**API Strategy and Design**

**Richardson Maturity Model - RESTful API**

**The Richardson Maturity Model (RMM), proposed by Leonard Richardson,** is a model used to assess the maturity of a RESTful API based on its implementation levels. It consists of four levels, each representing a stage of maturity in the design and implementation of RESTful principles. Let's delve into each level to understand how APIs progress through the model. In determining the maturity of a service, Richardson emphasized three main factors. They include:

* URI
* HTTP Methods
* HATEOAS (Hypermedia)

**URI**

A Uniform Resource Identifier (URI) is a unique sequence of characters used by web technologies to identify resources on the web.

**HTTP Methods**

Hypertext Transfer Protocol (HTTP) is a protocol used to transfer hypermedia documents. HTTP requests are sent to servers by HTTP clients in the form of request messages. HTTP defines a set of request methods to specify the action to be taken on a given resource.

* **GET:** The GET method retrieves a representation of the specified resource.
* **POST:** A POST request transmits data to the server.
* **PUT:** The PUT method replaces all existing representations of the resource.
* **PATCH:** A PATCH request makes partial changes to a resource.
* **DELETE:** The DELETE method removes the specified resource.

**HATEOAS**

HATEOAS (Hypermedia as the Engine of Application State ) refers to discoverability. The client can interact with a REST API solely through the server’s responses. It is a self-documentary Hypermedia. Clients need not refer to any documentation to interact with a new API.

**REST services are divided into maturity levels according to the Richardson Maturity Model.**

| **Level** | **Description** | **Key Feature** |
| --- | --- | --- |
| **0** | **Swamp of POX**: Single URI,  all actions via POST | RPC-like, no REST principles |
| **1** | Resources | Uses distinct URIs for each resource |
| **2** | HTTP Verbs | Uses GET, POST, PUT, DELETE appropriately |
| **3** | HATEOAS | Hypermedia links in responses to guide client |

**♻️ 2. API Design Reusability**

Design your API to maximize reuse across modules, teams, and products.

**🔧 Guidelines:**

* ✅ **Use standard HTTP methods** (GET, POST, PUT, DELETE)
* ✅ **Return standard response envelopes** (e.g., with status, message, data)
* ✅ **Define common DTOs or schemas** for errors, pagination, metadata
* ✅ **Use OpenAPI/Swagger** specs for consistent documentation and codegen
* ✅ **Create API versioning strategy** (e.g., /api/v1/)

**✅ Reusable Elements:**

* PaginationDTO
* ErrorResponse
* SortingRequest
* StandardHeader

**📛 3. API Naming Conventions**

Consistent naming improves readability and maintainability.

| **Element** | **Convention** |
| --- | --- |
| Resources | Use **plural nouns**: /users, /orders |
| Sub-resources | Nested routes: /users/{id}/orders |
| Actions | Use **verbs only when needed**: /users/{id}/deactivate |
| Query params | For filtering/sorting: /orders?status=pending&sort=date |
| Versions | /api/v1/users, not query params |
| File upload | Use /upload, not /uploadFile.do |

**🔧 DOs:**

✅ /users  
✅ /users/{id}  
✅ /users/{id}/orders  
✅ /orders?status=shipped&sort=asc

**❌ DON'Ts:**

🚫 /getUserById  
🚫 /doLogin  
🚫 /listAllOrders

**REST services are divided into maturity levels according to the Richardson Maturity Model.**

**Level 0: The Swamp of POX (Plain Old XML)**

**Characteristics:**

* Uses HTTP as a transport protocol but doesn't leverage its features.
* Often relies on a single endpoint (e.g., POST /api) for all operations.
* Treats HTTP as a tunnel for remote procedure calls (RPC).
* Doesn't follow resource-oriented architecture.
* Encode the **operation type and parameters in the request body**
* Often use **XML** (or JSON) for data transfer

**Key Issues:**

* Lack of resource-based URLs.
* No use of HTTP methods like GET, POST, PUT, DELETE.
* Absence of hypermedia controls.

**HTTP Request (XML-based RPC over HTTP)**

POST /api HTTP/1.1

Content-Type: application/xml

Host: example.com

**Request Body (XML):**

<request>

<action>getUser</action>

<userId>123</userId>

</request>

🔍 No meaningful use of HTTP verbs or URLs; everything is tunneled through the body.

**💻 HTTP Response (XML)**

<response>

<status>success</status>

<user>

<id>123</id>

<name>Alice</name>

<email>alice@example.com</email>

</user>

</response>

**Level 1: Resources**

**Characteristics:**

* Introduces the concept of resources as the key abstraction.
* Resources are exposed as individual URLs (e.g., /users, /products).

**Key Issues:**

* May not use HTTP methods correctly (e.g., using only POST for all operations).
* Does not leverage hypermedia controls.

**🎯 Example: "User Management API" — Level 1 Design**

**🔵 HTTP Request to Fetch a User**

POST /users/123 HTTP/1.1

Content-Type: application/json

**Request Body:**

{

"action": "getUser"

}

**🔵 HTTP Request to Create a User**

POST /users HTTP/1.1

Content-Type: application/json

**Request Body:**

{

"name": "Alice",

"email": "alice@example.com"

}

**✅ What's Improved from Level 0?**

| **Feature** | **Level 0** | **Level 1** |
| --- | --- | --- |
| Single Endpoint | /api | ❌ |
| Resource-based URIs | ❌ | ✅ /users, /users/123 |
| Meaningful Paths | ❌ | ✅ |
| HTTP Verbs Used Properly | ❌ | ❌ (still mostly POST) |

🧠 You now model **"nouns" as URIs** — like users, orders, products.

**⚠️ What's Still Missing?**

* You're still **misusing POST** for reads/updates
* Not leveraging **HTTP semantics** (GET/PUT/DELETE)
* No **hypermedia (HATEOAS)**

**🛠 Spring Boot Code Example (Level 1)**

Here’s how a Level 1 controller might look:

@RestController

@RequestMapping("/users")

public class UserController {

@PostMapping("/{id}")

public ResponseEntity<User> getUser(@PathVariable int id, @RequestBody Map<String, Object> request) {

if ("getUser".equals(request.get("action"))) {

User user = userService.findById(id);

return ResponseEntity.ok(user);

}

return ResponseEntity.badRequest().build();

}

@PostMapping

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

User saved = userService.save(user);

return ResponseEntity.ok(saved);

}

}

Note how:

* We're using resource paths: /users, /users/{id}
* But still using **POST for everything**

**Level 2: HTTP Verbs**

**Characteristics:**

* Adheres to proper usage of HTTP methods (GET, POST, PUT, DELETE).
* Uses HTTP headers (e.g., Content-Type, Accept) for negotiation and metadata.
* Each HTTP method has a specific role in manipulating resources (e.g., GET for retrieval, PUT for update).
* Leverages status codes (e.g., 200 OK, 404 Not Found) to convey operation results.

**Key Advancements:**

* Enhances API consistency and predictability.
* Facilitates better understanding and debugging of API interactions.
* Enables caching and optimization based on HTTP features.

**📘 Example Domain: User Management**

We'll manage a User resource using proper HTTP methods.

**🔧 HTTP Verbs in Action**

| **Operation** | **HTTP Verb** | **URI** | **Description** |
| --- | --- | --- | --- |
| Get all users | GET | /users | Fetch all users |
| Get single user | GET | /users/{id} | Fetch a specific user |
| Create new user | POST | /users | Create a new user |
| Update existing user | PUT | /users/{id} | Replace user info completely |
| Partially update user | PATCH | /users/{id} | Modify specific fields |
| Delete user | DELETE | /users/{id} | Delete a specific user |

**🧪 Sample HTTP Requests**

**✅ Create a User**

POST /users

Content-Type: application/json

{

"name": "Alice",

"email": "alice@example.com"

}

**✅ Get User by ID**

GET /users/123

**✅ Update User**

PUT /users/123

Content-Type: application/json

{

"name": "Alice B.",

"email": "aliceb@example.com"

}

**✅ Delete User**

DELETE /users/123

**🛠 Spring Boot Code Example (Level 2)**

@RestController

@RequestMapping("/users")

public class UserController {

@GetMapping

public List<User> getAllUsers() {

return userService.getAll();

}

@GetMapping("/{id}")

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

User user = userService.findById(id);

return user != null ? ResponseEntity.ok(user) : ResponseEntity.notFound().build();

}

@PostMapping

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

return ResponseEntity.status(HttpStatus.CREATED).body(userService.save(user));

}

@PutMapping("/{id}")

public ResponseEntity<User> updateUser(@PathVariable int id, @RequestBody User updatedUser) {

return ResponseEntity.ok(userService.update(id, updatedUser));

}

@DeleteMapping("/{id}")

public ResponseEntity<Void> deleteUser(@PathVariable int id) {

userService.delete(id);

return ResponseEntity.noContent().build();

}

}

**✅ Level 2 Benefits**

| **Feature** | **Benefit** |
| --- | --- |
| Uses URI as resource | /users, /users/1 |
| HTTP verbs for action | Clear intent via GET/POST/PUT |
| Cacheable operations | GET responses can be cached |
| Standardized errors | Can return 404, 204, 400, etc. |

**🚫 Still Missing (for Level 3)**

* No **hypermedia links** (e.g., \_links.self, \_links.orders)
* Client doesn’t yet discover how to proceed from responses

**Level 3: Hypermedia Controls (HATEOAS)**

**Characteristics:**

* Represents the highest level of RESTful maturity.
* Integrates hypermedia controls within API responses (e.g., links, actions).
* Allows clients to navigate the API dynamically without prior knowledge of all endpoints.
* Responses include links to related resources and actions that clients can perform next.

**Key Advantages:**

* Improves API discoverability and usability.
* Reduces client coupling to API structure, promoting flexibility and evolution.
* Enables server-driven application state (clients follow links to discover and interact with resources).

**📘 Use Case: User Management**

Let’s enhance our existing RESTful API to include **hypermedia links** in the responses using Spring HATEOAS.

**⚙️ Technologies Used**

* **Spring Boot 3.2+**
* **Spring HATEOAS** (org.springframework.boot:spring-boot-starter-hateoas)
* **JSON HAL (Hypertext Application Language)** format

**🔧 Add HATEOAS Dependency in pom.xml**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-hateoas</artifactId>

</dependency>

**🧱 Example Response: GET /users/123**

{

"id": 123,

"name": "Alice",

"email": "alice@example.com",

"\_links": {

"self": {

"href": "http://localhost:8080/users/123"

},

"update": {

"href": "http://localhost:8080/users/123"

},

"delete": {

"href": "http://localhost:8080/users/123"

},

"all-users": {

"href": "http://localhost:8080/users"

}

}

}

🧠 Now the **client doesn’t need to guess** — it follows the links to perform the next action.

**🧠 HATEOAS Spring Boot Example Code**

**📄 User.java**

public class User {

private int id;

private String name;

private String email;

// Constructors, Getters, Setters

}

**📄 UserController.java**

@RestController

@RequestMapping("/users")

public class UserController {

@Autowired

private UserService userService;

@GetMapping("/{id}")

public EntityModel<User> getUserById(@PathVariable int id) {

User user = userService.findById(id);

EntityModel<User> resource = EntityModel.of(user);

WebMvcLinkBuilder linkToSelf = linkTo(methodOn(this.getClass()).getUserById(id));

WebMvcLinkBuilder linkToAllUsers = linkTo(methodOn(this.getClass()).getAllUsers());

resource.add(linkToSelf.withSelfRel());

resource.add(linkToAllUsers.withRel("all-users"));

resource.add(linkTo(methodOn(this.getClass()).deleteUser(id)).withRel("delete"));

resource.add(linkTo(methodOn(this.getClass()).updateUser(id, user)).withRel("update"));

return resource;

}

@GetMapping

public CollectionModel<EntityModel<User>> getAllUsers() {

List<EntityModel<User>> users = userService.getAll().stream().map(user -> {

return EntityModel.of(user,

linkTo(methodOn(this.getClass()).getUserById(user.getId())).withSelfRel()

);

}).toList();

return CollectionModel.of(users,

linkTo(methodOn(this.getClass()).getAllUsers()).withSelfRel()

);

}

@PutMapping("/{id}")

public ResponseEntity<User> updateUser(@PathVariable int id, @RequestBody User user) {

return ResponseEntity.ok(userService.update(id, user));

}

@DeleteMapping("/{id}")

public ResponseEntity<Void> deleteUser(@PathVariable int id) {

userService.delete(id);

return ResponseEntity.noContent().build();

}

}

**🔎 Benefits of Level 3 / HATEOAS**

| **✅ Feature** | **Benefit** |
| --- | --- |
| Hypermedia links | Client knows "what to do next" |
| Discoverable API | No need to hardcode URLs |
| Decoupled client | Frontends can adapt without versioning |
| Extensible | Easy to add new actions/links |

**⚠️ Challenges**

* Requires more implementation effort
* Some clients may not support HAL/JSON Hypermedia easily
* Often combined with tools like **Spring HATEOAS**, **HAL Browser**, or **Siren**