Spring Advanced

Object Mapping and Validation

think.
create.
accelerate.

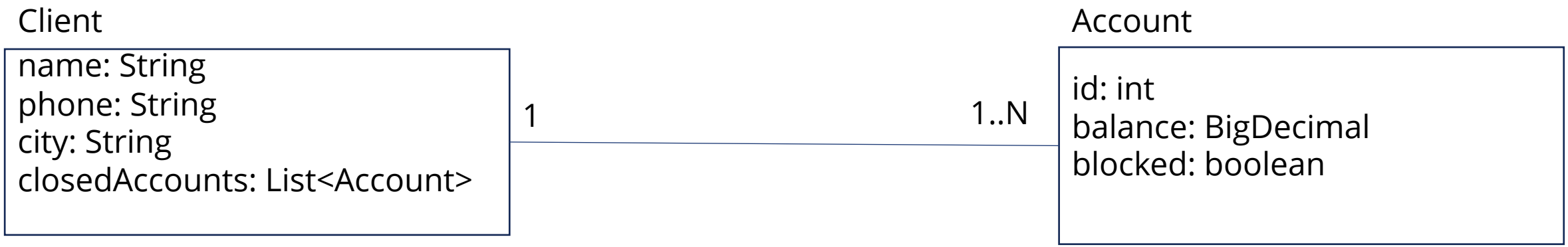
Luxoft
A DXC Technology Company

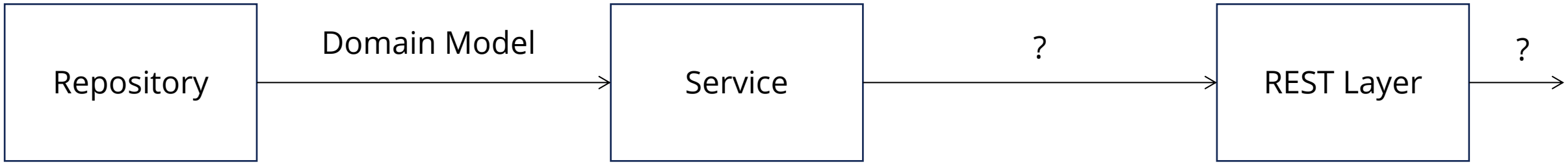
Limiting Data Passed to the Client

Multi-Layered Systems

- A well-layered system separates code that handles the user interface from code that handles the business logic.
- You may want several interfaces for similar business logic; the user interface becomes too complicated if it does both.
- Although the behavior can be easily separated, the data often cannot. Data needs to be embedded in GUI control that has the same meaning as data that lives in the domain model.

Typical Multi-Layered Systems





Reducing the Data Passed to the Client

1. Data Transfer Object (DTO)
2. Nullifying fields
3. Using Jackson annotations (@JsonIgnore, etc.)

1. Using DTO

Multi-Layered Systems Using DTO

Data Transfer Object (DTO)

```
ClientDTO getClientForm(id) {
    Client client = db.getClient(id);
    ClientDTO dto = new ClientDTO();
    dto.setName(client.getName());
    dto.setPhone(client.getPhone());
    dto.setCity(client.getCity());
    return dto;
}
```

2. Fields nullification

Fields Nullification

- Fields that are not shown on the GUI side may be made `null`. In our example, **`closedAccounts`** may be nullified.

```
@GetMapping
Client getClientForm(id) {
    Client client = db.getClient(id);
    client.closedAccounts = null;
    return client;
}
```

3. Using Jackson annotations

Working with Jackson

- Jackson - Java based library to serialize or map Java objects to JSON and vice versa.
- The `ObjectMapper` API provides a straightforward way to parse and generate JSON response objects.
- The method `writeValueAsString` generates a JSON from a Java object and returns the generated JSON as a string.

Using DTO

```
public class Client {

    private String name;
    private int expenses;
    private boolean vip;

    …

}
```

Testing that null Fields Are Ignored

```
public void testIgnoreNullFields()
            throws JsonProcessingException {
    ObjectMapper mapper = new ObjectMapper();
    mapper.setSerializationInclusion(Include.NON_NULL);
    Client entityObject = new Client();

    String entityAsString = mapper.writeValueAsString(entityObject);

    assertThat(entityAsString, containsString("expenses"));
    assertThat(entityAsString, containsString("vip"));
    assertThat(entityAsString, not(containsString("name")));
    System.out.println(entityAsString);
}
```

ClientIgnoreField

```
public class ClientIgnoreField {

    private String name;
    @JsonIgnore
    private int expenses;
    private boolean vip;

    …
}
```

Testing that null Fields Are Ignored

```
@Test
public void testFieldIgnoredWithJsonIgnore() throws IOException {
    ObjectMapper mapper = new ObjectMapper();
    ClientIgnoreField entityObject = new ClientIgnoreField();

    String entityAsString = mapper.writeValueAsString(entityObject);

    assertThat(entityAsString, not(containsString("expenses")));
    assertThat(entityAsString, containsString("name"));
    assertThat(entityAsString, containsString("vip"));
    System.out.println(entityAsString);
}
```

ClientDtoIgnoreFieldByName

```
@JsonIgnoreProperties(value = { "expenses" })
public class ClientIgnoreFieldByName {

    private String name;
    private int expenses;
    private boolean vip;

    …

}
```

Testing the Field Is Ignored by Name

```
@Test
public void testFieldIgnoredByNameWithJsonIgnoreProperties()
        throws IOException {
    ObjectMapper mapper = new ObjectMapper();
    ClientIgnoreFieldByName entityObject =
                    new ClientIgnoreFieldByName();
 entityObject.setVip(true);

    String entityAsString = mapper.writeValueAsString(entityObject);

    assertThat(entityAsString, not(containsString("expenses")));
    assertThat(entityAsString, containsString("name"));
    assertThat(entityAsString, containsString("vip"));
    System.out.println(entityAsString);
}
```

ClientDtoIgnoreNull

The @JsonInclude(Include.NON_NULL) annotation tells to include only the non null values of the fields.

```
@JsonInclude(Include.NON_NULL)
public class ClientDtoIgnoreNull {

    private String name;
    private int expenses;
    private boolean vip;

    …

}
```

Testing Including only Non-Null Fields

```
@Test
public void testIncludeNonNull() throws JsonProcessingException {
    ObjectMapper mapper = new ObjectMapper();
    ClientDtoIgnoreNull dtoObject = new ClientDtoIgnoreNull();

    String dtoAsString = mapper.writeValueAsString(dtoObject);

    assertThat(dtoAsString, containsString("expenses"));
    assertThat(dtoAsString, containsString("vip"));
    assertThat(dtoAsString, not(containsString("name")));
    System.out.println(dtoAsString);
}
```

ClientDtoIncludeNonDefault

The @JsonInclude(Include.NON_DEFAULT) annotation tells to include only the non-default values. The default values are: false for boolean type, 0 for numeric types, null for reference types.

```
@JsonInclude(Include.NON_DEFAULT)
public class ClientDtoIncludeNonDefault {

    private String name;
    private int expenses;
    private boolean vip;

    …

}
```

Testing Including only Non-Default Fields

```
@Test
public void testIncludeNonDefault() throws IOException {
    ObjectMapper mapper = new ObjectMapper();
    ClientDtoIncludeNonDefault dtoObject =
            new ClientDtoIncludeNonDefault();
    dtoObject.setVip(true);

    String dtoAsString = mapper.writeValueAsString(dtoObject);

    assertThat(dtoAsString, not(containsString("expenses")));
    assertThat(dtoAsString, not(containsString("name")));
    assertThat(dtoAsString, containsString("vip"));
    System.out.println(dtoAsString);
}
```

Projections and Excerpts

Introducing Projections and Excerpts

- Spring Data REST presents a default view of the domain model you export.
- Sometimes you may need to alter the view of that model for various reasons.
- Projections and excerpts serve as simplified and reduced views of resources.

The Customer Class

```
@Entity
public class Customer {

    @GeneratedValue
    @Id
    private Long id;
    private String firstname, lastname;
    private Gender gender;

    @OneToOne(cascade = CascadeType.ALL, orphanRemoval = true)
    private Address address;

    …
}
```

The Address class

```
@Entity
public class Address {

    @GeneratedValue
    @Id
    private Long id;
    private String street, zipCode, city, state;

    …
}
```

Exporting the Domain Object

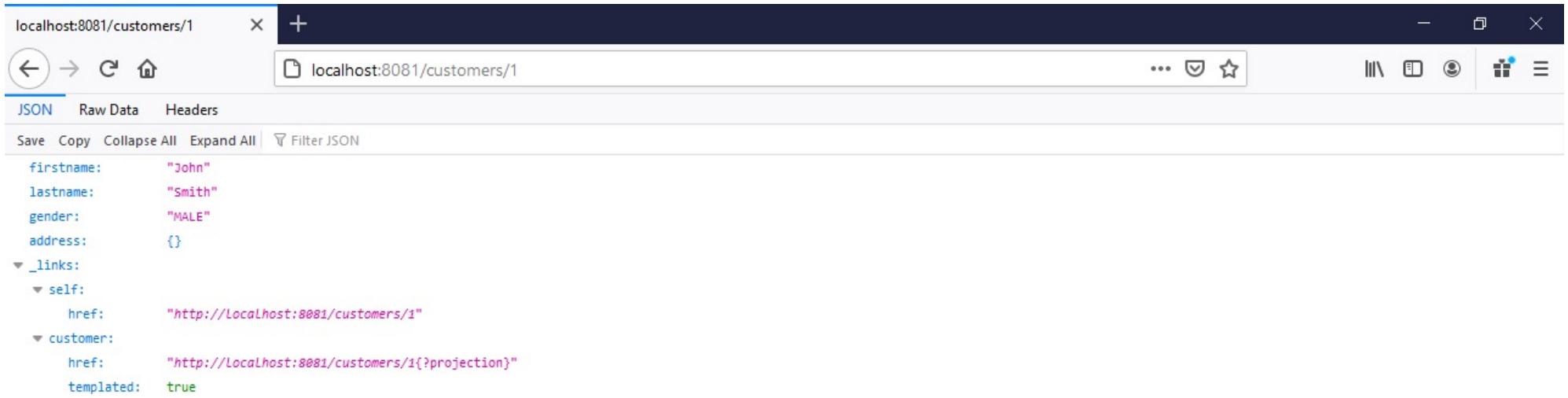
- By default, Spring Data REST exports this domain object, including all of its attributes:
  - firstname
  - lastname
  - gender
  - address

The CustomerProjection Interface

```
@Projection(name = "myprojection", types = Customer.class)
public interface CustomerProjection {

   String getFirstname();

   String getLastname();

   @Value("#{target.address.toString()}")
   String getAddress();
}
```

Accessing the Usual Interface

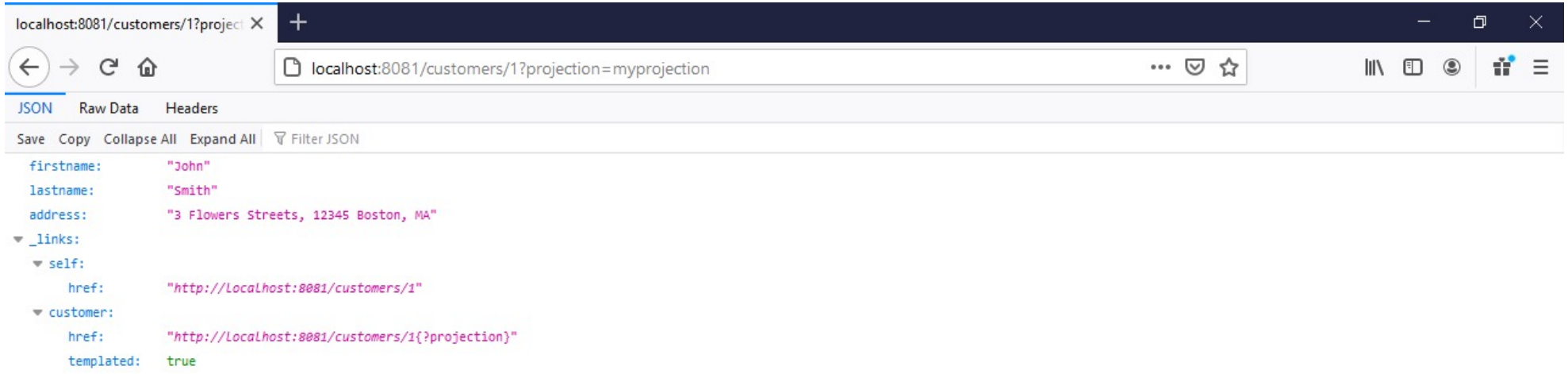
http://localhost:8081/customers/1



.rest file: resources/rest/commands.rest

Accessing the Projected Interface

http://localhost:8081/customers/1?**projection=myprojection**



.rest file: resources/rest/commands.rest

Working with Excerpts

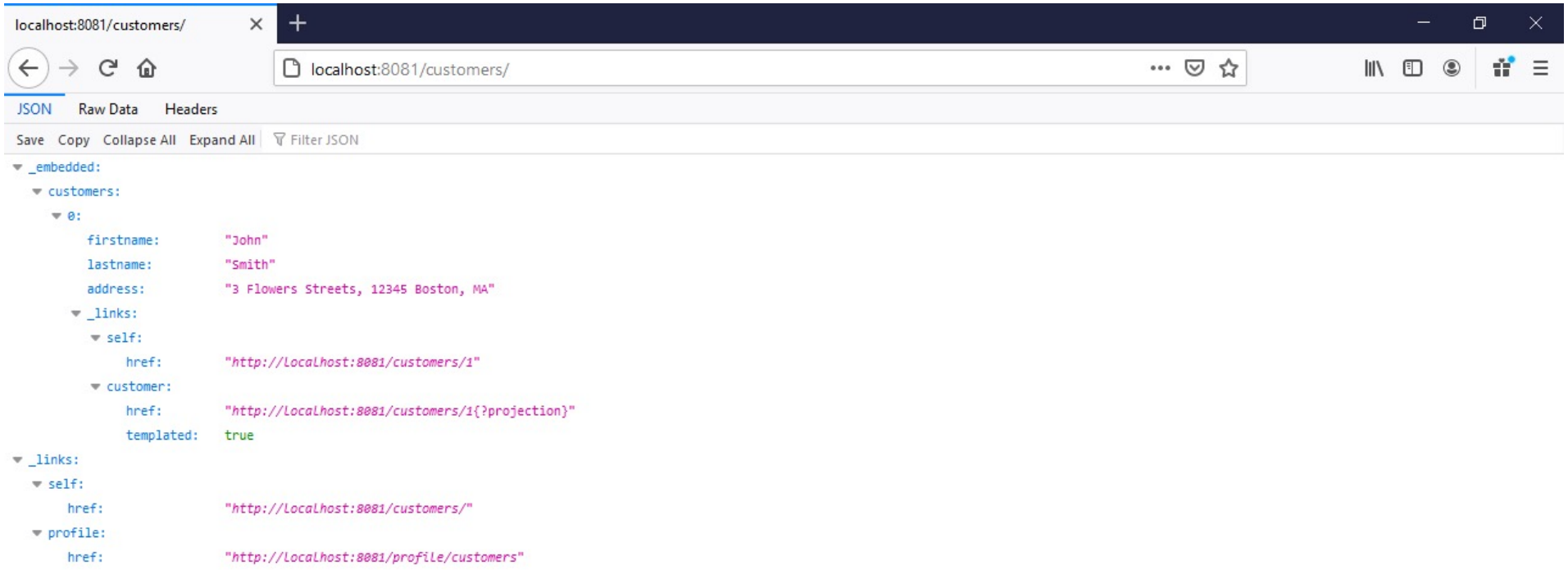
- Excerpts are also projections
- They are applied as default views to resource collections
- `@RepositoryRestResource` is used to change the default view for the whole resource collection
- Accessing `localhost:8081/customers` will not include the gender field

Working with Excerpts

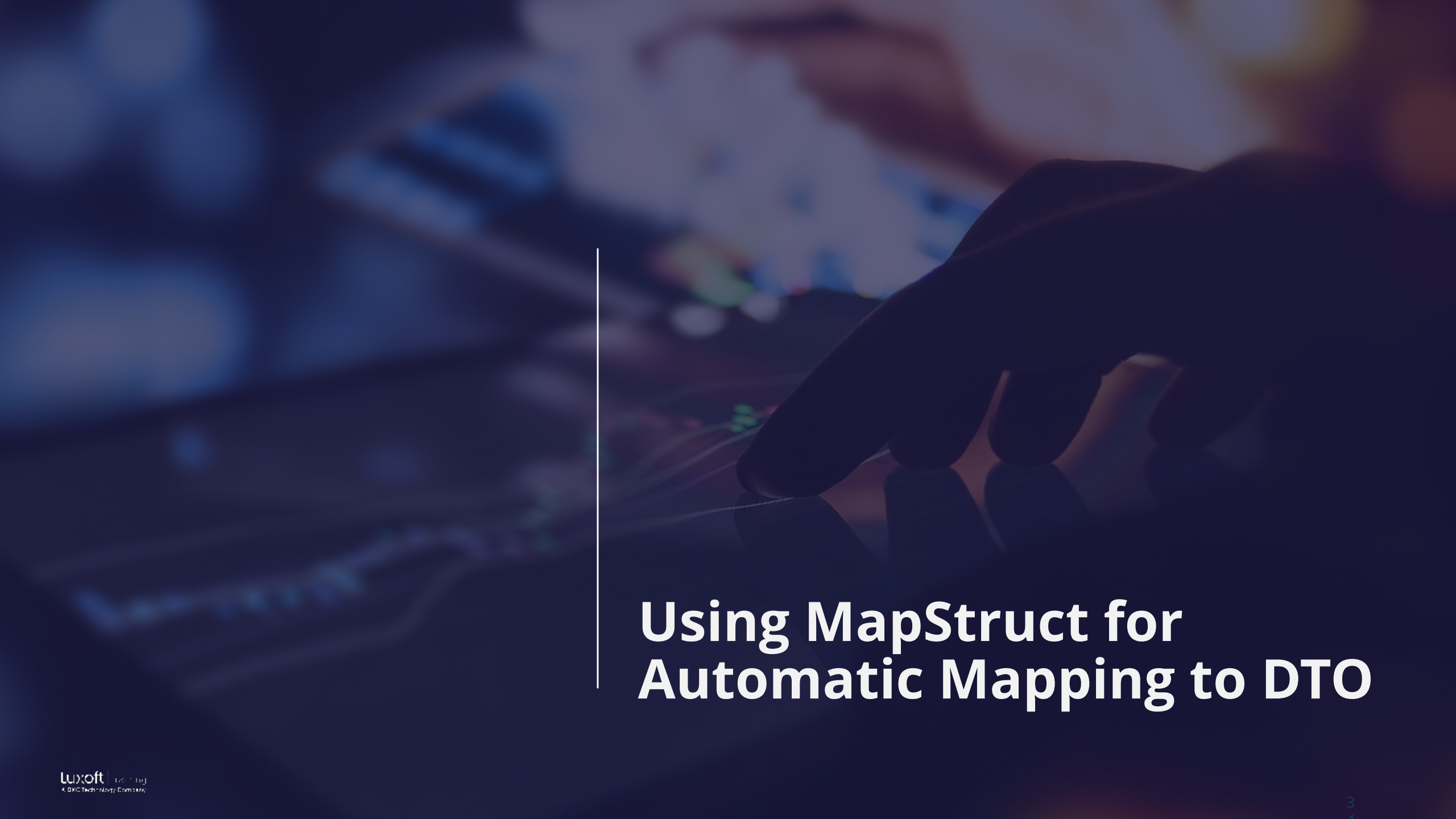
```
@RepositoryRestResource(excerptProjection =
                        CustomerProjection.class)
public interface CustomerRepository extends
                                    CrudRepository<Customer, Long> {

}
```

Accessing the Collection Resource

localhost:8081/customers



.rest file: resources/rest/commands.rest

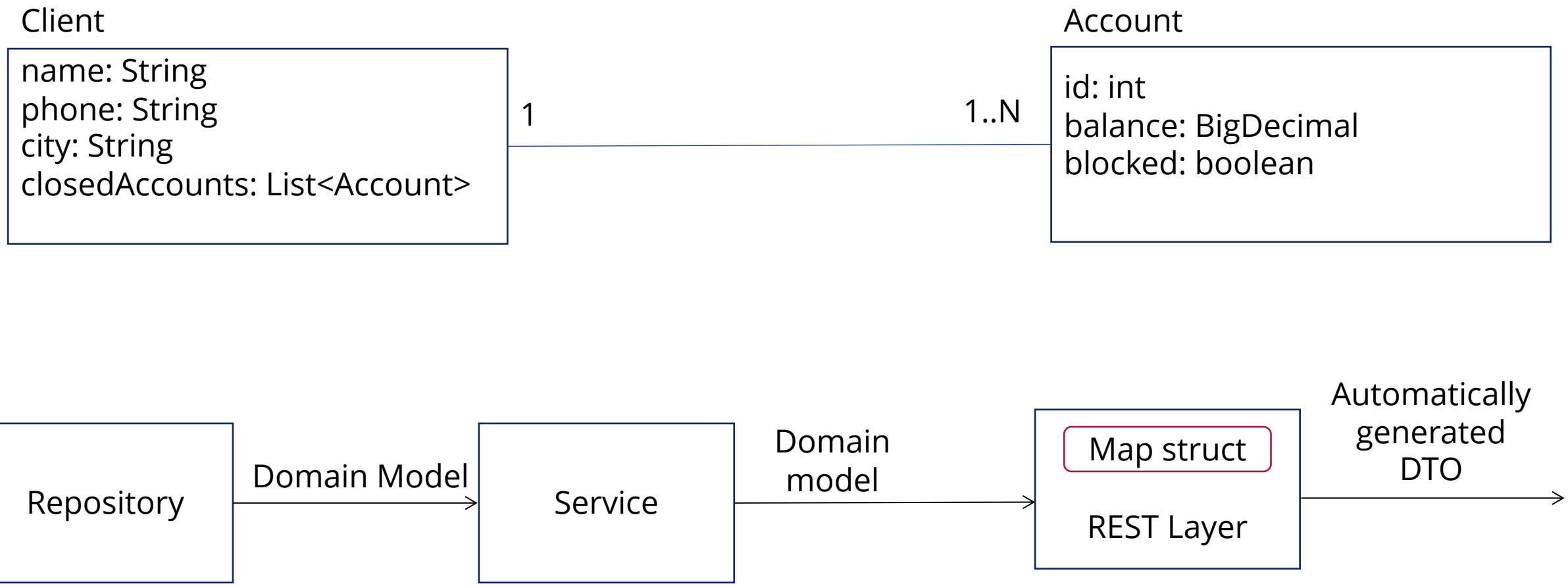


Using MapStruct for Automatic Mapping to DTO

Introduction to MapStruct

- The `MapStruct` API contains functions that automatically maps between two Java Beans.
- We only need to create the interface.
- The library will automatically create a concrete implementation at compile time.

Multi-Layered System Using MapStruct



Introduction to MapStruct

- A lot of boilerplate code may convert POJOs to other POJOs.
- A common type of conversion happens between persistence-backed entities and DTOs.
- MapStruct generates bean mapper classes automatically.

MapStruct Dependency

```
<dependency>
    <groupId>org.mapstruct</groupId>
    <artifactId>mapstruct</artifactId>
    <version>1.4.0.Beta1</version>
</dependency>
```

Using 1.4.0.Beta1 as it has only introduced the @Mapper annotation.

The MapStruct Processor

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-compiler-plugin</artifactId>
   <version>3.8.1</version>
   <configuration>
      <source>11</source>
      <target>11</target>
      <annotationProcessorPaths>
         <path>
            <groupId>org.mapstruct</groupId>
            <artifactId>mapstruct-processor</artifactId>
            <version>1.4.0.Beta1</version>
         </path>
      </annotationProcessorPaths>
   </configuration>
</plugin>
```

mapstruct-processor can be used from the Maven or Gradle configuration by the executor (including Intellij IDEA) to generate the mapper implementation during the build.

Creating POJOs

```
public class SimpleSource {

    private String name;
    private String description;
    …
}

public class SimpleDestination {

    private String name;
    private String description;
    …
}
```

The Mapper Interface

```
@Mapper(componentModel = "spring")
public interface SimpleSourceDestinationMapper {

    SimpleDestination sourceToDestination(SimpleSource source);

    SimpleSource destinationToSource(
                 SimpleDestination destination);

}
```

The Generated Map Implementation

```
@Generated(
    value = "org.mapstruct.ap.MappingProcessor",
    date = "2020-10-28T14:49:17+0200",
    comments = "version: 1.4.0.Beta1, compiler: javac, environment: Java 11.0.7 (Oracle
Corporation)"
)
@Component
public class SimpleSourceDestinationMapperImpl implements SimpleSourceDestinationMapper {

    @Override
    public SimpleDestination sourceToDestination(SimpleSource source) {
        if ( source == null ) {
            return null;
        }

        SimpleDestination simpleDestination = new SimpleDestination();

        simpleDestination.setName( source.getName() );
        simpleDestination.setDescription( source.getDescription() );

        return simpleDestination;
    }
```

The Generated Map Implementation

```
    @Override
    public SimpleSource destinationToSource(SimpleDestination destination) {
        if ( destination == null ) {
            return null;
        }

        SimpleSource simpleSource = new SimpleSource();

        simpleSource.setName( destination.getName() );
        simpleSource.setDescription( destination.getDescription() );

        return simpleSource;
    }
}
```

target\generated-sources\annotations\com\luxoft\springadvanced\mapper\SimpleSourceDestinationMapperImpl.java

Testing the Mapper Interface

```
SimpleSourceDestinationMapper simpleSourceDestinationMapper =
                Mappers.getMapper(SimpleSourceDestinationMapper.class);

@Test
@DisplayName("Given a SimpleSource with String fields, when we map it to a SimpleDestination
              with the same fields names, then the fields are correctly mapped")
public void testMapSourceToDestination() {
    SimpleSource simpleSource = new SimpleSource();
    simpleSource.setName("SourceName");
    simpleSource.setDescription("SourceDescription");

    SimpleDestination destination =
                    simpleSourceDestinationMapper.sourceToDestination(simpleSource);

    assertEquals(simpleSource.getName(), destination.getName());
    assertEquals(simpleSource.getDescription(), destination.getDescription());
}
```

Testing the Mapper Interface

```
@Test
@DisplayName("Given a SimpleDestination with String fields, when we map it to a SimpleSource
              with the same fields names, then the fields are correctly mapped")
public void testMapDestinationToSource() {
    SimpleDestination destination = new SimpleDestination();
    destination.setName("DestinationName");
    destination.setDescription("DestinationDescription");

    SimpleSource source = simpleSourceDestinationMapper.destinationToSource(destination);

    assertEquals(destination.getName(), source.getName());
    assertEquals(destination.getDescription(), source.getDescription());
}
```

Injecting the Mapper Interface

- `MapStruct` has support for Spring Dependency Injection.

- To use Spring IoC, we add the `componentModel attribute` to `@Mapper` with the value `"spring"`.

- Other possible `componentModel` option is `“cdi”`.

Injecting the Mapper Interface

```
@Mapper(componentModel = "spring")
public interface SimpleSourceDestinationMapper {

    SimpleDestination sourceToDestination(SimpleSource source);

    SimpleSource destinationToSource(SimpleDestination destination);

}

@ExtendWith(SpringExtension.class)
@ContextConfiguration("classpath:application-context.xml")
public class SimpleSourceDestinationMapperIntegrationTest {

    @Autowired
    SimpleSourceDestinationMapper simpleSourceDestinationMapper;
```

Mapping Fields with Different Names

```
public class Employee {

    private int id;
    private String name;
    …
}

public class EmployeeDTO {

    private int employeeId;
    private String employeeName;
  …
}
```

The EmployeeMapper Interface

```
@Mapper
public interface EmployeeMapper {

    @Mappings({ @Mapping(target = "employeeId", source = "entity.id"),
      @Mapping(target = "employeeName", source = "entity.name")})
     EmployeeDTO employeeToEmployeeDTO(Employee entity);

    @Mappings({ @Mapping(target = "id", source = "dto.employeeId"),
      @Mapping(target = "name", source = "dto.employeeName")})
     Employee employeeDTOtoEmployee(EmployeeDTO dto);

}
```

Testing the EmployeeMapper Interface

```
@Test
@DisplayName("Given an EmployeeDTO, when we map it to an Employee,
              then the fields with different names are correctly mapped")
public void testMappingEmployeeDTOToEmployeeWithDifferentFieldsNames() {
    EmployeeDTO dto = new EmployeeDTO();
    dto.setEmployeeId(1);
    dto.setEmployeeName("John");

    Employee entity = mapper.employeeDTOtoEmployee(dto);

    assertEquals(dto.getEmployeeId(), entity.getId());
    assertEquals(dto.getEmployeeName(), entity.getName());
}
```

Mapping Beans with Child Beans

```
public class Employee {
    private int id;
    private String name;
    private Division division;
    …
}

public class EmployeeDTO {
    private int employeeId;
    private String employeeName;
    private DivisionDTO division;
    …
}
```

The Division and DivisionDTO Classes

```
public class Division {

    private int id;
    private String name;
    …
}

public class DivisionDTO {

    private int id;
    private String name;
    …
}
```

The Modified EmployeeMapper Interface

```
@Mapper
public interface EmployeeMapper {

    @Mappings({ @Mapping(target = "employeeId", source = "entity.id"),
                @Mapping(target = "employeeName", source = "entity.name")})
    EmployeeDTO employeeToEmployeeDTO(Employee entity);

    @Mappings({ @Mapping(target = "id", source = "dto.employeeId"),
                @Mapping(target = "name", source = "dto.employeeName")})
    Employee employeeDTOtoEmployee(EmployeeDTO dto);

    DivisionDTO divisionToDivisionDTO(Division entity);

    Division divisionDTOtoDivision(DivisionDTO dto);


}
```

Adding New Tests to the EmployeeMapper Interface

```
@Test
@DisplayName("Given an EmployeeDTO with a nested object, when we map it
              to an Employee, then the fields are correctly mapped")
public void testEmployeeDTOWithNestedObjectToEmployee() {
    EmployeeDTO dto = new EmployeeDTO();
    dto.setDivision(new DivisionDTO(1, "Division1"));

    Employee entity = mapper.employeeDTOtoEmployee(dto);

    assertEquals(dto.getDivision().getId(), entity.getDivision().getId());
    assertEquals(dto.getDivision().getName(), entity.getDivision().getName());
}
```

Adding New Tests to the EmployeeMapper Interface

```
@Test
@DisplayName("Given an Employee with a nested object, when we map it
              to an EmployeeDTO, then the fields are correctly mapped")
public void testEmployeeWithNestedObjectToEmployeeDTO() {
    Employee entity = new Employee();
    entity.setDivision(new Division(1, "Division1"));

    EmployeeDTO dto = mapper.employeeToEmployeeDTO(entity);

    assertEquals(dto.getDivision().getId(), entity.getDivision().getId());
    assertEquals(dto.getDivision().getName(), entity.getDivision().getName());
}
```

Mapping Collections of Beans

```
@Mapper
public interface EmployeeMapper {

    @Mappings({ @Mapping(target = "employeeId", source = "entity.id"),
                @Mapping(target = "employeeName", source = "entity.name")})
    EmployeeDTO employeeToEmployeeDTO(Employee entity);

    @Mappings({ @Mapping(target = "id", source = "dto.employeeId"),
                @Mapping(target = "name", source = "dto.employeeName")})
    Employee employeeDTOtoEmployee(EmployeeDTO dto);

    DivisionDTO divisionToDivisionDTO(Division entity);

    Division divisionDTOtoDivision(DivisionDTO dto);

    List<Employee> convertEmployeeDTOListToEmployeeList(List<EmployeeDTO> list);

    List<EmployeeDTO> convertEmployeeListToEmployeeDTOList(List<Employee> list);

}
```

Mapping Collections of Beans

- We added two new methods to the mapper interface:

```
List<Employee>
convertEmployeeDTOListToEmployeeList(List<EmployeeDTO> list);

List<EmployeeDTO>
convertEmployeeListToEmployeeDTOList(List<Employee> list);
```

Adding New Tests to the EmployeeMapper Interface

```
@Test
@DisplayName("Given a list of Employee, when we map it to a list of EmployeeDTO,
              then the fields with different names are correctly mapped")
public void testEmployeeListToEmployeeDTOList() {
    List<Employee> employeeList = new ArrayList<>();
    Employee emp = new Employee();
    emp.setId(1);
    emp.setName("EmpName");
    emp.setDivision(new Division(1, "Division1"));
    employeeList.add(emp);

    List<EmployeeDTO> employeeDtoList =
            mapper.convertEmployeeListToEmployeeDTOList(employeeList);
    EmployeeDTO employeeDTO = employeeDtoList.get(0);
    assertEquals(employeeDTO.getEmployeeId(), emp.getId());
    assertEquals(employeeDTO.getEmployeeName(), emp.getName());
    assertEquals(employeeDTO.getDivision().getId(), emp.getDivision().getId());
    assertEquals(employeeDTO.getDivision().getName(), emp.getDivision().getName());
}
```

Adding New Tests to the EmployeeMapper Interface

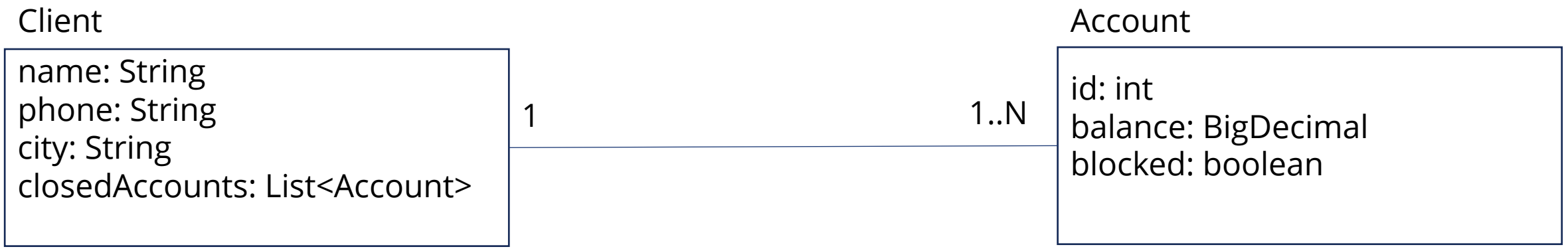
```
@Test
@DisplayName("Given a list of EmployeeDTO, when we map it to a list of Employee,
              then the fields with different names are correctly mapped")
public void testEmployeeDTOListToEmployeeList() {
    List<EmployeeDTO> employeeDTOList = new ArrayList<>();
    EmployeeDTO empDTO = new EmployeeDTO();
    empDTO.setEmployeeId(1);
    empDTO.setEmployeeName("EmpName");
    empDTO.setDivision(new DivisionDTO(1, "Division1"));
    employeeDTOList.add(empDTO);

    List<Employee> employeeList =
                  mapper.convertEmployeeDTOListToEmployeeList(employeeDTOList);
    Employee employee = employeeList.get(0);
    assertEquals(employee.getId(), empDTO.getEmployeeId());
    assertEquals(employee.getName(), empDTO.getEmployeeName());
    assertEquals(employee.getDivision().getId(), empDTO.getDivision().getId());
    assertEquals(employee.getDivision().getName(), empDTO.getDivision().getName());
}
```

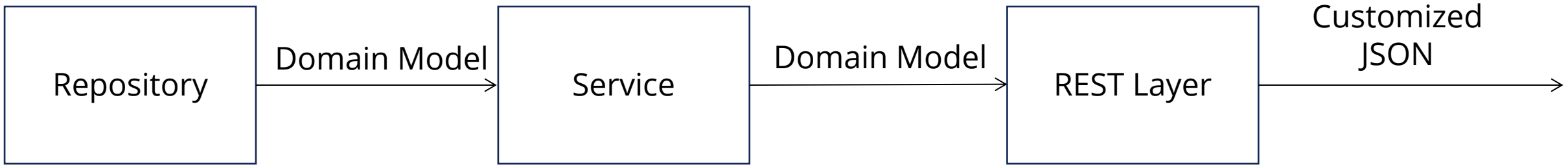
Adding New Tests to the EmployeeMapper Interface

```
@Test
@DisplayName("Given an Employee with a nested object, when we map it
              to an EmployeeDTO, then the fields are correctly mapped")
public void testEmployeeWithNestedObjectToEmployeeDTO() {
    Employee entity = new Employee();
    entity.setDivision(new Division(1, "Division1"));

    EmployeeDTO dto = mapper.employeeToEmployeeDTO(entity);

    assertEquals(dto.getDivision().getId(), entity.getDivision().getId());
    assertEquals(dto.getDivision().getName(), entity.getDivision().getName());
}
```

Custom Serializers and Deserializers

Multi-Layered System Using Serializer





The Item and User Classes

```
public class Item {
    private int id;
    private String itemName;
    private User owner;
    …
}
```

```
public class User {
    private int id;
    private String name;
    …
}
```

Serializing and Deserializing

```
@Test
@DisplayName("Given an Item, when it is serialized and deserialized,
    then the deserialized Item will be equal to the initial Item")
public void testSerializationDeserialization() throws IOException {
    Item myItem = new Item(1, "theItem", new User(2, "theUser"));
    String serialized =
        new ObjectMapper().writeValueAsString(myItem);
    System.out.println(serialized);
    Item readValue =
        new ObjectMapper().readValue(serialized, Item.class);
    assertEquals(myItem, readValue);
}
```

Defining a Custom Serializer

```
public class ItemSerializer extends StdSerializer<Item> {
    …
    @Override
    public void serialize(Item value, JsonGenerator jgen,
                SerializerProvider provider) throws IOException {
        jgen.writeStartObject();
        jgen.writeNumberField("id", value.getId());
        jgen.writeStringField("itemName", value.getItemName());
        jgen.writeNumberField("owner", value.getOwner().getId());
        jgen.writeStringField("name", value.getOwner().getName());
        jgen.writeEndObject();
    }
    …
}
```

Defining a Custom Deserializer

```
public class ItemDeserializer extends StdDeserializer<Item> {

    …

    @Override
    public Item deserialize(JsonParser jp, DeserializationContext ctxt)
                                throws IOException {
        JsonNode node = jp.getCodec().readTree(jp);
        int id = (Integer) (node.get("id")).numberValue();
        String itemName = node.get("itemName").asText();
        int userId = (Integer) (node.get("owner")).numberValue();
        String userName = node.get("name").asText();

        return new Item(id, itemName, new User(userId, userName));
    }
```

Custom Serializing and Deserializing

```
@Test
@DisplayName("Given an Item, when it is custom serialized and
              deserialized, then the deserialized Item will be equal
              to the initial Item")
public void testCustomSerialization() throws IOException {
    Item myItem = new Item(1, "theItem", new User(2, "theUser"));

    ObjectMapper mapper = new ObjectMapper();

    SimpleModule simpleModule = new SimpleModule();
    simpleModule.addSerializer(Item.class, new ItemSerializer());
    simpleModule.addDeserializer(Item.class, new ItemDeserializer());
    mapper.registerModule(simpleModule);
```

Custom Serializing and Deserializing

```
    String serialized = mapper.writeValueAsString(myItem);
    System.out.println(serialized);
    Item readValue = mapper.readValue(serialized, Item.class);
    assertEquals(myItem, readValue);

}
```

Serializer registration on the bean

```
@JsonSerialize(using = ItemSerializer.class)
@JsonDeserialize(using = ItemDeserializer.class)
public class Item {
    private int id;
    private String itemName;
    private User owner;
   …
}
```

We also can define custom serializer/deserializer for the field:

```
public class ItemBag {
    @JsonSerialize(using = ItemSerializer.class)
    private List<Item> items;
}
```

Bean Validation

The Java Bean Validation Framework

- Spring Boot provides support for bean validation.
- The Java Bean Validation framework **`javax.validation`** is the de facto standard
- Common annotations:
  - @NotNull
  - @NotEmpty
  - @Min
  - @Max
  - @DecimalMin
  - @DecimalMax

Enabling the Validation

- The bean validation will be enabled automatically if any validator implementation (e.g. Hibernate Validator) is available on the classpath.
- By default, Spring Boot will get and download the Hibernate Validator automatically.

Maven Dependencies

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

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```

The Person Class

```
@Entity
public class Person {

    @Id
    @GeneratedValue
    private Long id;

    @NotEmpty(message = "Please provide a name")
    private String name;

    @Country
    @NotEmpty(message = "Please provide a country")
    private String country;

    @NotNull(message = "Please provide a salary")
    @DecimalMin("1000.00")
    private BigDecimal salary;
```

The @Country Annotation

```
@Target({FIELD})
@Retention(RUNTIME)
@Constraint(validatedBy = CountryValidator.class)
@Documented
public @interface Country {

    String message() default "Country is not allowed.";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

}
```

The CountryValidator Class

```
public class CountryValidator implements
             ConstraintValidator<Country, String> {

    List<String> countries =
       Arrays.asList("Australia", "United Kingdom", "United States");

    @Override
    public boolean isValid(String value,
                           ConstraintValidatorContext context) {

        return countries.contains(value);

    }
}
```

The PersonController Class

```
@RestController
public class PersonController {

    @Autowired
    private PersonRepository repository;

    @PostMapping("/persons")
    @ResponseStatus(HttpStatus.CREATED)
    Person newPerson(@Valid @RequestBody Person newPerson) {
        return repository.save(newPerson);
    }
```

Error Handler for Request Body

- In POST and PUT requests, Spring automatically maps the incoming JSON to a Java object.
- We check if the incoming Java object meets our requirements.
- If the validation fails, it triggers a MethodArgumentNotValidException.
- We provide a handler for MethodArgumentNotValidException.

The CustomGlobalExceptionHandler Class

```
@ControllerAdvice
public class CustomGlobalExceptionHandler
        extends ResponseEntityExceptionHandler {

    // error handle for @Valid
    @Override
    protected ResponseEntity<Object> handleMethodArgumentNotValid(
       MethodArgumentNotValidException ex, HttpHeaders headers,
       HttpStatus status, WebRequest request) {

        Map<String, Object> body = new LinkedHashMap<>();
        body.put("timestamp", new Date());
        body.put("status", status.value());
```

The CustomGlobalExceptionHandler Class

```
        //Get all errors
        List<String> errors = ex.getBindingResult()
                .getFieldErrors()
                .stream()
                .map(x -> x.getDefaultMessage())
                .collect(Collectors.toList());

        body.put("errors", errors);

        return new ResponseEntity<>(body, headers, status);

    }
```

Error Handler for Request Parameters

- We annotate the method parameter in the Spring controller.
- If the validation fails, it triggers a `ConstraintViolationException`.
- We provide a handler for `ConstraintViolationException`.

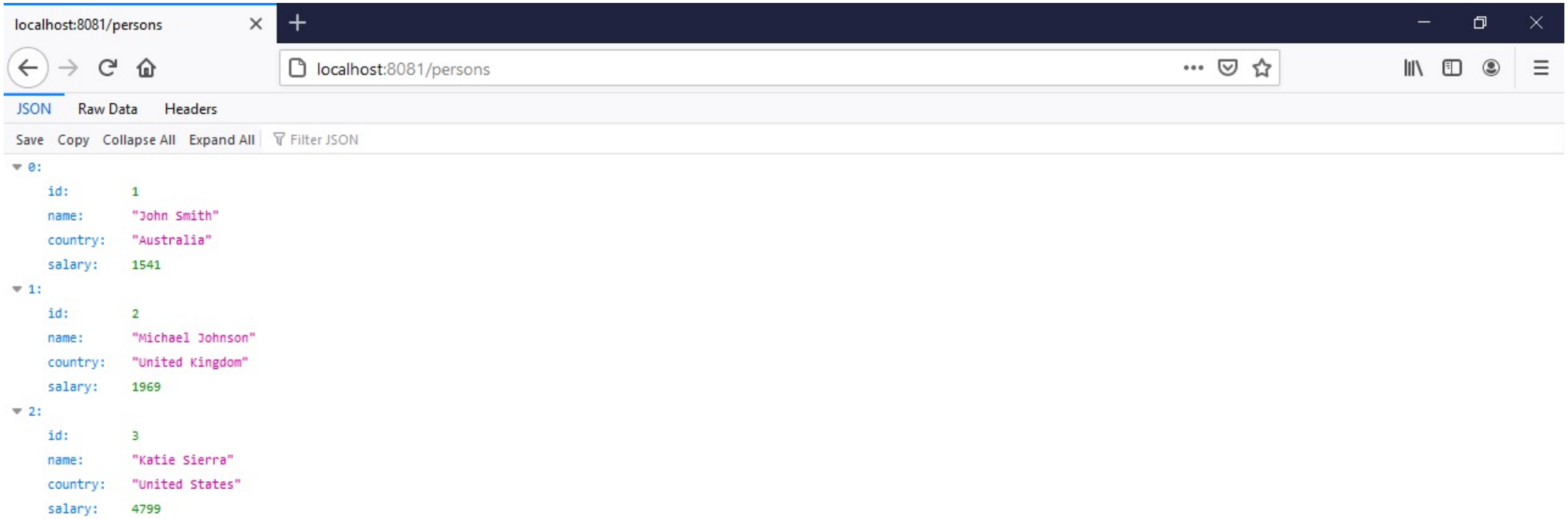
Error Handler for Request Parameters

```
@RestController
@Validated
public class PersonController {


    …
    @GetMapping("/persons/{id}")
    Person findOne(@PathVariable @Min(1) Long id) {
        return repository.findById(id)
                .orElseThrow(() -> new PersonNotFoundException(id));
    }
```

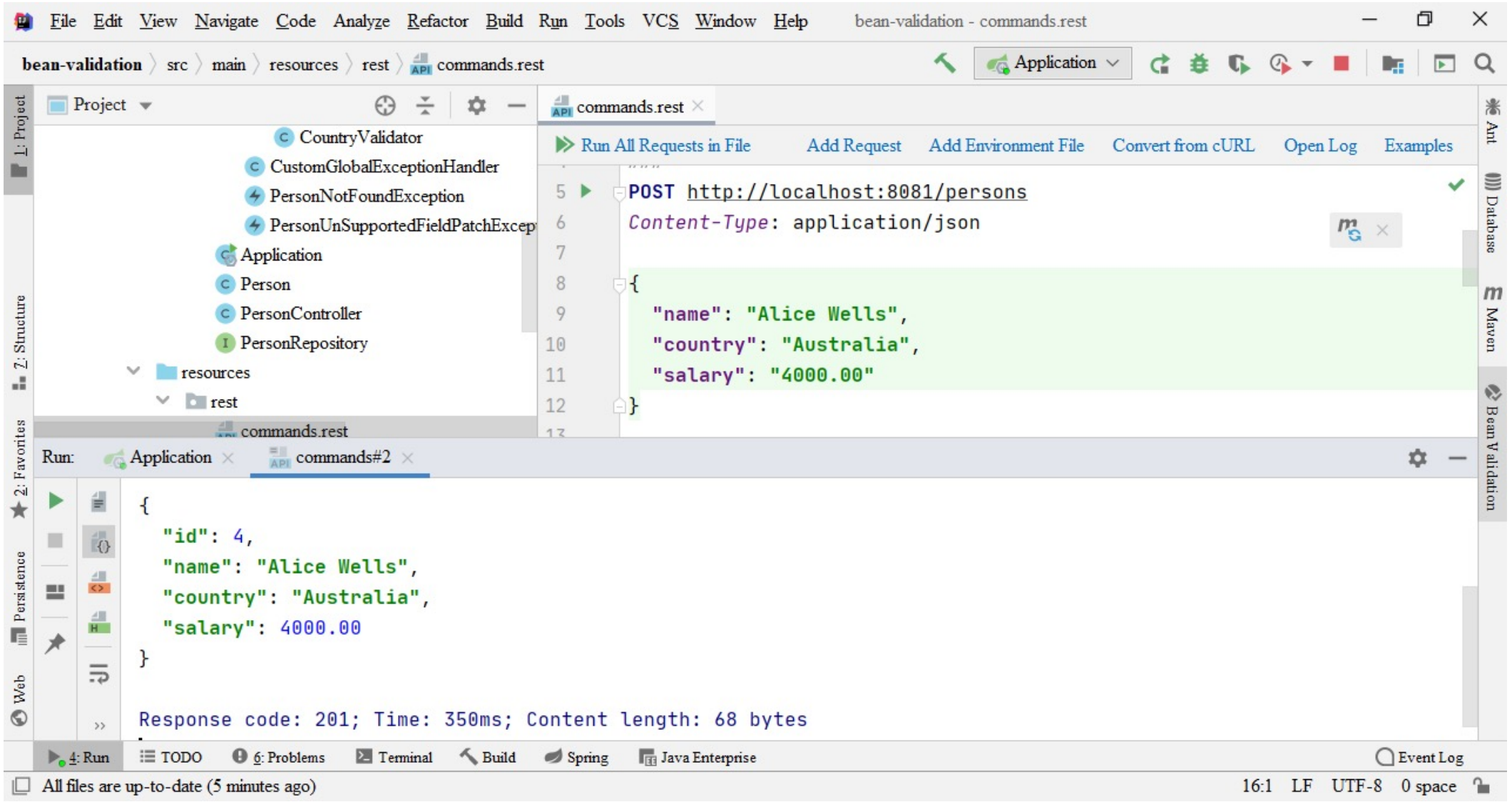
Error Handler for Request Parameters

```
@ControllerAdvice
public class CustomGlobalExceptionHandler
            extends ResponseEntityExceptionHandler {

    //Error handler for request parameters
    @ExceptionHandler(ConstraintViolationException.class)
    public void constraintViolationException
       (HttpServletResponse response) throws IOException {
        response.sendError(HttpStatus.BAD_REQUEST.value());
    }
```

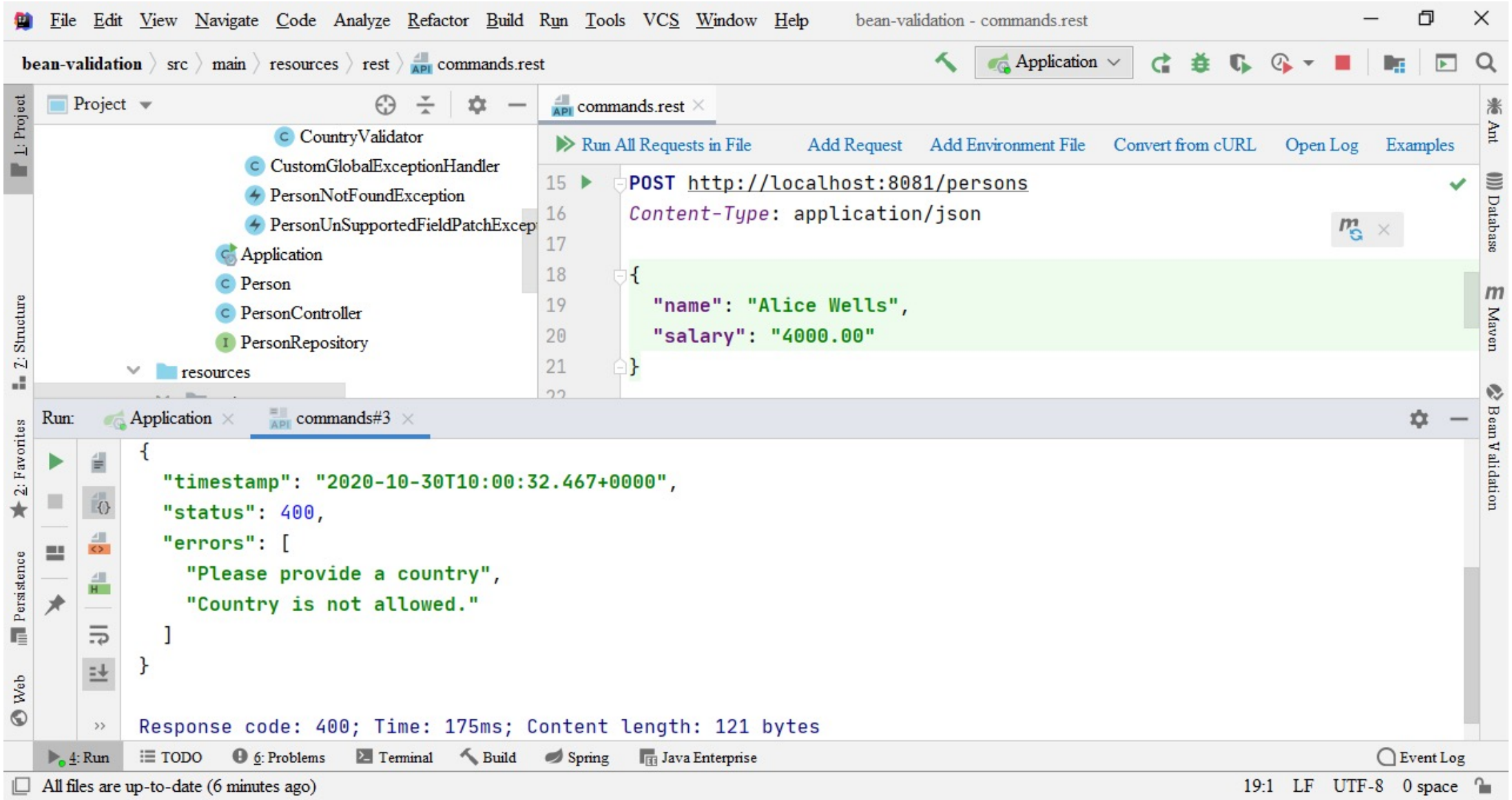
Starting the Application



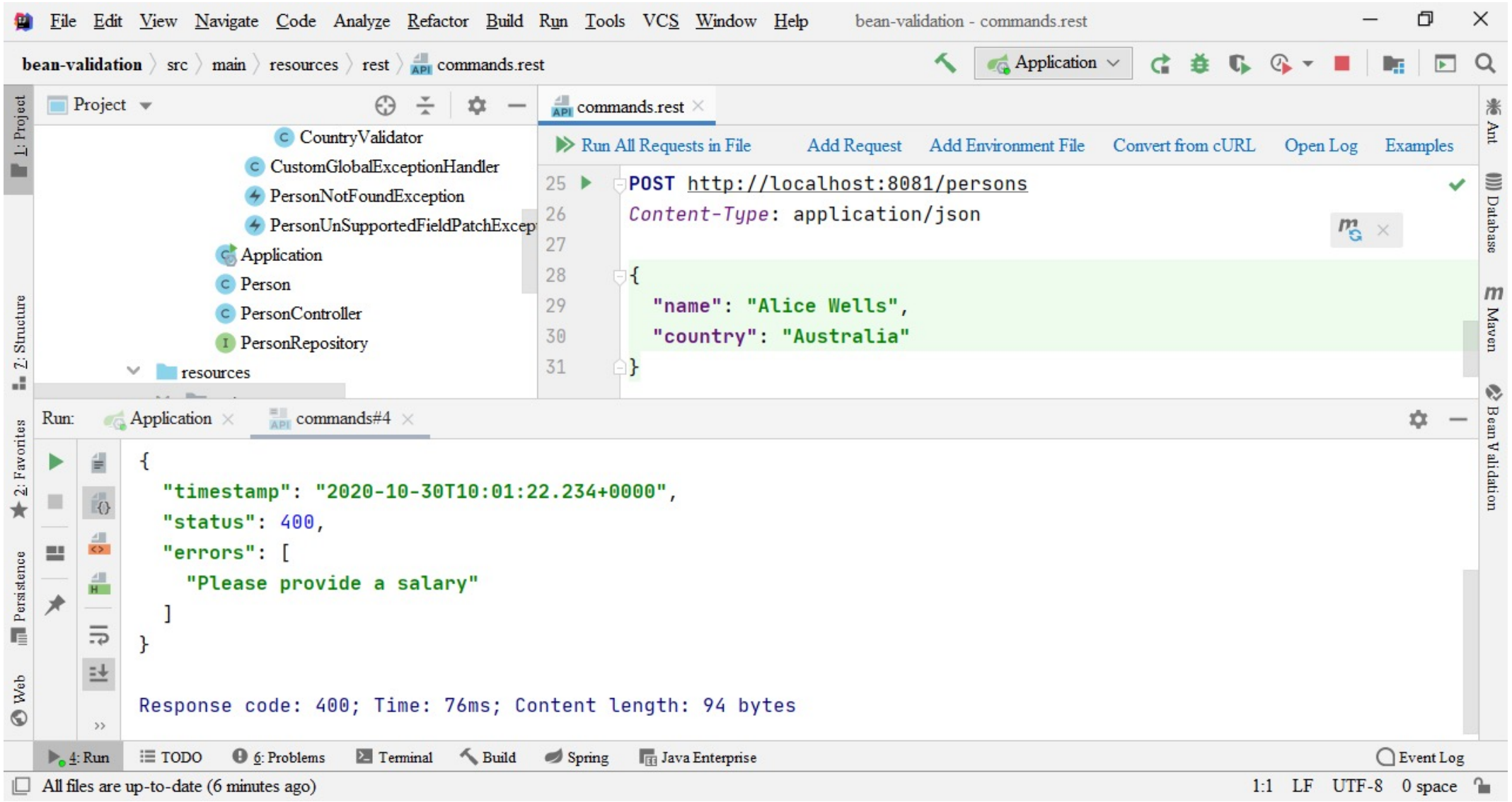
Adding a Valid Person



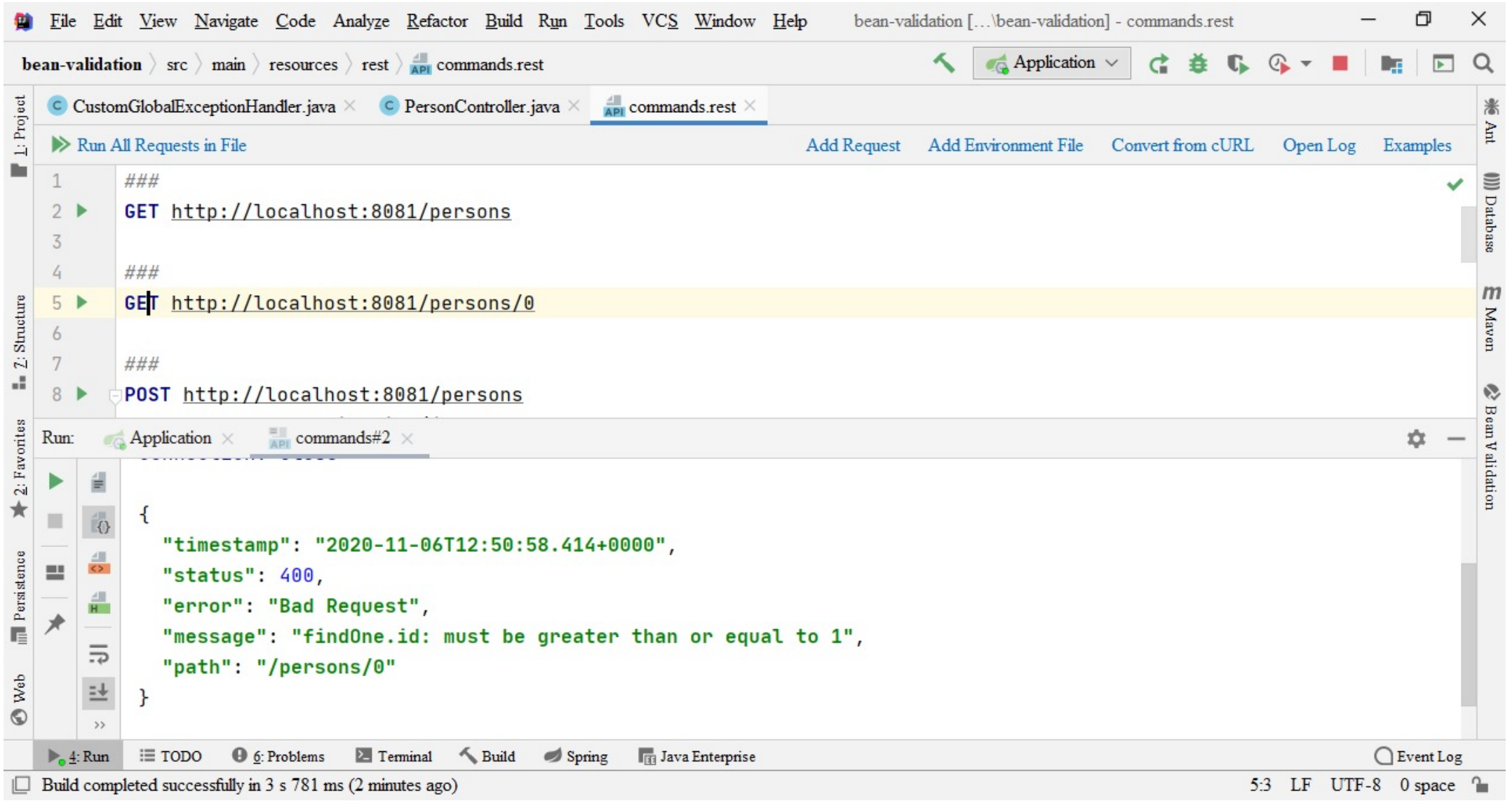
Adding a Person without a Country



Adding a Person without a Salary



Getting a Person with the ID Less than the Minimum



Thank You!

think.
create.
accelerate.

Пройди опрос
— получи **скидку 10%!**

https://ru.surveymonkey.com/r/promo_JSON_2_july

Luxoft
A DXC Technology Company

Luxoft Training – ведущий провайдер обучения, консалтинга и оценки IT-специалистов в сфере Software Engineering

15	**2000 г.**	**5+**
СТРАН: РОССИЯ, СНГ, ВОСТОЧНАЯ ЕВРОПА	СОЗДАНИЕ УЧЕБНОГО ЦЕНТРА	ПРОГРАММ ПОДГОТОВКИ К МЕЖДУНАРОДНЫМ СЕРТИФИКАЦИЯМ
200+	**250+**	**800+**
СООБЩЕСТВО ТРЕНЕРОВ-ПРАКТИКОВ	ТРЕНИНГОВ И ПРОГРАММ	КОРПОРАТИВНЫХ КЛИЕНТОВ

Заинтересовались темой?

Продолжите обучение на наших курсах!

Luxoft
A DXC Technology Company

luxoft-training.ru

Spring Advanced

Online, 30.08-09.09

education@luxoft.com