

1. Get all customers:

- Client sends GET request to controller with endpoint '/customers'.
- Controller maps the request to getAllCustomers() and calls customerService.getAllCustomers().

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

- Service calls Repository to query the database.
- Repository returns an entity list to service.
- Service gets the entity and maps it to a ResponseDTO.

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

- Service returns the DTO to Controller.
- Controller returns a response with the DTO converted to JSON format to the client.

The screenshot shows a REST client interface. On the left, the 'Query' tab is active, displaying the URL 'http://localhost:8080/api/customers' and a 'Send' button. Below the URL bar, there are tabs for 'Query', 'Headers', 'Auth', 'Body', 'Tests', and 'Pre Run'. The 'Query' tab shows 'Query Parameters' with a table with columns 'parameter' and 'value'. On the right, the 'Response' tab is active, showing the status '200 OK', size '1.48 KB', and time '38 ms'. The response body is a JSON array with one object representing a customer.

```
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": "2025-12-06T16:19:21"
11  },
12 }
```

2. Get customer by id:

- Client sends GET request to controller with endpoint '/customers/id'.
- Controller maps the request to getAllCustomers() and calls customerService.getCustomerById().

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

- Service calls Repository to query the database.
- Repository returns an entity to service.
- Service gets the entity and maps it to a RequestDTO.

```
@Override
public CustomerResponseDTO getCustomerById(Long id) {
    Customer customer = customerRepository.findById(id)
        .orElseThrow(() -> new ResourceNotFoundException("Customer not found wi
    return convertToResponseDTO(customer);
}
```

- Service returns the DTO to Controller.
- Controller returns a response with the DTO converted to JSON format to the client.

The screenshot shows a REST client interface. On the left, the 'Query' tab is active, displaying the URL 'http://localhost:8080/api/customers/2'. On the right, the 'Response' tab is active, showing a JSON response with the following details:

- Status: 200 OK
- Size: 249 Bytes
- Time: 12 ms

The JSON response is as follows:

```
{
  "id": 2,
  "customerCode": "C002",
  "fullName": "Jane Smith",
  "email": "jane.smith@example.com",
  "phone": "+1-555-0102",
  "address": "456 Oak Ave, Los Angeles, CA 90001",
  "status": "ACTIVE",
  "createdAt": "2025-12-06T16:19:21"
}
```

3. Create customer:

- Client sends POST request to controller with endpoint '/customers' and a RequestBody with the customer information in JSON format.
- Controller maps the request to createCustomer() and calls customerService.createCustomer().

```
// 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);
}
```

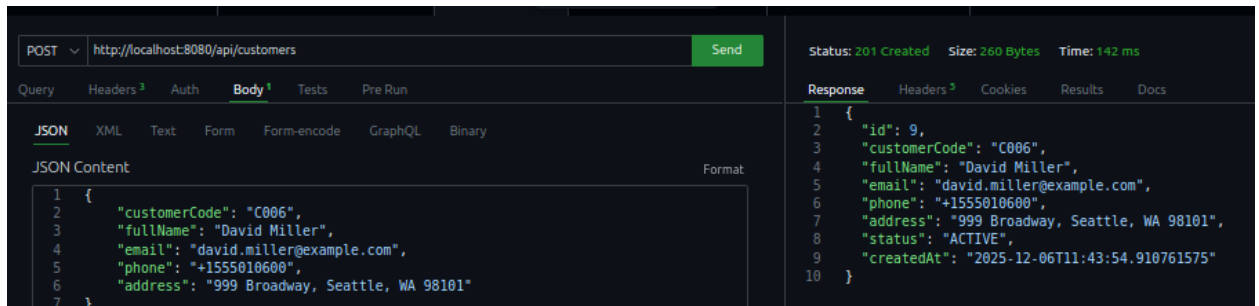
- Service converts the JSON to Entity, then calls Repository to query the database.
- Repository returns an entity to service.
- Service gets the entity and maps it to a ResponseDTO.

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

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

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

- Service returns the DTO to Controller.
- Controller returns a response with the status CREATED and the new customer in JSON body.



4. Update customer:

- Client sends PUT request to controller with endpoint '/customers/id' and a RequestBody with the customer information in JSON format.
- Controller maps the request to updateCustomer() and calls customerService.updateCustomer().

```

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

```

- Service calls Repository to check if the customer exist and create a customer entity.

```

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

```

- Service update the fields off the entity.
- Service calls Repository to save the entity and convert it to a ResponseDTO.

```

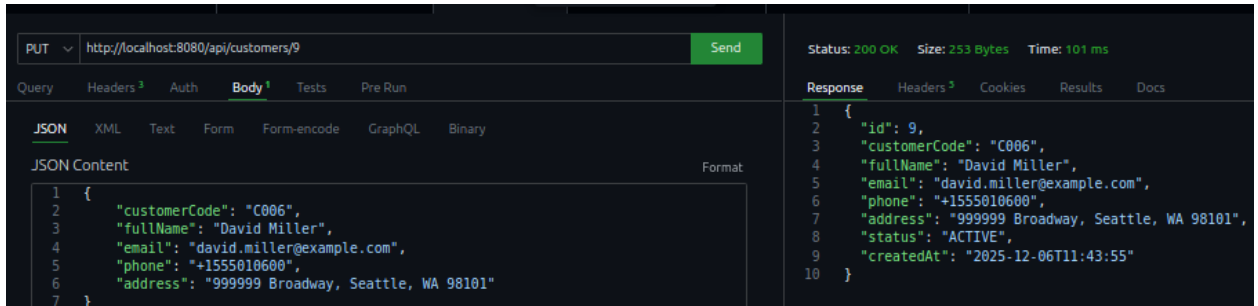
// Update fields
existingCustomer.setFullName(requestDTO.getFullName());
existingCustomer.setEmail(requestDTO.getEmail());
existingCustomer.setPhone(requestDTO.getPhone());
existingCustomer.setAddress(requestDTO.getAddress());

// Don't update customerCode (immutable)

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

```

- Service returns the DTO to Controller.
- Controller returns a response with ok status and the updated customer in JSON body.



5. Delete customer:

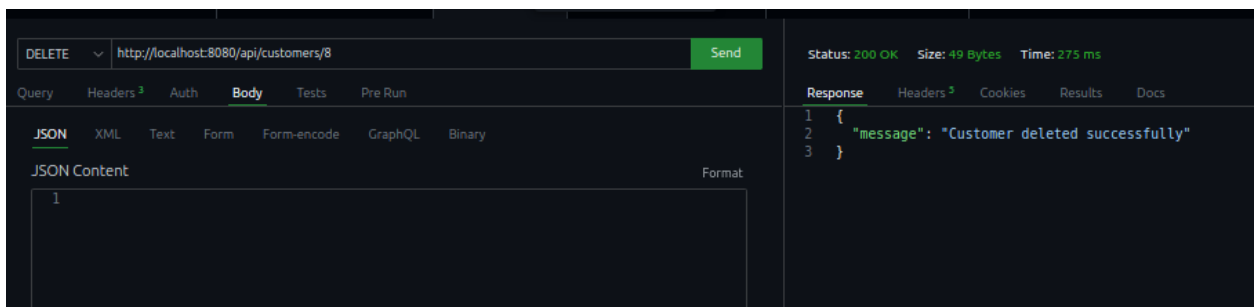
- Client sends DELETE request to controller with endpoint '/customers/id'.
- Controller maps the request to deleteCustomer() and calls customerService.deleteCustomer().

```
// 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);
}
```

- Service calls repository to check if a customer with the id exists and delete the customer.

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

- Controller returns a response with the status ok.



6. Search customer by keyword:

- Client sends GET request to controller with endpoint '/customers/search?keyword='.
- Controller maps the request to searchCustomers() and calls customerService.searchCustomers().

```
// GET search customers
@GetMapping("/search")
public ResponseEntity<List<CustomerResponseDTO>> searchCustomers(@RequestParam String keyword) {
    List<CustomerResponseDTO> customers = customerService.searchCustomers(keyword);
    return ResponseEntity.ok(customers);
}
```

- Service calls Repository to query the database.
- Repository returns an entity to service.
- Service gets the entity and maps it to a ResponseDTO.

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

- Service returns the DTO to Controller.
- Controller returns a response with the DTO with status ok and JSON body to the client.

The screenshot shows the Thunder Client interface. The top bar displays the method 'GET', the URL 'http://localhost:8080/api/customers/search?keyword=john', and a 'Send' button. Below the bar, the 'Query' tab is selected, showing the URL. The 'Headers' tab is also visible, showing the 'Accept' header set to '*/*' and the 'User-Agent' set to 'Thunder Client (https://www.thunderclient.com)'. The 'Response' tab is selected, showing the status '200 OK', size '495 Bytes', and time '14 ms'. The response body is a JSON array of two customer objects.

```
[
  {
    "id": 1,
    "customerCode": "C001",
    "fullName": "John Doe",
    "email": "john.doe@example.com",
    "phone": "+1-555-0101",
    "address": "123 Main St, New York, NY 10001",
    "status": "ACTIVE",
    "createdAt": "2025-12-06T16:19:21"
  },
  {
    "id": 3,
    "customerCode": "C003",
    "fullName": "Bob Johnson",
    "email": "bob.johnson@example.com",
    "phone": "+1-555-0103",
    "address": "789 Pine Rd, Chicago, IL 60601",
    "status": "ACTIVE",
    "createdAt": "2025-12-06T16:19:21"
  }
]
```

7. Validation error:

- Client sends a request (POST or PUT) to controller with invalid data in RequestBody.
- Controller validates the RequestBody using @Valid annotation and catches MethodArgumentNotValidException.

```
// POST create new customer
@PostMapping
public ResponseEntity<CustomerResponseDTO> createCustomer(@Valid @RequestBody CustomerRequest requestDTO) {
    // ...
}
```

- GlobalExceptionHandler intercepts the exception and calls handleValidationException().
- Handler extracts field errors from BindingResult and creates a list of validation error details.
- Handler creates an ErrorResponseDTO with status BAD_REQUEST and the validation error details.
- Handler returns the ErrorResponseDTO to Controller.

```
// 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);
}
```

- Controller returns a response with status 400 (BAD_REQUEST) and the error details in JSON body to the client.

The screenshot shows a REST client interface. On the left, a POST request is configured to `http://localhost:8080/api/customers`. The request body is a JSON object representing a customer: `{ "customerCode": "C006", "fullName": "David Miller", "email": "david.miller@example.com", "phone": "+15550106", "address": "999999 Broadway, Seattle, WA 98101" }`. On the right, the response is displayed with status `400 Bad Request`, size `223 Bytes`, and time `8 ms`. The response body is a JSON object: `{ "status": 400, "error": "Validation Failed", "message": "Invalid input data", "path": "/api/customers", "details": [{ "phone": "Invalid phone number format" }], "timestamp": "2025-12-06T11:45:31.137352258" }`.

8. Resource not found:

- Client sends a request (GET, PUT, or DELETE) to controller with an invalid customer id.
- Controller maps the request and calls the corresponding `customerService` method.
- Service calls Repository to query the database by id.
- Repository returns empty Optional because customer doesn't exist.
- Service throws `ResourceNotFoundException`.
- `GlobalExceptionHandler` intercepts the exception and calls `handleResourceNotFoundException()`.
- Handler creates an `ErrorResponseDTO` with status `NOT_FOUND`.
- Handler returns the `ErrorResponseDTO` to Controller.

```
// Handle ResourceNotFoundException (404)
@ExceptionHandler(ResourceNotFoundException.class)
public ResponseEntity<ErrorResponseDTO> handleResourceNotFoundException(
    ResourceNotFoundException ex,
    WebRequest request) {

    ErrorResponseDTO error = new ErrorResponseDTO(
        HttpStatus.NOT_FOUND.value(),
        error: "Not Found",
        ex.getMessage(),
        request.getDescription(false).replace("uri=", "")
    );

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

- Controller returns a response with status 404 (NOT_FOUND) and the error details in JSON body to the client.

The screenshot shows a web browser with the URL `http://localhost:8080/api/customers/999`. The status bar indicates a **404 Not Found** error with a size of 195 Bytes and a time of 12 ms. The response body is displayed in JSON format, showing the following details:

```
{
  "status": 404,
  "error": "Not Found",
  "message": "Customer not found with id: 999",
  "path": "/api/customers/999",
  "details": null,
  "timestamp": "2025-12-06T11:45:54.60998681"
}
```

9. Duplicate resource:

- Client sends a POST request to create a customer with existing customerCode or email.
- Controller maps the request to `createCustomer()` and calls `customerService.createCustomer()`.
- Service calls Repository to check if customerCode or email already exists.
- Repository returns true for `existsByCustomerCode()` or `existsByEmail()`.
- Service throws `DuplicateResourceException`.
- `GlobalExceptionHandler` intercepts the exception and calls `handleDuplicateResourceException()`.
- Handler creates an `ErrorResponseDTO` with status `CONFLICT`.
- Handler returns the `ErrorResponseDTO` to Controller.

```
// Handle DuplicateResourceException (409)
@ExceptionHandler(DuplicateResourceException.class)
public ResponseEntity<ErrorResponseDTO> handleDuplicateResourceException(
    DuplicateResourceException ex,
    WebRequest request) {

    ErrorResponseDTO error = new ErrorResponseDTO(
        HttpStatus.CONFLICT.value(),
        error: "Conflict",
        ex.getMessage(),
        request.getDescription(false).replace("uri=", "")
    );

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

- Controller returns a response with status 409 (CONFLICT) and the error details in JSON body to the client.

The screenshot displays a REST client interface with a POST request to `http://localhost:8080/api/customers`. The request body is a JSON object containing customer details. The response status is 409 Conflict, with a size of 194 Bytes and a time of 29 ms. The response body is a JSON object indicating a conflict because the customer code already exists.

Request:

```
POST http://localhost:8080/api/customers
```

JSON Content:

```
1 {
2   "customerCode": "C006",
3   "fullName": "David Miller",
4   "email": "david.miller@example.com",
5   "phone": "+1555010600",
6   "address": "999999 Broadway, Seattle, WA 98101"
7 }
```

Response:

```
1 {
2   "status": 409,
3   "error": "Conflict",
4   "message": "Customer code already exists: C006",
5   "path": "/api/customers",
6   "details": null,
7   "timestamp": "2025-12-06T11:46:35.742996658"
8 }
```