reference of other object. This is also called as cascading of validation. To do this you have to annotate the reference of other object with @Valid annotation. For example, in following CustomerDTO class which has a reference of AddressDTO class which is annotated with @Valid annotation to cascade validation from CustomerDTO to AddressDTO.

```
public class CustomerDTO {
    private Integer customerId;
    @Email(message = "Please provide valid email address")
    @NotNull(message = "Please provide email address")
    private String emailId;
    @NotNull(message = "Please provide customer name")
    @Pattern(regexp="[A-Za-z]+( [A-Za-z]+)*", message="Name should contain 
only alphabets and space")
    private String name;
    @PastOrPresent(message = "Date of birth should be past or present date")
    private LocalDate dateOfBirth;
    @NotNull
    @Valid
    private AddressDTO addressDTO;

    //getter and setters
}


public class AddressDTO {
    private Integer addressId;
    @NotNull(message = "Please provide street")
    private String street;
    @NotNull(message = "Please provide city")
    private String city;
    //getter and setters
 }
```

Data Validation using Bean Validation API - Demo

Objective:

This demo illustrates how to do data validation in Spring REST application.

Steps:

Step 1: Using Spring Initializr, create a Spring Boot project with following specifications:

- Spring Boot Version: 2.3.1

- Group: com.cg
- Artifact: Demo_SpringREST_Validation
- Name: Demo_SpringREST_Validation
- Package name: com.cg
- Java Version: 11
- Dependencies: Spring Data JPA, MySQL Driver,Spring Web,Spring Boot DevTools and Spring Validation.

Now import this project in Eclipse.

Step 2: Create the database and table

Open MySQL terminal and execute the following command:

```
drop database if exists customer_db;
create database customer_db;
use  customer_db;

create table address(
   address_id int auto_increment,
   street varchar(50),
   city varchar(20),
   constraint ps_address_id_pk primary key (address_id)
);
create table customer(
   customer_id int auto_increment,
   email_id varchar(50),
   name varchar(20),
   date_of_birth date,
   address_id int,
   constraint ps_customer_id_pk primary key (customer_id),
   constraint ps_address_id_fk foreign key (address_id) references
address(address_id)
);
insert into address values (1, '15 Yemen Road', 'Yemen');
insert into address values (2, 'Wallstreet', 'New York');
insert into address values (3, 'Houston Street', 'New York');

insert into customer values (1, 'martin@cg.com', 'Martin', sysdate()-
interval 9000 day, 1);
insert into customer values (2, 'tim@cg.com', 'Tim', sysdate()- interval
5000 day, 2);
insert into customer values (3, 'jack@cg.com', 'Jack', sysdate()- interval
6000 day, 3);

commit;
select * from customer;


select * from address;
```

Step 3: Configure the data source

Open application.properties in src/main/resources folder and add following properties for MySQL:

```
# MySQL settings
#Change these settings according to database you are using
spring.datasource.url=jdbc:mysql://localhost:3306/customer_db
spring.datasource.username=root

#If MySQL installation is password proctored,then use below property to set
password
#spring.datasource.password=root

#JPA settings
```

```
spring.jpa.show-sql=true

spring.jpa.properties.hibernate.format_sql=true
```

Step 4: Create the following AddressDTO class in com.cg.dto package :

```java
public class AddressDTO {
    private Integer addressId;
    @NotNull(message = "Please provide street")
    private String street;
    @NotNull(message = "Please provide city")
    private String city;

  public Integer getAddressId() {
          return addressId;
  }
  public void setAddressId(Integer addressId) {
          this.addressId = addressId;
  }
  public String getStreet() {
          return street;
  }
  public void setStreet(String street) {
          this.street = street;
  }
  public String getCity() {
          return city;
  }
  public void setCity(String city) {
          this.city = city;
  }

  @Override
  public String toString() {
          return "AddressDTO [addressId=" + addressId + ", street=" + street
+ ", city=" + city + "]";
  }

}

```

Step 5: Create the following CustomerDTO class in com.cg.dto package :

```java
public class CustomerDTO {

  private Integer customerId;

  @Email(message = "{customer.emailid.invalid}")
  @NotNull(message = "{customer.emailid.absent}")
  private String emailId;

  @NotNull(message = "{customer.name.absent}")
  @Pattern(regexp = "[A-Za-z]+( [A-Za-z]+)*", message =
"{customer.name.invalid}")
  private String name;

  @PastOrPresent(message = "{customer.dob.invalid}")
  private LocalDate dateOfBirth;

    @NotNull
  @Valid
  private AddressDTO addressDTO;

  public Integer getCustomerId() {
      return customerId;
  }

  public void setCustomerId(Integer customerId) {
      this.customerId = customerId;
  }

  public String getEmailId() {
      return emailId;
  }

  public void setEmailId(String emailId) {
      this.emailId = emailId;
  }

  public String getName() {
      return name;
  }

  public void setName(String name) {
      this.name = name;
  }

  public LocalDate getDateOfBirth() {
      return dateOfBirth;
  }

  public void setDateOfBirth(LocalDate dateOfBirth) {
      this.dateOfBirth = dateOfBirth;
  }

  public AddressDTO getAddressDTO() {
      return addressDTO;
  }

  public void setAddressDTO(AddressDTO addressDTO) {
      this.addressDTO = addressDTO;
  }

  @Override
  public String toString() {
      return "CustomerDTO [customerId=" + customerId + ", emailId=" + emailId + ", name=" + name + ", dateOfBirth="
              + dateOfBirth + ", addressDTO=" + addressDTO + "]";
  }

}

```

Step 6: Create the following Address class in com.cg.entity package :

```java
@Entity
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer addressId;
    private String street;
    private String city;

    public Integer getAddressId() {
        return addressId;
    }

    public void setAddressId(Integer addressId) {
        this.addressId = addressId;
```

```java
	}

	public String getStreet() {
		return street;
	}

	public void setStreet(String street) {
		this.street = street;
	}

	public String getCity() {
		return city;
	}

	public void setCity(String city) {
		this.city = city;
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + ((this.getAddressId() == null) ? 0 : this.getAddressId().hashCode());
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Address other = (Address) obj;
		if (this.getAddressId() == null) {
			if (other.getAddressId() != null)
				return false;
		}
		else if (!this.getAddressId().equals(other.getAddressId()))
			return false;
		return true;
	}

	@Override
	public String toString() {
		return "Address [addressId=" + addressId + ", street=" + street + ", city=" + city + "]";
	}

}
```

Step 7: Create the following Customer class in com.cg.entity package :

```java
@Entity
public class Customer {

	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private Integer customerId;
	private String emailId;
	private String name;
	private LocalDate dateOfBirth;
	@OneToOne(cascade = CascadeType.ALL)
	@JoinColumn(name = "address_id", unique = true)
```

```java
    private Address address;

    public Integer getCustomerId() {
        return customerId;
    }

    public void setCustomerId(Integer customerId) {
        this.customerId = customerId;
    }

    public String getEmailId() {
        return emailId;
    }

    public void setEmailId(String emailId) {
        this.emailId = emailId;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public LocalDate getDateOfBirth() {
        return dateOfBirth;
    }

    public void setDateOfBirth(LocalDate dateOfBirth) {
        this.dateOfBirth = dateOfBirth;
    }

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((this.getCustomerId() == null) ? 0 : this.getCustomerId().hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Customer other = (Customer) obj;
        if (this.getCustomerId() == null) {
            if (other.getCustomerId() != null)
                return false;
        } else if (!this.getCustomerId().equals(other.getCustomerId()))
            return false;
        return true;
    }

```

```
	@Override
	public String toString() {
		return "Customer [customerId=" + customerId + ", emailId=" + emailId + ", name=" + name + ", dateOfBirth="
				+ dateOfBirth + ", address=" + address + "]";
	}


}
```

Step 8: Create the following CustomerRepository interface in com.cg.repository package :

```
public interface CustomerRespository extends CrudRepository<Customer, Integer> {

}
```

Step 9: Create the following CustomerService interface in com.cg.service package :

```
public interface CustomerService {
	public Integer addCustomer(CustomerDTO customerDTO) throws CgBankException;
	public CustomerDTO getCustomer(Integer customerId) throws CgBankException;
	public void updateCustomer(Integer customerId, String emailId)throws CgBankException;
	public void deleteCustomer(Integer customerId)throws CgBankException;
	public List<CustomerDTO> getAllCustomers() throws CgBankException;

}
```

Step 10: Create the following CustomerServiceImpl class in com.cg.service package :

```
@Service(value = "customerService")
@Transactional
public class CustomerServiceImpl implements CustomerService {

	@Autowired
	private CustomerRespository customerRespository;

	@Override
	public CustomerDTO getCustomer(Integer customerId) throws CgBankException {
		Optional<Customer> optional = customerRespository.findById(customerId);
		Customer customer = optional.orElseThrow(() -> new CgBankException("Service.CUSTOMER_NOT_FOUND"));
		CustomerDTO customerDTO = new CustomerDTO();
		customerDTO.setCustomerId(customer.getCustomerId());
		customerDTO.setDateOfBirth(customer.getDateOfBirth());
		customerDTO.setEmailId(customer.getEmailId());
		customerDTO.setName(customer.getName());
		AddressDTO addressDTO = new AddressDTO();
		addressDTO.setAddressId(customer.getAddress().getAddressId());
		addressDTO.setStreet(customer.getAddress().getStreet());
		addressDTO.setCity(customer.getAddress().getCity());
		customerDTO.setAddressDTO(addressDTO);
		return customerDTO;
	}

	@Override
	public Integer addCustomer(CustomerDTO customerDTO) throws CgBankException {
		Customer customer = new Customer();
		customer.setDateOfBirth(customerDTO.getDateOfBirth());
```

```java
        customer.setEmailId(customerDTO.getEmailId());
        customer.setName(customerDTO.getName());
        customer.setCustomerId(customerDTO.getCustomerId());
        Address address = new Address();
        address.setStreet(customerDTO.getAddressDTO().getStreet());
        address.setCity(customerDTO.getAddressDTO().getCity());
        customer.setAddress(address);
        customerRespository.save(customer);
        return customer.getCustomerId();
    }

    @Override
    public void updateCustomer(Integer customerId, String emailId) throws CgBankException {
        Optional<Customer> customer = customerRespository.findById(customerId);
        Customer c = customer.orElseThrow(() -> new CgBankException("Service.CUSTOMER_NOT_FOUND"));
        c.setEmailId(emailId);
    }

    @Override
    public void deleteCustomer(Integer customerId) throws CgBankException {
        Optional<Customer> customer = customerRespository.findById(customerId);
        customer.orElseThrow(() -> new CgBankException("Service.CUSTOMER_NOT_FOUND"));
        customerRespository.deleteById(customerId);
    }

    @Override
    public List<CustomerDTO> getAllCustomers() throws CgBankException {
        Iterable<Customer> customers = customerRespository.findAll();
        List<CustomerDTO> customerDTOs = new ArrayList<>();
        customers.forEach(customer -> {
            CustomerDTO customerDTO = new CustomerDTO();
            customerDTO.setCustomerId(customer.getCustomerId());
            customerDTO.setDateOfBirth(customer.getDateOfBirth());
            customerDTO.setEmailId(customer.getEmailId());
            customerDTO.setName(customer.getName());

            AddressDTO addressDTO = new AddressDTO();
            addressDTO.setAddressId(customer.getAddress().getAddressId());
            addressDTO.setStreet(customer.getAddress().getStreet());
            addressDTO.setCity(customer.getAddress().getCity());
            customerDTO.setAddressDTO(addressDTO);
            customerDTOs.add(customerDTO);
        });
        if (customerDTOs.isEmpty())
            throw new CgBankException("Service.CUSTOMERS_NOT_FOUND");
        return customerDTOs;
    }


}
```

Step 11: Create an aspect LoggingAspect class in com.cg.utility package:

```java
@Component
@Aspect
public class LoggingAspect {
    public static final Log LOGGER = LogFactory.getLog(LoggingAspect.class);
    @AfterThrowing(pointcut = "execution(* com.cg.service.*Impl.*(..))", throwing = "exception")
    public void logServiceException(Exception exception) {
        LOGGER.error(exception.getMessage(), exception);
    }

}
```

Step 12: Create the following CgBankException class in com.cg.exception package :

```
public class CgBankException extends Exception {

    private static final long serialVersionUID = 1L;

    public CgBankException(String message) {
            super(message);
    }
}

```

Step 13: Create the following ErrorInfo class in com.cg.utility package :

```
public class ErrorInfo {
    private String errorMessage;
    private Integer errorCode;
    private LocalDateTime timestamp;

    public String getErrorMessage() {
            return errorMessage;
    }

    public void setErrorMessage(String errorMessage) {
            this.errorMessage = errorMessage;
    }

    public Integer getErrorCode() {
            return errorCode;
    }

    public void setErrorCode(Integer errorCode) {
            this.errorCode = errorCode;
    }

    public LocalDateTime getTimestamp() {
            return timestamp;
    }

    public void setTimestamp(LocalDateTime timestamp) {
            this.timestamp = timestamp;
    }

}

```

Step 14: Create the following ExceptionControllerAdvice class in com.cg.utility package :

```
@RestControllerAdvice
public class ExceptionControllerAdvice {
    @Autowired
    Environment environment;

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorInfo> exceptionHandler(Exception exception) {
            ErrorInfo error = new ErrorInfo();
```

```
  error.setErrorMessage(environment.getProperty("General.EXCEPTION_MESSAGE"));
          error.setErrorCode(HttpStatus.INTERNAL_SERVER_ERROR.value());
          error.setTimestamp(LocalDateTime.now());
          return new ResponseEntity<ErrorInfo>(error, HttpStatus.INTERNAL_SERVER_ERROR);
  }

  @ExceptionHandler(CgBankException.class)
  public ResponseEntity<ErrorInfo> cgBankexceptionHandler(CgBankException exception) {
          ErrorInfo error = new ErrorInfo();

  error.setErrorMessage(environment.getProperty(exception.getMessage()));
          error.setTimestamp(LocalDateTime.now());
          error.setErrorCode(HttpStatus.NOT_FOUND.value());
          return new ResponseEntity<ErrorInfo>(error, HttpStatus.NOT_FOUND);
  }
  @ExceptionHandler(MethodArgumentNotValidException.class)
  public ResponseEntity<ErrorInfo> exceptionHandler(MethodArgumentNotValidException exception) {

          ErrorInfo errorInfo = new ErrorInfo();
          errorInfo.setErrorCode(HttpStatus.BAD_REQUEST.value());


          String errorMsg = exception.getBindingResult().getAllErrors().stream().map(x -> x.getDefaultMessage())
                              .collect(Collectors.joining(", "));

          errorInfo.setErrorMessage(errorMsg);
          errorInfo.setTimestamp(LocalDateTime.now());
          return new ResponseEntity<>(errorInfo, HttpStatus.BAD_REQUEST);
  }


}
```

Step 15: Create the following CustomerAPI class in com.cg.api package:

```
@RestController
@RequestMapping(value = "/cgbank")
public class CustomerAPI {

  @Autowired
  private CustomerService customerService;

  @Autowired
  private Environment environment;

  @GetMapping(value = "/customers")
  public ResponseEntity<List<CustomerDTO>> getAllCustomerDetails() throws Exception {
          List<CustomerDTO> customerList = customerService.getAllCustomers();
          return  new ResponseEntity<>(customerList, HttpStatus.OK);
  }

  @GetMapping(value = "/customers/{customerId}")
  public ResponseEntity<CustomerDTO> getCustomerDetails(@PathVariable Integer customerId) throws Exception {
          CustomerDTO customer = customerService.getCustomer(customerId);
          return new ResponseEntity<>(customer, HttpStatus.OK);
  }

```

```java
	@PostMapping(value = "/customers")
	public ResponseEntity<String> addCustomer(@Valid @RequestBody CustomerDTO customerDTO) throws Exception {
		Integer customerId = customerService.addCustomer(customerDTO);
		String successMessage = environment.getProperty("API.INSERT_SUCCESS") + customerId;
		return new ResponseEntity<>(successMessage, HttpStatus.CREATED);
	}

	@PutMapping(value = "/customers/{customerId}")
	public ResponseEntity<String> updateCustomer(@PathVariable Integer customerId, @RequestBody CustomerDTO customer)
			throws Exception {
		customerService.updateCustomer(customerId, customer.getEmailId());
		String successMessage = environment.getProperty("API.UPDATE_SUCCESS");
		return new ResponseEntity<>(successMessage, HttpStatus.OK);
	}

	@DeleteMapping(value = "/customers/{customerId}")
	public ResponseEntity<String> deleteCustomer(@PathVariable Integer customerId) throws Exception {
		customerService.deleteCustomer(customerId);
		String successMessage = environment.getProperty("API.DELETE_SUCCESS");
		return new ResponseEntity<>(successMessage, HttpStatus.OK);
	}

}
```

Step 16: Add the following in application.properties :

```
Service.CUSTOMER_NOT_FOUND=No customer found with given customer id.
Service.CUSTOMERS_NOT_FOUND=No customers found.
General.EXCEPTION_MESSAGE=Request could not be processed due to some issue. Please try again!

API.INSERT_SUCCESS=Customer added successfully with customer id :
API.UPDATE_SUCCESS=Customer emailid successfully updated.
API.DELETE_SUCCESS=Customer details deleted successfully.

server.port=8765

```

Step 17: Create ValidationMessages.properties in src/main/resources folder :

```
customer.emailid.absent=Please provide email address
customer.emailid.invalid=Please provide valid email address
customer.name.absent=Please provide customer name
customer.name.invalid=Name should contain only alphabets and space
customer.dob.invalid=Date of birth should be past or present date
```

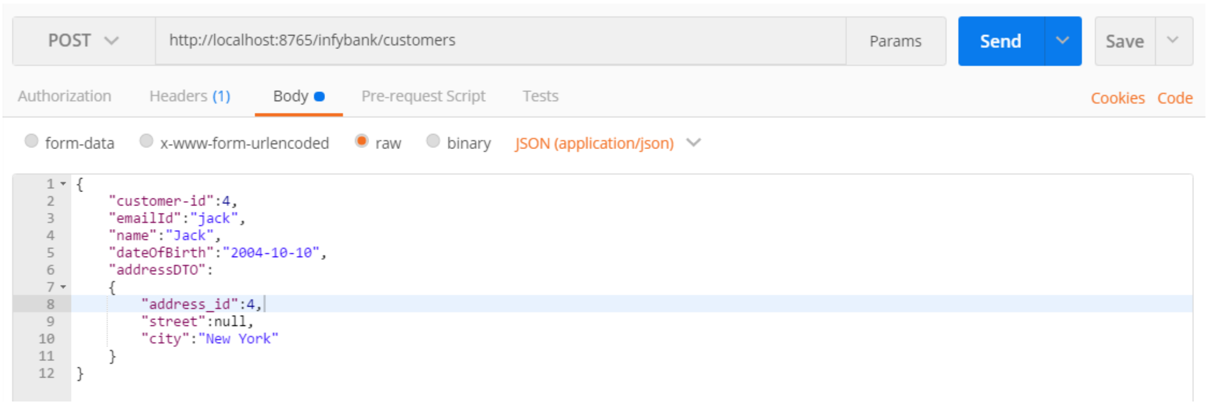
Step 18 : Deploy the application on the server by executing the class containing the main method.

We have successfully implemented global exception handling. Now, you can test it using postman client.

Testing Web Service using Postman

Step 1: Launch Postman.

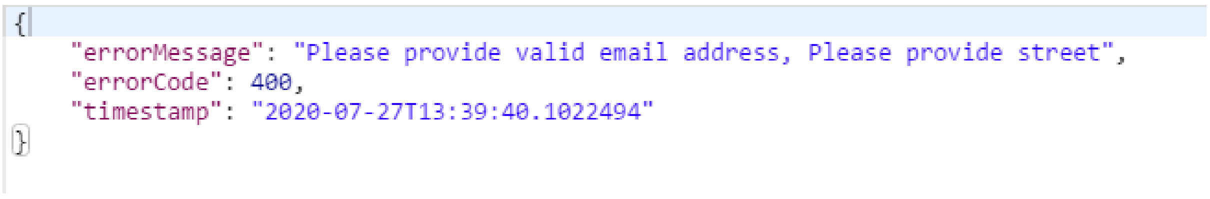
Step 2: Provide HTTP POST request - **http://localhost:8765/cgbank/customers** by passing an incorrect email id as shown below.



http://localhost:8765/infybank/customers

```
{
   "customer-id":4,
   "emailId":"jack",
   "name":"Jack",
   "dateOfBirth":"2004-10-10",
   "addressDTO":
   {
       "address_id":4,
       "street":null,
       "city":"New York"
   }
}
```

We observe the following exception is thrown.



```
{
    "errorMessage": "Please provide valid email address, Please provide street",
    "errorCode": 400,
    "timestamp": "2020-07-27T13:39:40.1022494"
}
```

The error message indicates the attribute of the bean which failed during validation.

Thus, we can conclude that, bean validation has been implemented successfully.

Path Variable Validation

So far, you have learnt how to validate the data sent in request body. But what to do if path variable or request parameter is invalid. For example, if you want to restrict the value of customerId in path variable between 1 and 100. This requires validation of path variable or request parameter. You can also do this validation using Bean Validation API. Lets see how to do this.

For this, annotate the controller class with **@Validated** annotation to tell Spring to validate path variables and request parameters. Then annotate the parameters of the handler methods which needs to be validated with constraint annotations.

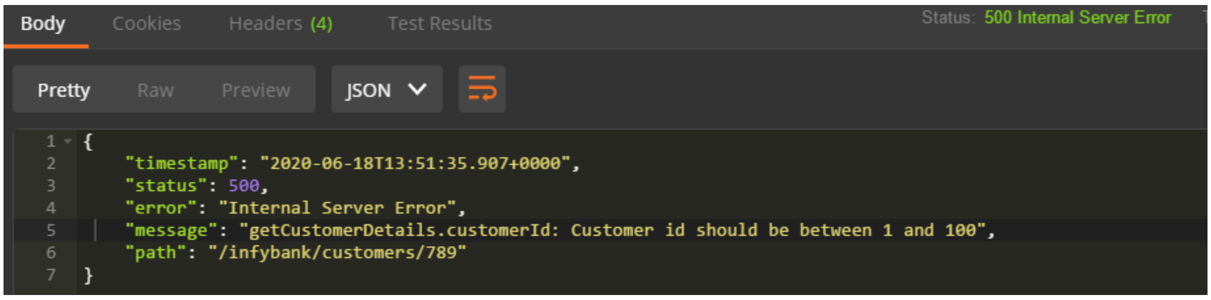
For example, if you want to specify that customerId should be between 1 and 100 then modify the controller class as follows:

```
@RestController
@RequestMapping(value="/cgbank")
@Validated
public class CustomerAPI {

  @Autowired
  private CustomerService customerService;

  @GetMapping(value = "/customers/{customerId}")
  public ResponseEntity<Customer> getCustomerDetails(@PathVariable @Min(value
= 1, message = "Customer id should be between 1 and 100") @Max(value = 100,
message = "Customer id should be between 1 and 100") Integer customerId)
throws Exception  {
```

```
            Customer customer = customerService.getCustomer(customerId);
            ResponseEntity<Customer> response = new
ResponseEntity<Customer>(customer, HttpStatus.OK);
            return response;
  }

```

In above code if customerId validation fails then ConstraintViolationException will be thrown instead of MethodArgumentNotValidException. The Spring will translate this exception to a response with HTTP status code as 500 which means internal server error.

Now, if you try to fetch details of a customer with customerId = 200 you will get following response:



```
{
    "timestamp": "2020-06-18T13:51:35.907+0000",
    "status": 500,
    "error": "Internal Server Error",
    "message": "getCustomerDetails.customerId: Customer id should be between 1 and 100",
    "path": "/infybank/customers/789"
}
```

In above response status code 500 indicates internal server error. But actually it is a bad request with status code as 400. You can customize this by adding exception handler method to handle ConstraintViolationException in handler class. In this way the error message can also be customized. For this create the following ErrroInfo class:

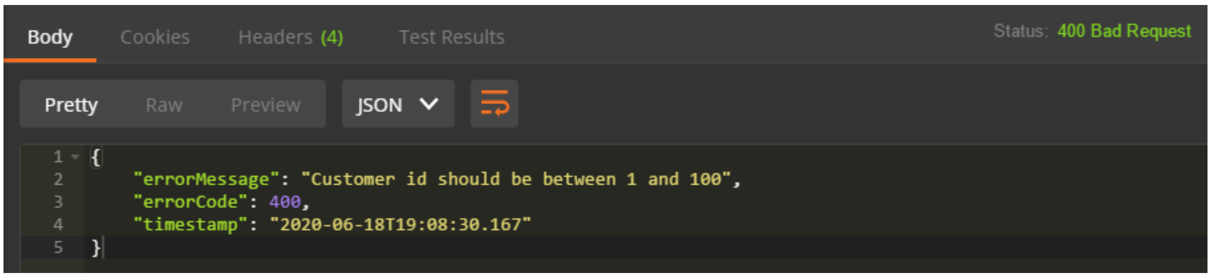
```
public class ErrorInfo  {
  private String errorMessage;
  private Integer errorCode;
  private LocalDateTime timestamp;
  // getter and setter

}
```

The object of this class is created and populated with values in exception handler. When path variables or request parameter validation fails the ConstraintViolationException is thrown. So to handle these exceptions, a method has to be added to exception handler as shown below:

```
@RestControllerAdvice
public class RestExceptionHandler {

  @ExceptionHandler(ConstraintViolationException.class)
  public ResponseEntity<ErrorInfo>
pathExceptionHandler(ConstraintViolationException exception) {

           ErrorInfo errorInfo = new ErrorInfo();
           errorInfo.setErrorCode(HttpStatus.BAD_REQUEST.value());

           String errorMsg =
exception.getConstraintViolations().stream().map(x -> x.getMessage())
                               .collect(Collectors.joining(", "));
           errorInfo.setErrorMessage(errorMsg);
           errorInfo.setTimestamp(LocalDateTime.now());
           return new ResponseEntity<>(errorInfo, HttpStatus.BAD_REQUEST);
  }

}
```

In above code, first an object of ErrorInfo class is created and populated with value of HTTP status code and timestamp of exception. Then received exception object is used to get all the errors and associated messages. All these messages are concatenated into single message separated by comma (,). Then this error message is set as message of ErrorInfo object. Then object of ResponseEntity is returned with ErrorInfo object.

For example, if you try to fetch details of a customer with customerId = 200 you will get following response:



```
{
    "errorMessage": "Customer id should be between 1 and 100",
    "errorCode": 400,
    "timestamp": "2020-06-18T19:08:30.167"
}
```

Path Variable Validation - Demo

Objective:

This demo illustrates how to do data validation of path variable in Spring REST application.

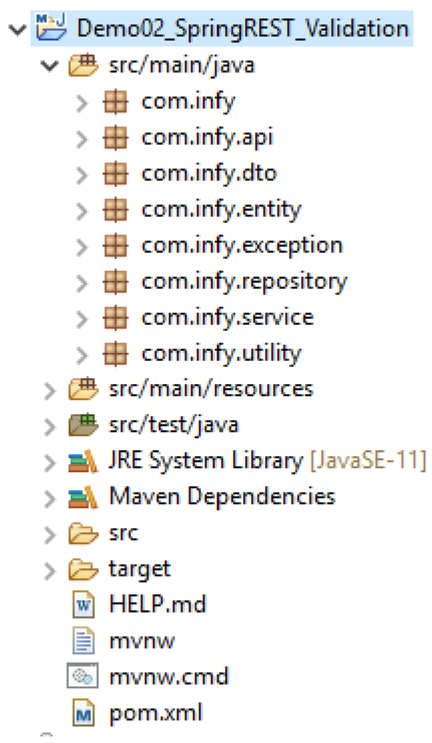
Steps:

Step 1: Using Spring Initializr, create a Spring Boot project with following specifications:

- Spring Boot Version: 2.3.1
- Group: com.cg
- Artifact: Demo02_SpringREST_Validation
- Name: Demo02_SpringREST_Validation
- Package name: com.cg
- Java Version: 11
- Dependencies: Spring Data JPA, MySQL Driver, Spring Web, Spring Boot DevTools, Spring Validation

Now import this project in Eclipse.

Step 2: Modify the imported project according to the following project structure :



```
Demo02_SpringREST_Validation
  src/main/java
    com.infy
    com.infy.api
    com.infy.dto
    com.infy.entity
    com.infy.exception
    com.infy.repository
    com.infy.service
    com.infy.utility
  src/main/resources
  src/test/java
  JRE System Library [JavaSE-11]
  Maven Dependencies
  src
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

Step 3: Create the database and table

Open MySQL terminal and execute the following command:

```sql
drop database if exists customer_db;
create database customer_db;
use  customer_db;
create table customer(
   customer_id int auto_increment,
   email_id varchar(50),
   name varchar(20),
   date_of_birth date,
   constraint ps_customer_id_pk primary key (customer_id)
);
insert into customer (customer_id, email_id, name, date_of_birth) values (1, 'martin@cg.com', 'Martin', sysdate()- interval 9000 day);
insert into customer (customer_id, email_id, name, date_of_birth) values (2, 'tim@cg.com', 'Tim', sysdate()- interval 5000 day);
insert into customer (customer_id, email_id, name, date_of_birth) values (3, 'jack@cg.com', 'Jack', sysdate()- interval 6000 day);
commit;

select * from customer;
```

Step 4: Configure the data source

Open application.properties in src/main/resources folder and add following properties for MySQL:

```
# MySQL settings
#Change these settings according to database you are using
spring.datasource.url=jdbc:mysql://localhost:3306/customer_db
spring.datasource.username=root
#If MySQL installation is password proctored,then use below property to set password
#spring.datasource.password=root
#JPA settings
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

Step 5: Create the following CustomerDTO class in com.cg.dto package :

```java
public class CustomerDTO {

  private Integer customerId;

  @Email(message = "{customer.emailid.invalid}")
  @NotNull(message = "{customer.emailid.absent}")
  private String emailId;

  @NotNull(message = "{customer.name.absent}")
  @Pattern(regexp = "[A-Za-z]+( [A-Za-z]+)*", message = "{customer.name.invalid}")
  private String name;

  @PastOrPresent(message = "customer.dob.invalid")
  private LocalDate dateOfBirth;

  public CustomerDTO() {

  }

  public CustomerDTO(Integer customerId, String emailId, String name) {
          super();
          this.customerId = customerId;
          this.emailId = emailId;
          this.name = name;
  }

  public Integer getCustomerId() {
          return customerId;
  }

  public void setCustomerId(Integer customerId) {
          this.customerId = customerId;
  }

  public String getEmailId() {
          return emailId;
  }

  public void setEmailId(String emailId) {
          this.emailId = emailId;
  }

  public String getName() {
          return name;
  }

  public void setName(String name) {
          this.name = name;
  }

  public LocalDate getDateOfBirth() {
          return dateOfBirth;
  }

  public void setDateOfBirth(LocalDate dateOfBirth) {
          this.dateOfBirth = dateOfBirth;
  }

  @Override
  public String toString() {
```

```java
		return "CustomerDTO [customerId=" + customerId + ", emailId=" + emailId + ", name=" + name + ", dateOfBirth="
				+ dateOfBirth + "]";
	}


}
```

Step 6: Create the following Customer class in com.cg.entity package :

```java
@Entity
public class Customer {

	@Id
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	private Integer customerId;
	private String emailId;
	private String name;
	private LocalDate dateOfBirth;


	public Integer getCustomerId() {
		return customerId;
	}

	public void setCustomerId(Integer customerId) {
		this.customerId = customerId;
	}

	public String getEmailId() {
		return emailId;
	}

	public void setEmailId(String emailId) {
		this.emailId = emailId;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public LocalDate getDateOfBirth() {
		return dateOfBirth;
	}

	public void setDateOfBirth(LocalDate dateOfBirth) {
		this.dateOfBirth = dateOfBirth;
	}

	@Override
	public int hashCode() {

		return 31;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Customer other = (Customer) obj;
		if (this.getCustomerId() == null) {
			if (other.getCustomerId() != null)
				return false;
		} else if (!this.getCustomerId().equals(other.getCustomerId()))
			return false;
		return true;
	}

	@Override
	public String toString() {
		return "Customer [customerId=" + customerId + ", emailId=" + emailId + ", name=" + name + ", dateOfBirth="
				+ dateOfBirth + "]";
	}


}

```

Step 7: Create the following CustomerRepository interface in com.cg.repository package :

```java
public interface CustomerRespository extends CrudRepository<Customer, Integer> {

}

```

Step 8: Create the following CustomerService interface in com.cg.service package :

```java
public interface CustomerService {
	public Integer addCustomer(CustomerDTO customerDTO) throws CgBankException;
	public CustomerDTO getCustomer(Integer customerId) throws CgBankException;
	public void updateCustomer(Integer customerId, String emailId)throws CgBankException;
	public void deleteCustomer(Integer customerId)throws CgBankException;
	public List<CustomerDTO> getAllCustomers() throws CgBankException;

}
```

Step 9: Create the following CustomerServiceImpl class in com.cg.service package :

```java
@Service(value = "customerService")
@Transactional
public class CustomerServiceImpl implements CustomerService {
	@Autowired
	private CustomerRepository customerRepository;
	@Override
	public CustomerDTO getCustomer(Integer customerId) throws CgBankException {
		Optional<Customer> optional = customerRepository.findById(customerId);
		Customer customer = optional.orElseThrow(() -> new CgBankException("Service.CUSTOMER_NOT_FOUND"));
		CustomerDTO customer2 = new CustomerDTO();
		customer2.setCustomerId(customer.getCustomerId());
		customer2.setDateOfBirth(customer.getDateOfBirth());
		customer2.setEmailId(customer.getEmailId());
		customer2.setName(customer.getName());
		return customer2;
	}
	@Override
```

```java
    public Integer addCustomer(CustomerDTO customerDTO) throws CgBankException {
        Customer customerEntity = new Customer();
        customerEntity.setDateOfBirth(customerDTO.getDateOfBirth());
        customerEntity.setEmailId(customerDTO.getEmailId());
        customerEntity.setName(customerDTO.getName());
        customerEntity.setCustomerId(customerDTO.getCustomerId());
        Customer customerEntity2 = customerRepository.save(customerEntity);
        return customerEntity2.getCustomerId();
    }
    @Override
    public void updateCustomer(Integer customerId, String emailId) throws CgBankException {
        Optional<Customer> customer = customerRepository.findById(customerId);
        Customer c = customer.orElseThrow(() -> new CgBankException("Service.CUSTOMER_NOT_FOUND"));
        c.setEmailId(emailId);
    }
    @Override
    public void deleteCustomer(Integer customerId) throws CgBankException {
        Optional<Customer> customer = customerRepository.findById(customerId);
        customer.orElseThrow(() -> new CgBankException("Service.CUSTOMER_NOT_FOUND"));
        customerRepository.deleteById(customerId);
    }
    @Override
    public List<CustomerDTO> getAllCustomers() throws CgBankException {
        Iterable<Customer> customers = customerRepository.findAll();
        List<CustomerDTO> customerDTOs = new ArrayList<>();
        customers.forEach(customer -> {
            CustomerDTO cust = new CustomerDTO();
            cust.setCustomerId(customer.getCustomerId());
            cust.setDateOfBirth(customer.getDateOfBirth());
            cust.setEmailId(customer.getEmailId());
            cust.setName(customer.getName());
            customerDTOs.add(cust);
        });
        if (customerDTOs.isEmpty())
            throw new CgBankException("Service.CUSTOMERS_NOT_FOUND");
        return customerDTOs;
    }


}
```

Step 10: Create an aspect LoggingAspect class in com.cg.utility package:

```java
@Component
@Aspect
public class LoggingAspect {
    public static final Log LOGGER = LogFactory.getLog(LoggingAspect.class);
    @AfterThrowing(pointcut = "execution(* com.cg.service.*Impl.*(..))", throwing = "exception")
    public void logServiceException(Exception exception) {
        LOGGER.error(exception.getMessage(), exception);
    }


}
```

Step 11: Create the following CgBankException class in com.cg.exception package :

```java
public class CgBankException extends Exception {

	private static final long serialVersionUID = 1L;

	public CgBankException(String message) {
		super(message);
	}
}

```

Step 12: Create the following ErrorInfo class in com.cg.utility package :

```java
public class ErrorInfo {
	private String errorMessage;
	private Integer errorCode;
	private LocalDateTime timestamp;

	public String getErrorMessage() {
		return errorMessage;
	}

	public void setErrorMessage(String errorMessage) {
		this.errorMessage = errorMessage;
	}

	public Integer getErrorCode() {
		return errorCode;
	}

	public void setErrorCode(Integer errorCode) {
		this.errorCode = errorCode;
	}

	public LocalDateTime getTimestamp() {
		return timestamp;
	}

	public void setTimestamp(LocalDateTime timestamp) {
		this.timestamp = timestamp;
	}

}

```

Step 13: Create the following ExceptionControllerAdvice class in com.cg.utility package :

```java
@RestControllerAdvice
public class ExceptionControllerAdvice {
	@Autowired
	Environment environment;

	@ExceptionHandler(Exception.class)
	public ResponseEntity<ErrorInfo> exceptionHandler(Exception exception) {
		ErrorInfo error = new ErrorInfo();

	error.setErrorMessage(environment.getProperty("General.EXCEPTION_MESSAGE"))
;
		error.setErrorCode(HttpStatus.INTERNAL_SERVER_ERROR.value());
		error.setTimestamp(LocalDateTime.now());
		return new ResponseEntity<ErrorInfo>(error,
HttpStatus.INTERNAL_SERVER_ERROR);
	}

	@ExceptionHandler(CgBankException.class)
	public ResponseEntity<ErrorInfo> cgBankexceptionHandler(CgBankException
exception) {
```

```java
          ErrorInfo error = new ErrorInfo();

          error.setErrorMessage(environment.getProperty(exception.getMessage()));
          error.setTimestamp(LocalDateTime.now());
          error.setErrorCode(HttpStatus.NOT_FOUND.value());
          return new ResponseEntity<ErrorInfo>(error, HttpStatus.NOT_FOUND);
  }
  @ExceptionHandler(MethodArgumentNotValidException.class)
  public ResponseEntity<ErrorInfo> exceptionHandler(MethodArgumentNotValidException exception) {

          ErrorInfo errorInfo = new ErrorInfo();
          errorInfo.setErrorCode(HttpStatus.BAD_REQUEST.value());


          String errorMsg = exception.getBindingResult().getAllErrors().stream().map(x -> x.getDefaultMessage())
                                 .collect(Collectors.joining(", "));

          errorInfo.setErrorMessage(errorMsg);
          errorInfo.setTimestamp(LocalDateTime.now());
          return new ResponseEntity<>(errorInfo, HttpStatus.BAD_REQUEST);
  }



          @ExceptionHandler(ConstraintViolationException.class)
          public ResponseEntity<ErrorInfo> pathExceptionHandler(ConstraintViolationException exception) {

                   ErrorInfo errorInfo = new ErrorInfo();
                   errorInfo.setErrorCode(HttpStatus.BAD_REQUEST.value());

                   String errorMsg = exception.getConstraintViolations().stream().map(x -> x.getMessage())
                                          .collect(Collectors.joining(", "));
                   errorInfo.setErrorMessage(errorMsg);
                   errorInfo.setTimestamp(LocalDateTime.now());
                   return new ResponseEntity<>(errorInfo, HttpStatus.BAD_REQUEST);
          }

}
```

Step 14: Create the following controller class in com.cg.api package:

```java
@RestController
@RequestMapping(value = "/cgbank")
@Validated
public class CustomerAPI {

  @Autowired
  private CustomerService customerService;

  @Autowired
  private Environment environment;

  @GetMapping(value = "/customers")
  public ResponseEntity<List<CustomerDTO>> getAllCustomers() throws CgBankException {
      List<CustomerDTO> customerList = customerService.getAllCustomers();
      return new ResponseEntity<>(customerList, HttpStatus.OK);
  }

  @GetMapping(value = "/customers/{customerId}")
  public ResponseEntity<CustomerDTO> getCustomerDetails(@PathVariable @Min(value = 1, message = "{customer.customerid.invalid}") @Max(value = 100, message = "{customer.customerid.invalid}") Integer customerId)  throws CgBankException{
  //public ResponseEntity<Customer> getCustomerDetails(@PathVariable Integer customerId) throws Exception {
      CustomerDTO customer = customerService.getCustomer(customerId);
      return new ResponseEntity<>(customer, HttpStatus.OK);
  }

  @PostMapping(value = "/customers")
  public ResponseEntity<String> addCustomer(@Valid @RequestBody CustomerDTO customer) throws CgBankException {
      Integer customerId = customerService.addCustomer(customer);
      String successMessage = environment.getProperty("API.INSERT_SUCCESS") + customerId;
      return new ResponseEntity<>(successMessage, HttpStatus.CREATED);
  }

  @PutMapping(value = "/customers/{customerId}")
  public ResponseEntity<String> updateCustomer(@PathVariable Integer customerId, @RequestBody CustomerDTO customer)
                   throws CgBankException {
      customerService.updateCustomer(customerId, customer.getEmailId());
      String successMessage = environment.getProperty("API.UPDATE_SUCCESS");
      return new ResponseEntity<>(successMessage, HttpStatus.OK);
  }

  @DeleteMapping(value = "/customers/{customerId}")
  public ResponseEntity<String> deleteCustomer(@PathVariable Integer customerId) throws CgBankException {
      customerService.deleteCustomer(customerId);
      String successMessage = environment.getProperty("API.DELETE_SUCCESS");
      return new ResponseEntity<>(successMessage, HttpStatus.OK);
  }


}
```

Step 15: Add the following properties in application.properties :

```
Service.CUSTOMER_NOT_FOUND=No customer found with given customer id.
Service.CUSTOMERS_NOT_FOUND=No customers found.
General.EXCEPTION_MESSAGE=Request could not be processed due to some issue. Please try again!

API.INSERT_SUCCESS=Customer added successfully with customer id :
API.UPDATE_SUCCESS=Customer emailid successfully updated.
API.DELETE_SUCCESS=Customer details deleted successfully.

server.port=8765


```

Step 16: Create ValidationMessages.properties file in src/main/resources folder :

```
customer.emailid.absent=Please provide email address
customer.emailid.invalid=Please provide valid email address
customer.name.absent=Please provide customer name
customer.name.invalid=Name should contain only alphabets and space
customer.dob.invalid=Date of birth should be past or present date

customer.customerid.invalid=Customer id should be between 1 and 100
```

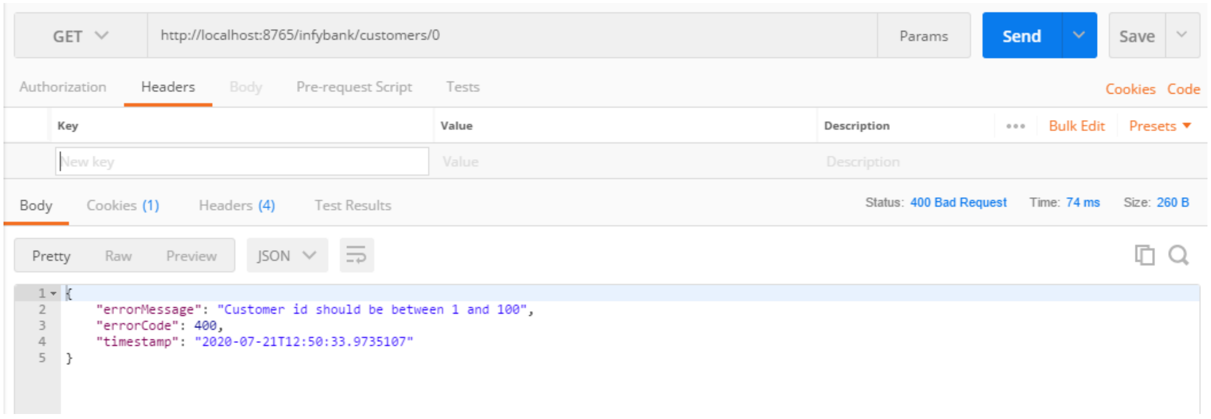
Step 17: Deploy the application on the server by executing the class containing the main method.

We have successfully implemented data valdiation of path variable. Now, you can test it using postman client.

Testing Web Service using Postman

Step 1: Launch Postman.

Step 2: Provide HTTP GET request - http://localhost:8765/cgbank/customers/0 using by passing customerId.



```
{
    "errorMessage": "Customer id should be between 1 and 100",
    "errorCode": 400,
    "timestamp": "2020-07-21T12:50:33.9735107"
}
```

When we pass 0 as customerId, which is invalid we get an error with its message stating the same. Thus we have successfully implemented data validation of path variable.

Best Practices in Spring REST

Some of the best practices that should be followed while developing Web API Layer are as follows:

1. Using a Consumer First Approach

When we are designing a RESTful Web Service, it is a best practice to consider a "Consumer First" approach. The idea behind this approach is to keep the API design simple enough for the consumer to be:

- able to understand the APIs
- able to locate and access the exposed resources of the APIs
- able to understand the URIs

2. Avoid Implementing Business Logic in API classes

The Web API classes in a Spring Boot-based application are designed with only one specific task: connecting the front-end of the application to the back-end of the application and delegating the further tasks to the Service Layer classes. Hence, it is a best practice to not couple the Web API Layer and Service Layer with implementing business logic in the API classes.

3. Using Nouns for Resource Identification

For creating easy to understand Web API methods it is a best practice to name the resources with nouns. For example:

- For fetching a lists of trainees (GET) – "/trainees"
- For fetching a specific trainee (GET) – "/trainee/100"

The idea behind this practice is, if a URL is compared to a sentence, then the resources are the nouns and the HTTP methods are the verbs. Hence, naming the resources as verbs should be avoided. For example:

- For fetching a lists of items (GET) - "/getAllItems"
- For fetching a specific item by item id (GET) - "/getItemById/100"

4. Avoid using GET methods that Alter the State

Let us consider a situation where a GET method and its query parameters alter states. Designing such a GET method is a design flaw. This is because, a GET method by default is used for fetching. Hence, instead of using GET methods for such operations using PUT, POST and DELETE must be considered.

For example: A GET method which modifies the state of the fetched record - "/items/343?modifying" or "/items/343/modifying"

5. Using Searching, Sorting, Filtering and Pagination features in GET Methods

When we are fetching data from larger databases, we often obtain the data (more than required most of the times) in an unsorted and unfiltered order. Hence, using the features of Sorting, Searching, Filtering and Pagination in the GET methods is a good practice as we can obtain the optimized result from any GET method. For example:

- **Sorting:** This feature can be added to the GET method(s) to obtain the results in a required order. For eg. this GET method fetches list of results by ordering them in ascending order.

```
"/result?sort=rank_asc"
```

- **Filtering:** This feature can be added to the GET method(s) to filter the results according to required criteria(s). For eg. this GET method fetches list of employees by filtering them on the basis of technology and location of the employees.

```
"/employees?technology=ivs&location=india"
```

- **Searching:** This feature can be added to the GET method(s) to obtain specific results. For eg. this GET method fetches list of technologies by using "java" as search keyword.

```
“/technology?search=java”
```

- **Pagination:** This feature can be added to the GET method(s) to obtain a certain section of results by paginating the complete fetched results. For eg. this GET method fetches list of employees from page 12 of the complete employees list.

```
“/employees?page=10”
```

6. Correct usage of HTTP methods

Whenever we want to implement an operation using RESTful web services, the resource has to be identified appropriately on which operation is to be performed. Then the relevant HTTP method has to be selected.

Accordingly, use GET for retrieval, POST for creation, PUT for updating and DELETE for deletion.

7. Appropriate response status and exception handling

It is also equally important to return the appropriate status while sending back the response for the request send by the consumer.

For example, when a POST method is successful, status returned should be CREATED and not OK.

```
return new ResponseEntity<>(successMessage, HttpStatus.CREATED);
```

This is also applicable when an exception occurs. It should be appropriately handled and the correct response should be sent back.

Consider a case if we send a GET request for fetching a non-existing customer using the URI **http://localhost:8765/cgbank/customers/25**

```
throw new ResponseStatusException(HttpStatus.NOT_FOUND,

environment.getProperty(exception.getMessage()), exception);
```

The above code for handling the exception is a good practice as it returns the correct status NOT_FOUND when a resource is not found, rather than sending the status INTERNAL_SERVER_ERROR.

Also, as good practice, provide appropriate error data like the date and time of exception and message along with the response status. As you have already seen, custom error class ErrorInfo can be used for setting this error data. Thus, a standardized and readable response can be maintained