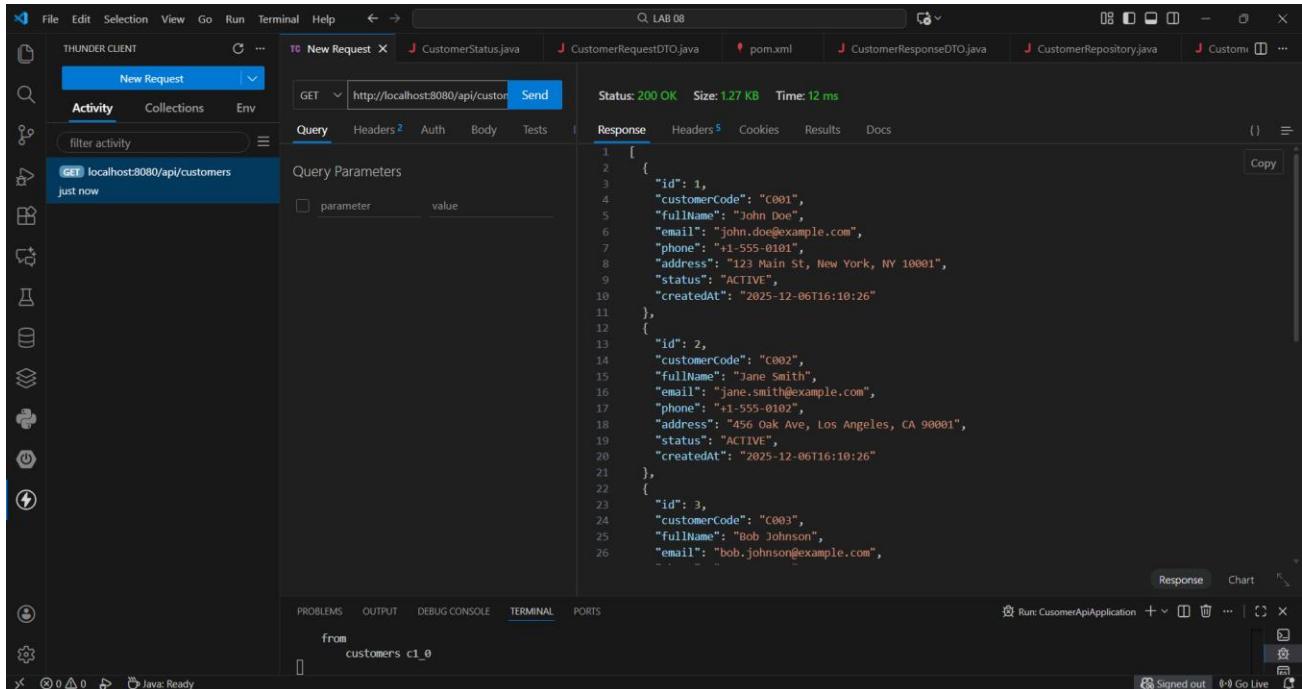


# Web Application Development Lab 08

## REST API & DTO PATTERN

### Part A – INCLASS EXERCISES

## I. RESULTS & CODE FLOW



*API Testing 'getAllCustomers'*

```

@RestController
@RequestMapping("/api/customers")
@CrossOrigin(origins = "*") // Allow CORS for frontend
public class CustomerRestController {

    private final CustomerService customerService;

    @Autowired
    public CustomerRestController(CustomerService customerService) {
        this.customerService = customerService;
    }

    // GET all customers
    @GetMapping
    public ResponseEntity<List<CustomerResponseDTO>> getAllCustomers() {
        List<CustomerResponseDTO> customers = customerService.getAllCustomers();
        return ResponseEntity.ok(customers);
    }
}
  
```

'/api/customers' invoke getAllCustomers(), which returns in a form of a response JSON file. The @RequestMapping and @GetMapping let us know what /api/customers does.

CustomerService maps to CustomerRepository function, which performs the necessary query on the database. findAll() is a basic CRUD function from JPA.

```
@Override
public List<CustomerResponseDTO> getAllCustomers() {
    return customerRepository.findAll()
        .stream()
        .map(this::convertToResponseDTO)
        .collect(Collectors.toList());
}
```

The screenshot shows the Thunder Client interface. A POST request is being made to `http://localhost:8080/api/customers`. The request body is a JSON object:

```
{
  "customerCode": "C006",
  "fullName": "David Miller Jr.",
  "email": "david.miller.jr@example.com",
  "phone": "0901112226",
  "address": "1000 Broadway, Seattle, WA 98101"
}
```

The response status is `201 Created`, size is `273 Bytes`, and time is `77 ms`. The response body is:

```
1 {
2   "id": 6,
3   "customerCode": "C006",
4   "fullName": "David Miller Jr.",
5   "email": "david.miller.jr@example.com",
6   "phone": "0901112226",
7   "address": "1000 Broadway, Seattle, WA 98101",
8   "status": "ACTIVE",
9   "createdAt": "2025-12-06T10:18:54.288786"
10 }
```

*Adding a customer*

```
// POST create new customer
@PostMapping
public ResponseEntity<CustomerResponseDTO> createCustomer(@Valid @RequestBody CustomerRequestDTO requestDTO) {
    CustomerResponseDTO createdCustomer = customerService.createCustomer(requestDTO);
    return ResponseEntity.status(HttpStatus.CREATED).body(createdCustomer);
}
```

@PostMapping handles POST methods made to the @RequestMapping('/api/customers'), when a POST method is called, it invokes createCustomer(), which return a JSON response file.

Similarly, CustomerService maps to CustomerRepository function. Some validation is in place.

```
@Override
public CustomerResponseDTO createCustomer(CustomerRequestDTO requestDTO) {
    // Check for duplicates
    if (customerRepository.existsByCustomerCode(requestDTO.getCustomerCode())) {
        throw new DuplicateResourceException("Customer code already exists: " + requestDTO.getCustomerCode());
    }

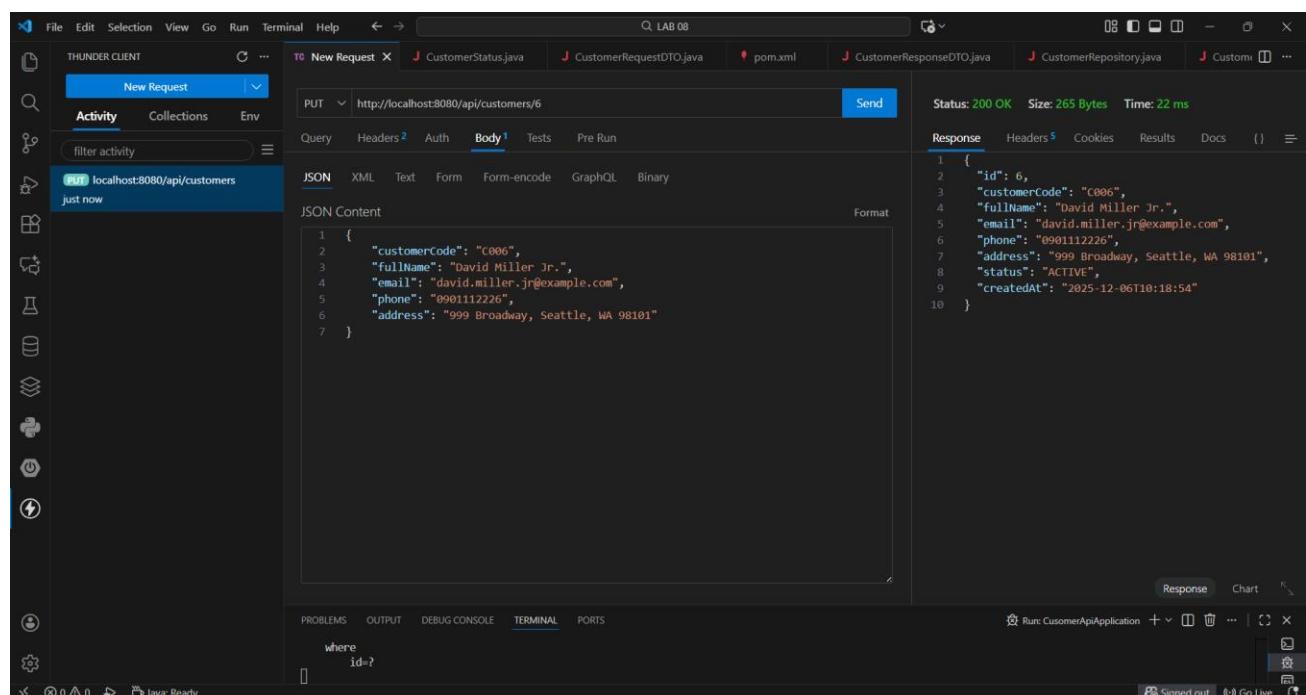
    if (customerRepository.existsByEmail(requestDTO.getEmail())) {
        throw new DuplicateResourceException("Email already exists: " + requestDTO.getEmail());
    }

    // Convert DTO to Entity
    Customer customer = convertToEntity(requestDTO);

    // Save to database
    Customer savedCustomer = customerRepository.save(customer);

    // Convert Entity to Response DTO
    return convertToResponseDTO(savedCustomer);
}
```

The convertToEntity() method turn the request into a Customer in the database, then they are saved.



### *Updating a customer*

```
// PUT update customer
@GetMapping("/{id}")
public ResponseEntity<CustomerResponseDTO> updateCustomer(
    @PathVariable Long id,
    @Valid @RequestBody CustomerRequestDTO requestDTO) {
    CustomerResponseDTO updatedCustomer = customerService.updateCustomer(id, requestDTO);
    return ResponseEntity.ok(updatedCustomer);
}
```

In addition the api root url ‘/api/customers/’, PUT requests need to have an addition ‘/id’, where ID is an integer. Again, CustomerService maps to CustomerRepository function. updateCustomer() simply parse all the data necessary to the Repository and do some basic validation. Save() is a basic CRUD function from JPA.

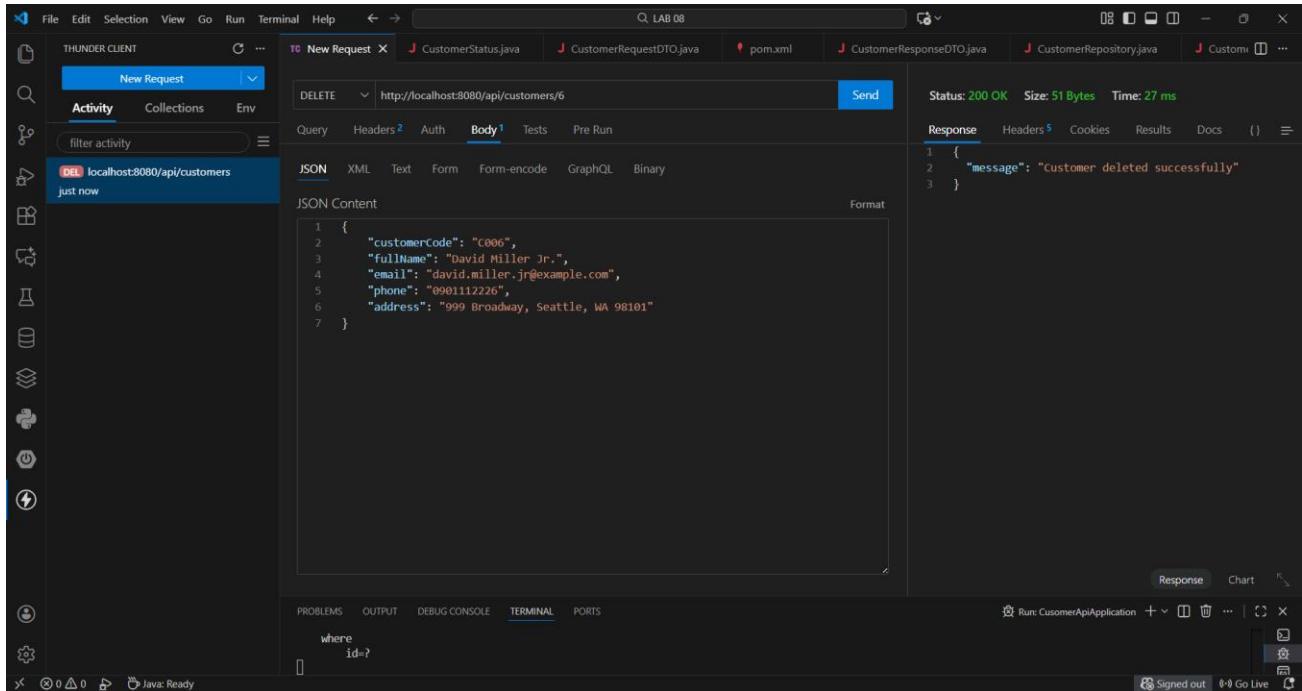
```
@Override
public CustomerResponseDTO updateCustomer(Long id, CustomerRequestDTO requestDTO) {
    Customer existingCustomer = customerRepository.findById(id)
        .orElseThrow(() -> new ResourceNotFoundException("Customer not found with id: " + id));

    // Check if email is being changed to an existing one
    if (!existingCustomer.getEmail().equals(requestDTO.getEmail())
        && customerRepository.existsByEmail(requestDTO.getEmail())) {
        throw new DuplicateResourceException("Email already exists: " + requestDTO.getEmail());
    }

    // Update fields
    existingCustomer.setFullName(requestDTO.getFullName());
    existingCustomer.setEmail(requestDTO.getEmail());
    existingCustomer.setPhone(requestDTO.getPhone());
    existingCustomer.setAddress(requestDTO.getAddress());

    // Don't update customerCode (immutable)

    Customer updatedCustomer = customerRepository.save(existingCustomer);
    return convertToResponseDTO(updatedCustomer);
}
```



### Delete user

The code flow is very similar to update

```

// DELETE customer
@DeleteMapping("/{id}")
public ResponseEntity<Map<String, String>> deleteCustomer(@PathVariable Long id) {
    customerService.deleteCustomer(id);
    Map<String, String> response = new HashMap<>();
    response.put("message", "Customer deleted successfully");
    return ResponseEntity.ok(response);
}

```

Instead of saving it to the database, we delete the user with the matching parameters.

```

@Override
public void deleteCustomer(Long id) {
    if (!customerRepository.existsById(id)) {
        throw new ResourceNotFoundException("Customer not found with id: " + id);
    }
    customerRepository.deleteById(id);
}

```

The screenshot shows the Thunder Client interface. In the top navigation bar, there are tabs for 'File', 'Edit', 'Selection', 'View', 'Go', 'Run', 'Terminal', and 'Help'. The current tab is 'New Request'. Below the tabs, the URL is set to 'http://localhost:8080/api/customers/search?keyword=john'. The 'Send' button is highlighted in blue. To the right, the status bar shows 'Status: 200 OK', 'Size: 513 Bytes', and 'Time: 23 ms'. The main area is divided into sections: 'Query Parameters' (with 'keyword' set to 'john'), 'Headers' (with 'Content-Type' set to 'application/json'), 'Body' (empty), 'Tests' (empty), and 'Pre Run' (empty). On the right side, the 'Response' tab is selected, displaying a JSON array of customer objects. The JSON data is as follows:

```

1  [
2    {
3      "id": 1,
4      "customerCode": "C001",
5      "fullName": "John Doe",
6      "email": "john.doe@example.com",
7      "phone": "+1-555-0101",
8      "address": "123 Main St, New York, NY 10001",
9      "status": "ACTIVE",
10     "createdAt": "2025-12-06T16:10:26"
11   },
12   {
13     "id": 3,
14     "customerCode": "C003",
15     "fullName": "Bob Johnson",
16     "email": "bob.johnson@example.com",
17     "phone": "+1-555-0103",
18     "address": "789 Pine Rd, Chicago, IL 60601",
19     "status": "ACTIVE",
20     "createdAt": "2025-12-06T16:10:26"
21   }
22 ]

```

Below the response, there are tabs for 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', 'TERMINAL', and 'PORTS'. The 'TERMINAL' tab is currently active, showing the command 'Run: CustomerApiApplication'. At the bottom, there are icons for Java Ready, Signed out, and Go Live.

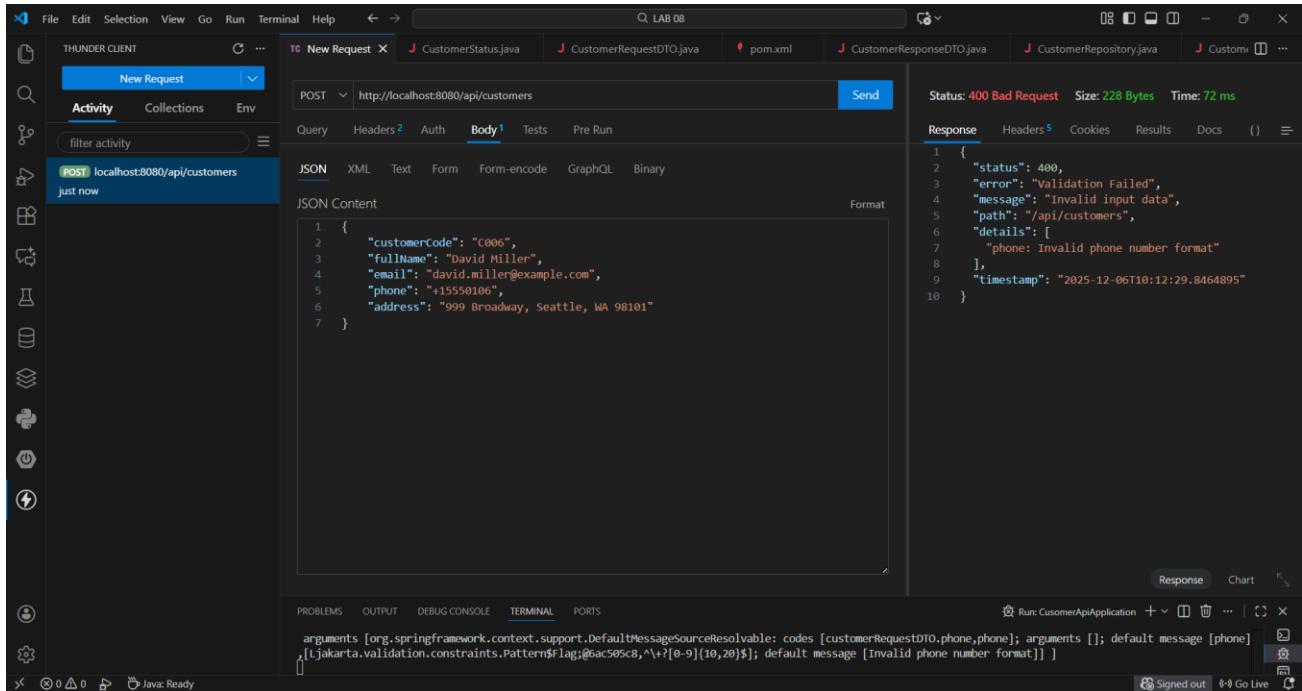
*Searching for 'john'*

```

// GET customer by ID
@GetMapping("/{id}")
public ResponseEntity<CustomerResponseDTO> getCustomerById(@PathVariable Long id) {
    CustomerResponseDTO customer = customerService.getCustomerById(id);
    return ResponseEntity.ok(customer);
}

```

Handles any GET method followed by /id. getCustomerByID maps to CustomerRepository, which is a findById() CRUD operation.



## Validation

Validation is done through the DTO

```

@NotBlank(message = "Customer code is required")
@Size(min = 3, max = 20, message = "Customer code must be 3-20 characters")
@Pattern(regexp = "^[C]\\d{3,}$", message = "Customer code must start with C followed by numbers")
private String customerCode;

@NotBlank(message = "Full name is required")
@Size(min = 2, max = 100, message = "Name must be 2-100 characters")
private String fullName;

@NotBlank(message = "Email is required")
@Email(message = "Invalid email format")
private String email;

@Pattern(regexp = "^[+]?[0-9]{10,20}$", message = "Invalid phone number format")
private String phone;

@Size(max = 500, message = "Address too long")
private String address;

private String status;
  
```

If any of the parameters is violated, it returns a message

```
// Handle Validation Errors (400)
@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity<ErrorResponseDTO> handleValidationException(
    MethodArgumentNotValidException ex,
    WebRequest request) {

    List<String> details = new ArrayList<>();
    for (FieldError error : ex.getBindingResult().getFieldErrors()) {
        details.add(error.getField() + ": " + error.getDefaultMessage());
    }

    ErrorResponseDTO error = new ErrorResponseDTO(
        HttpStatus.BAD_REQUEST.value(),
        error: "Validation Failed",
        message: "Invalid input data",
        request.getDescription(false).replace("uri=", ""))
    );
    error.setDetails(details);

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

The message is used by the exception handler to build the response. That is all.