Java Technologies

Lecture 5

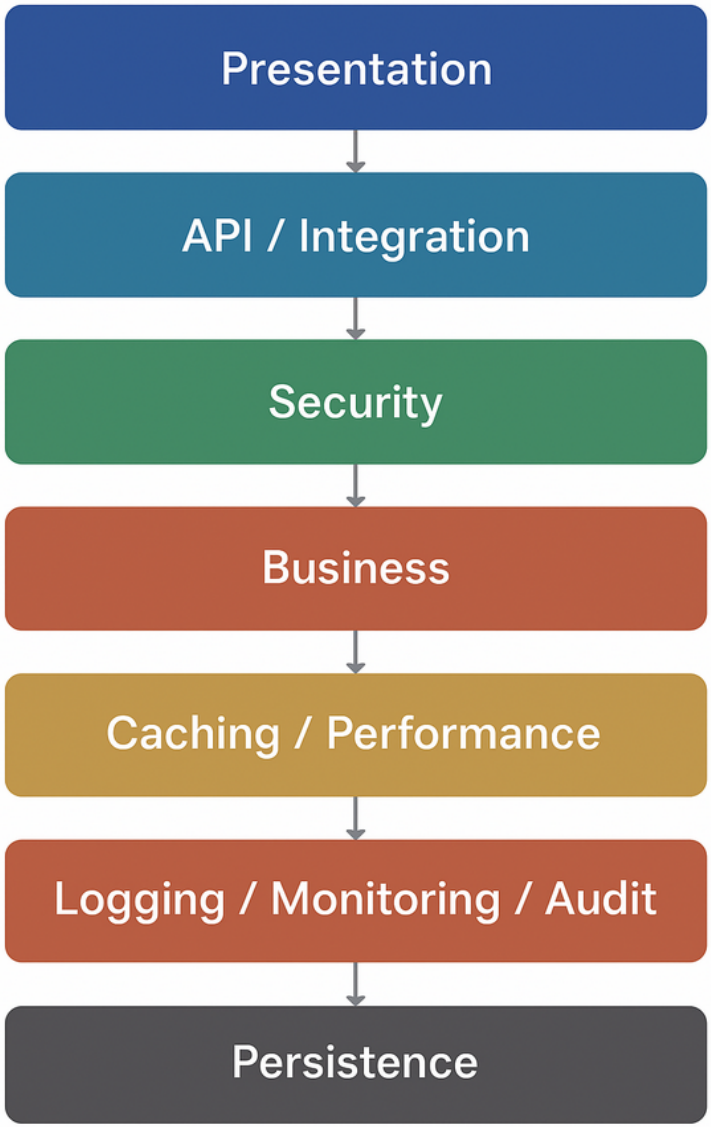
API Development: RESTful services

Fall, 2025

Agenda

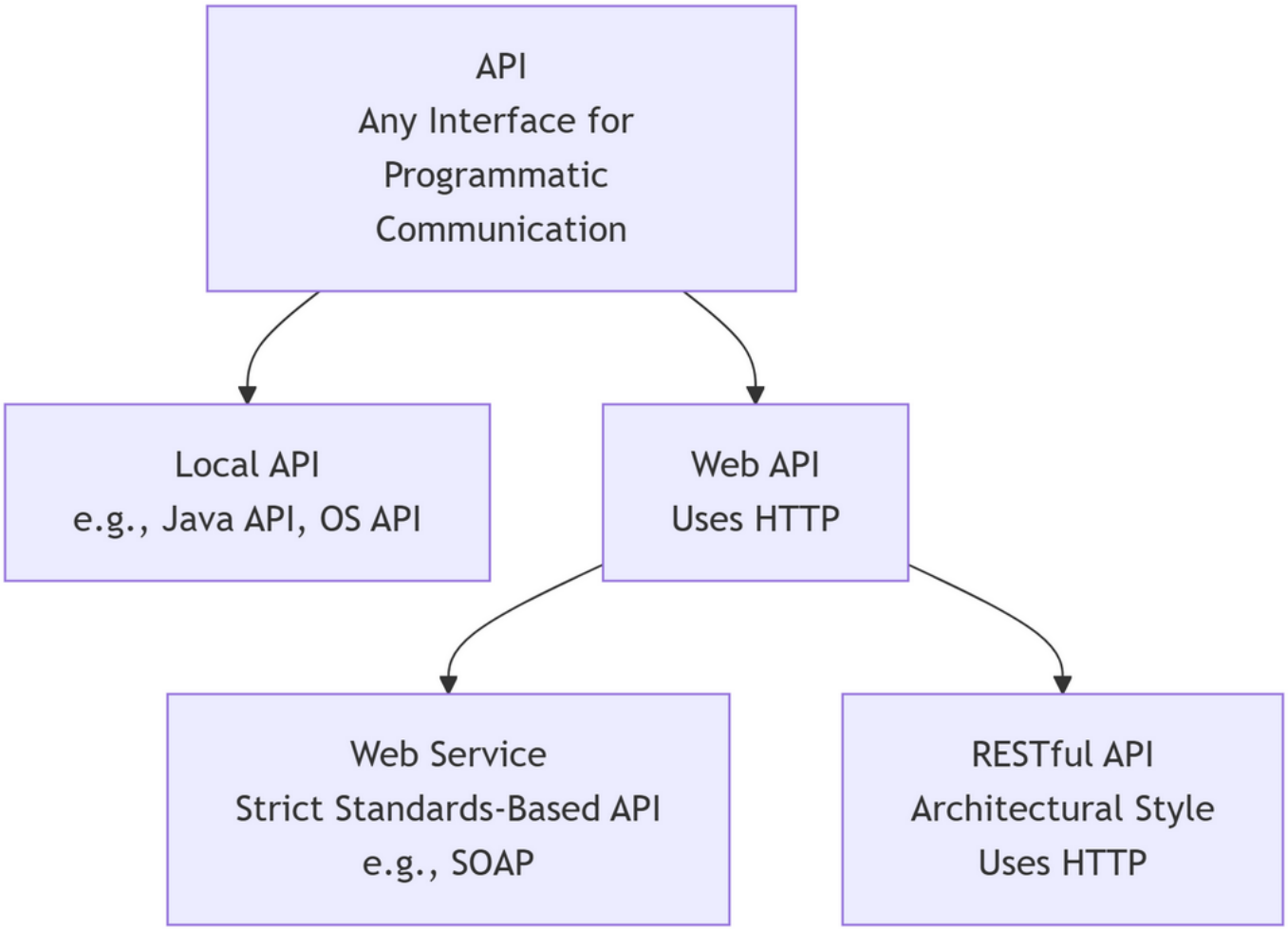
- Layered Architectures, Application Programming Interface (API)
- Service Oriented Architectures (SOA)
- Web API Development Paradigms
- REST, RESTful, The Richardson Maturity Model
- Creating REST Services in Spring
- Receiving Data, Responding, MIME Types, Naming Conventions
- Controller-Service-Repository Pattern
- The Data Transfer Object (DTO) Pattern
- Bean Validation, Handling Runtime Exceptions
- Web Filters vs. Spring Interceptors, CORS, Cookies
- HATEOAS (Hypermedia as the Engine of Application State)
- OpenAPI Specification (OAS), Swagger, Springdoc OpenAPI

Layered Architectures



Application Pogramming Interface (API)

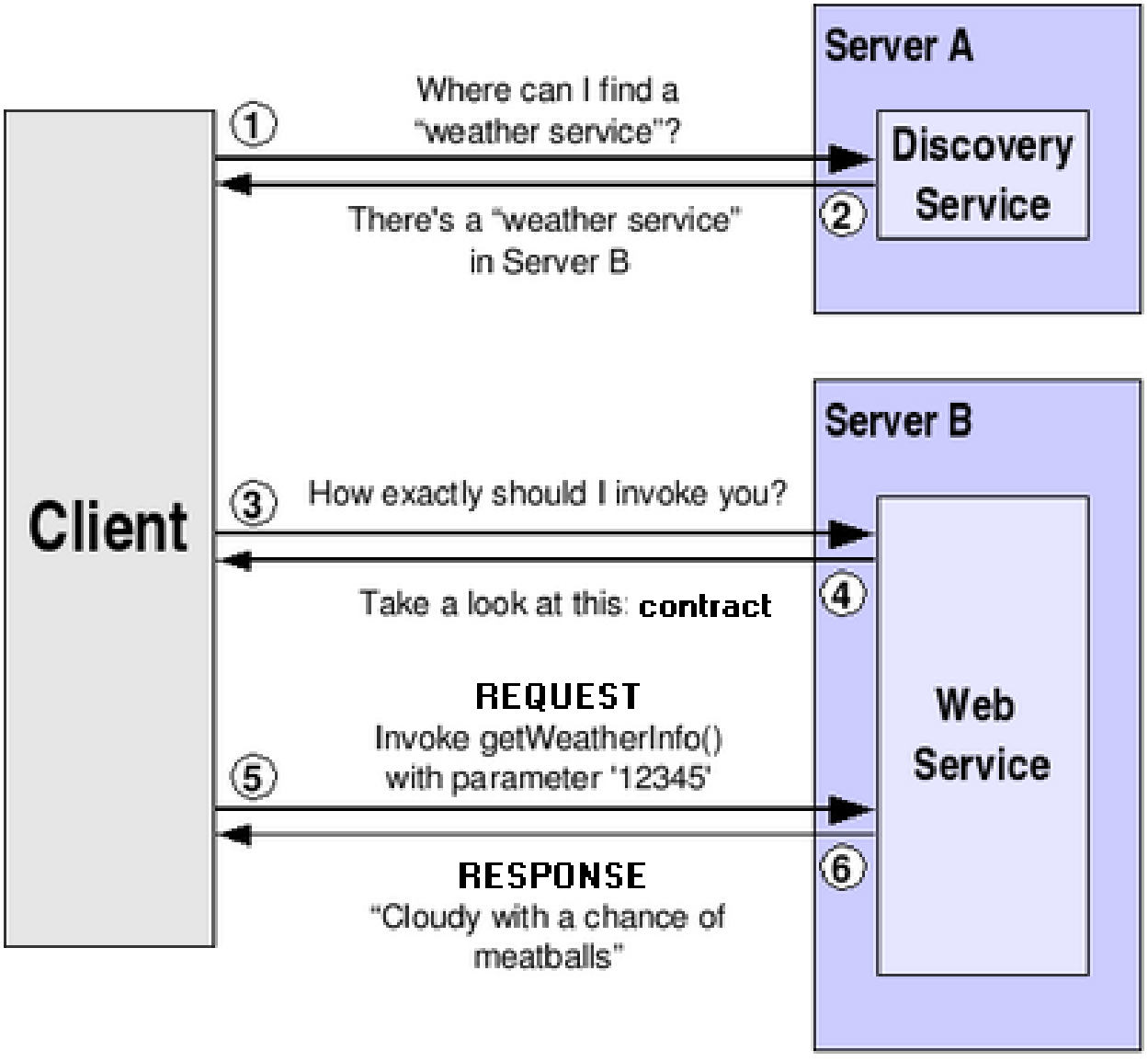
Does your application have an API? 😳



Service Oriented Architectures (SOA)

- We are in the context of developing distributed components that must be **shared by various** applications, in a **standard** way.
- How to expose a functionality in an **heterogeneous** (mixed) environment, independent of platform, vendor, or technology?
- How to **locate and invoke** this functionality?
- What exactly is a **service**? *An act of helpful activity.*
  - Someone is **offering**: How can you offer a service?
  - Someone is **using** it: How can you use a service?
  - **Protocols are needed**.
- **Service Oriented Architecture (SOA)**
  "A paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations."

How Should A Service Work?



Short History of Web Services

1. **The Pre-Web Services Era** (before 2000s)
   → The early era of distributed computing.
   → Proprietary RPC technologies: CORBA, DCOM, RMI.
2. **The Age of Standards** (c. 2000-2005)
   → Interoperability over custom integration.
   → XML based services: SOAP, WSDL, and UDDI.
3. **The Rise of Simplicity** (c. 2005-Present)
   → REST becomes the de facto standard.

Web API Development Paradigms

- **Presentation Layer**: External API (Frontend → Backend)
  - **REST** (REspresentational State Transfer)
    → Stateless, resource-oriented, uses standard HTTP verbs.
  - GraphQL (Graph Query Language)
    →Single endpoint for queries and mutations.
  - WebSockets
    → Full-duplex, persistent bidirectional connection.
  - SSE (Server-Sent Events)
    → Server push over a unidirectional HTTP connection.

Web API Development Paradigms

- **Presentation Layer**: External API (Frontend → Backend)
  - **REST** (REspresentational State Transfer)
    → Stateless, resource-oriented, uses standard HTTP verbs.
  - GraphQL (Graph Query Language)
    →Single endpoint for queries and mutations.
  - WebSockets
    → Full-duplex, persistent bidirectional connection.
  - SSE (Server-Sent Events)
    → Server push over a unidirectional HTTP connection.

- **Service Layer**: Internal API (Backend → Backend specific)
  - RPC (Remote Procedure Call)
    → Client invokes methods/functions on the server as if local.
  - Messaging
    → Asynchronous, decoupled communication using brokers.

What is REST?

- REST (REpresentational State Transfer) is **an architectural style** for designing networked applications - web services.
- Roy Fielding, 2000, PhD dissertation, *"Architectural Styles and the Design of Network-based Software Architectures"*.

- The core idea: **Resources and Representations**.

- **Architectural constraints**
  1. Client-Server (Separation of Concerns)
  2. Statelessness
  3. Cacheability
  4. Uniform Interface
  5. Layered System
  6. Code-On-Demand (Optional)

The Richardson Maturity Model

A way to evaluate how **"RESTful" a web API is**.
"RESTful" emphasizes proper adherence to REST constraints

- **Level 0**: The Swamp of POX (Plain Old XML)
  → Single endpoint (URI), RPC-like behavior over HTTP
- **Level 1**: Resources
  → Separate URIs for different resources.
- **Level 2**: HTTP Verbs
  → GET, POST, PUT, DELETE
- **Level 3**: Hypermedia (HATEOAS)
  → Hypermedia As The Engine Of Application State 
  → Responses include hyperlinks to other related resources/actions.

Technologies Used to Implement REST Services

- **Core Java/Jakarta EE**: **JAX-RS** (Java API for RESTful Web Services) – Standard API specifications. Implemented by:
  - Jersey (reference implementation, rich ecosystem)
  - RESTEasy (from JBoss/WildFly)
  - Apache CXF (also supports SOAP + REST)
- **Spring Ecosystem**
  - Spring MVC / Spring Boot (Spring Web)
  - Spring HATEOAS: Build Level 3 REST APIs with hypermedia.
  - Spring Data REST: Expose repository-based REST endpoints.
- **Other Frameworks**
  - Dropwizard, Micronaut, Quarkus  , Vert.x
- **Supporting Technologies**
  - JSON serialization and deserialization: Jackson, Gson, JSON-B.
  - Validation: Hibernate Validator (Bean Validation / JSR 380)
  - Security, Testing

Creating a REST Service in Spring

- Add the dependency `spring-boot-starter-web`.
  → Spring MVC, Jackson, Tomcat.
- Create a **`@RestController`** (specialization of `@Component`)

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String sayHello() {
        return "Hello, World!";
    }
    @GetMapping("/hello/{name}")
    public String sayHelloTo(@PathVariable String name) {
        return "Hello, " + name + "!";
    }
}
```

- Test the service from the browser, using `curl` or Postman.

```
http://localhost:8080/hello          → Hello, World!
http://localhost:8080/hello/Alice    → Hello, Alice!
```

HTTP Methods

Verb	Use Case	Idemp.	Safe
GET	Retrieve a resource or collection.	Yes	**Yes**
POST	Create a new resource.	No	No
PUT	Update a **complete** resource.	**Yes**	No
PATCH	Update **part** of a resource.	No	No
DELETE	Delete a resource.	**Yes**	No

- **Idempotent**: Multiple identical requests have the same effect as a single one. Idempotency is about **side effects** on the server.
- **Safe**: Does not change the server's state.
- Spring **annotations** for HTTP methods are are built on top of `@RequestMapping(method = RequestMethod.{VERB})`: `@GetMapping`, `@PostMapping`, `@PutMapping`, `@PatchMapping`, `@DeleteMapping`

Receiving Data in a REST Controller

- **Path Variables**: `@PathVariable`

```
http://localhost:8080/products/123
```

- **Query Parameters**: `@RequestParam`

```
http://localhost:8080/search?keyword=laptop&maxPrice=2000
```

- **Request Body**: `@RequestBody`

```
@PostMapping("/persons")
public String createPerson(@RequestBody Person person)
```

- **HTTP Headers**: `@RequestHeader`

```
@GetMapping("/user-agent")
public String getAgent(@RequestHeader("User-Agent") String agent)
```

- **Form Data**: `@RequestParam` or `@ModelAttribute`

```
@PostMapping("/upload")
public String upload(@RequestParam("file") MultipartFile file)
```

Responding from a REST Controller

- Return **void**, a **string** or an **object** (auto-converted to JSON).
- Return **ResponseEntity** when you need status codes/headers.

```
@GetMapping("/custom")
public ResponseEntity<String> customResponse() {
  return ResponseEntity.status(201).body("Resource created");
} // You may also use @ResponseStatus(HttpStatus.CREATED)
```

- Use **exception handling** 

```
@GetMapping("/error")
public User triggerError() {
    throw new RuntimeException("Something went wrong");
} // Don't do this, throw meaningful custom exceptions.
```

- Use **Resource/File response** for downloads/streams.

```
@GetMapping("/download")
public ResponseEntity<Resource> downloadFile() {
    Resource file = new FileSystemResource("example.txt");
    return ResponseEntity.ok().header(...).body(file);
}
```

Example: Returning an Object

```
public record User(int id, String name, List<String> roles) {}
```

```
@RestController
@RequestMapping("/users")
public class UserController {
    private List<User> users = Arrays.asList(
        new User(1, "Alice", Arrays.asList("ADMIN", "USER")),
        new User(2, "Bob", Arrays.asList("USER"))
    );

    @GetMapping("/{id}")
    public User getUserById(@PathVariable int id) {
        return users.stream().filter(u -> u.getId() == id)
                    .findFirst().orElse(null);
    }
} // An HttpMessageConverter is responsible for the conversion
  // (String, MappingJackson2/2Xml, ByteArrayHttp)HttpMessageConverter
```

GET /users/1

```
{ "id": 1, "name": "Alice", "roles": ["ADMIN", "USER"] }
```

Controlling the Response Format

A **MIME type (media type)** is a standard label that identifies the format of a piece of data.

```java
@RestController
public class XmlController {
    @GetMapping(value = "/user/xml", produces = "application/xml")
    public User getUserAsXml() {
        return new User(1, "Alice");
    }
} // Add the dependency jackson-dataformat-xml
```

```
<User>
  <id> 1 </id>
  <name> Alice </name>
</User>
```

```java
@RestController
public class ImageController {
    @GetMapping(value = "/image",
                produces = MediaType.IMAGE_PNG_VALUE)
    public ResponseEntity<Resource> getImage() {
        Resource image = new ClassPathResource("duke-power.png");
        return ResponseEntity.ok().contentType(MediaType.IMAGE_PNG)
                             .body(image);
    }
}
```

Content Negotiation

- The client can tell the server in which format it wants the response (JSON, XML, etc.) using either the **`Accept` HTTP header** (recommended), or the `Content-Type` header.

```
GET /students/1     Accept: application/json
GET /students/1     Accept: application/xml
```

- The server returns the resource in the requested format.

```
    @GetMapping("/{id}", produces = {
       MediaType.APPLICATION_JSON_VALUE,
       MediaType.APPLICATION_XML_VALUE })
    public Student getStudent(@PathVariable Long id) {
        return new Student(id, "Alice"); }
```

- Other forms of negotiation: URL suffixes (`/resource.xml`) and query parameters (`/resource?format=xml`) ← Don't do this.
- Return appropriate status code, when there is a problem: 406 (Not Acceptable) or 415 (Unsupported Media Type).

Naming Conventions

- Use **plural nouns** for resources – not verbs.

```
✔ GET /products                    # Retrieve all products
✔ POST /products                   # Create a new product
✔ DELETE /products/123             # Delete a specific product
✘ GET /getAllProducts              # Incorrect
✘ GET /product                     # Incorrect
```

- **Nest resources** to show relationships.

```
✔ GET /products/123/reviews        # Get all reviews of a product
✔ POST /products/123/reviews       # Create a new review
✘ GET /reviews?productId=123       # Sorting, filtering, pagination
```

- Use **kebab-case** for resources and **camelCase** for parameters.

```
✔ GET /user-accounts ✘ GET /user_accounts ✘ GET /userAccounts
✔ GET /products?minPrice=100&maxPrice=500
✘ GET /products?min-price=100&max-price=500
```

- **Version** your API (using URL path or `Accept` header)

```
✔ GET /v1/products/123          ✔ POST /v2/orders
```

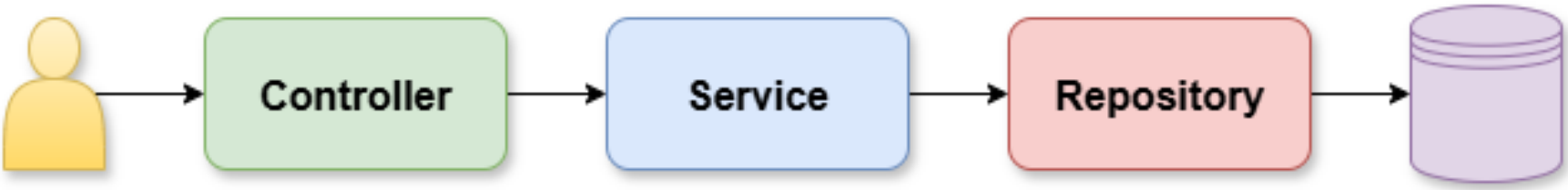
HTTP Status Response Codes

- **2xx - Success**
  - **200 OK** – Standard success response
  - 201 Created, 202 Accepted, 204 No Content
- **4xx - Client Errors**
  - **404 Not Found** – Resource not found
  - 400 Bad Request, 401 Unauthorized, 403 Forbidden, 405 Method Not Allowed, 422 Unprocessable Entity, etc.
- **5xx - Server Errors**
  - **500 Internal Server Error** – Unexpected server issue
  - 502 Bad Gateway, 503 Service Unavailable, etc.
- Match method to status code properly.

```
POST /users          → 201 Created
GET /users/123       → 200 OK (or 404 if not found)
PUT /users/123       → 200 OK (resource updated) or 204 No Content
DELETE /users/123    → 204 No Content
```

- Don't overuse 200. Don't say it OK when it's NOT OK.

Controller-Service-Repository Pattern



- **Controller**: Handles HTTP requests and responses

```
@RestController
@RequestMapping("/users")
public class UserController {
    @Autowired
    private UserService userService;
} //  Why not inject repository into the controller?
```

- **Service**: Business logic. 🤔

```
@Service
public class UserService {
    @Autowired
    private UserRepository userRepo;
}
```

- **Repository**: Data persistence (usually a `JpaRepository`)

The Data Transfer Object (DTO) Pattern

- A DTO is a **plain object that carries data** between layers of an application: Client $\leftrightarrow$ Controller, Controller $\leftrightarrow$ Service, or between microservices. It **does not contain any business logic**.
- **Why use DTOs** and not the entity objects?
  - **Encapsulation**: Hide internal entity structure.
  - **Decoupling**: Keep API contracts separate from database entities.
  - **Validation, Transformation**: Carry only the required fields.
- Best practice is to have:
  - **Request DTO**: Data coming from the client to the server.
  - **Response DTO**: Data going from the server to the client.

Example: Using DTOs

```
public record UserRequestDto
        (String username, String password, String email) {}
```

```
public record UserResponseDto
        (long id, String username, String email) {
    public UserResponseDto(User entity) { this.id = id; ... }
}
```

```
@RestController
@RequestMapping("/users")
public class UserController {
  @Autowired private UserService userService;

  @PostMapping
  public ResponseEntity<UserResponseDto>
        createUser(@RequestBody UserRequestDto requestDto) {
    UserResponseDto responseDto = userService.createUser(requestDto);
    return ResponseEntity.status(HttpStatus.CREATED)
                         .body(responseDto);
  }
}
```

Java/Jakarta EE Bean Validation

- **Standard specifications** for validating objects.
- Declare validation rules on fields, methods, or constructors.

```
@NotBlank(message = "Name is required")
@Size(min = 2, max = 20, message = "Name must be 2-20 characters")
private String name;
```

- Spring Boot integrates seamlessly with Bean Validation, using the starter `spring-boot-starter-validation`.
- **Validate incoming request DTOs** before processing them.

```
@PostMapping
public ResponseEntity<ResponseUserDto> createUser(
        @Valid @RequestBody RequestUserDto requestDto)  ...
```

- Spring will automatically respond with **400 Bad Request** if validation fails, along with a detailed error message.

Handling Runtime Exceptions

- If a runtime exception occurs in a REST controller, Spring returns a **generic HTTP 500 response** with a JSON body like:

```
{
  "timestamp": "2025-08-22T12:00:00.000+00:00",
  "status": 500,
  "error": "Internal Server Error",
  "path": "/api/users/1"
}
```

- Use `@ResponseStatus` for custom exceptions simple mapping.

```
@ResponseStatus(HttpStatus.NOT_FOUND)
public class UserNotFoundException extends RuntimeException {...}
```

- Use `@ExceptionHandler` for **controller-specific** handling.
- Use `@ControllerAdvice` for **global exception** handling.

Example: Using an @ExceptionHandler

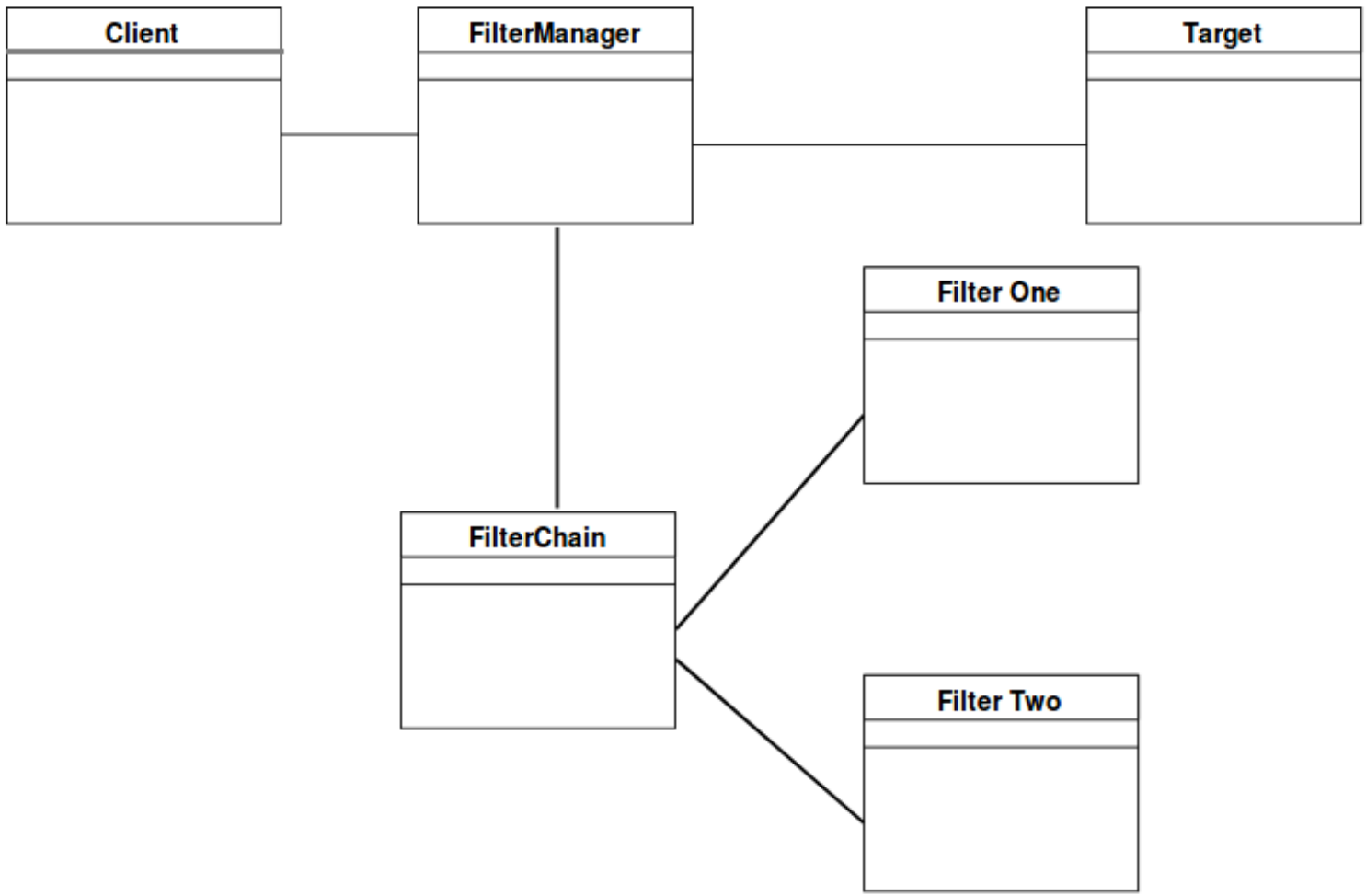
```
@RestController
@RequestMapping("/api/users")
public class UserController {

  @GetMapping("/{id}")
  public User getUser(@PathVariable Long id) {
    return repo.findById(id)
               .orElseThrow(() -> new UserNotFoundException(id));
  }

  @ExceptionHandler(UserNotFoundException.class)
  @ResponseStatus(HttpStatus.NOT_FOUND)
  public Map<String, String> handleNotFound(UserNotFoundException ex) {

        return Map.of(
            "error", ex.getMessage(),
            "status", "404"
        );
        // Include helpful info in the error response.
  }
}
```

Example: Using a @ControllerAdvice

```
@ControllerAdvice
public class GlobalExceptionHandler {

  @ExceptionHandler(PersonNotFoundException.class)
  public ResponseEntity<Map<String, String>>
          handlePersonNotFound(PersonNotFoundException ex) {

      Map<String, String> body = Map.of(
            "error", ex.getMessage(), "status", "404");
      return new ResponseEntity<>(body, HttpStatus.NOT_FOUND);
  }

  @ExceptionHandler(Exception.class)
  public ResponseEntity<Map<String, String>>
          handleAll(Exception ex) {

      Map<String, String> body = Map.of(
            "error", ex.getMessage(), "status", "500");
      return new ResponseEntity<>(
          body, HttpStatus.INTERNAL_SERVER_ERROR);
  }
}
```

Intercepting Filter Design Pattern

Insert **processing logic before and/or after** the actual processing of a request, without modifying the core business logic.



Web Filters (Servlet Filters)

- Servlet filters are part of the **Java EE specifications** and operate at a servlet-container level, **outside Spring app context**.
- **Common use cases**: Authentication and authorization, logging and auditing, compression or decompression of data streams, URL rewriting and request header manipulation, handling CORS.

```
@WebFilter("/*")  // applies to all requests
public class LoggingFilter implements Filter {
    @Override
    public void doFilter(ServletRequest request,
                         ServletResponse response,
                         FilterChain chain)
            throws IOException, ServletException {

        System.out.println("Request received");
        chain.doFilter(request, response); // continue
        System.out.println("Response sent");
    }
}
```

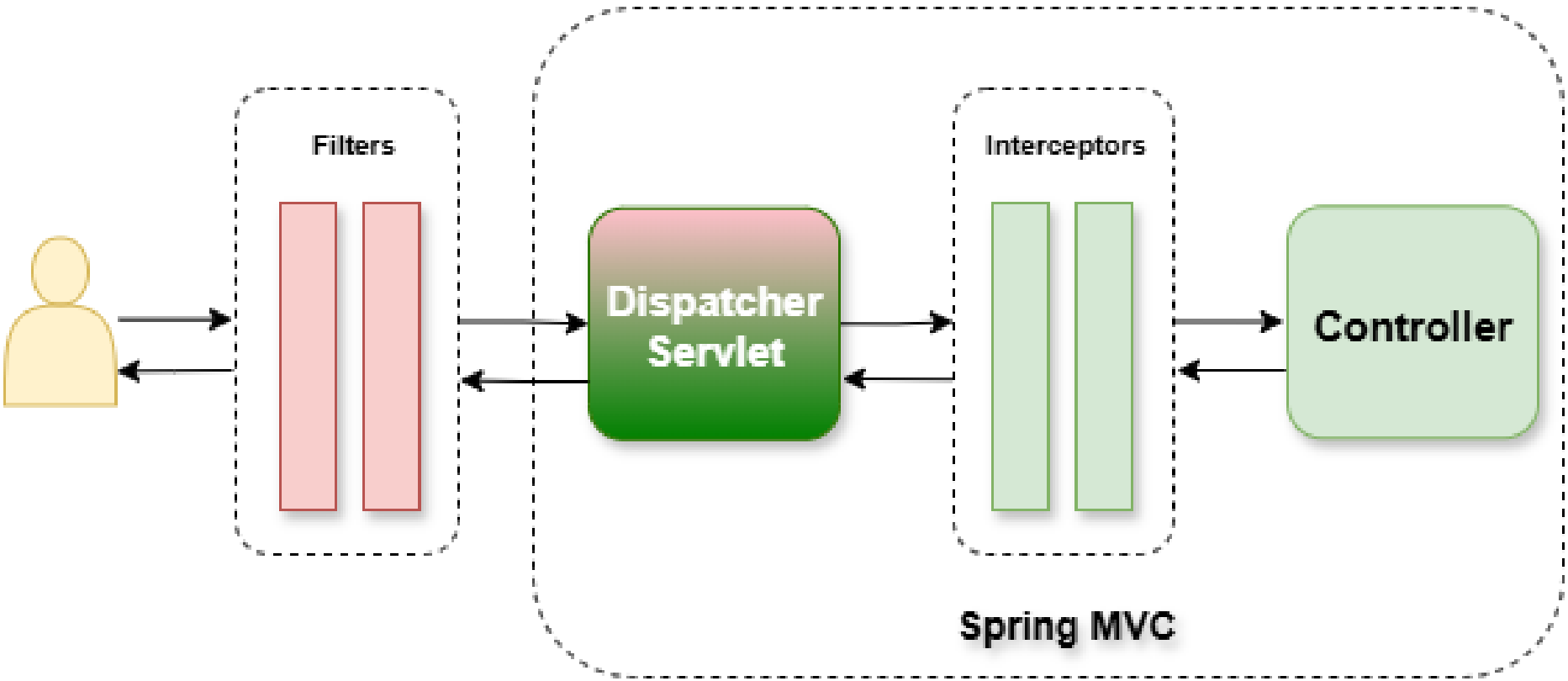
Spring Interceptors

- Interceptors are a feature of the Spring MVC framework and operate at a higher level, **within the Spring app context**.
- They are executed **after the `DispatcherServlet`** has received the request but **before the controller method** is executed.

```
@Component
public class LoggingInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request,
                             HttpServletResponse response,
                             Object handler) {
        System.out.println("Before controller");
        return true; // continue to controller
    }
} //  postHandle and afterCompletion can be overriden also.
```

- Must be registered via `WebMvcConfigurer`.

Web Filters vs. Spring Interceptors



- Use filters for things that do not depend on Spring app context.
- Use interceptors for concerns related to Spring MVC controllers.

What Exactly Means CORS?

- **Cross-Origin Resource Sharing**
- Your frontend app runs at `http://localhost:3000` (React).
- Your backend API runs at `http://localhost:8080` (Spring).
- Browser blocks cross-origin request for security reasons:

```
fetch("http://localhost:8080/api/data")
Origin = http://localhost:3000
Target = http://localhost:8080
```

- Enable CORS on the backend (Access-Control-Allow-Origin).

```
@Configuration
public class WebConfig implements WebMvcConfigurer {
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
                .allowedOrigins("http://localhost:3000")
                .allowedMethods("GET","POST","PUT","DELETE");
    }        // .allowedHeaders, .allowCredentials
} // Can be configured per controller or per method
```

Setting Cookies in the Response

- **Session Management** in traditional web apps.
- **Authentication**: JWT or custom tokens.
- Client-specific **preferences**: theme, language, UI layout, etc.
- **Tracking / Analytics**: Set by analytics tools.
- Spring uses **Java EE Servlet API** to handle cookies.

```
@GetMapping("/set-cookie")
public ResponseEntity<String> setCookie(HttpServletResponse response) {
    Cookie cookie = new Cookie("userId", "12345");
    cookie.setPath("/");
    // Send this cookie with every request to this domain,
    // regardless of the path.

    cookie.setHttpOnly(true);  // prevents JavaScript access
    cookie.setMaxAge(24 * 60 * 60); // 1 day

    response.addCookie(cookie);
    return ResponseEntity.ok("Cookie set");
}
```

HATEOAS

- HATEOAS = Hypermedia as the Engine of Application State
- A client should be able to interact with a RESTful service entirely through **hyperlinks provided by the server**.
  - Instead of returning just raw data, your API responses also contain links that describe actions the client can take next.
  - The client discovers functionality by following links, not by hardcoding endpoints.
- **No need to have prior knowledge** of the URI structure.
- The API becomes **self-descriptive**.
- The server can **evolve independently** without breaking clients.
- **Disadvantages**: Server/Client-side overhead, larger payloads, "chatty" API, lack of standardization, limited adoption and tooling.

HATEOAS Example

- Without HATEOAS

```
{
  "id": 1,
  "name": "Alice"
}
```

- With HATEOAS (HAL syntax)

```
{
  "id": 1,
  "name": "Alice",
  "_links": {
    "self": { "href": "http://localhost:8080/api/persons/1" },
    "all":  { "href": "http://localhost:8080/api/persons" }
  }
}
```

The `_links` tell the client:

how to get this person again: `self`

how to navigate to the list of all persons: `all`

Hypertext Application Language (HAL)

- HAL is a **lightweight JSON (or XML) format** for representing resources and hypermedia links.
- It defines a **convention** for how to represent:
  - Resources (the data)
  - Links (navigation/action hints)
  - Embedded resources (nested objects)
- You can **add any link relation (rel)**:
  - `self` → the canonical link to this resource (mandatory).
  - `collection / all` → link to the collection resource.
  - `next / prev` → pagination (next page, previous page).
  - `first / last` → start/end pages in paginated responses.
  - `create` → endpoint to create a new resource.
  - `update / delete` → actions on the current resource.

Spring HATEOAS

spring-boot-starter-hateoas

```
@GetMapping("/{id}")
public EntityModel<Person> one(@PathVariable Long id) {
  Person person = repo.findById(id).orElseThrow();

  return EntityModel.of(person,
    linkTo(methodOn(PersonController.class).one(id)).withSelfRel(),
    linkTo(methodOn(PersonController.class).all()).withRel("all"));
  }

@GetMapping
public CollectionModel<EntityModel<Person>> all() {
  var persons = repo.findAll().stream()
        .map(p -> EntityModel.of(p,
            linkTo(methodOn(PersonController.class).one(p.getId()))
            .withSelfRel()))
        .toList();

  return CollectionModel.of(persons,
    linkTo(methodOn(PersonController.class).all()).withSelfRel());
}
```

OpenAPI Specification (OAS) & Swagger

- **OpenAPI Specification (OAS)** is a formal, language-agnostic **standard for describing RESTful APIs**.
- It allows both humans and machines to understand an API's capabilities without having to access its source code or documentation.
- **Design-first approach**: The API contract is defined and agreed upon before a single line of code is written.
- An **OAS document** (typically in YAML or JSON format) defines the API's endpoints, operations, request and response models, authentication methods, and more.

OpenAPI Specification (OAS) & Swagger

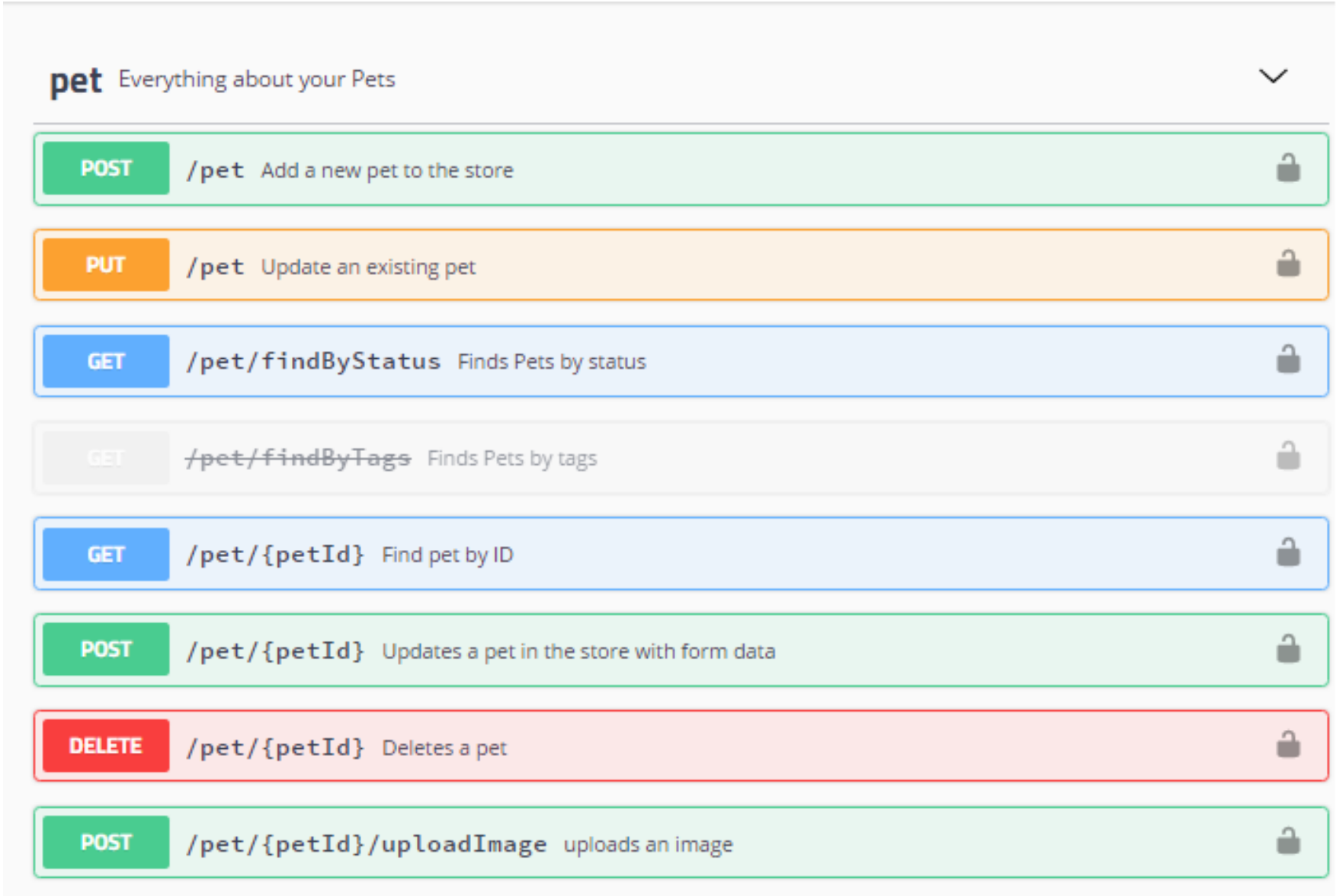
- **OpenAPI Specification (OAS)** is a formal, language-agnostic **standard for describing RESTful APIs**.
- It allows both humans and machines to understand an API's capabilities without having to access its source code or documentation.
- **Design-first approach**: The API contract is defined and agreed upon before a single line of code is written.
- An **OAS document** (typically in YAML or JSON format) defines the API's endpoints, operations, request and response models, authentication methods, and more.
- **Swagger** is a set of open-source tools that work with OAS.
  - **Swagger UI**: An interactive web UI to explore and test APIs
  - **Swagger Editor**: Browser-based editor OAS documents.
  - **Swagger Codegen**: Code generator for various languages.

Example of OAS Document

```yaml
# hello.yaml
# Description of the "Hello World" service
openapi: 3.0.0
info:
  title: Hello World API
  version: 1.0.0
paths:
  /hello:
    get:
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                type: object
                properties:
                  message:
                    type: string
```

```
GET /hello → { "message": "Hello World" }
```

Swagger UI


pet Everything about your Pets
POST /pet Add a new pet to the store
PUT /pet Update an existing pet
GET /pet/findByStatus Finds Pets by status
GET /pet/findByTags Finds Pets by tags
GET /pet/{petId} Find pet by ID
POST /pet/{petId} Updates a pet in the store with form data
DELETE /pet/{petId} Deletes a pet
POST /pet/{petId}/uploadImage uploads an image

Springdoc OpenAPI

- Add the starter **`springdoc-openapi-starter-webmvc-ui`**
- Spring scans `@RestController` and Spring MVC endpoints and automatically exposes the following URLs :

```
http://localhost:8080
    /v3/api-docs          → raw OpenAPI JSON
    /swagger-ui.html      → interactive Swagger UI
```

- **Enhance documentation** with annotations.

```java
@GetMapping("/{id}")
@Operation(summary = "Get a person by ID",
           description = "Returns a single person")
@ApiResponse(responseCode = "200",
             description = "Person found")
@ApiResponse(responseCode = "404",
             description = "Person not found")
public ResponseEntity<Person> getPerson(@PathVariable Long id) {
    // ...
}
```

Spring Data REST

- Framework that **simplifies the creation of RESTful APIs** by automatically exposing Spring Data repositories as hypermedia-driven web services → reduces boilerplate code.
- It automatically **generates endpoints for CRUD operations**.
- It adheres to the **HATEOAS** principle.
- It provides out-of-the-box support for features like **pagination, sorting, and dynamic filtering**.
- **Convention over configuration** – still customizable.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
```

Example: Using Spring Data REST

- Create a JPA repository class.

```
public interface BookRepository extends JpaRepository<Book, Long>
```

- The following endpoint are generated automatically:

```
GET    /books        → list all books
POST   /books        → create a new book
GET    /books/{id}   → get book by ID
PUT    /books/{id}   → update book
DELETE /books/{id}   → delete book
```

- The JSON responses include navigational links:

```
GET /books/1
{ "id":1, "title":"The Iliad and The Odyssey", "author":"Homer",
  "_links": {
    "self": { "href": "http://localhost:8080/books/1" },
    "publisher":
        { "href": "http://localhost:8080/books/1/publisher" }
  }
}
```

Spring Data REST Customization

- Change endpoint names/paths.

```
@RepositoryRestResource(path = "library")
public interface BookRepository extends JpaRepository<Book, Long>
```

- Expose only certain methods.

```
@RestResource(exported = false) // hides this method
void deleteById(Long id);
```

- Control JSON output using projections for custom fields.

```
@Projection(name = "summary", types = { Book.class })
public interface BookSummary {
    String getTitle();
} // GET /books?projection=summary
```

- Create custom controllers.
- Create event handlers, such `@HandleBeforeSave`.
- Configure global parameters: IDs, base path, CORS.

REST vs GraphQL

- **REST:**
  - Fixed endpoints (e.g., /users, /orders)
  - Multiple requests for different resources
  - Over-fetching or under-fetching possible
- **GraphQL:**
  - Single endpoint
  - Client specifies exactly what data is needed
  - Reduces over-fetching and under-fetching
- **Summary:** GraphQL offers more flexibility and efficiency for complex queries, while REST is simple and widely adopted.

REST vs WebSockets

- **REST:**
  - Request-response model
  - Stateless, each request independent
  - Good for CRUD APIs
- **WebSockets:**
  - Persistent, bidirectional connection
  - Server can push data to clients in real-time
  - Ideal for chat, notifications, live dashboards
- **Summary:** REST is simple and works for most APIs, WebSockets are for real-time, low-latency communication.