CustomerRestController.java:

- Incoming HTTP requests hit the controller class (@RestController, @RequestMapping("/api/customers")).
- Each endpoint method (@GetMapping, @PostMapping, @PutMapping, @DeleteMapping) extracts inputs via @PathVariable, @RequestParam, or @RequestBody (e.g., getCustomerById(Long id)).
- The controller forwards all logic to customerService (e.g., customerService.getAllCustomers(), customerService.createCustomer(requestDTO)).
- Service returns DTO results (CustomerResponseDTO or List<CustomerResponseDTO>), which are wrapped in ResponseEntity (e.g., ResponseEntity.ok(customers)).
- The method returns the HTTP response with proper status codes (e.g., HttpStatus.CREATED for POST, ok() for others).

CustomerRequestDTO.java:

- Client sends JSON → Spring maps it into CustomerRequestDTO via @RequestBody (controller calls like createCustomer(@Valid @RequestBody CustomerRequestDTO requestDTO)).
- Validation runs automatically because of @Valid: annotations such as @NotBlank, @Size, @Email, @Pattern check fields (customerCode, fullName, email, phone, address).
- If validation fails, Spring returns 400 with messages defined in the annotations (e.g., "Customer code is required").
- If valid, the DTO is passed to customerService methods (e.g., createCustomer(requestDTO)), acting as the input model for business logic.
- Getters provide values for entity creation, and the controller never exposes this DTO directly to clients (only CustomerResponseDTO is returned).

CustomerResponseDTO.java:

- After service methods finish processing, they build a CustomerResponseDTO (using constructor or setters) to represent returned customer data.
- Fields like id, customerCode, fullName, email, phone, address, status, and createdAt are filled from the Customer entity before sending to the controller.
- Controller methods return this DTO directly to the client (e.g., return ResponseEntity.ok(customer) in getCustomerById).

- This class contains **no validation**—it is purely an output model with getters used by JSON serialization.
- Spring converts the DTO into JSON and sends it back to the frontend as the API response.

ErrorResponseDTO.java:

- When an exception occurs, the global exception handler creates an ErrorResponseDTO (e.g., using the constructor new ErrorResponseDTO(status, error, message, path)).
- The DTO automatically sets timestamp = LocalDateTime.now() in both constructors to record the error time.
- Fields (status, error, message, path, details) store HTTP status code, error type, readable message, request path, and additional validation errors.
- The handler fills optional details for cases like validation failures (List<String>).
- Controller returns this DTO as a JSON error response, ensuring a consistent error format for the frontend.

Customer.java:

- This class maps to the database table customers using @Entity and @Table(name = "customers").
- Each field is mapped to a column (@Column), including constraints like unique, nullable, and lengths (e.g., customerCode, email, fullName).
- status uses an enum stored as a string because of @Enumerated(EnumType.STRING).
- Lifecycle callbacks @PrePersist and @PreUpdate automatically set createdAt and updatedAt timestamps before insert and update.
- Service layer creates or updates this entity using setters, and Spring Data JPA persists it to the database.

CustomerStatus.java:

- CustomerStatus defines two allowed values: ACTIVE and INACTIVE.
- The Customer entity stores this enum using @Enumerated(EnumType.STRING), meaning it is saved as text in the database.
- Default status is CustomerStatus.ACTIVE as set in the entity field.
- The service layer updates the status by calling customer.setStatus(CustomerStatus.ACTIVE or INACTIVE).

- API endpoints like /api/customers/status/{status} use this enum to filter customers by their status.

DuplicateResourceException.java:

- This is a custom runtime exception extending RuntimeException, used to signal duplicate data errors (e.g., existing email or customerCode).
- It provides two constructors: one accepting only a message, and another accepting both a message and a cause.
- The service layer throws this exception when detecting duplicates before saving a Customer (e.g., throw new DuplicateResourceException("Email already exists")).
- The global exception handler catches it and converts it into an ErrorResponseDTO.
- The controller then returns a structured error JSON to the client with a proper HTTP status code (usually 409 Conflict).

ResourceNotFoundException.java:

- This class extends RuntimeException, representing cases where a requested customer cannot be found in the database.
- It provides two constructors: one with a message, and one with both message and cause (super(message, cause)).
- The service layer throws this exception when methods like getCustomerById(id) fail to locate a Customer entity.
- The global exception handler catches it and converts it into an ErrorResponseDTO with an appropriate HTTP status (usually 404).
- The controller then returns this structured error response back to the client in JSON format.

GlobalExceptionHandler.java:

- @RestControllerAdvice makes this class catch exceptions thrown anywhere in your controllers or service layer.
- Each @ExceptionHandler maps a specific exception (e.g., ResourceNotFoundException, DuplicateResourceException, MethodArgumentNotValidException) to an HTTP status.
- For each error, it builds an ErrorResponseDTO using details like status, error type, message, and request path (request.getDescription(false).replace("uri=", "")).
- Validation errors iterate through FieldError list and add them to error.setDetails(details) for 400 responses.

- All unhandled errors fall into the generic Exception handler, returning a 500 Internal Server Error wrapped as an ErrorResponseDTO.

CustomerRepository.java:

- CustomerRepository extends JpaRepository<Customer, Long>, giving CRUD operations automatically (save, findById, deleteById, etc.).
- Custom finder methods like findByCustomerCode() and findByEmail() return Optional<Customer> for lookup during validation and updates.
- Methods existsByCustomerCode() and existsByEmail() help detect duplicates before inserting data.
- findByStatus(String status) retrieves customers using the status column (mapped from CustomerStatus).
- The @Query method searchCustomers() performs a case-insensitive search across fullName, email, and customerCode using JPQL with LIKE and CONCAT.

CustomerService.java:

- CustomerService defines the contract for all customer-related operations used by the controller (getAllCustomers(), getCustomerById(), createCustomer(), etc.).
- Methods accept CustomerRequestDTO for input and return CustomerResponseDTO or lists, ensuring controllers never interact directly with entities.
- createCustomer() and updateCustomer() handle validation, duplication checks, and entity saving in the implementation class.
- searchCustomers(keyword) and getCustomersByStatus(status) delegate filtering logic to the repository layer.
- The controller depends on this interface, while the actual logic lives in the CustomerServiceImpl implementation.

CustomerServiceImpl.java:

- Each service method interacts with CustomerRepository for database operations (e.g., findAll(), findById(), save(), deleteById()), converting entities to CustomerResponseDTO using convertToResponseDTO().
- getCustomerById() and deleteCustomer() throw ResourceNotFoundException when the repository returns empty results.
- createCustomer() checks duplicates using existsByCustomerCode() and existsByEmail(), then converts the request DTO into a Customer entity via convertToEntity(), and saves it.

- updateCustomer() fetches the existing customer, validates email changes for conflicts, updates fields, and saves the updated entity.
- Search and filtering methods (searchCustomers(), getCustomersByStatus()) delegate queries to repository methods and map results to response DTOs via streams.