**REST Architectural Constraints**

**1. Client-Server Architecture**

* Separates the **client** (user interface) from the **server** (data storage & logic).
* Separates UI and backend logic.
* Improves scalability by allowing each to evolve independently.
* Example: **A mobile app (client)** communicates with a **REST API (server).**

**2. Statelessness**

* Each request from the client to the server must contain all the necessary information.
* The server **does not store any session** **information** about the client.
* Example: No session storage in the server; authentication is handled via tokens (JWT, OAuth).

**3. Cacheability**

* Responses from the server can be cached to improve performance.
* The server should explicitly define whether a response is **cacheable or non-cacheable**.
* Example: API responses include headers like Cache-Control, ETag, or Expires to manage caching.

**4. Layered System**

* A RESTful system can have multiple layers (e.g., **load balancers, proxies, security layers**) between client and server.
* Each layer should function independently and not interfere with others.
* Example: A **reverse proxy (like Nginx)** handles API requests before reaching the backend server.

**5. Uniform Interface**

* Standardized interface for interaction between client and server.
* Enforces API design principles like:
  + **Resource-Based URLs** (/users, /orders/123).
  + **Standard HTTP Methods** (GET, POST, PUT, DELETE).
  + **Stateless communication**.
* Example: Instead of using different endpoints for different clients, a single API (/users) serves web and mobile apps.

**6. Code on Demand (Optional)**

* Allows the server to send **executable code (like JavaScript, Java Applets) to the client** for execution.
* Example: A **web API sends JavaScript code** to a browser, dynamically updating the UI.

**Exception Handling for Rest APIs**

**1. Using @ControllerAdvice for Global Exception Handling**

* @ControllerAdvice allows you to write a single exception handler for multiple controllers.
* Write custom exceptions using with extends RuntimeException class.

**2. Returning a Custom JSON Response for Exceptions**

* Instead of returning just a string, we can return a **structured JSON response** with using Custom Error Response Class using properties with message key, message description and system message details and timestamp

**3. Throwing the Exception in a Controller**

* userService.findById(id).orElseThrow(() -> new ResourceNotFoundException("User not found with id: " + id));

**4. Validation Exception Handling**

* Handles validation failures using MethodArgumentNotValidException.
* When using **@Valid** in Spring Boot, we can catch validation errors globally.
* Add validation annotations like @NotBlank and @Email in DTOs

[**@RestControllerAdvice**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestControllerAdvice.html) is the combination of both **@ControllerAdvice** and @**ResponseBody**:

@RestControllerAdvice

public class ExceptionHandlerControllerAdvice {

// Handle specific exception (e.g., Resource Not Found)

@ExceptionHandler(ResourceNotFoundException.class)

@ResponseStatus(value = HttpStatus.NOT\_FOUND)

public @ResponseBody ExceptionResponse handleResourceNotFound(final ResourceNotFoundException exception,

final HttpServletRequest request) {

ExceptionResponse error = new ExceptionResponse();

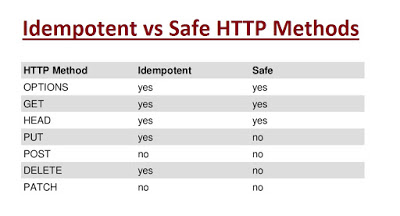
error.setErrorMessage(exception.getMessage());

error.callerURL(request.getRequestURI());

return error;}

**Idempotent:**

These are methods which are safe from multiple calls i.e. they produce the same result irrespective of how many times you call them. They change the resource in Server every time you call them, but the end result is always same

[[](https://2.bp.blogspot.com/-6JFwUOCTM7w/V02b8UY9y_I/AAAAAAAAGE0/IP-CEIcWZ34deazdz7StE71gBHXkv9dEwCLcB/s1600/Safe%2Band%2BIdempotent%2Bmethods%2Bof%2BHTTP%2Band%2BREST.jpg)](https://2.bp.blogspot.com/-6JFwUOCTM7w/V02b8UY9y_I/AAAAAAAAGE0/IP-CEIcWZ34deazdz7StE71gBHXkv9dEwCLcB/s1600/Safe%2Band%2BIdempotent%2Bmethods%2Bof%2BHTTP%2Band%2BREST.jpg)

suppose a client wants to update a resource through POST. Since **POST is not an idempotent method**, calling it multiple times may result in incorrect updates.

**Both @RequestParam and @PathVariable can be optional.**

1. Use @PathParam - when it is a mandatory item such as an Id

GET /balloon/{id}

1. Use @QueryParam - when you have the exact resource but need to filter that on some optional condition such as color, size, etc.

GET /balloon/123?color=red&size=large

1. **Path Parameters**:
   * Path parameters are a part of the URL path itself.
   * They are used to identify a specific resource or endpoint.
   * Path parameters are typically used when the parameter value is essential to the resource's identity.
   * Example: **https://api.example.com/users/{userId}**
   * In this example, **{userId}** is a path parameter, and it represents the ID of a specific user resource.
2. **Query Parameters**:
   * Query parameters are appended to the URL after a question mark **?**.
   * They are used to filter, sort, or modify the results returned by an endpoint.
   * Query parameters are optional and can be used for pagination, searching, filtering, etc.
   * Example: **https://api.example.com/users?status=active&sort=desc**
   * In this example, **status** and **sort** are query parameters, and they specify the filter condition and sorting order for the list of users.
3. **Request Parameters**:
   * Request parameters are included in the body of an HTTP request, typically in POST or PUT requests.
   * They are used to send additional data to the server when creating, updating, or deleting a resource.
   * Request parameters are commonly used when dealing with complex data structures or large amounts of data.
   * Example (in JSON format):
   * {
   * "name": "John Doe",
   * "email": "john@example.com",
   * "age": 30
   * }
   * In this example, **name**, **email**, and **age** are request parameters sent in the body of a POST request to create a new user.

**@Controller**

* Used for **traditional MVC** applications.
* Returns **views** (like JSP, Thymeleaf) instead of raw data.
* Requires @ResponseBody to return JSON or XML in RESTful responses.
* **Without @ResponseBody**, it assumes you're returning a **view name**.

**@RestController**

* Specialized version of @Controller.
* Automatically adds @ResponseBody to all methods.
* Used to build **RESTful APIs** returning JSON or XML directly.
* Automatically **serializes the response** to JSON/XML.
* No need for @**ResponseBody** since it's implied.

**SOA: SOA** is an architectural pattern where applications are built as a collection of loosely coupled **services** that communicate via **protocols like SOAP, RPC, or REST**.

* Uses SOAP (Simple Object Access Protocol) or REST
* Services are **stateful** or **stateless**
* Uses **ESB (Enterprise Service Bus)** for communication
* Supports multiple communication protocols (SOAP, MQ, HTTP, FTP)
* More suited for **enterprise-level, complex applications**
* When to Use: Stateful transactions are required
* You need **complex enterprise applications**
* You require **multiple protocols (SOAP, MQ, FTP, HTTP)**

| **Feature** | **SOA (Service-Oriented Architecture)** | **RESTful APIs** |
| --- | --- | --- |
| Communication | SOAP, REST, RPC, JMS, MQ | HTTP (REST) |
| Protocol | Uses SOAP (XML-based), messaging queues | Uses REST over HTTP (JSON/XML) |
| State Management | Can be stateful or stateless | Always stateless |
| Message Format | XML (SOAP) or JSON | JSON, XML |
| Scalability | More complex due to ESB | Lightweight, easy to scale |
| Performance | Slower (SOAP messages are heavy) | Faster (Uses HTTP methods directly) |
| Security | WS-Security, SSL for authentication | OAuth2, JWT, API keys |
| Use Case | Enterprise applications (Banking, Telecom) | Web & mobile APIs |

**HTTP Status codes**

**1XX**

This is the **Informational status code** used to indicate that the **client request** is **received**and is **under process**.

This is just like an **acknowledgement to the client**.

**2XX**

This is the **success status code**.

This is used to indicate that the requestfrom the clientis receivedand processed successfully.

**3XX**

This is the **Redirection success code**.

This is used to indicate that further action is required from **client side** to **fulfill**the **request**.

It may be **redirected automatically** if the **second request** is **GET**or **HEAD**.

**4XX**

This is the **Client error status code**.

This is used to indicate that there is an **error**in the **client request**.

If client is sending **invalid data**, then it can lead to**client error** with this **status code**.

**5XX**

This is the **Server Error status code**.

This is used to indicate that the **server is failed** to process the **valid request**.

This status code indicates that the **request is valid** but **server is not capable of processing** it due to some of its **internal errors**.

**Difference between URI and URL?**

**URL:** URLs always start with a protocol (http) and usually contain information such as the network host name (example.com) and often a document path (/foo/mypage.html). URLs may have query parameters and fragment identifiers.

* Ex:http://example.com/mypage.html

**URI:**

Basically URIs are a superset of URLs and of URNs. A URI identifies a resource either by location, or a name, or both.

**Internal process for spring boot rest api**

**🔄 Complete Request Flow Summary**

1. **Client** sends an HTTP request (e.g., GET /users/1).
2. **Embedded server (Tomcat)** receives the request.
3. **DispatcherServlet** determines the correct controller.
4. **Controller method** executes the request.
5. **Service layer** processes business logic.
6. **Repository layer** retrieves data from the database.
7. **Data is returned** to the controller.
8. **Jackson serializes** Java object to JSON.
9. **Response is sent** to the client.

**1. Client Sends an HTTP Request**

* A client (e.g., browser, Postman, frontend app) sends an **HTTP request** (GET, POST, PUT, DELETE) to a Spring Boot application.
* Example request:

GET /users/1 HTTP/1.1

Host: localhost:8080

**2️ Embedded Server Receives Request**

* Spring Boot applications run on an **embedded server** (Tomcat, Jetty, or Undertow).
* The **server listens** for incoming HTTP requests and passes them to Spring’s DispatcherServlet.

✅ **Example: Default embedded server (Tomcat) in application.properties**

properties

CopyEdit

server.port=8080

server.servlet.context-path=/api

**3️ DispatcherServlet Handles the Request**

* DispatcherServlet is the **central controller** in Spring MVC.
* It performs:
  + **Routing** – Identifies which controller method should handle the request.
  + **Request Parsing** – Converts JSON/XML data into Java objects.
  + **Response Handling** – Converts Java objects back to JSON/XML.

**4️ Request Mapping to Controller**

* The request is matched to a **controller method** using @RequestMapping, @GetMapping, @PostMapping, etc.

✅ **Example: Controller with Request Mapping**

@RestController

@RequestMapping("/users")

public class UserController {

@GetMapping("/{id}")

public ResponseEntity<User> getUser(@PathVariable Long id) {

return ResponseEntity.ok(userService.getUserById(id));

}

}

* Here, Spring matches GET /users/1 to the getUser method.

**5️ Request Processing in Service Layer**

* The controller calls a **service layer** method for business logic.

✅ **Example: Service Layer**

@Service

public class UserService {

public User getUserById(Long id) {

return userRepository.findById(id).orElseThrow(() -> new UserNotFoundException("User not found"));

}

}

**6️ Data Retrieval from Repository (Database)**

* The service interacts with the **repository layer (DAO)** to fetch data from a database.
* Spring Boot uses **Spring Data JPA** to handle database operations.

✅ **Example: Repository Layer**

@Repository

public interface UserRepository extends JpaRepository<User, Long> {

}

* The repository queries the database and retrieves the **User** entity.

**7️ Response Sent Back to DispatcherServlet**

* The **User object** is returned to the controller.
* The controller wraps it in an ResponseEntity and sends it back to the DispatcherServlet.

**8️ Data Serialization (Java Object → JSON)**

* The **Jackson Library (default in Spring Boot)** converts Java objects into JSON format.
* Example JSON Response:

{

"id": 1,

"name": "John Doe",

"email": "john@example.com"

}

✅ **Example: Object Conversion using Jackson**

@JsonInclude(JsonInclude.Include.NON\_NULL)

@JsonIgnoreProperties(ignoreUnknown = true)

public class User {

private Long id;

private String name;

private String email;

}

**9️ Response Sent to Client**

* The final JSON response is sent back to the client.
* Example HTTP response:

HTTP/1.1 200 OK

Content-Type: application/json

{

"id": 1,

"name": "John Doe",

"email": "john@example.com"

}

Designing RESTful APIs involves following certain principles to ensure that your API is easy to use, scalable, maintainable, and adheres to the REST architecture style. Here are the key REST API design principles:

**1. Use of HTTP Methods**

* **GET**: Retrieve a resource or a collection of resources.
* **POST**: Create a new resource.
* **PUT**: Update an existing resource or create one if it doesn’t exist (idempotent).
* **PATCH**: Partially update a resource.
* **DELETE**: Remove a resource.

**2. Statelessness**

* **No Server-Side State**: Each API request should contain all the information needed for the server to understand and process the request. The server should not store any state between requests. This makes the API scalable and easier to manage.

**3. Resource-Based URIs**

* **Nouns, Not Verbs**: Use nouns to represent resources in URIs. For example, /users, /orders, /products, not /getUsers or /createUser.
* **Hierarchical URIs**: Structure URIs hierarchically to represent resource relationships. For example, /users/{userId}/orders indicates orders belonging to a specific user.

**4. HTTP Status Codes**

* **Standard Codes**: Use standard HTTP status codes to indicate the result of the request. Some common codes include:
  + 200 OK: Request succeeded.
  + 201 Created: Resource successfully created.
  + 204 No Content: Request succeeded but no content to return.
  + 400 Bad Request: The request was invalid or cannot be served.
  + 401 Unauthorized: Authentication is required.
  + 403 Forbidden: The server understood the request but refuses to authorize it.
  + 404 Not Found: The requested resource was not found.
  + 500 Internal Server Error: Generic server error.

**5. Consistency**

* **Uniform Interface**: Follow consistent patterns for URIs, HTTP methods, and response formats across the API. Consistency helps developers predict how the API will behave and makes it easier to use.

**6. Versioning**

* **Version in the URI**: Include the version number in the URI, such as /api/v1/users, to maintain backward compatibility when making changes to the API.
* **Header Versioning**: Alternatively, use custom headers for versioning, although URI versioning is more explicit and easier to manage.

**7. Stateless Authentication**

* **Use Tokens**: Implement stateless authentication using tokens such as JWT (JSON Web Tokens) instead of sessions. This keeps the API stateless and scalable.
* **OAuth2**: Use OAuth2 for authorization in more complex scenarios, especially for APIs that need to interact with third-party services.

**8. Data Format and Serialization**

* **JSON by Default**: Use JSON as the default data format for request and response bodies since it's widely supported and easy to parse. XML can be used if necessary.
* **Content Negotiation**: Allow clients to specify the format they prefer using the Accept header (application/json, application/xml, etc.).

**9. HATEOAS (Hypermedia as the Engine of Application State)**

* **Hypermedia Links**: Include links in your responses that guide the client on what they can do next. For example, after retrieving a list of users, provide links to individual user details or actions like updating or deleting a user.

**10. Error Handling**

* **Clear Error Messages**: Return meaningful error messages that help developers understand what went wrong. Include error codes, descriptions, and possibly links to documentation.
* **Consistent Error Format**: Use a consistent error response structure, such as:

json

Copy code

{

"error": "Bad Request",

"message": "The 'email' field is required.",

"code": 400

}

**11. Pagination, Filtering, and Sorting**

* **Pagination**: When returning large datasets, use pagination to split the data into manageable chunks. Common query parameters include page and limit.
* **Filtering and Sorting**: Allow clients to filter and sort data using query parameters, such as ?sort=price&filter[category]=electronics.

**12. Rate Limiting**

* **Throttling**: Implement rate limiting to prevent abuse and ensure fair usage. Provide headers (like X-Rate-Limit-Limit, X-Rate-Limit-Remaining) to inform clients about their current rate limits.

**13. Caching**

* **Cache Control**: Use HTTP caching headers like Cache-Control, ETag, and Last-Modified to manage cache behavior. Caching can significantly improve performance by reducing the load on the server.

**14. Security**

* **HTTPS**: Always use HTTPS to encrypt data in transit and protect against man-in-the-middle attacks.
* **Input Validation**: Validate all inputs to protect against common security vulnerabilities like SQL injection, XSS, and CSRF.
* **Authorization**: Implement proper authorization checks to ensure that users can only access resources they are permitted to.

**15. Documentation**

* **Comprehensive Documentation**: Provide detailed and easy-to-understand documentation that covers all aspects of the API, including endpoints, request/response formats, authentication methods, and error codes.
* **Interactive Documentation**: Use tools like Swagger (OpenAPI) to generate interactive API documentation that allows developers to test API calls directly from the documentation.

**16. Idempotency**

* **Safe and Idempotent Methods**: Ensure that GET, PUT, and DELETE operations are idempotent, meaning multiple identical requests should produce the same result. This is crucial for reliability, especially in distributed systems.

**17. Localization and Internationalization**

* **Locale Support**: Consider supporting multiple locales for error messages and content, especially if your API is used globally.
* **Timezone Awareness**: Use UTC for date and time fields, and allow clients to specify time zones if needed.

**18. Scalability**

* **Efficient Query Handling**: Design the API to handle large volumes of data and requests efficiently. Use techniques like pagination, query optimization, and database indexing.
* **Microservices**: Consider using a microservices architecture to break down the API into smaller, manageable, and scalable components.

**How Spring Boot Determines JSON or XML Response?**

Spring Boot checks the **Accept** header from the client request and determines the response format accordingly.

✅ **1️ If Client Requests JSON (Accept: application/json)**

* Spring Boot uses **Jackson (MappingJackson2HttpMessageConverter)** to serialize Java objects into JSON.
* **Default behavior** if no explicit request format is mentioned.

✅ **2️ If Client Requests XML (Accept: application/xml or Accept: text/xml)**

* Spring Boot uses **JAXB (Jaxb2RootElementHttpMessageConverter)** to serialize Java objects into XML.
* JAXB must be present in the classpath.

✅ **3️ If No Accept Header is Sent**

* By default, Spring Boot returns **JSON** (as it is most common).
* However, if XML is explicitly configured in the project, it can also be returned.

**Spring Boot Auto-Configuration for JSON & XML**

* **JSON Support** (Enabled by default via Jackson)
* **XML Support** (Requires JAXB dependency)

✅ **To enable XML support, add the following dependency (if using Spring Boot 3+ and Jakarta API)**:

<dependency>

<groupId>org.glassfish.jaxb</groupId>

<artifactId>jaxb-runtime</artifactId>

</dependency>

1️ Spring Boot determines the response format based on the **Accept** header.  
2️ JSON is the **default format** if no Accept header is specified.  
3️ XML is returned if the Accept: application/xml header is present and JAXB is available.  
4️ **Jackson** is used for JSON, and **JAXB** is used for XML serialization.

**When we call a REST API endpoint in Spring Boot, several internal processes to handle the request and generate a response.**

1. **Client** sends an HTTP request (e.g., GET /users/1).

2. **Embedded server (Tomcat)** receives the request.

3. **DispatcherServlet** determines the correct controller.

4. **Controller method** executes the request.

5. **Service layer** processes business logic.

6. **Repository layer** retrieves data from the database.

7. **Data is returned** to the controller.

8. **Jackson serializes** Java object to JSON.

9. **Response is sent** to the client.

A screenshot of a computer

Description automatically generated