DAY 8-9: BUILDING RESTFUL APIS

1. RESTFUL ARCHITECTURE PRINCIPLES

REST (Representational State Transfer) is an architectural style for designing networked applications.

Key constraints of REST:

1. **Client-Server**
   - Separation of concerns.
   - Client handles UI and user experience.
   - Server handles data storage, business logic, and security.
2. **Stateless**
   - Each request contains all necessary information.
   - Server does not store client session.
   - Easier to scale horizontally.
3. **Cacheable**
   - Responses must declare if they are cacheable or not.
   - Reduces client-server interactions, improves performance.
4. **Uniform Interface**
   - Consistent way to access resources.
   - Includes standard URIs, HTTP methods, media types, and status codes.
5. **Layered System**
   - Client cannot tell whether it is connected directly to the server or through intermediaries (like proxies, load balancers).
   - Helps scalability and security.
6. **Code on Demand (optional)**
   - Server can send executable code (e.g., JavaScript) to clients to extend functionality.
   - Optional constraint, rarely used in most APIs.

2. HTTP METHODS AND THEIR USAGE

HTTP Method	Description	Idempotent	Safe
GET	Retrieve resources or data	Yes	Yes
POST	Create a new resource	No	No
PUT	Update a resource entirely	Yes	No
PATCH	Partial update of a resource	No	No (depends)
DELETE	Delete a resource	Yes	No

Definitions:

- **Idempotent**: Multiple identical requests have the same effect as a single request.
- **Safe**: Does not change server state; only reads data.

Usage Examples:

- **GET /users** → Get all users
- **POST /users** → Create a new user
- **PUT /users/1** → Update the entire user with ID 1
- **PATCH /users/1** → Update selected fields of user with ID 1
- **DELETE /users/1** → Remove user with ID 1

3. HTTP STATUS CODES

Status Code	Meaning	Example Usage
200 OK	Request succeeded	Returning a resource list
201 Created	Resource successfully created	After POST /users
204 No Content	Successful, but no data returned	DELETE /users/1
400 Bad Request	Invalid request syntax or parameters	Missing required fields
401 Unauthorized	Client not authenticated	API key or token missing
403 Forbidden	Client authenticated but not allowed	Access restricted
404 Not Found	Resource does not exist	GET /users/999
409 Conflict	Duplicate resource or conflicting data	POST duplicate email
500 Internal Server Error	Server-side error	Unexpected exception in code

4. REQUEST AND RESPONSE DTOS

DTO (Data Transfer Object):

- A simple object used to transfer data between client and server.

Types of DTOs:

1. **Request DTO**

   Represents input data from client to API.

Example:

```
public class UserRequestDTO {
    private String name;
    private String email;
}
```

2. **Response DTO**

   Represents data sent from API to client.

   Example:

```
public class UserResponseDTO {
    private Long id;
    private String name;
    private String email;
    private LocalDateTime createdAt;
}
```

Advantages of DTOs:

- Decouples internal entity structure from API response.
- Prevents overexposing sensitive fields.
- Allows customizing API responses independently.

5. ENTITY ↔ DTO MAPPING

Manual Mapping:

- Copying values field by field between entities and DTOs.
- Example:

```
UserResponseDTO dto = new UserResponseDTO();
dto.setId(userEntity.getId());
dto.setName(userEntity.getName());
dto.setEmail(userEntity.getEmail());
```

Automated Mapping:

- Libraries: **MapStruct**, **ModelMapper**.
- Example using MapStruct:

```
@Mapper
public interface UserMapper {
   UserMapper INSTANCE = Mappers.getMapper(UserMapper.class);
   UserResponseDTO toResponse(UserEntity entity);
   UserEntity toEntity(UserRequestDTO dto);
}
```

Benefits:

- Reduces boilerplate code.
- Ensures consistency and maintainability.

6. EXCEPTION HANDLING IN SPRING

Global Exception Handling:

- Use **@ControllerAdvice** to handle exceptions across all controllers.
- Use **@ExceptionHandler** to handle specific exceptions.

Example:

```
@ControllerAdvice
public class GlobalExceptionHandler {
   @ExceptionHandler(ResourceNotFoundException.class)
   public ResponseEntity<String> handleNotFound(ResourceNotFoundException ex) {
      return ResponseEntity.status(HttpStatus.NOT_FOUND).body(ex.getMessage());
   }
   @ExceptionHandler(Exception.class)
   public ResponseEntity<String> handleGeneral(Exception ex) {
      return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                  .body("Something went wrong");
   }
}
```

Key points:

- Provides consistent error responses.
- Avoids duplicating try-catch in controllers.

7. API VERSIONING STRATEGIES

Purpose: Maintain backward compatibility when API changes.

1. **URI Versioning**
   - Example: /api/v1/users
   - Simple and widely used.
2. **Request Parameter Versioning**
   - Example: /users?version=1
   - Flexible, but less visible.
3. **Header Versioning**

    - Example: X-API-VERSION: 1
    - Does not clutter URI, used in some enterprise APIs.
4. **Content Negotiation Versioning**
    - Example: Accept: application/vnd.app.v1+json
    - Version is part of Accept header.

Best practice: URI versioning is simplest and easiest to manage.