Module 3: Building APIs and Web Services

HTTP Basics

1. Overview of HTTP Protocol

What is HTTP?

HTTP (HyperText Transfer Protocol) is an application-layer protocol that defines how messages are formatted and transmitted between web clients and servers. It serves as the foundation of data communication on the World Wide Web. Understanding its structure, methods, and status codes is essential for web development and API design. Each HTTP method serves a specific purpose, from retrieving data with GET to creating resources with POST. Status codes provide standardized communication about request outcomes, enabling robust error handling and user feedback. Modern applications leverage HTTP's flexibility while adhering to RESTful principles to create scalable and maintainable systems.

Key Characteristics

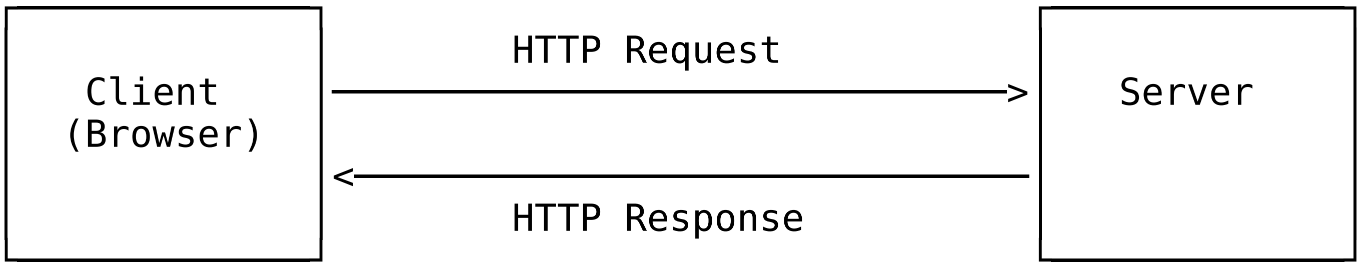
Stateless Protocol: Each request-response pair is independent. The server doesn't retain information about previous requests from the same client by default.

Client-Server Model: HTTP follows a request-response pattern where clients (browsers, mobile apps) initiate requests and servers provide responses.

Text-Based Protocol: HTTP messages are human-readable text, making debugging and analysis straightforward.

Port Usage: By default, HTTP uses port 80, while HTTPS (secure version) uses port 443.

HTTP Architecture



2. Standard Request and Response Structure

HTTP Request Structure

An HTTP request consists of four main components:

Request Line
Headers (multiple key-value pairs)
Blank Line
Body (optional) (message payload)

Request Line Format

```
METHOD /path/to/resource HTTP/version
```

Example Request:

```
POST /api/users HTTP/1.1
Host: example.com
Content-Type: application/json
Content-Length: 45
User-Agent: Mozilla/5.0
Accept: application/json
Authorization: Bearer token123

{"username":"john","email":"john@example.com"}
```

Request Components Breakdown

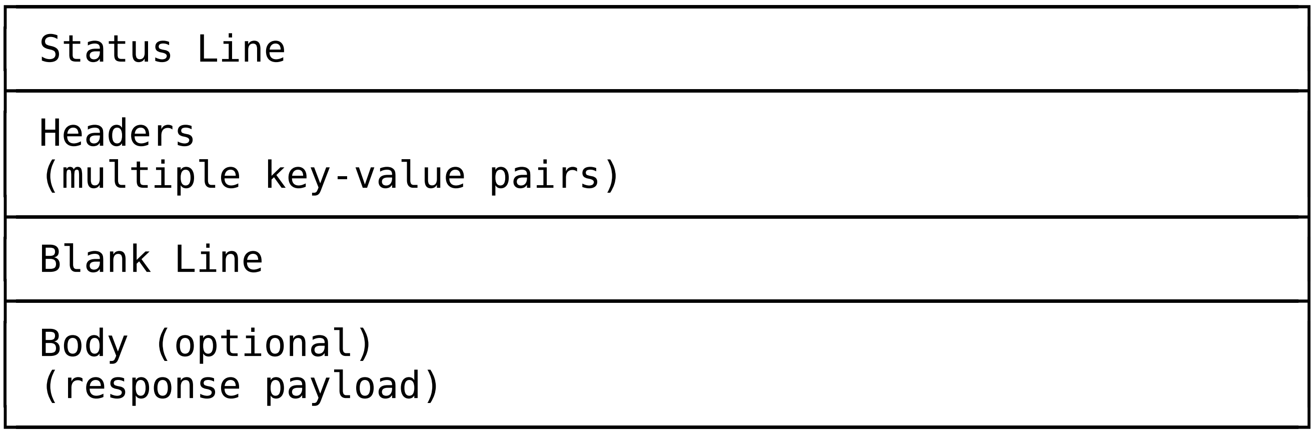
Request Line: Contains the HTTP method, resource path, and protocol version.

Headers: Metadata about the request providing additional context like content type, accepted formats, authentication credentials, and client information.

Blank Line: Separates headers from the body (mandatory even if there’s no body).

Body: Contains data being sent to the server (used in POST, PUT, PATCH requests).

HTTP Response Structure



Status Line Format

```
HTTP/version StatusCode ReasonPhrase
```

Example Response:

```
HTTP/1.1 200 OK
Date: Thu, 22 Jan 2026 10:30:00 GMT
Content-Type: application/json
Content-Length: 89
Server: Apache/2.4.41
Cache-Control: no-cache

{"id":123,"username":"john","email":"john@example.com","created":"2026-01-22T10:30:00Z"}
```

3. HTTP Methods: Structure and Usage

HTTP methods (also called verbs) indicate the desired action to be performed on a resource.

GET Method

Purpose: Retrieve data from the server without modifying it.

Request Body: Not used (data sent via URL query parameters).

Structure:

```
GET /api/users?page=1&limit=10 HTTP/1.1
Host: api.example.com
Accept: application/json
```

Response Example:

```
HTTP/1.1 200 OK
Content-Type: application/json

[{"id":1,"name":"Alice"},{"id":2,"name":"Bob"}]
```

Use Cases: Fetching web pages, retrieving API data, searching, filtering results.

POST Method

Purpose: Submit data to create a new resource or trigger processing.

Request Body: Contains the data being sent.

Structure:

```
POST /api/users HTTP/1.1
Host: api.example.com
Content-Type: application/json
Content-Length: 58

{"name":"Charlie","email":"charlie@example.com","age":28}
```

Response Example:

```
HTTP/1.1 201 Created
Location: /api/users/3
Content-Type: application/json

{"id":3,"name":"Charlie","email":"charlie@example.com"}
```

Use Cases: Creating new resources, submitting forms, uploading files, triggering server-side operations.

PUT Method

Purpose: Update an existing resource or create it if it doesn’t exist (full replacement).

Structure:: Similar to POST but targets a specific resource.

Use Cases: Complete resource updates, replacing entire documents.

PATCH Method

Purpose: Partially update an existing resource.

Request Body: Contains only the fields to be updated.

Structure: Similar to PUT but with partial data.

Use Cases: Updating specific fields without sending the entire resource.

DELETE Method

Purpose: Remove a resource from the server.

Request Body: Usually empty.

Structure:

```
DELETE /api/users/3 HTTP/1.1
Host: api.example.com
```

Response Example:

```
HTTP/1.1 204 No Content
```

Use Cases: Removing resources, canceling subscriptions, clearing data.

HEAD Method

Purpose: Same as GET but retrieves only headers, not the body.

Request Body: Not used.

Structure:

```
HEAD /api/users/3 HTTP/1.1
Host: api.example.com
```

Response Example:

```
HTTP/1.1 200 OK
Content-Type: application/json
```

```
Content-Length: 89
Last-Modified: Thu, 22 Jan 2026 10:30:00 GMT

(no body)
```

Use Cases: Checking if a resource exists, getting metadata, checking last modification time.

OPTIONS Method

Purpose: Describe communication options for the target resource.

Request Body: Not used.

Structure:

```
OPTIONS /api/users HTTP/1.1
Host: api.example.com
```

Response Example:

```
HTTP/1.1 200 OK
Allow: GET, POST, PUT, DELETE, OPTIONS
Access-Control-Allow-Methods: GET, POST, PUT, DELETE
Access-Control-Allow-Origin: *
```

Use Cases: CORS preflight requests, discovering allowed methods on a resource.

4. Important HTTP Status Codes

Status codes are three-digit numbers that indicate the result of an HTTP request. They are grouped into five categories.

Status Code Categories

```
Status Codes (3-digit)
│
├── 1xx: Informational (request received, processing)
│   └── Rarely used in modern applications
│
├── 2xx: Success (request successfully processed)
│   └── Action completed successfully
│
├── 3xx: Redirection (further action needed)
```

```
│     └── Client must take additional action
│
├── 4xx: Client Error (request contains errors)
│     └── Problem with the request itself
│
└── 5xx: Server Error (server failed to fulfill request)
      └── Server encountered an error
```

200 OK: Standard success response. Request succeeded.

201 Created: New resource created successfully (typically after POST).

301 Moved Permanently: Resource permanently moved to a new URL.

```
HTTP/1.1 301 Moved Permanently
Location: https://newsite.com/resource
```

400 Bad Request: Server cannot process the request due to client error (malformed syntax).

```
HTTP/1.1 400 Bad Request
Content-Type: application/json

{"error":"Invalid JSON format in request body"}
```

401 Unauthorized: Authentication required or failed.

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="API"

{"error":"Authentication required"}
```

403 Forbidden: Server understood request but refuses to authorize it.

```
HTTP/1.1 403 Forbidden

{"error":"You don't have permission to access this resource"}
```

404 Not Found: Requested resource doesn’t exist.

```
HTTP/1.1 404 Not Found
```

```
{"error":"User not found"}
```

405 Method Not Allowed: HTTP method not supported for this resource.

```
HTTP/1.1 405 Method Not Allowed
Allow: GET, POST

{"error":"DELETE method not allowed"}
```

500 Internal Server Error: Generic server error.

```
HTTP/1.1 500 Internal Server Error

{"error":"An unexpected error occurred"}
```

Status Code Quick Reference (For understanding purposes)

Code	Name	Category	Meaning
200	OK	Success	Request succeeded
201	Created	Success	New resource created
204	No Content	Success	Success but no content to return
301	Moved Permanently	Redirection	Resource moved permanently
302	Found	Redirection	Resource temporarily moved
304	Not Modified	Redirection	Resource unchanged (cache valid)
400	Bad Request	Client Error	Malformed request
401	Unauthorized	Client Error	Authentication required
403	Forbidden	Client Error	Access denied
404	Not Found	Client Error	Resource doesn’t exist
409	Conflict	Client Error	Request conflicts with current state
422	Unprocessable Entity	Client Error	Validation failed
429	Too Many Requests	Client Error	Rate limit exceeded
500	Internal Server Error	Server Error	Generic server error
502	Bad Gateway	Server Error	Invalid upstream response
503	Service Unavailable	Server Error	Server temporarily unavailable

REST APIs Quick Overview

What is REST?

REST (Representational State Transfer) is an architectural style for building web services that use HTTP methods to perform operations on resources. RESTful APIs are designed to be stateless, scalable, and easy to maintain. They use standard HTTP methods (GET, POST, PUT, DELETE) to perform CRUD operations on resources identified by URIs. REST emphasizes a uniform interface and resource-based interactions, making it a popular choice for web API development.

Key Principles:

- **Resources**: Everything is a resource (User, Product, Order)
- **URIs**: Each resource has a unique identifier (URL)
- **HTTP Methods**: Use standard methods (GET, POST, PUT, DELETE)
- **Stateless**: Each request contains all necessary information
- **JSON/XML**: Data exchange format

REST vs Traditional Web Services

Traditional web services often use RPC-style endpoints that are action-based and may not follow RESTful principles. In contrast, RESTful APIs focus on resources and use standard HTTP methods to perform operations on those resources. This leads to more intuitive and scalable APIs. Using RESTful design allows for better separation of concerns, easier maintenance, and improved scalability compared to traditional web services. HTTP methods are used to indicate the desired action on a resource, and URIs are designed to represent resources rather than actions, making RESTful APIs more intuitive and easier to use. Common mappings of HTTP methods to CRUD operations in RESTful APIs are as follows:

```
Traditional:

POST /getUserById
POST /createUser
POST /updateUser
POST /deleteUser

RESTful:

GET    /users/{id}     - Get user
POST   /users          - Create user
PUT    /users/{id}     - Update user
DELETE /users/{id}     - Delete user
```

OpenAPI

What is OpenAPI?

OpenAPI Specification (OAS) is a standard, language-agnostic interface description for HTTP APIs. It allows developers to define the structure of their APIs in a machine-readable format (YAML or JSON). This specification can be used to generate interactive documentation, client SDKs, server stubs, and perform automated testing. OpenAPI promotes consistency and standardization in API design, making it easier for developers to understand and consume APIs.

Key Benefits

- **Standardization**: Industry-standard format for API documentation
- **Auto-generation**: Generate client SDKs, server stubs, and documentation
- **Validation**: Validate requests and responses automatically
- **Testing**: Enable automated API testing
- **Discoverability**: Make APIs easier to understand and consume

Version Information

- **Current Version**: OpenAPI 3.0.4 (March 2024)
- **Format**: YAML or JSON
- **Previous Version**: OpenAPI 3.0.3, Swagger 2.0

OpenAPI Specification Structure

High-Level Architecture

```
┌────────────────────────────────────────────────────┐
│             OpenAPI 3.0.4 Document                 │
├────────────────────────────────────────────────────┤
│                                                    │
│  ┌──────────────────────────────────────────────┐  │
│  │  1. Metadata (info, servers, tags)           │  │
│  └──────────────────────────────────────────────┘  │
│                      ↓                             │
│  ┌──────────────────────────────────────────────┐  │
│  │  2. Paths (API endpoints)                    │  │
│  │     - HTTP methods                           │  │
│  │     - Parameters                             │  │
│  │     - Request/Response definitions           │  │
│  └──────────────────────────────────────────────┘  │
│                      ↓                             │
│  ┌──────────────────────────────────────────────┐  │
│  │  3. Components (Reusable definitions)        │  │
│  │     - Schemas (Data models)                  │  │
```

```
|  |     - Parameters                           |  |
|  |     - Responses                            |  |
|  |     - Security schemes                     |  |
|  └────────────────────────────────────────────┘  |
|                       ↓                          |
|  ┌────────────────────────────────────────────┐  |
|  |  4. Security (Authentication requirements)  |  |
|  └────────────────────────────────────────────┘  |
|                                                  |
└──────────────────────────────────────────────────┘
```

Example OpenAPI Document

Below is an example of a simple OpenAPI 3.0.4 document in YAML format that defines a basic API for managing users.

```yaml
openapi: 3.0.4
info:
  title: User Management API
  version: 1.0.0
  description: API for managing users in the system
servers:
  - url: https://api.example.com/v1
paths:
  /users:
    get:
      summary: Get a list of users
      responses:
        '200':
          description: A list of users
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/User'
    post:
      summary: Create a new user
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/User'
      responses:
        '201':
          description: User created successfully
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/User'
  /users/{id}:
    get:
      summary: Get a user by ID
      parameters:
        - name: id
          in: path
          required: true
          schema:
            type: integer
      responses:
        '200':
          description: User details
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/User'
        '404':
          description: User not found
security:
  - bearerAuth: []
components:
  schemas:
    User:
      type: object
      properties:
        id:
          type: integer
        name:
          type: string
        email:
          type: string
      required:
        - name
        - email
  securitySchemes:
    bearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
```

Student CRUD API - Complete Example

Below is a complete OpenAPI 3.0.4 specification for a Student Management REST API with CRUD operations.

```
openapi: 3.0.4

info:
  title: Student Management API
  version: 1.0.0
  description: |
    Comprehensive REST API for managing student records in an educational institution.

    Features:
    - Complete CRUD operations
    - Advanced search and filtering
    - Pagination support
    - Input validation
    - Error handling

  contact:
    name: API Support Team
    email: support@university.edu
    url: https://university.edu/support
  license:
    name: MIT
    url: https://opensource.org/licenses/MIT

servers:
  - url: https://api.university.edu/v1
    description: Production server
  - url: https://api-staging.university.edu/v1
    description: Staging server
  - url: http://localhost:8080/v1
    description: Development server

tags:
  - name: Students
    description: Student management operations
  - name: Health
    description: API health check endpoints

paths:
  # ==================== Health Check ====================
  /health:
    get:
      tags:
        - Health
```

```yaml
      summary: Health check endpoint
      description: Check if the API is running
      operationId: healthCheck
      responses:
        '200':
          description: API is healthy
          content:
            application/json:
              schema:
                type: object
                properties:
                  status:
                    type: string
                    example: UP
                  timestamp:
                    type: string
                    format: date-time
                    example: '2024-02-06T10:30:00Z'

  # ==================== Get All Students ====================
  /students:
    get:
      tags:
        - Students
      summary: Get all students
      description: |
        Retrieve a paginated list of all students with optional filtering.

        Query parameters allow you to:
        - Paginate through results
        - Filter by name (partial match)
        - Filter by minimum CGPA
        - Filter by city
        - Sort results
      operationId: getAllStudents
      parameters:
        - $ref: '#/components/parameters/PageParam'
        - $ref: '#/components/parameters/SizeParam'
        - name: name
          in: query
          description: Filter by student name (partial match, case-insensitive)
          required: false
          schema:
            type: string
            example: John
        - name: minCgpa
          in: query
          description: Filter by minimum CGPA
          required: false
          schema:
            type: number
            format: double
            minimum: 0.0
            maximum: 10.0
            example: 7.5
        - name: city
          in: query
          description: Filter by city
          required: false
          schema:
            type: string
            example: Mumbai
        - name: sortBy
          in: query
          description: Field to sort by
          required: false
          schema:
            type: string
            enum: [studentNumber, name, cgpa, createdDate]
            default: studentNumber
        - name: sortOrder
          in: query
          description: Sort order
          required: false
          schema:
            type: string
            enum: [asc, desc]
            default: asc
      responses:
        '200':
          description: Successful response with paginated student list
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/StudentPageResponse'
              examples:
                successExample:
                  summary: Example response with two students
                  value:
                    content:
                      - studentNumber: STU001
                        name: John Doe
                        address:
                          street: 123 Main Street
                          city: Mumbai
                          state: Maharashtra
                          country: India
                        cgpa: 8.5
                  backlogs: 0
                - studentNumber: STU002
                  name: Jane Smith
                  address:
                    street: 456 Park Avenue
                    city: Delhi
                    state: Delhi
                    country: India
                  cgpa: 9.2
                  backlogs: 1
              page:
                number: 0
                size: 20
                totalElements: 2
                totalPages: 1
    '400':
      $ref: '#/components/responses/BadRequest'
    '500':
      $ref: '#/components/responses/InternalServerError'
  security:
    - bearerAuth: []

# ==================== Create Student ====================
post:
  tags:
    - Students
  summary: Create a new student
  description: |
    Create a new student record in the system.

    Required fields:
    - studentNumber (unique)
    - name
    - address (complete)
    - cgpa
    - backlogs
  operationId: createStudent
  requestBody:
    required: true
    description: Student object to be created
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/StudentCreateRequest'
        examples:
          validStudent:
            summary: Valid student creation request
            value:
              studentNumber: STU003
            name: Alice Johnson
            address:
              street: 789 Oak Street
              city: Bangalore
              state: Karnataka
              country: India
            cgpa: 7.8
            backlogs: 2
responses:
  '201':
    description: Student created successfully
    headers:
      Location:
        description: URI of the created student
        schema:
          type: string
          example: /v1/students/STU003
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/Student'
        example:
          studentNumber: STU003
          name: Alice Johnson
          address:
            street: 789 Oak Street
            city: Bangalore
            state: Karnataka
            country: India
          cgpa: 7.8
          backlogs: 2
          createdDate: '2024-02-06T10:30:00Z'
          lastModifiedDate: '2024-02-06T10:30:00Z'
  '400':
    $ref: '#/components/responses/BadRequest'
  '409':
    description: Student with the same student number already exists
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/Error'
        example:
          timestamp: '2024-02-06T10:30:00Z'
          status: 409
          error: Conflict
          message: Student with student number STU003 already exists
          path: /v1/students
  '500':
    $ref: '#/components/responses/InternalServerError'
    security:
      - bearerAuth: []

# =================== Get Student by Number ===================
/students/{studentNumber}:
  get:
    tags:
      - Students
    summary: Get student by student number
    description: Retrieve a single student record by their unique student number
    operationId: getStudentByNumber
    parameters:
      - $ref: '#/components/parameters/StudentNumberParam'
    responses:
      '200':
        description: Student found
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Student'
            example:
              studentNumber: STU001
              name: John Doe
              address:
                street: 123 Main Street
                city: Mumbai
                state: Maharashtra
                country: India
              cgpa: 8.5
              backlogs: 0
              createdDate: '2024-01-15T09:00:00Z'
              lastModifiedDate: '2024-02-01T14:30:00Z'
      '404':
        $ref: '#/components/responses/NotFound'
      '500':
        $ref: '#/components/responses/InternalServerError'
    security:
      - bearerAuth: []

  # =================== Update Student ===================
  put:
    tags:
      - Students
    summary: Update student by student number
    description: |
      Update an existing student record.

      All fields in the request body will replace existing values.
      Student number cannot be changed.
```

```yaml
operationId: updateStudent
parameters:
  - $ref: '#/components/parameters/StudentNumberParam'
requestBody:
  required: true
  description: Updated student object
  content:
    application/json:
      schema:
        $ref: '#/components/schemas/StudentUpdateRequest'
      examples:
        updateExample:
          summary: Update student information
          value:
            name: John Michael Doe
            address:
              street: 123 Main Street, Apt 4B
              city: Mumbai
              state: Maharashtra
              country: India
            cgpa: 8.7
            backlogs: 0
responses:
  '200':
    description: Student updated successfully
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/Student'
        example:
          studentNumber: STU001
          name: John Michael Doe
          address:
            street: 123 Main Street, Apt 4B
            city: Mumbai
            state: Maharashtra
            country: India
          cgpa: 8.7
          backlogs: 0
          createdDate: '2024-01-15T09:00:00Z'
          lastModifiedDate: '2024-02-06T10:45:00Z'
  '400':
    $ref: '#/components/responses/BadRequest'
  '404':
    $ref: '#/components/responses/NotFound'
  '500':
    $ref: '#/components/responses/InternalServerError'
security:
  - bearerAuth: []
```

```yaml
# =================== Partial Update Student ===================
patch:
  tags:
    - Students
  summary: Partially update student
  description: |
    Update specific fields of a student record.

    Only the fields provided in the request body will be updated.
    Other fields will remain unchanged.
  operationId: partialUpdateStudent
  parameters:
    - $ref: '#/components/parameters/StudentNumberParam'
  requestBody:
    required: true
    description: Fields to update
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/StudentPartialUpdateRequest'
        examples:
          updateCgpa:
            summary: Update only CGPA and backlogs
            value:
              cgpa: 9.0
              backlogs: 0
  responses:
    '200':
      description: Student updated successfully
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Student'
    '400':
      $ref: '#/components/responses/BadRequest'
    '404':
      $ref: '#/components/responses/NotFound'
    '500':
      $ref: '#/components/responses/InternalServerError'
  security:
    - bearerAuth: []

# =================== Delete Student ===================
delete:
  tags:
    - Students
  summary: Delete student by student number
  description: |
        Permanently delete a student record from the system.

        **Warning**: This operation cannot be undone.
      operationId: deleteStudent
      parameters:
        - $ref: '#/components/parameters/StudentNumberParam'
      responses:
        '204':
          description: Student deleted successfully
        '404':
          $ref: '#/components/responses/NotFound'
        '500':
          $ref: '#/components/responses/InternalServerError'
      security:
        - bearerAuth: []

  # =================== Search Students ===================
  /students/search:
    post:
      tags:
        - Students
      summary: Advanced student search
      description: |
        Perform advanced search with multiple criteria.

        Supports complex queries with multiple filters combined.
      operationId: searchStudents
      requestBody:
        required: true
        description: Search criteria
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/StudentSearchRequest'
            examples:
              searchExample:
                summary: Search for high-performing students in Mumbai
                value:
                  name: John
                  city: Mumbai
                  minCgpa: 8.0
                  maxBacklogs: 1
      responses:
        '200':
          description: Search results
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Student'
        '400':
          $ref: '#/components/responses/BadRequest'
        '500':
          $ref: '#/components/responses/InternalServerError'
      security:
        - bearerAuth: []

  # ==================== Get Student Statistics ====================
  /students/statistics:
    get:
      tags:
        - Students
      summary: Get student statistics
      description: Retrieve statistical information about all students
      operationId: getStudentStatistics
      responses:
        '200':
          description: Statistics retrieved successfully
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/StudentStatistics'
              example:
                totalStudents: 150
                averageCgpa: 7.85
                studentsWithNoBacklogs: 120
                studentsWithBacklogs: 30
                topPerformers: 15
                cityDistribution:
                  Mumbai: 45
                  Delhi: 35
                  Bangalore: 40
                  Chennai: 30
      security:
        - bearerAuth: []

# ==================== Components Section ====================
components:
  schemas:
    # ==================== Student Entity ====================
    Student:
      type: object
      required:
        - studentNumber
        - name
        - address
        - cgpa
```

```yaml
      - backlogs
    properties:
      studentNumber:
        type: string
        description: Unique student identifier
        pattern: '^STU[0-9]{3,6}$'
        example: STU001
      name:
        type: string
        description: Full name of the student
        minLength: 2
        maxLength: 100
        example: John Doe
      address:
        $ref: '#/components/schemas/Address'
      cgpa:
        type: number
        format: double
        description: Cumulative Grade Point Average
        minimum: 0.0
        maximum: 10.0
        example: 8.5
      backlogs:
        type: integer
        format: int32
        description: Number of backlog subjects
        minimum: 0
        example: 0
      createdDate:
        type: string
        format: date-time
        description: Date and time when the student record was created
        readOnly: true
        example: '2024-01-15T09:00:00Z'
      lastModifiedDate:
        type: string
        format: date-time
        description: Date and time when the student record was last updated
        readOnly: true
        example: '2024-02-06T10:30:00Z'
    description: Complete student record with all fields

  # ==================== Address Schema ====================
  Address:
    type: object
    required:
      - street
      - city
      - state
    - country
  properties:
    street:
      type: string
      description: Street address
      minLength: 3
      maxLength: 200
      example: 123 Main Street
    city:
      type: string
      description: City name
      minLength: 2
      maxLength: 100
      example: Mumbai
    state:
      type: string
      description: State or province
      minLength: 2
      maxLength: 100
      example: Maharashtra
    country:
      type: string
      description: Country name
      minLength: 2
      maxLength: 100
      example: India
  description: Student's residential address

# =================== Create Student Request ===================
StudentCreateRequest:
  type: object
  required:
    - studentNumber
    - name
    - address
    - cgpa
    - backlogs
  properties:
    studentNumber:
      type: string
      description: Unique student identifier
      pattern: '^STU[0-9]{3,6}$'
      example: STU001
    name:
      type: string
      description: Full name of the student
      minLength: 2
      maxLength: 100
      example: John Doe
```

```yaml
        address:
          $ref: '#/components/schemas/Address'
        cgpa:
          type: number
          format: double
          description: Cumulative Grade Point Average
          minimum: 0.0
          maximum: 10.0
          example: 8.5
        backlogs:
          type: integer
          format: int32
          description: Number of backlog subjects
          minimum: 0
          example: 0
      description: Request body for creating a new student

    # =================== Update Student Request ===================
    StudentUpdateRequest:
      type: object
      required:
        - name
        - address
        - cgpa
        - backlogs
      properties:
        name:
          type: string
          description: Full name of the student
          minLength: 2
          maxLength: 100
          example: John Doe
        address:
          $ref: '#/components/schemas/Address'
        cgpa:
          type: number
          format: double
          description: Cumulative Grade Point Average
          minimum: 0.0
          maximum: 10.0
          example: 8.5
        backlogs:
          type: integer
          format: int32
          description: Number of backlog subjects
          minimum: 0
          example: 0
      description: Request body for updating a student (all fields required)
```

```yaml
# =================== Partial Update Request ===================
StudentPartialUpdateRequest:
  type: object
  properties:
    name:
      type: string
      minLength: 2
      maxLength: 100
    address:
      $ref: '#/components/schemas/Address'
    cgpa:
      type: number
      format: double
      minimum: 0.0
      maximum: 10.0
    backlogs:
      type: integer
      format: int32
      minimum: 0
  description: Request body for partial update (all fields optional)

# =================== Search Request ===================
StudentSearchRequest:
  type: object
  properties:
    name:
      type: string
      description: Search by name (partial match)
    city:
      type: string
      description: Filter by city
    state:
      type: string
      description: Filter by state
    country:
      type: string
      description: Filter by country
    minCgpa:
      type: number
      format: double
      minimum: 0.0
      maximum: 10.0
      description: Minimum CGPA
    maxCgpa:
      type: number
      format: double
      minimum: 0.0
      maximum: 10.0
      description: Maximum CGPA
    maxBacklogs:
      type: integer
      format: int32
      minimum: 0
      description: Maximum number of backlogs
  description: Advanced search criteria

# =================== Paginated Response ===================
StudentPageResponse:
  type: object
  properties:
    content:
      type: array
      items:
        $ref: '#/components/schemas/Student'
      description: Array of student records
    page:
      $ref: '#/components/schemas/PageInfo'
  description: Paginated list of students

PageInfo:
  type: object
  properties:
    number:
      type: integer
      format: int32
      description: Current page number (0-based)
      example: 0
    size:
      type: integer
      format: int32
      description: Number of items per page
      example: 20
    totalElements:
      type: integer
      format: int64
      description: Total number of items
      example: 150
    totalPages:
      type: integer
      format: int32
      description: Total number of pages
      example: 8
  description: Pagination information

# =================== Statistics ===================
StudentStatistics:
  type: object
  properties:
```

```yaml
    totalStudents:
      type: integer
      format: int64
      example: 150
    averageCgpa:
      type: number
      format: double
      example: 7.85
    studentsWithNoBacklogs:
      type: integer
      format: int64
      example: 120
    studentsWithBacklogs:
      type: integer
      format: int64
      example: 30
    topPerformers:
      type: integer
      format: int64
      description: Students with CGPA >= 9.0
      example: 15
    cityDistribution:
      type: object
      additionalProperties:
        type: integer
      description: Number of students per city
      example:
        Mumbai: 45
        Delhi: 35
  description: Statistical information about students

# ==================== Error Response ====================
Error:
  type: object
  required:
    - timestamp
    - status
    - error
    - message
    - path
  properties:
    timestamp:
      type: string
      format: date-time
      description: Error occurrence timestamp
      example: '2024-02-06T10:30:00Z'
    status:
      type: integer
      format: int32
```

```yaml
          description: HTTP status code
          example: 400
        error:
          type: string
          description: Error type
          example: Bad Request
        message:
          type: string
          description: Detailed error message
          example: Validation failed for field 'cgpa'
        path:
          type: string
          description: Request path that caused the error
          example: /v1/students
        errors:
          type: array
          items:
            $ref: '#/components/schemas/ValidationError'
          description: List of validation errors (if applicable)
      description: Standard error response

    ValidationError:
      type: object
      properties:
        field:
          type: string
          description: Field that failed validation
          example: cgpa
        message:
          type: string
          description: Validation error message
          example: must be between 0.0 and 10.0
        rejectedValue:
          type: string
          description: Value that was rejected
          example: '12.5'
      description: Individual validation error

  # ==================== Parameters ====================
  parameters:
    StudentNumberParam:
      name: studentNumber
      in: path
      description: Unique student identifier
      required: true
      schema:
        type: string
        pattern: '^STU[0-9]{3,6}$'
      example: STU001
  PageParam:
    name: page
    in: query
    description: Page number (0-based)
    required: false
    schema:
      type: integer
      format: int32
      minimum: 0
      default: 0
    example: 0

  SizeParam:
    name: size
    in: query
    description: Number of items per page
    required: false
    schema:
      type: integer
      format: int32
      minimum: 1
      maximum: 100
      default: 20
    example: 20

# ==================== Responses ====================
responses:
  BadRequest:
    description: Bad request - Invalid input
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/Error'
        examples:
          validationError:
            summary: Validation error example
            value:
              timestamp: '2024-02-06T10:30:00Z'
              status: 400
              error: Bad Request
              message: Validation failed
              path: /v1/students
              errors:
                - field: cgpa
                  message: must be between 0.0 and 10.0
                  rejectedValue: '12.5'
                - field: studentNumber
                  message: must match pattern ^STU[0-9]{3,6}$
```

~~~yaml
                    rejectedValue: INVALID

    NotFound:
      description: Resource not found
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
          example:
            timestamp: '2024-02-06T10:30:00Z'
            status: 404
            error: Not Found
            message: Student not found with student number STU999
            path: /v1/students/STU999

    InternalServerError:
      description: Internal server error
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
          example:
            timestamp: '2024-02-06T10:30:00Z'
            status: 500
            error: Internal Server Error
            message: An unexpected error occurred
            path: /v1/students

  # ==================== Security Schemes ====================
  securitySchemes:
    bearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
      description: |
        JWT-based authentication. Include the token in the Authorization header:

        ```
        Authorization: Bearer <your_jwt_token>
        ```

# ==================== Global Security ====================
security:
  - bearerAuth: []
~~~

Component Breakdown

1. Schemas (Data Models)

Schemas define the structure of request and response bodies.

Key Schema Features

```
Student:
  type: object
  required:              # Required fields
    - studentNumber
    - name
  properties:
    studentNumber:
      type: string
      pattern: '^STU[0-9]{3,6}$'    # Regex validation
      example: STU001
    name:
      type: string
      minLength: 2                   # Length validation
      maxLength: 100
    cgpa:
      type: number
      format: double
      minimum: 0.0                   # Range validation
      maximum: 10.0
    createdDate:
      type: string
      format: date-time
      readOnly: true                 # Read-only field
```

Schema Reusability

```
# Define once
components:
  schemas:
    Address:
      type: object
      properties:
        city:
          type: string

# Reuse multiple times
Student:
  properties:
    address:
      $ref: '#/components/schemas/Address'

Teacher:
  properties:
    address:
      $ref: '#/components/schemas/Address'
```

2. Parameters

Parameters can be in path, query, header, or cookie.

```
parameters:
  # Path parameter
  StudentNumberParam:
    name: studentNumber
    in: path              # location: path, query, header, cookie
    description: Student ID
    required: true        # Always required for path params
    schema:
      type: string

  # Query parameter
  PageParam:
    name: page
    in: query
    required: false       # Optional
    schema:
      type: integer
      default: 0          # Default value
      minimum: 0

  # Header parameter
  ApiKeyParam:
    name: X-API-Key
    in: header
    required: true
    schema:
      type: string
```

3. Request Bodies

Define the structure of request payloads.

```
requestBody:
  required: true
  description: Student to create
  content:
    application/json:         # Content type
      schema:
        $ref: '#/components/schemas/StudentCreateRequest'
      examples:              # Multiple examples
        example1:
          summary: Basic student
          value:
            studentNumber: STU001
            name: John Doe
        example2:
          summary: Student with high CGPA
          value:
            studentNumber: STU002
            name: Jane Smith
            cgpa: 9.5
```

4. Responses

Define possible API responses.

```
responses:
  '200':                    # HTTP status code
    description: Success
    headers:                # Response headers
      X-Rate-Limit:
        schema:
          type: integer
        description: Requests per hour
    content:
      application/json:     # Content type
        schema:
          $ref: '#/components/schemas/Student'
        examples:
          example1:
            value:
              studentNumber: STU001
              name: John Doe

  '404':
    description: Not found
    content:
      application/json:
```

```
      schema:
        $ref: '#/components/schemas/Error'
```

5. Security Schemes

Define authentication methods.

Bearer Authentication (JWT)

```
securitySchemes:
  bearerAuth:
    type: http
    scheme: bearer
    bearerFormat: JWT
    description: JWT token authentication
```

API Key

```
securitySchemes:
  apiKey:
    type: apiKey
    in: header           # Can be: header, query, cookie
    name: X-API-Key
```

OAuth2

```
securitySchemes:
  oauth2:
    type: oauth2
    flows:
      authorizationCode:
        authorizationUrl: https://example.com/oauth/authorize
        tokenUrl: https://example.com/oauth/token
        scopes:
          read:students: Read student data
          write:students: Modify student data
```

Basic Authentication

```
securitySchemes:
  basicAuth:
    type: http
    scheme: basic
```

6. Tags

Organize endpoints into logical groups.

```
tags:
  - name: Students
    description: Student management operations
    externalDocs:
      description: Find out more
      url: https://docs.example.com/students
  - name: Admin
    description: Administrative operations
```

Best Practices

1. Versioning

```
# URL versioning (recommended)
servers:
  - url: https://api.example.com/v1

# Header versioning
parameters:
  - name: API-Version
    in: header
    schema:
      type: string
      enum: [v1, v2]
```

2. Error Handling

Provide consistent error responses:

```
Error:
  type: object
  required:
    - timestamp
    - status
    - error
    - message
  properties:
    timestamp:
      type: string
      format: date-time
    status:
      type: integer
    error:
      type: string
    message:
      type: string
    path:
      type: string
    errors:
      type: array
      items:
        type: object
```

3. Pagination

Always paginate large collections:

```
parameters:
  - name: page
    in: query
    schema:
      type: integer
      default: 0
      minimum: 0
  - name: size
    in: query
    schema:
      type: integer
      default: 20
      minimum: 1
      maximum: 100
```

4. Filtering and Sorting

```
parameters:
  # Filtering
  - name: status
    in: query
    schema:
      type: string
      enum: [active, inactive]

  # Sorting
  - name: sortBy
    in: query
    schema:
      type: string
      enum: [name, createdDate, cgpa]
  - name: sortOrder
    in: query
    schema:
      type: string
      enum: [asc, desc]
      default: asc
```

5. Field Selection

Allow clients to select specific fields:

```
parameters:
  - name: fields
    in: query
    description: Comma-separated list of fields to return
    schema:
      type: string
    example: studentNumber,name,cgpa
```

6. Documentation

- Use clear, descriptive summaries and descriptions
- Provide examples for all requests and responses
- Document error scenarios
- Include external documentation links

```
paths:
  /students:
    get:
      summary: Get all students           # Brief summary
      description: |                      # Detailed description
        Retrieve a paginated list of all students.

        This endpoint supports:
        - Pagination
        - Filtering by name and city
        - Sorting by multiple fields
      externalDocs:
        description: API documentation
        url: https://docs.example.com
```

7. Validation Rules

Be explicit about validation:

```
studentNumber:
  type: string
  pattern: '^STU[0-9]{3,6}$'
  minLength: 6
  maxLength: 9
  example: STU001

cgpa:
  type: number
  format: double
  minimum: 0.0
  maximum: 10.0
  multipleOf: 0.01           # Two decimal places
```

8. Examples

Provide comprehensive examples:

```
examples:
  validStudent:
    summary: Valid student
    description: Example of a valid student object
    value:
      studentNumber: STU001
      name: John Doe
      cgpa: 8.5
```

```
invalidStudent:
  summary: Invalid CGPA
  description: Example showing validation error
  value:
    studentNumber: STU001
    name: John Doe
    cgpa: 12.5              # Invalid - exceeds maximum
```

REST API using Spring Boot 3

What is @RestController?

`@RestController` is a specialized version of the `@Controller` annotation used to create RESTful web services in Spring Boot. RestControllers are designed to handle HTTP requests and return data (usually in JSON or XML format) rather than rendering views. When you annotate a class with `@RestController`, Spring automatically converts the return values of the methods into the appropriate format based on the client's request (using content negotiation). RestControllers are typically used to build APIs that serve data to clients, such as web applications, mobile apps, or other services. In spring boot `@RestController` is a convenient annotation that combines `@Controller` and `@ResponseBody`, eliminating the need to annotate each method with `@ResponseBody` to indicate that the return value should be serialized directly to the HTTP response body.

```
@RestController = @Controller + @ResponseBody
```

Key Characteristics:

- Automatically converts return values to JSON/XML
- Eliminates need for `@ResponseBody` on every method
- Designed for REST API development
- Returns data instead of views

Basic Usage

Simple RestController

```
@RestController
@RequestMapping("/api")
public class HelloController {

    @GetMapping("/hello")
    public String hello() {
        return "Hello, World!";
    }
}
```

Above example defines a simple REST endpoint that returns a greeting message. The `@RequestMapping("/api")` sets the base path for all endpoints in this controller, and the `@GetMapping("/hello")` maps GET requests to the `hello()` method.

Access: `GET http://localhost:8080/api/hello`

Key Annotations

1. @RequestMapping

Maps HTTP requests to handler methods. Can be used at class and method level.

```
@RestController
@RequestMapping("/api/users")  // Base path for all methods
public class UserController {

    @RequestMapping("/all")  // /api/users/all
    public List<User> getAllUsers() {
        return userList;
    }
}
```

2. HTTP Method Annotations

```
@RestController
@RequestMapping("/api/products")
public class ProductController {

    @GetMapping          // GET - Read
    @PostMapping         // POST - Create
    @PutMapping          // PUT - Update
    @PatchMapping        // PATCH - Partial Update
    @DeleteMapping       // DELETE - Delete
}
```

3. @PathVariable

Path variables for dynamic URL segments. These values are extracted from the URL and passed as method parameters. The name in `@PathVariable` should match the placeholder in the URL.

```
// GET /api/users/123
@GetMapping("/users/{id}")
public User getUser(@PathVariable Long id) {
    return findUserById(id);
}

// GET /api/users/123/orders/456
@GetMapping("/users/{userId}/orders/{orderId}")
public Order getOrder(
    @PathVariable Long userId,
    @PathVariable Long orderId) {
    return findOrder(userId, orderId);
}
```

4. @RequestParam

Query parameters are passed in the URL after the `?` and are used to filter or modify the request. They are optional by default, but you can make them required.

```
// GET /api/search?keyword=java&page=1
@GetMapping("/search")
public List<Item> search(
    @RequestParam String keyword,
    @RequestParam(defaultValue = "0") int page) {
    return searchItems(keyword, page);
}

// Optional parameter
@GetMapping("/filter")
public List<Item> filter(
    @RequestParam(required = false) String category) {
    return filterItems(category);
}
```

5. @RequestBody

Request body is used to pass complex objects in the request payload, typically in JSON format. Spring automatically deserializes the JSON into the specified Java object.

```
@PostMapping("/users")
public User createUser(@RequestBody User user) {
    return saveUser(user);
}
```

```
// Request JSON:
{
    "name": "John Doe",
    "email": "john@example.com"
}
```

6. @RequestHeader

RequestHeader is used to access HTTP headers sent by the client. You can specify the header name and Spring will inject its value into the method parameter.

```
@GetMapping("/info")
public String getInfo(
    @RequestHeader("User-Agent") String userAgent) {
    return "Browser: " + userAgent;
}
```

Complete Example

```
@RestController
@RequestMapping("/api/books")
public class BookController {

    private List<Book> books = new ArrayList<>();
    private Long nextId = 1L;

    // GET all books
    @GetMapping
    public List<Book> getAllBooks() {
        return books;
    }

    // GET book by ID
    @GetMapping("/{id}")
    public Book getBook(@PathVariable Long id) {
        return books.stream()
            .filter(b -> b.getId().equals(id))
            .findFirst()
            .orElse(null);
    }

    // POST - Create book
    @PostMapping
    public Book createBook(@RequestBody Book book) {
        book.setId(nextId++);
        books.add(book);
        return book;
    }

    // PUT - Update book
    @PutMapping("/{id}")
    public Book updateBook(
        @PathVariable Long id,
        @RequestBody Book updatedBook) {

        for (int i = 0; i < books.size(); i++) {
            if (books.get(i).getId().equals(id)) {
                updatedBook.setId(id);
                books.set(i, updatedBook);
                return updatedBook;
            }
        }
        return null;
    }

    // DELETE book
    @DeleteMapping("/{id}")
    public String deleteBook(@PathVariable Long id) {
        books.removeIf(b -> b.getId().equals(id));
        return "Book deleted";
    }

    // Search with query parameter
    @GetMapping("/search")
    public List<Book> search(@RequestParam String title) {
        return books.stream()
            .filter(b -> b.getTitle().contains(title))
            .toList();
    }
}

// Book Model
class Book {
    private Long id;
    private String title;
    private String author;

    // Constructors, Getters, Setters
}
```

Response Handling

1. Return Simple Types

```
@GetMapping("/message")
public String getMessage() {
    return "Hello";  // Returns: "Hello"
}
```

2. Return Objects (Auto-converted to JSON)

```
@GetMapping("/user")
public User getUser() {
    return new User("John", "john@example.com");
}

// Response:
{
    "name": "John",
    "email": "john@example.com"
}
```

3. Return Collections

```
@GetMapping("/users")
public List<User> getUsers() {
    return List.of(
        new User("John", "john@example.com"),
        new User("Jane", "jane@example.com")
    );
}

// Response: [ {...}, {...} ]
```

4. ResponseEntity for Status Control

```
@GetMapping("/user/{id}")
public ResponseEntity<User> getUser(@PathVariable Long id) {
    User user = findUser(id);
    if (user != null) {
        return ResponseEntity.ok(user);  // 200 OK
    } else {
        return ResponseEntity.notFound().build();  // 404 Not Found
    }
}

@PostMapping("/users")
public ResponseEntity<User> createUser(@RequestBody User user) {
    User created = saveUser(user);
    return ResponseEntity
        .status(HttpStatus.CREATED)  // 201 Created
        .body(created);
}
```

HTTP Status Codes

```
@RestController
public class StatusController {

    @GetMapping("/ok")
    public String ok() {
        return "Success";  // 200 OK (default)
    }

    @PostMapping("/create")
    @ResponseStatus(HttpStatus.CREATED)  // 201
    public String create() {
        return "Created";
    }

    @DeleteMapping("/delete")
    @ResponseStatus(HttpStatus.NO_CONTENT)  // 204
    public void delete() {
        // No content returned
    }
}
```

Common Status Codes:

- `200 OK` - Success (default)
- `201 Created` - Resource created
- `204 No Content` - Success, no response body
- `400 Bad Request` - Invalid request
- `404 Not Found` - Resource not found
- `500 Internal Server Error` - Server error

Exception Handling

Method-Level Exception Handler

Method level exception handlers allow you to handle exceptions specific to a controller. You can use the `@ExceptionHandler` annotation to define methods that will handle specific exceptions thrown by any method in the controller. This is useful for handling exceptions that are relevant only to that controller, allowing you to return custom error responses or status codes.

```
@RestController
public class ProductController {

    @GetMapping("/products/{id}")
    public Product getProduct(@PathVariable Long id) {
        if (id < 1) {
            throw new IllegalArgumentException("Invalid ID");
        }
        return findProduct(id);
    }

    @ExceptionHandler(IllegalArgumentException.class)
    public ResponseEntity<String> handleBadRequest(
        IllegalArgumentException ex) {

        return ResponseEntity
            .status(HttpStatus.BAD_REQUEST)
            .body(ex.getMessage());
    }
}
```

Global Exception Handler

Global exception handlers allow you to handle exceptions across the entire application in a centralized manner. By using the `@RestControllerAdvice` annotation, you can define a class that will intercept exceptions thrown by any controller and provide a consistent error response format. This is particularly useful for handling common exceptions like `ResourceNotFoundException`, validation errors, or any unhandled exceptions, ensuring that clients receive meaningful error messages and appropriate HTTP status codes.

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleNotFound(
        ResourceNotFoundException ex) {

        ErrorResponse error = new ErrorResponse(
            404,
            ex.getMessage(),
            System.currentTimeMillis()
        );

        return ResponseEntity
            .status(HttpStatus.NOT_FOUND)
            .body(error);
    }
}
```

Data Validation

Add Validation Annotations

Validation annotations from the `jakarta.validation.constraints` package can be used to enforce constraints on request data. When combined with `@Valid`, Spring will automatically validate the incoming request body against these constraints and return a 400 Bad Request response if validation fails. Inorder to customize the error response, you can create a global exception handler that catches `MethodArgumentNotValidException` and formats the validation errors in a user-friendly way.

```
import jakarta.validation.constraints.*;

public class User {

    @NotBlank(message = "Name is required")
    @Size(min = 2, max = 50)
    private String name;

    @Email(message = "Invalid email")
    @NotBlank
    private String email;

    @Min(18)
    @Max(100)
    private int age;

    // Getters, Setters
}
```

Use @Valid in Controller

@Valid triggers the validation process for the request body. If any validation constraints are violated, Spring will throw a `MethodArgumentNotValidException`, which can be handled to return detailed error messages to the client.

```
@RestController
@RequestMapping("/api/users")
public class UserController {

    @PostMapping
    public ResponseEntity<User> createUser(
        @Valid @RequestBody User user) {

        // If validation fails, returns 400 Bad Request
        User created = saveUser(user);
        return ResponseEntity.status(HttpStatus.CREATED)
            .body(created);
    }
}
```

Handle Validation Errors

Handle validation errors globally to provide consistent error responses. The example below captures all validation errors and returns a structured response containing the field names and corresponding error messages. MethodArgumentNotValidException is thrown when validation on an argument annotated with @Valid fails. By catching this exception in a global exception handler, you can extract the validation errors and return them in a user-friendly format, such as a JSON object containing the field names and error messages.

```
@RestControllerAdvice
public class ValidationExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Map<String, String>> handleValidation(
        MethodArgumentNotValidException ex) {

        Map<String, String> errors = new HashMap<>();

        ex.getBindingResult().getAllErrors().forEach(error -> {
            String field = ((FieldError) error).getField();
            String message = error.getDefaultMessage();
            errors.put(field, message);
        });
        return ResponseEntity
            .status(HttpStatus.BAD_REQUEST)
            .body(errors);
    }
}
```

Common Validation Annotations

```
@NotNull        // Cannot be null
@NotEmpty       // Cannot be null or empty (String, Collection)
@NotBlank       // Cannot be null, empty, or whitespace (String only)
@Size(min, max) // Size constraints (String, Collection, Array)
@Min(value)     // Minimum numeric value
@Max(value)     // Maximum numeric value
@Email          // Valid email format
@Pattern(regexp)// Matches regex pattern
@Past           // Date in the past
@Future         // Date in the future
@Positive       // Positive number
@Negative       // Negative number
```

Best Practices

1. Use Proper HTTP Methods

Use the correct HTTP method for each operation to follow RESTful principles. This makes your API more intuitive and easier to use. Don’t use POST for actions that are meant to read data, and avoid using GET for operations that modify data.

```
// GOOD
@GetMapping("/users")           // Read
@PostMapping("/users")          // Create
@PutMapping("/users/{id}")      // Update
@DeleteMapping("/users/{id}")   // Delete

// BAD
@PostMapping("/getUsers")
@PostMapping("/createUser")
```

2. Use Meaningful URIs

Resource-based URIs are more intuitive and easier to understand than action-based URIs. Use nouns to represent resources and avoid verbs in the URI path. Don’t include implementation details or actions in the URI. Instead, use HTTP methods to indicate the action being performed on the resource.

```
// GOOD - Resource-based
/api/customers
/api/customers/{id}
/api/customers/{id}/orders

// BAD - Action-based
/api/getCustomers
/api/createCustomer
```

3. Return Appropriate Status Codes

Returning the correct HTTP status codes helps clients understand the result of their requests and handle responses appropriately. Use 200 OK for successful GET requests, 201 Created for successful POST requests that create resources, and 204 No Content for successful DELETE requests. Avoid returning 200 OK for all responses, as it can be misleading and does not provide enough information about the outcome of the request.

```
@PostMapping("/users")
public ResponseEntity<User> create(@RequestBody User user) {
    return ResponseEntity
        .status(HttpStatus.CREATED)  // 201 instead of 200
        .body(user);
}

@DeleteMapping("/users/{id}")
public ResponseEntity<Void> delete(@PathVariable Long id) {
    deleteUser(id);
    return ResponseEntity.noContent().build();  // 204
}
```

4. Use Constructor Injection

Constructor injection is generally recommended over field injection because it promotes immutability, makes dependencies explicit, and is easier to test. With constructor injection, you can easily see what dependencies a class requires by looking at its constructor. It also allows you to create immutable objects, which can help prevent bugs and improve thread safety. Field injection, on the other hand, can lead to hidden dependencies and makes it harder to write unit tests for the class.

```
// GOOD
```

```
@RestController
public class UserController {
    private final UserService userService;

    public UserController(UserService userService) {
        this.userService = userService;
    }
}

// AVOID
@RestController
public class UserController {
    @Autowired
    private UserService userService;  // Field injection
}
```

5. Version Your API

API versioning allows you to make changes to your API without breaking existing clients. You can version your API using the URL, request header, or query parameter. URL versioning is the most common and straightforward approach, where you include the version number in the URI path. This makes it clear to clients which version of the API they are using and allows you to maintain multiple versions simultaneously. Alternatively you can use header versioning or query parameter versioning, but these approaches can be less visible to clients and may require additional documentation to ensure proper usage.

```
@RestController
@RequestMapping("/api/v1/products")  // Version in URL
public class ProductController {
    // ...
}
```

Quick Reference

Annotation Summary

Annotation	Purpose
`@RestController`	Defines REST controller
`@RequestMapping`	Maps requests to handler
`@GetMapping`	Maps GET requests
`@PostMapping`	Maps POST requests
`@PutMapping`	Maps PUT requests

Annotation	Purpose
`@DeleteMapping`	Maps DELETE requests
`@PathVariable`	Extracts URI variable
`@RequestParam`	Extracts query parameter
`@RequestBody`	Binds request body to object
`@RequestHeader`	Extracts HTTP header
`@Valid`	Enables validation
`@ResponseStatus`	Sets HTTP status code

Common Patterns

```
// Basic CRUD endpoints
GET    /api/resources           // List all
GET    /api/resources/{id}      // Get one
POST   /api/resources           // Create
PUT    /api/resources/{id}      // Update
DELETE /api/resources/{id}      // Delete

// Nested resources
GET    /api/users/{id}/orders
POST   /api/users/{id}/orders

// Search/Filter
GET    /api/products?category=books&sort=price
```

Testing with cURL

cURL is a command-line tool for making HTTP requests. You can use it to test your REST API endpoints by sending various types of requests (GET, POST, PUT, DELETE) and inspecting the responses. Below are examples of how to use cURL to interact with a REST API built with Spring Boot.

```
# GET request
curl http://localhost:8080/api/books

# GET with path variable
curl http://localhost:8080/api/books/1

# POST request
curl -X POST http://localhost:8080/api/books \
  -H "Content-Type: application/json" \
  -d '{"title":"Spring Boot","author":"John Doe"}'
```

```
# PUT request
curl -X PUT http://localhost:8080/api/books/1 \
  -H "Content-Type: application/json" \
  -d '{"title":"Updated Title","author":"Jane Doe"}'

# DELETE request
curl -X DELETE http://localhost:8080/api/books/1
```

Summary

@RestController simplifies REST API development by:

- Automatically converting responses to JSON
- Providing specialized annotations for HTTP methods
- Integrating with Spring's dependency injection
- Supporting validation and exception handling
- Following RESTful principles

Key Points:

- Use `@RestController` for REST APIs
- Use `@Controller` for traditional MVC (returning views)
- Leverage HTTP method annotations (`@GetMapping`, `@PostMapping`, etc.)
- Return `ResponseEntity` for fine-grained control
- Implement proper exception handling
- Validate input with `@Valid`
- Follow REST best practices