**International University**

School of Computer Science and Engineering

# Web Application Development Laboratory

# IT093IU

# Lab #8

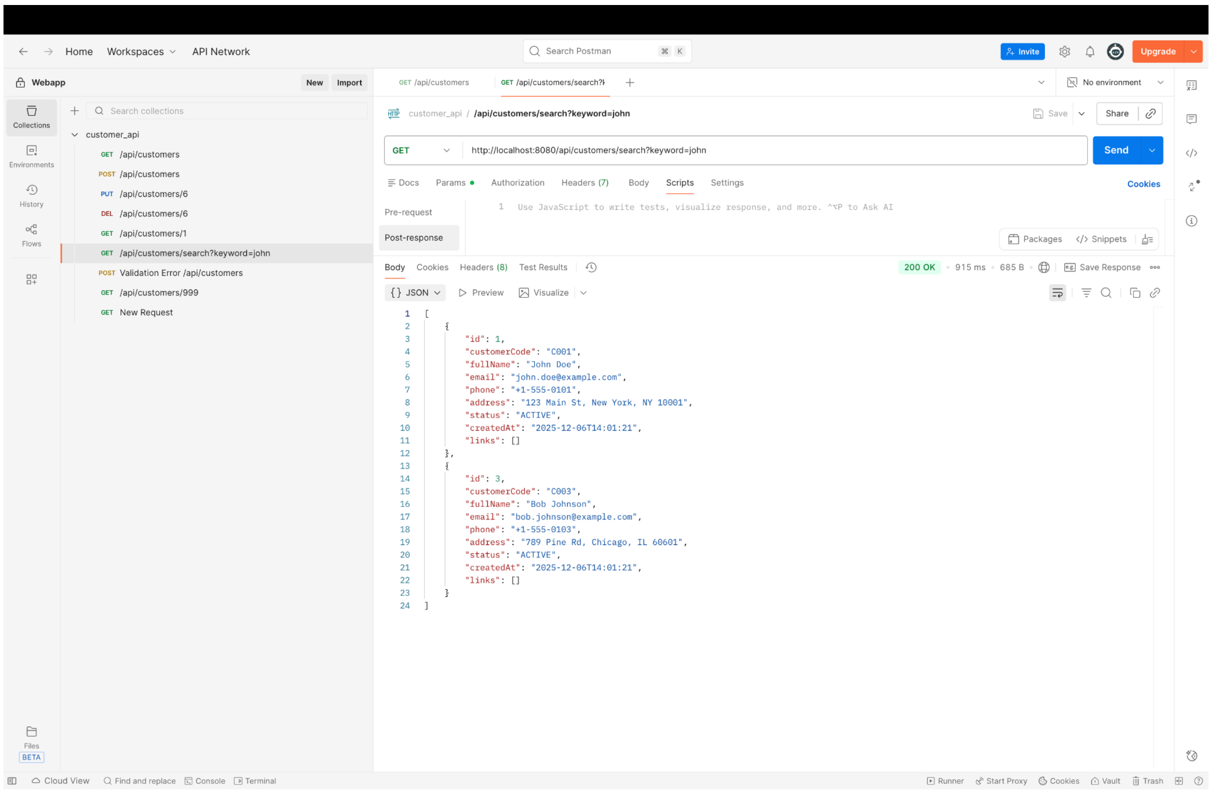
**Submitted by**

**Nguyễn Hồng Ngọc Hân - ITCSIU22229**

## EXERCISE 5: SEARCH & FILTER ENDPOINTS

### Task 5.1: Search Customers

### GET /api/customers/search?keyword=john



GET http://localhost:8080/api/customers/search?keyword=john

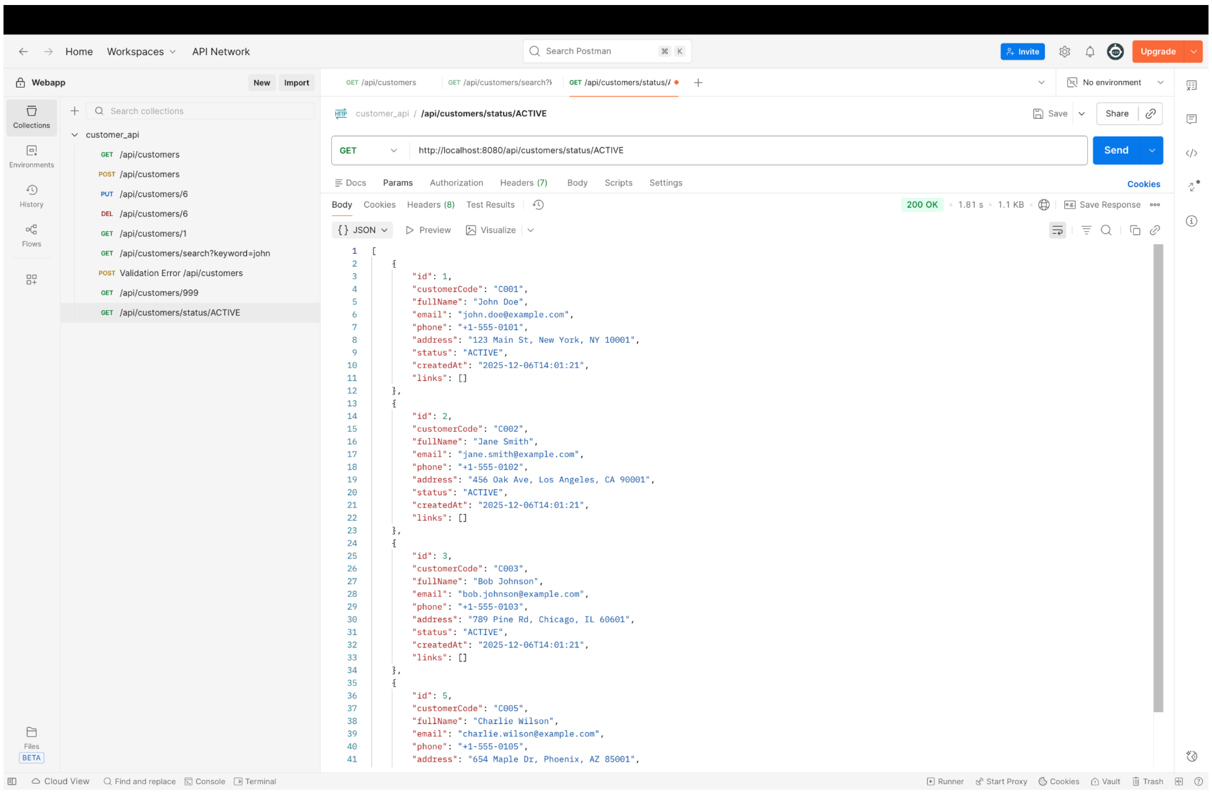
200 OK · 915 ms · 685 B

```
[
    {
        "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": "2025-12-06T14:01:21",
        "links": []
    },
    {
        "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": "2025-12-06T14:01:21",
        "links": []
    }
]
```

When a request is sent to the endpoint GET /api/customers/search?keyword=john, the processing flow begins in the Controller. At this level, the keyword parameter is extracted from the query string using the @RequestParam annotation. The Controller then calls the Service layer by invoking the searchCustomers(keyword) method. Inside the Service, the search logic is executed by delegating the task to the Repository, typically through a method such as customerRepository.searchCustomers(keyword) or built-in query methods like findByFullNameContainingIgnoreCase(keyword) or findByEmailContainingIgnoreCase(keyword). The Repository is responsible for executing a corresponding SQL SELECT query in the database to find customers whose fields match the given keyword. The result of this query is a list of Customer entities. The Service then converts these Customer entities into CustomerResponseDTO objects before returning them to the Controller. Finally, the Controller responds to the client with the DTO list and an HTTP status 200 OK.

### Task 5.2: Filter by Status

### GET /api/customers/status/ACTIVE

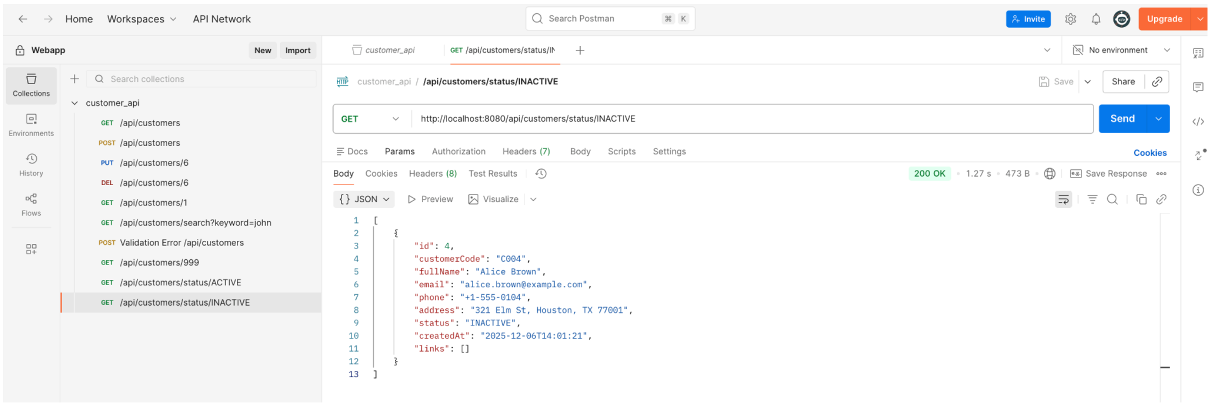

GET http://localhost:8080/api/customers/status/ACTIVE

200 OK · 1.81 s · 1.1 KB

```
[
    {
        "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": "2025-12-06T14:01:21",
        "links": []
    },
    {
        "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": "2025-12-06T14:01:21",
        "links": []
    },
    {
        "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": "2025-12-06T14:01:21",
        "links": []
    },
    {
        "id": 5,
        "customerCode": "C005",
        "fullName": "Charlie Wilson",
        "email": "charlie.wilson@example.com",
        "phone": "+1-555-0105",
        "address": "654 Maple Dr, Phoenix, AZ 85001",
```

when a request is made to GET /api/customers/status/ACTIVE, the process again starts at the Controller, where the value `ACTIVE` is extracted as a `@PathVariable` and mapped to the `CustomerStatus` enum. The Controller calls the `getCustomersByStatus(status)` method in the Service layer. Inside the Service, the method delegates to the Repository, typically through `findByStatus(status)`. This Repository method executes a SQL SELECT query with a condition such as `WHERE status = 'ACTIVE'` to retrieve all customers with the specified status. After the Repository returns a list of Customer entities, the Service converts them into a list of CustomerResponseDTOs. The DTO list is then sent back to the Controller, which returns the response to the client along with an HTTP status 200 OK.

**GET /api/customers/status/INACTIVE**


GET http://localhost:8080/api/customers/status/INACTIVE

200 OK · 1.27 s · 473 B

```
[
    {
        "id": 4,
        "customerCode": "C004",
        "fullName": "Alice Brown",
        "email": "alice.brown@example.com",
        "phone": "+1-555-0104",
        "address": "321 Elm St, Houston, TX 77001",
        "status": "INACTIVE",
        "createdAt": "2025-12-06T14:01:21",
        "links": []
    }
]
```

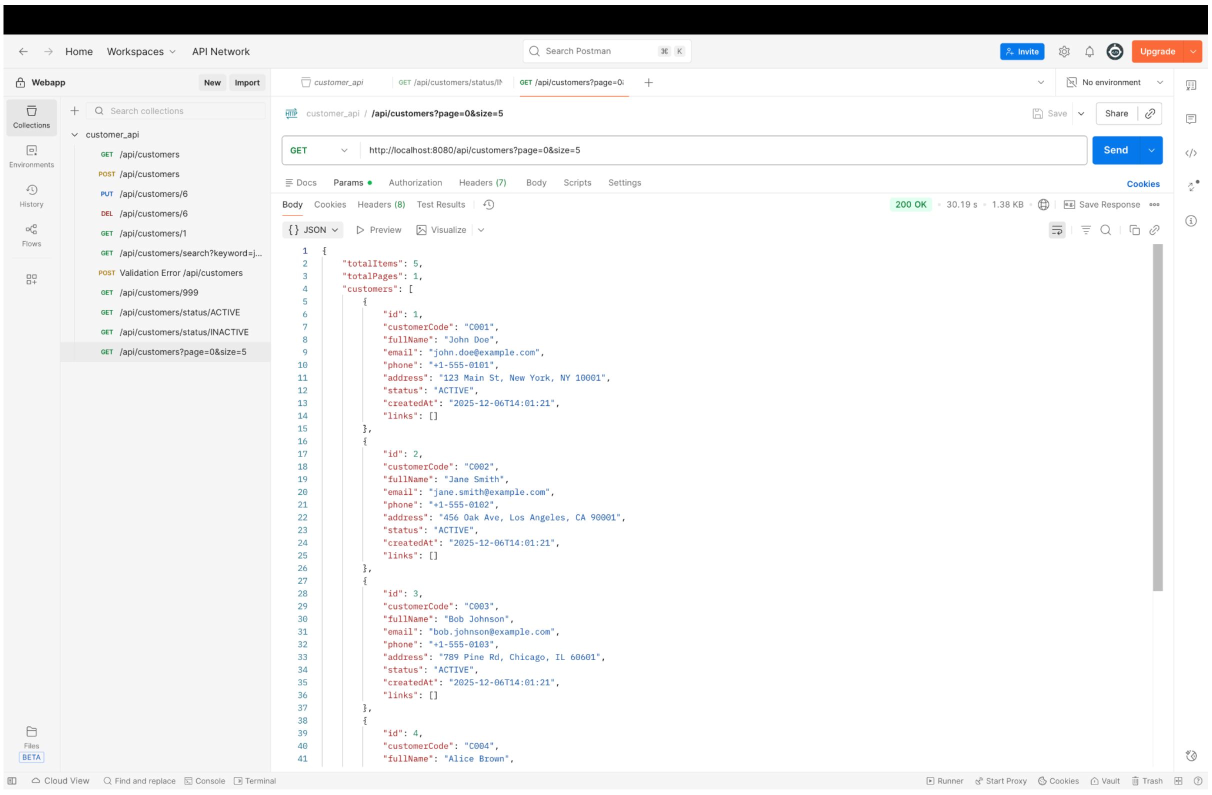
When a request is made to GET /api/customers/status/INACTIVE, the processing flow begins at the Controller. The value INACTIVE is captured from the URL using the @PathVariable annotation and is automatically mapped to the CustomerStatus enum. The Controller then invokes the Service layer by calling the method getCustomersByStatus(status). Inside the Service, the method delegates the filtering task to the Repository, typically through a query method such as findByStatus(status). The Repository executes a SQL SELECT statement with a condition like WHERE status = 'INACTIVE' to fetch all customers whose status is marked as INACTIVE in the database. Once the Repository returns the resulting list of Customer entities, the Service converts each entity into a CustomerResponseDTO to ensure that only the appropriate response data is exposed. After the conversion is complete, the Service returns the DTO list back to the Controller. Finally, the Controller sends the list of INACTIVE customers to the client along with an HTTP status 200 OK.

**Task 5.3: Advanced Search with Multiple Criteria**

## EXERCISE 6: PAGINATION & SORTING

**Task 6.1: Add Pagination**

**GET /api/customers?page=0&size=5**



customer_api / /api/customers?page=0&size=5

GET http://localhost:8080/api/customers?page=0&size=5

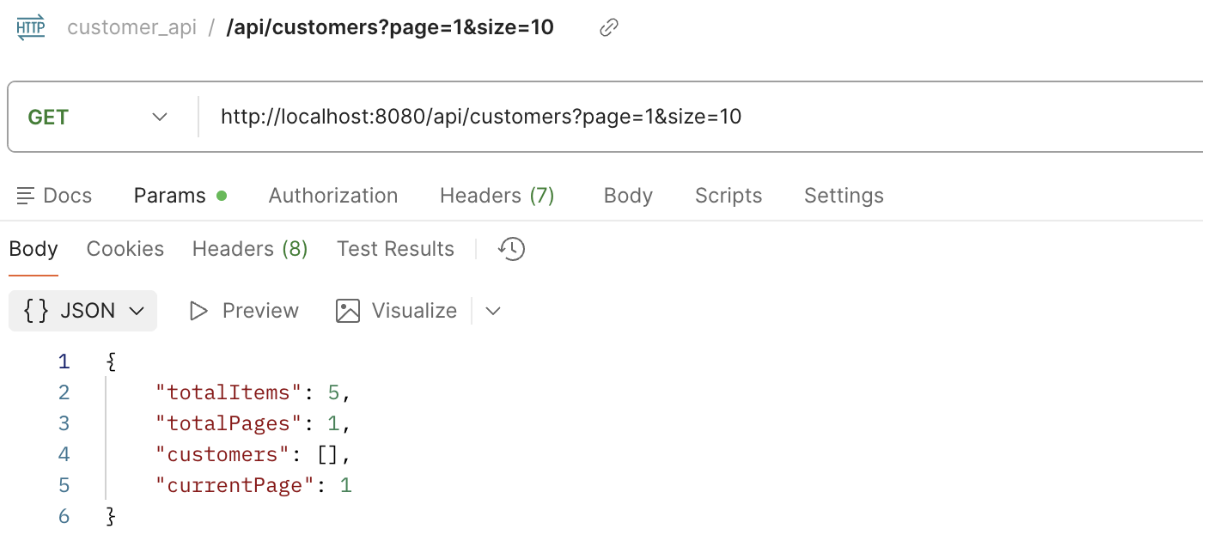
200 OK • 30.19 s • 1.38 KB

```
{
    "totalItems": 5,
    "totalPages": 1,
    "customers": [
        {
            "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": "2025-12-06T14:01:21",
            "links": []
        },
        {
            "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": "2025-12-06T14:01:21",
            "links": []
        },
        {
            "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": "2025-12-06T14:01:21",
            "links": []
        },
        {
            "id": 4,
            "customerCode": "C004",
            "fullName": "Alice Brown",
```

```
{
   "totalItems": 5,
   "totalPages": 1,
   "customers": [
      {
         "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": "2025-12-06T14:01:21",
         "links": []
      },
      {
         "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": "2025-12-06T14:01:21",
         "links": []
      },
      {
         "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": "2025-12-06T14:01:21",
         "links": []
      },
      {
         "id": 4,
         "customerCode": "C004",
         "fullName": "Alice Brown",
         "email": "alice.brown@example.com",
         "phone": "+1-555-0104",
         "address": "321 Elm St, Houston, TX 77001",
         "status": "INACTIVE",
         "createdAt": "2025-12-06T14:01:21",
         "links": []
      },
      {
         "id": 5,
         "customerCode": "C005",
         "fullName": "Charlie Wilson",
         "email": "charlie.wilson@example.com",
         "phone": "+1-555-0105",
         "address": "654 Maple Dr, Phoenix, AZ 85001",
         "status": "ACTIVE",
         "createdAt": "2025-12-06T14:01:21",
         "links": []
      }
   ],
   "currentPage": 0
}
```

When a request is sent to GET /api/customers?page=0&size=5, the handling process begins at the Controller. The query parameters page and size are extracted using the @RequestParam annotation and used to construct a PageRequest object that defines the pagination settings. The Controller then calls the Service method getAllCustomers(page, size), passing along the pagination parameters. Inside the Service, a Pageable instance (typically PageRequest.of(page, size)) is created and used to call the Repository method findAll(pageable). The Repository is responsible for executing a SQL SELECT query with pagination applied, retrieving only the specific subset of customers corresponding to the requested page. The result returned from the Repository is a Page<Customer>, which contains both the list of Customer entities and additional metadata such as total pages and total elements. The Service converts the Customer entities into CustomerResponseDTOs, preserving the pagination structure if necessary. After this conversion, the Service returns the paginated DTO data to the Controller. Finally, the Controller responds to the client with the paginated list of customers and an HTTP status 200 OK.

**GET /api/customers?page=1&size=10**



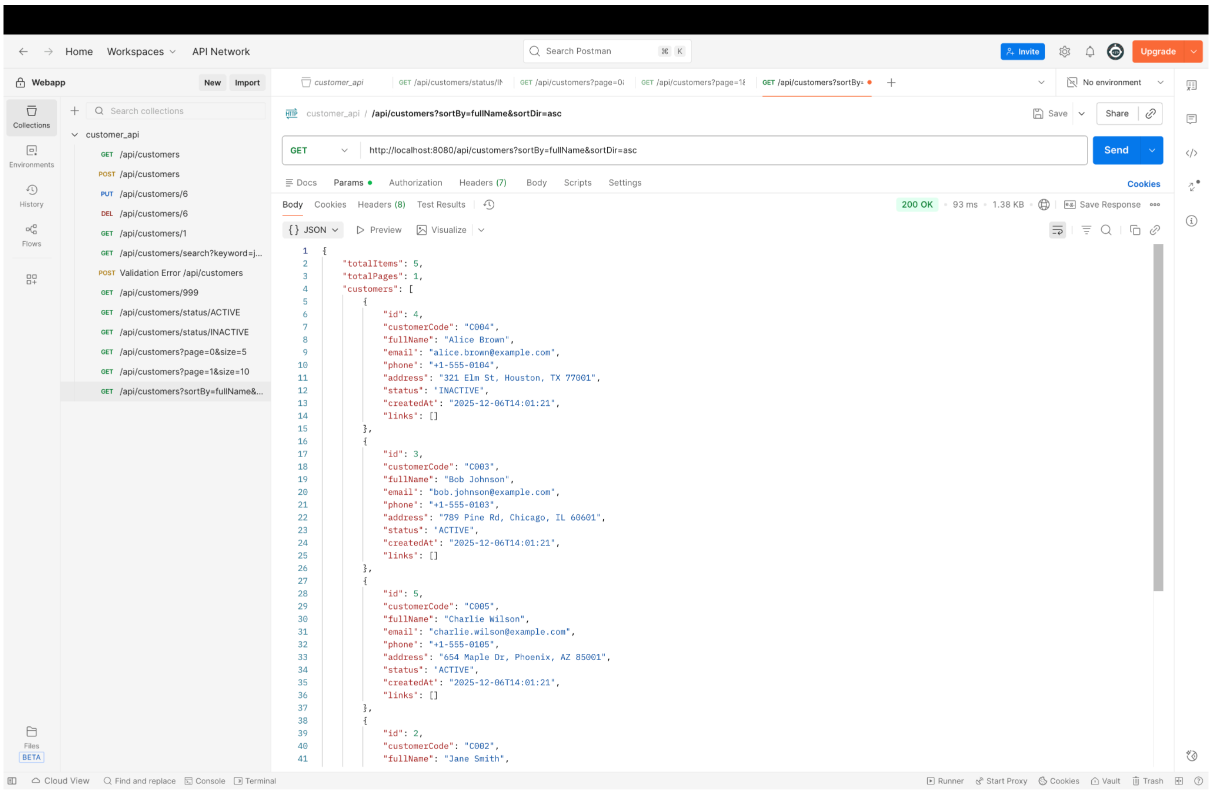
http://localhost:8080/api/customers?page=1&size=10

```json
{
    "totalItems": 5,
    "totalPages": 1,
    "customers": [],
    "currentPage": 1
}
```

When a request is sent to GET /api/customers?page=1&size=10, the processing flow starts at the Controller. The query parameters page and size are extracted via the @RequestParam annotation. These values are then used to construct a PageRequest object that determines which segment of customer data should be retrieved—specifically page 1, with 10 items per page. After extracting the parameters, the Controller invokes the Service layer by calling the method getAllCustomers(page, size). Inside the Service, a Pageable object is created using PageRequest.of(page, size) and passed to the Repository through the findAll(pageable) method. The Repository executes a paginated SQL SELECT query that fetches only the records belonging to page 1 based on the specified page size. The Repository returns the result as a Page<Customer>, which includes both the Customer entities and pagination metadata. The Service then converts the retrieved Customer entities into CustomerResponseDTO objects while preserving the structure of the paginated data. Once the conversion is complete, the DTO page is returned to the Controller. Finally, the Controller sends the paginated customer data back to the client with an HTTP status 200 OK.

**Task 6.2: Add Sorting**

**GET /api/customers?sortBy=fullName&sortDir=asc**

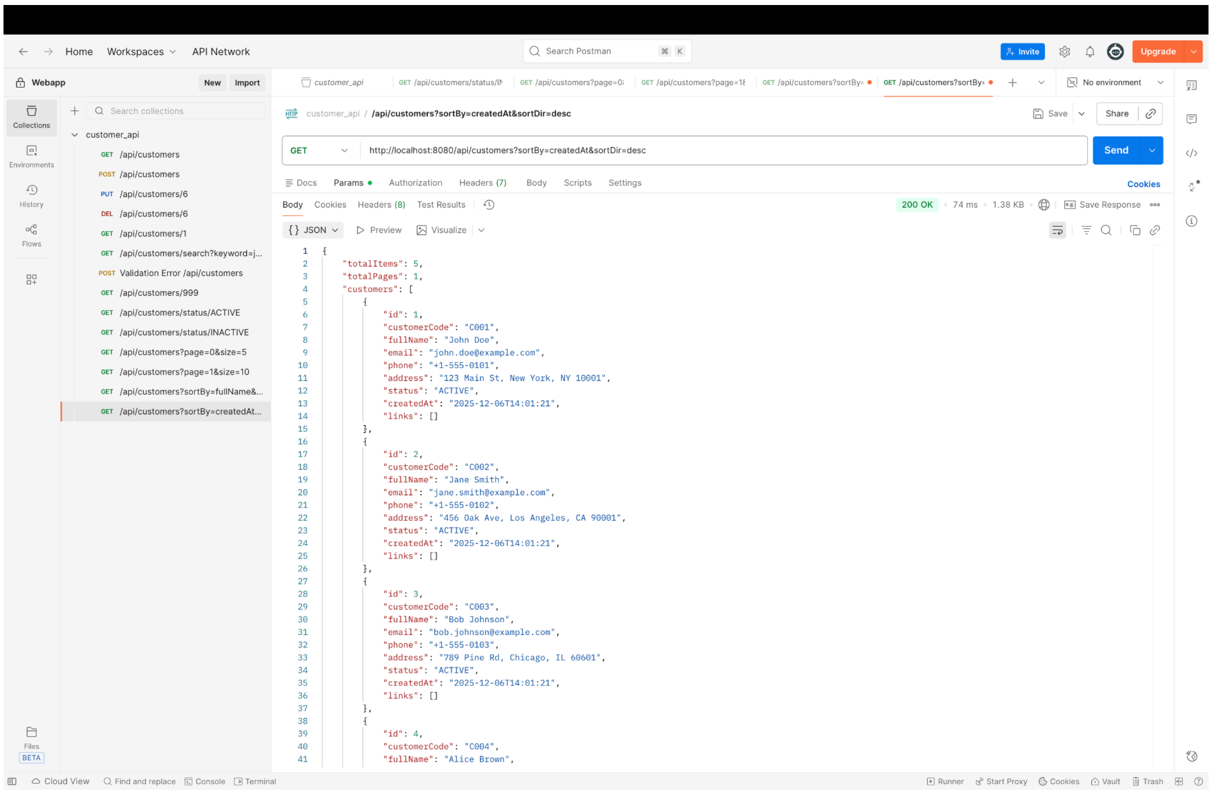

GET http://localhost:8080/api/customers?sortBy=fullName&sortDir=asc

200 OK · 93 ms · 1.38 KB

```
{
    "totalItems": 5,
    "totalPages": 1,
    "customers": [
        {
            "id": 4,
            "customerCode": "C004",
            "fullName": "Alice Brown",
            "email": "alice.brown@example.com",
            "phone": "+1-555-0104",
            "address": "321 Elm St, Houston, TX 77001",
            "status": "INACTIVE",
            "createdAt": "2025-12-06T14:01:21",
            "links": []
        },
        {
            "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": "2025-12-06T14:01:21",
            "links": []
        },
        {
            "id": 5,
            "customerCode": "C005",
            "fullName": "Charlie Wilson",
            "email": "charlie.wilson@example.com",
            "phone": "+1-555-0105",
            "address": "654 Maple Dr, Phoenix, AZ 85001",
            "status": "ACTIVE",
            "createdAt": "2025-12-06T14:01:21",
            "links": []
        },
        {
            "id": 2,
            "customerCode": "C002",
            "fullName": "Jane Smith",
```

When a request is sent to GET /api/customers?sortBy=fullName&sortDir=asc, the processing begins in the Controller. At this layer, the query parameters sortBy and sortDir are extracted using the @RequestParam annotation. These parameters determine which field the results should be sorted by (in this case, fullName) and whether the sorting should be ascending (asc) or descending. After retrieving the sorting parameters, the Controller invokes the Service method getAllCustomers(sortBy, sortDir). Inside the Service, a Sort object is constructed using Sort.by(sortBy) combined with the direction derived from sortDir, such as Sort.Direction.ASC. This sorting configuration is then applied when calling the Repository method findAll(sort). The Repository executes a SQL SELECT query that includes an ORDER BY clause, ordering the results based on the requested field and direction—for example, ORDER BY fullName ASC. The Repository returns a sorted list of Customer entities to the Service layer. The Service then converts these entities into CustomerResponseDTO objects to ensure the response matches the required API format. Once the conversion is complete, the list of DTOs is returned to the Controller. Finally, the Controller sends the sorted customer list back to the client along with an HTTP status 200 OK.

**GET /api/customers?sortBy=createdAt&sortDir=desc**


GET http://localhost:8080/api/customers?sortBy=createdAt&sortDir=desc

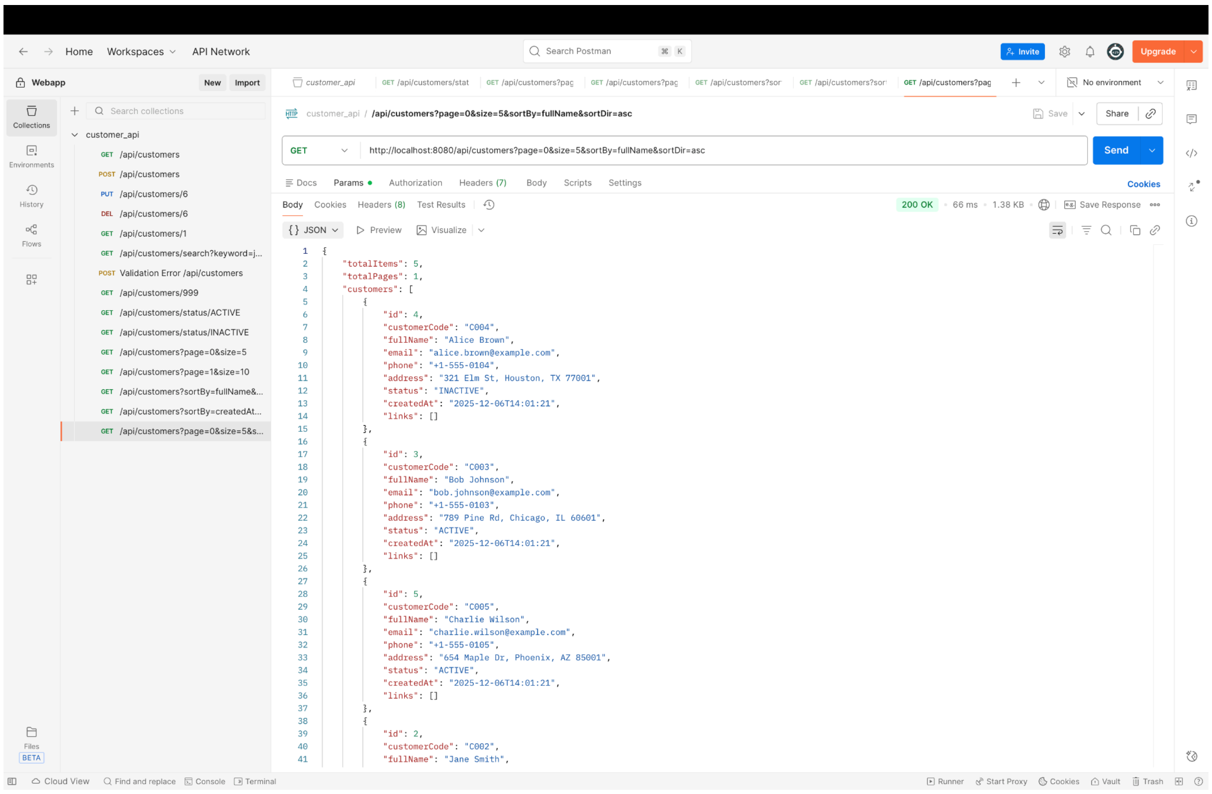
200 OK · 74 ms · 1.38 KB

```
{
    "totalItems": 5,
    "totalPages": 1,
    "customers": [
        {
            "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": "2025-12-06T14:01:21",
            "links": []
        },
        {
            "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": "2025-12-06T14:01:21",
            "links": []
        },
        {
            "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": "2025-12-06T14:01:21",
            "links": []
        },
        {
            "id": 4,
            "customerCode": "C004",
            "fullName": "Alice Brown",
```

When a request is made to GET /api/customers?sortBy=createdAt&sortDir=desc, the handling process begins in the Controller. At this layer, the query parameters sortBy and sortDir are extracted from the URL using the @RequestParam annotation. These parameters specify that the customer list should be sorted by the createdAt field and that the sorting direction should be descending (desc). After retrieving these values, the Controller calls the Service method getAllCustomers(sortBy, sortDir). Inside the Service, a Sort object is constructed using Sort.by(sortBy) combined with the direction derived from sortDir, typically Sort.Direction.DESC. This sort configuration is then passed to the Repository through the findAll(sort) method. The Repository executes a SQL SELECT query that includes an ORDER BY clause, such as ORDER BY createdAt DESC, ensuring that the most recently created customers appear first. The Repository returns a sorted list of Customer entities to the Service. The Service then converts the Customer entities into CustomerResponseDTO objects, ensuring that only the necessary response fields are included. After the conversion is complete, the list of DTOs is returned to the Controller. Finally, the Controller sends the sorted list of customers back to the client with an HTTP status 200 OK.

**Task 6.3: Combine Pagination and Sorting**

**GET /api/customers?page=0&size=5&sortBy=fullName&sortDir=asc**



customer_api / /api/customers?page=0&size=5&sortBy=fullName&sortDir=asc

```
GET http://localhost:8080/api/customers?page=0&size=5&sortBy=fullName&sortDir=asc
```

200 OK · 66 ms · 1.38 KB

```json
{
    "totalItems": 5,
    "totalPages": 1,
    "customers": [
        {
            "id": 4,
            "customerCode": "C004",
            "fullName": "Alice Brown",
            "email": "alice.brown@example.com",
            "phone": "+1-555-0104",
            "address": "321 Elm St, Houston, TX 77001",
            "status": "INACTIVE",
            "createdAt": "2025-12-06T14:01:21",
            "links": []
        },
        {
            "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": "2025-12-06T14:01:21",
            "links": []
        },
        {
            "id": 5,
            "customerCode": "C005",
            "fullName": "Charlie Wilson",
            "email": "charlie.wilson@example.com",
            "phone": "+1-555-0105",
            "address": "654 Maple Dr, Phoenix, AZ 85001",
            "status": "ACTIVE",
            "createdAt": "2025-12-06T14:01:21",
            "links": []
        },
        {
            "id": 2,
            "customerCode": "C002",
            "fullName": "Jane Smith",
```

When a request is sent to GET /api/customers?page=0&size=5&sortBy=fullName&sortDir=asc, the request handling begins in the Controller. At this layer, the query parameters page, size, sortBy, and sortDir are extracted using the @RequestParam annotation. These parameters indicate that the client is requesting page 0, with 5 customers per page, sorted by the fullName field in ascending order. The Controller then calls the Service method getAllCustomers(page, size, sortBy, sortDir), passing along all extracted parameters. Inside the Service, a Sort object is created using Sort.by(sortBy) combined with the direction specified by sortDir, such as Sort.Direction.ASC. This sort configuration is then combined with pagination by constructing a Pageable object using PageRequest.of(page, size, sort). The Service forwards this pageable object to the Repository via the findAll(pageable) method. The Repository executes a SQL SELECT query that incorporates both pagination and sorting through clauses like ORDER BY fullName ASC and LIMIT/OFFSET based on the page and size parameters. The Repository returns a Page<Customer> containing the Customer entities and pagination metadata. The Service then converts the returned Customer entities into CustomerResponseDTO objects, while preserving pagination information if required. Once the transformation is complete, the paginated and sorted DTO data is returned to the Controller. Finally, the Controller responds to the client with the structured result and an HTTP status 200 OK.

**EXERCISE 7: PARTIAL UPDATE WITH PATCH**

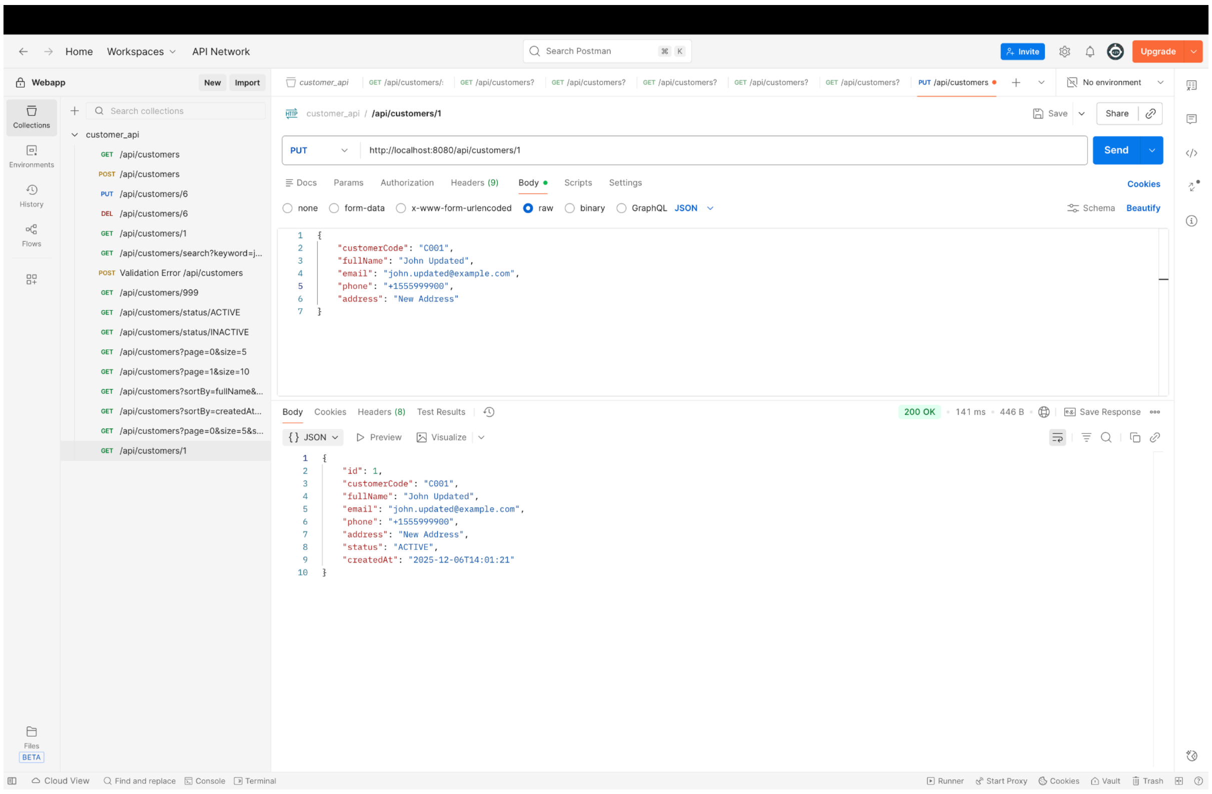
**Task 7.1: Create Update DTO**

**Task 7.2: Implement PATCH Endpoint**

**Task 7.3: Test PATCH vs PUT**

**PUT /api/customers/1**

```
{
    "customerCode": "C001",
    "fullName": "John Updated",
    "email": "john.updated@example.com",
    "phone": "+1-555-9999",
    "address": "New Address"
}
```



PUT http://localhost:8080/api/customers/1

```
{
    "customerCode": "C001",
    "fullName": "John Updated",
    "email": "john.updated@example.com",
    "phone": "+1555999900",
    "address": "New Address"
}
```

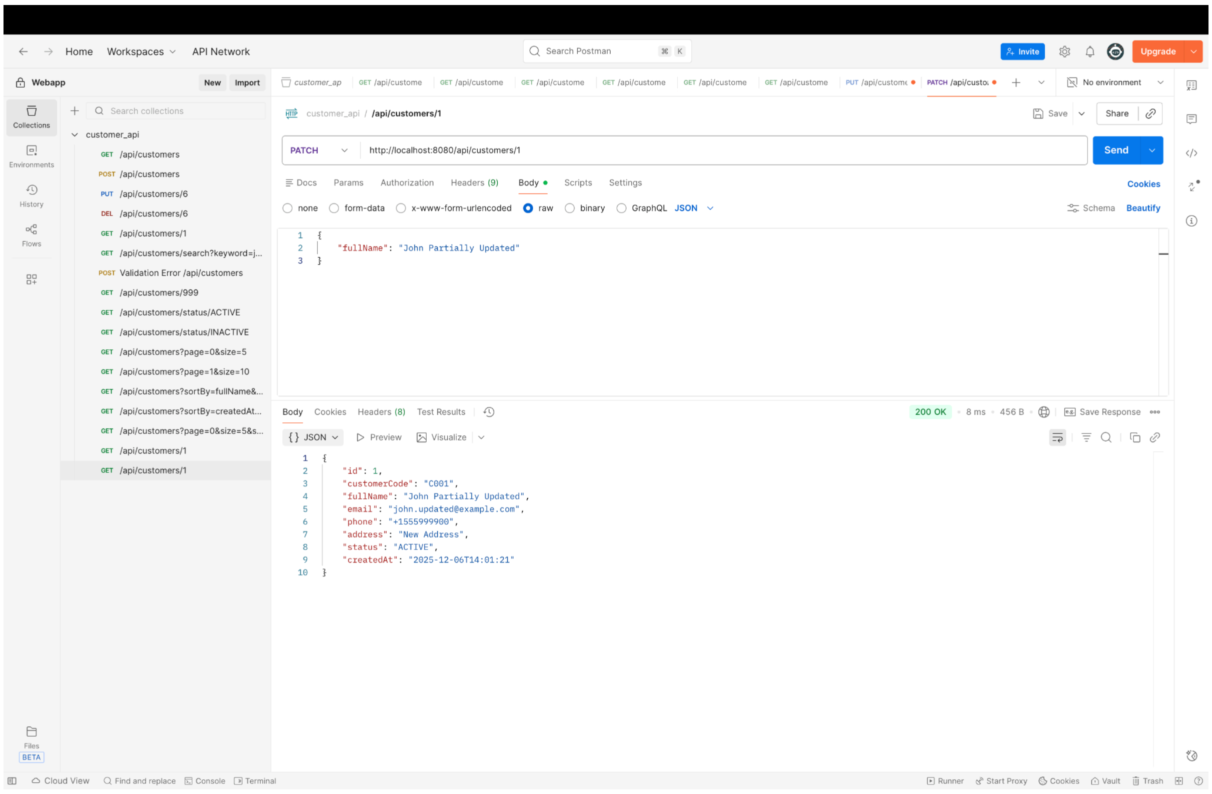
200 OK · 141 ms · 446 B

```
{
    "id": 1,
    "customerCode": "C001",
    "fullName": "John Updated",
    "email": "john.updated@example.com",
    "phone": "+1555999900",
    "address": "New Address",
    "status": "ACTIVE",
    "createdAt": "2025-12-06T14:01:21"
}
```

Home
Workspaces
API Network
Search Postman
Invite
Upgrade
Webapp
New
Import
Collections
Environments
History
Flows
Files BETA
customer_api
GET /api/customers
POST /api/customers
PUT /api/customers/6
DEL /api/customers/6
GET /api/customers/1
GET /api/customers/search?keyword=j...
POST Validation Error /api/customers
GET /api/customers/999
GET /api/customers/status/ACTIVE
GET /api/customers/status/INACTIVE
GET /api/customers?page=0&size=5
GET /api/customers?page=1&size=10
GET /api/customers?sortBy=fullName&...
GET /api/customers?sortBy=createdAt...
GET /api/customers?page=0&size=5&s...
GET /api/customers/1
PUT /api/customers
No environment
customer_api / /api/customers/1
Save
Share
Send
Docs Params Authorization Headers (9) Body Scripts Settings
Cookies
none form-data x-www-form-urlencoded raw binary GraphQL JSON
Schema
Beautify
Body Cookies Headers (8) Test Results
Save Response
JSON Preview Visualize
Cloud View Find and replace Console Terminal
Runner Start Proxy Cookies Vault Trash

When a request is sent to PUT /api/customers/1, the processing begins at the Controller. The path variable 1 is extracted from the URL using the @PathVariable annotation, and the updated customer information is received from the request body through @RequestBody. The Controller then calls the Service method updateCustomer(id, requestDTO), passing the extracted ID along with the incoming update data. Inside the Service, the method first checks whether the customer exists by calling the Repository’s findById(id) method. This triggers a SQL SELECT query to retrieve the customer with ID = 1. If the customer does not exist, the Service throws a ResourceNotFoundException. If the record is found, the Service updates the existing Customer entity with the new values provided in the request. After updating the fields, the Service calls the Repository’s save(customer) method, which executes an SQL UPDATE query to persist the changes in the database. Once the entity is successfully saved, the updated Customer is converted into a CustomerResponseDTO to ensure only appropriate response data is returned. The Service sends this DTO back to the Controller, and finally, the Controller responds to the client with the updated customer data along with an HTTP status 200 OK.

**PATCH /api/customers/1**
**{**
**"fullName": "John Partially Updated"**
**}**


customer_api / /api/customers/1

PATCH http://localhost:8080/api/customers/1

```
{
  "fullName": "John Partially Updated"
}
```

200 OK • 8 ms • 456 B

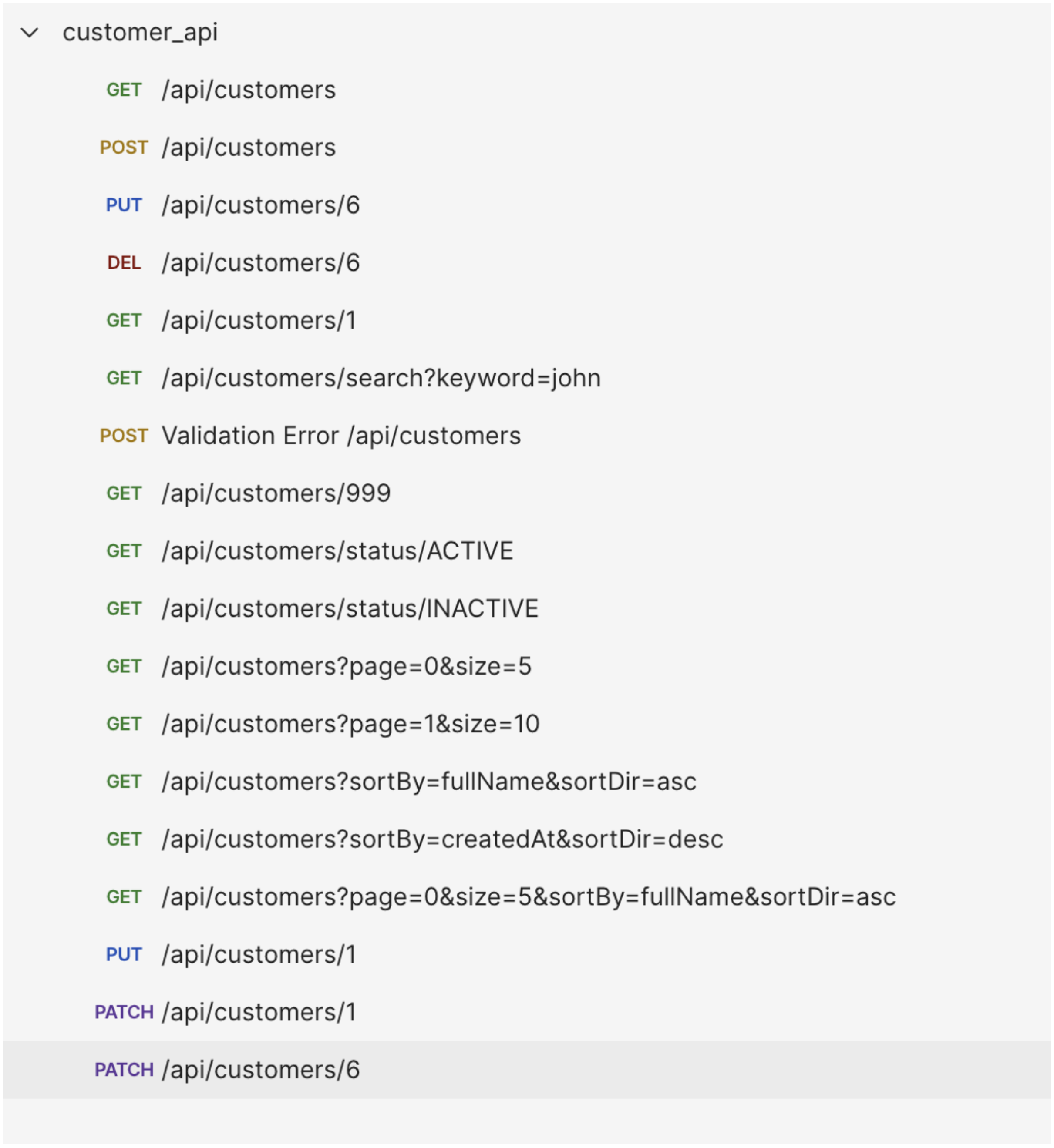
```
{
    "id": 1,
    "customerCode": "C001",
    "fullName": "John Partially Updated",
    "email": "john.updated@example.com",
    "phone": "+1555999900",
    "address": "New Address",
    "status": "ACTIVE",
    "createdAt": "2025-12-06T14:01:21"
}
```

When a request is sent to PATCH /api/customers/1, the processing flow begins at the Controller. The path variable 1 is extracted from the URL using the @PathVariable annotation, and the partial update data is received from the request body through @RequestBody. Unlike a full update, a PATCH request typically contains only the fields that need to be changed. The Controller then forwards the ID and the partial update payload to the Service layer by calling the method patchCustomer(id, requestDTO).

Inside the Service, the method first checks if the customer exists by invoking the Repository's findById(id) method. This triggers a SQL SELECT query to retrieve the customer with ID = 1. If the customer cannot be found, the Service throws a ResourceNotFoundException. If the record exists, the Service applies only the fields present in the PATCH request to the existing Customer entity—leaving all other fields unchanged. After updating the necessary fields, the Service calls the Repository's save(customer) method, which executes an SQL UPDATE query to persist the modified data.

Once the updated entity is successfully saved, the Service converts the Customer entity into a CustomerResponseDTO, ensuring that the response structure meets the API design. The Service returns this DTO back to the Controller. Finally, the Controller sends the updated customer information to the client with an HTTP status 200 OK.

**EXERCISE 8: API DOCUMENTATION**
**Task 8.1: Create Postman Collection (4 points)**



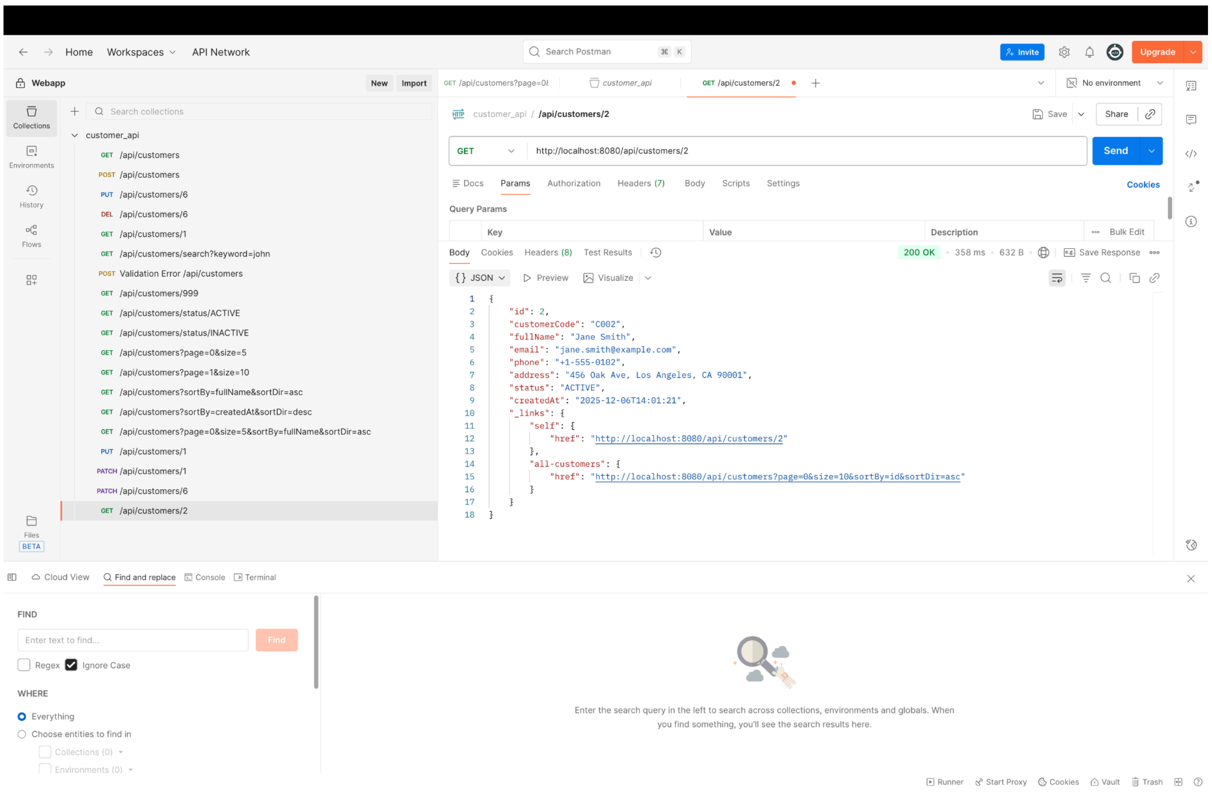
customer_api

GET /api/customers
POST /api/customers
PUT /api/customers/6
DEL /api/customers/6
GET /api/customers/1
GET /api/customers/search?keyword=john
POST Validation Error /api/customers
GET /api/customers/999
GET /api/customers/status/ACTIVE
GET /api/customers/status/INACTIVE
GET /api/customers?page=0&size=5
GET /api/customers?page=1&size=10
GET /api/customers?sortBy=fullName&sortDir=asc
GET /api/customers?sortBy=createdAt&sortDir=desc
GET /api/customers?page=0&size=5&sortBy=fullName&sortDir=asc
PUT /api/customers/1
PATCH /api/customers/1
PATCH /api/customers/6

## Task 8.2: Document API Responses

## BONUS 2: HATEOAS Links

### GET /api/customers/2



customer_api / /api/customers/2

GET http://localhost:8080/api/customers/2

Collections: customer_api

- GET /api/customers
- POST /api/customers
- PUT /api/customers/6
- DEL /api/customers/6
- GET /api/customers/1
- GET /api/customers/search?keyword=john
- POST Validation Error /api/customers
- GET /api/customers/999
- GET /api/customers/status/ACTIVE
- GET /api/customers/status/INACTIVE
- GET /api/customers?page=0&size=5
- GET /api/customers?page=1&size=10
- GET /api/customers?sortBy=fullName&sortDir=asc
- GET /api/customers?sortBy=createdAt&sortDir=desc
- GET /api/customers?page=0&size=5&sortBy=fullName&sortDir=asc
- PUT /api/customers/1
- PATCH /api/customers/1
- PATCH /api/customers/6
- GET /api/customers/2

200 OK · 358 ms · 632 B

```json
{
    "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": "2025-12-06T14:01:21",
    "_links": {
        "self": {
            "href": "http://localhost:8080/api/customers/2"
        },
        "all-customers": {
            "href": "http://localhost:8080/api/customers?page=0&size=10&sortBy=id&sortDir=asc"
        }
    }
}
```

When a request is sent to GET /api/customers/2, the processing begins at the Controller. The path variable 2 is extracted from the URL using the @PathVariable annotation. The Controller then calls the Service method getCustomerById(id), passing the extracted ID. Inside the Service, the method delegates to the Repository by invoking findById(id), which triggers a SQL SELECT query to retrieve the Customer entity with ID = 2. If the customer does not exist, the Service throws a ResourceNotFoundException. If the customer is found, the Service converts the entity into a CustomerResponseDTO.

At this stage—unlike a standard GET endpoint—the Service or Controller enriches the returned DTO with HATEOAS links. These links help the client navigate the API more easily by providing related hyperlinks directly in the response. Typically, links such as "self" (link to the current resource), "update" (link to update this customer), and "delete" (link to remove this customer) are added using Spring HATEOAS utilities like linkTo and methodOn. After the DTO is wrapped with its associated HATEOAS links, it is returned to the Controller.

Finally, the Controller sends the HATEOAS-enhanced response back to the client with an HTTP status 200 OK, allowing the client not only to view the customer with ID = 2 but also to discover additional API actions directly from the response.