REST API Design

The risks of returning an entity directly and the DTO pattern; pagination, sorting, and filtering with Pageable/Page/Sort and query parameters; URI versioning vs. header versioning; idempotency and making POST idempotent with the Idempotency-Key header; HATEOAS (a quick look).

Advanced 42 min

Last updated: August 18, 2026
Read online: https://www.learnforgex.com/en/topics/rest-api-design

---

REST API Design

Advanced Spring MVC covered how to add behavior **around** a request (interceptors, CORS, multipart) -- this lesson turns to the shape of the request/response **itself**. Request and Response Handling covered `@RequestBody`, `ResponseEntity`, and HTTP status codes; Validation & Exception Handling covered `ProblemDetail` for standard error bodies -- this lesson builds on those tools to address five concrete design problems real-world REST APIs run into often: moving data without leaking an entity's internals (DTOs), returning large collections a piece at a time (pagination/sorting/filtering), changing an API without breaking backward compatibility (versioning), preventing a request from being processed twice by accident (idempotency), and having a response carry its own navigation (HATEOAS).

What Is REST API Design?

REST (Representational State Transfer) is an architectural style built on principles we've already used in Request and Response Handling and Path Variables and Request Parameters -- resources are represented by URLs, HTTP methods carry a meaningful contract, responses speak through status codes. This lesson takes those principles beyond a **single** endpoint, to how an API is designed **as a whole**:

```
// A single "RESTful" endpoint isn't enough -- an entire API needs to be
// consistent: the same error shape, the same pagination pattern, the same
// versioning strategy.
@GetMapping("/api/v1/topics")
ResponseEntity<PagedResponse<TopicSummary>> listTopics(Pageable pageable) { ... }
```

Why Does It Exist?

If every endpoint invents its own convention (one names pagination `?page=`, another `?offset=`; one returns errors as plain text, another as JSON), a client consuming the API has to build a separate mental model for every endpoint. We saw `@RestControllerAdvice` gather error bodies into **one place** in Validation & Exception Handling -- every pattern in this lesson (DTOs, a pagination shape, a versioning strategy) shares the same motivation: making consistency central and predictable across endpoints.

History

The term REST was defined in Roy Fielding's 2000 doctoral dissertation -- not an architectural style as old as HTTP itself, but an observation about the **correct** use of HTTP. HATEOAS was part of Fielding's original dissertation, but in practice it became the least adopted of its principles -- most "REST APIs" are actually HATEOAS-free, plain JSON over HTTP. The Spring HATEOAS project started in 2012 to fill that gap (not used in this project, see "What Is HATEOAS? (A Quick Look)"). The Idempotency-Key header pattern is a convention Stripe's API popularized in 2017, later turning into an IETF draft. The debate between URI and header versioning strategies has been running since the 2010s, and still has no single settled answer.

The Risks of Returning an Entity Directly: Why a DTO?

Returning a JPA entity directly from a `@RestController` looks tempting -- Jackson can already turn it into JSON. But that carries two real risks:

```
// Returning a JPA entity directly from a @RestController -- letting Jackson serialize
// it as-is -- looks convenient, but couples your HTTP contract to your database
// schema and can leak things you never meant to expose.
class EntityLeakageRiskExample {

    // A typical entity: exactly what the database needs, nothing about what an API
    // consumer should see.
    static class UserEntity {
        Long id;
        String email;
        String passwordHash;       // never meant to leave the server
        String internalNotes;      // an admin-only field, added later by someone else
        java.util.List<String> roles; // in a real @Entity, this would be a LAZY collection

        UserEntity(Long id, String email, String passwordHash, String internalNotes, java.
util.List<String> roles) {
            this.id = id;
            this.email = email;
            this.passwordHash = passwordHash;
            this.internalNotes = internalNotes;
            this.roles = roles;
        }
    }

    // What Jackson would serialize if this entity were returned directly from a
    // @RestController method -- every field, by default, becomes a JSON property.
    static String naiveSerialize(UserEntity user) {
        return "{\"id\":" + user.id
                + ",\"email\":\"" + user.email + "\""
                + ",\"passwordHash\":\"" + user.passwordHash + "\""      // leaked
                + ",\"internalNotes\":\"" + user.internalNotes + "\""   // leaked
                + ",\"roles\":" + user.roles + "}";
    }

    public static void main(String[] args) {
        UserEntity user = new UserEntity(1L, "ada@example.com", "$2a$10$abcdef...",
                "flagged for review 2025-11-02", java.util.List.of("USER"));

        System.out.println(naiveSerialize(user));
        // {"id":1,"email":"ada@example.com","passwordHash":"$2a$10$abcdef...",
        //  "internalNotes":"flagged for review 2025-11-02","roles":[USER]}

        // Two separate problems bundled into one bad decision:
        // 1) passwordHash/internalNotes were never meant to be public API fields.
        // 2) In a REAL @Entity, "roles" would likely be a LAZY collection -- serializing
        //    it outside an open Hibernate session throws LazyInitializationException,
        //    which is exactly why this project's TopicController resolves associations
        //    with an explicit join fetch (see TopicRepository.
findBySlugWithCategoryAndCourse)
        //    instead of leaving them to be touched later, e.g. during serialization.
    }
}
```

First: an entity carries **every** field the database needs -- including things no client should ever see, like a password hash or internal notes. Second: a lazily-loaded collection on a real entity (like the `@ManyToOne(FetchType.LAZY)` fields this project's own `TopicRepository` deals

with) can throw `LazyInitializationException` if touched during serialization -- this project solves that risk upfront with a join fetch in `findBySlugWithCategoryAndCourse`, but the general principle is the same: don't let an entity's internal structure leak into the API contract.

The DTO Pattern: Separating Request/Response with Records

The fix is to define the shape the API actually needs **separately**:

```
// A DTO (Data Transfer Object) is a shape designed for the API contract, not the
// database. Records (see the Record lesson) are a natural fit -- immutable,
// concise, and each one describes exactly one direction of the conversation.
class DtoRecordExample {

    // Request DTO: only what a client is allowed to send. No id (the server assigns
    // it), no passwordHash (the client sends a plain password, the server hashes it).
    record CreateUserRequest(String email, String password) {
    }

    // Response DTO: only what a client is allowed to see. No passwordHash, no
    // internalNotes -- compare with EntityLeakageRiskExample.UserEntity.
    record UserResponse(Long id, String email, java.util.List<String> roles) {
    }

    public static void main(String[] args) {
        CreateUserRequest request = new CreateUserRequest("ada@example.com", "s3cret!");
        System.out.println(request);
        // DtoRecordExample$CreateUserRequest[email=ada@example.com, password=s3cret!]

        UserResponse response = new UserResponse(1L, request.email(), java.util.List.of
("USER"));
        System.out.println(response);
        // DtoRecordExample$UserResponse[id=1, email=ada@example.com, roles=[USER]]

        // Two different shapes for two different moments -- CreateUserRequest never
        // has an id (it doesn't exist yet), UserResponse never has a password (it
        // should never come back out). A single shared "User" shape used for both
        // directions can't express either constraint.
    }
}
```

As we saw in the Record lesson, a record is immutable and concise -- each one describes exactly one direction (request or response). `CreateUserRequest` has no `id` field (it doesn't exist yet); `UserResponse` has no `password` field (it should never leave the server). A single shared "User" shape couldn't express both constraints at once.

Entity ↔ DTO Mapping: By Hand

The DTO pattern only earns its keep once something actually **converts** between the two:

```
// The DTO pattern only pays off once something actually converts between entity and
// DTO. The simplest version is a plain static method -- no mapping library needed
// for a shape this small (see this lesson's "Example Writing Principles" -- don't add
// infrastructure a small example doesn't need).
class EntityToDtoMappingExample {

    record TopicSummary(String slug, String title, String difficulty) {
    }

    // Stands in for this project's real Topic/TopicTranslation entities -- a
    // simplified shape, just enough to show the mapping.
    record TopicEntityStub(String slug, String difficulty, String translatedTitle) {
    }

    static TopicSummary toDto(TopicEntityStub entity) {
        return new TopicSummary(entity.slug(), entity.translatedTitle(), entity.
difficulty());
    }

    public static void main(String[] args) {
        TopicEntityStub entity = new TopicEntityStub("advanced-spring-mvc", "ADVANCED",
"Advanced Spring MVC");

        TopicSummary dto = toDto(entity);
        System.out.println(dto);
        // TopicSummary[slug=advanced-spring-mvc, title=Advanced Spring MVC,
difficulty=ADVANCED]

        // At this project's actual scale, a hand-written toDto(...) per entity is
        // perfectly maintainable. Larger codebases often reach for a mapping library
        // (MapStruct is the common choice -- it generates this exact kind of method
        // at compile time instead of by hand) once there are dozens of DTOs and
        // fields change often enough that keeping mappings in sync by hand gets error-
prone.
    }
}
```

At this project's scale, a hand-written `toDto(...)` method is perfectly maintainable. In larger codebases, once there are dozens of DTOs and fields change often, keeping this mapping in sync by hand becomes error-prone -- a mapping library like MapStruct (which generates the same kind of method at compile time) usually steps in at that point.

Pagination: Pageable and Page

Instead of returning a large collection all at once, return it a piece at a time:

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageImpl;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;

import java.util.List;

// Pageable/Page are the same Spring Data types that back JpaRepository (see how
// TopicRepository extends it) -- when a @RestController method takes a Pageable
// parameter, Spring resolves it from ?page=/?size=/?sort= query parameters
// automatically, no manual parsing needed.
class PaginationExample {

    record Topic(String slug, String title) {
    }

    static Page<Topic> findTopics(List<Topic> allTopics, Pageable pageable) {
        int start = (int) pageable.getOffset();
        int end = Math.min(start + pageable.getPageSize(), allTopics.size());
        List<Topic> pageContent = start >= allTopics.size() ? List.of() : allTopics.subList
(start, end);
        // A real repository does this in the database (LIMIT/OFFSET); PageImpl here
        // just wraps an already-fetched in-memory list to demonstrate the shape.
        return new PageImpl<>(pageContent, pageable, allTopics.size());
    }

    public static void main(String[] args) {
        List<Topic> allTopics = List.of(
                new Topic("spring-mvc-fundamentals", "Spring MVC Fundamentals"),
                new Topic("mapping-annotations-http-methods", "Mapping Annotations and HTTP
Methods"),
                new Topic("path-variables-request-parameters", "Path Variables and Request
Parameters"),
                new Topic("request-response-handling", "Request and Response Handling"),
                new Topic("validation-exception-handling", "Validation and Exception
Handling"));

        // ?page=0&size=2 -- Spring resolves this into a Pageable automatically when
        // a controller method takes one as a parameter.
        Pageable firstPage = PageRequest.of(0, 2);
        Page<Topic> page1 = findTopics(allTopics, firstPage);

        System.out.println(page1.getContent());
        // [Topic[slug=spring-mvc-fundamentals, ...], Topic[slug=mapping-annotations-http-
methods, ...]]
        System.out.println("totalElements=" + page1.getTotalElements() + ", totalPages=" +
page1.getTotalPages());
        // totalElements=5, totalPages=3

        Pageable lastPage = PageRequest.of(2, 2);
        Page<Topic> page3 = findTopics(allTopics, lastPage);
        System.out.println(page3.getContent() + ", isLast=" + page3.isLast());
        // [Topic[slug=validation-exception-handling, ...]], isLast=true
    }
}
```

`Pageable` / `Page` come from the same `JpaRepository` family this project's own `TopicRepository` already extends -- when a `@RestController` method parameter is of type `Pageable`, Spring resolves it from `?page=` / `?size=` / `?sort=` query parameters automatically, no manual parsing needed. `page.getTotalElements()` / `getTotalPages()` let a client know how many more pages there are.

Sorting: Sort with Multiple Fields

`Sort` works together with `Pageable`, and can be chained across multiple fields:

```
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;

// Sort composes with Pageable -- a client can ask for ?sort=difficulty,asc&sort=title,desc
// and Spring resolves it into exactly the Sort object built here by hand.
class SortingExample {

    public static void main(String[] args) {
        Sort byDifficultyThenTitle = Sort.by(Sort.Direction.ASC, "difficulty")
                .and(Sort.by(Sort.Direction.DESC, "title"));

        System.out.println(byDifficultyThenTitle);
        // difficulty: ASC,title: DESC

        // Combined with paging into a single Pageable, exactly what a
        // @RestController parameter of type Pageable resolves to from query
        // parameters like ?page=0&size=10&sort=difficulty,asc&sort=title,desc:
        Pageable pageable = PageRequest.of(0, 10, byDifficultyThenTitle);
        System.out.println("page=" + pageable.getPageNumber() + ", sort=" + pageable.
getSort());
        // page=0, sort=difficulty: ASC,title: DESC

        // A shorthand for a single field:
        Pageable simpleSort = PageRequest.of(0, 10, Sort.by("title"));
        System.out.println(simpleSort.getSort());
        // title: ASC  -- Sort.by(String...) defaults to ascending
    }
}
```

`Sort.by(Sort.Direction.ASC, "difficulty").and(Sort.by(Sort.Direction.DESC, "title"))` is the server-side equivalent of what a client would request with `?sort=difficulty,asc&sort=title,desc` -- Spring resolves those query parameters into exactly this kind of `Sort` object automatically.

Filtering: Dynamic Queries from Query Parameters

Path Variables and Request Parameters showed that `@RequestParam` can be optional -- filtering is built on exactly that:

```
import java.util.List;
import java.util.Optional;
import java.util.function.Predicate;

// Query parameters like ?difficulty=ADVANCED&category=spring-mvc filter a collection --
// each present parameter narrows the result, each absent one is simply skipped.
// A real repository would push this down into a WHERE clause (or a JPA Specification
// for cases this dynamic); this example keeps the filtering logic itself visible by
// building it as a chain of optional Predicates over an in-memory list.
class DynamicFilterExample {

    record Topic(String slug, String category, String difficulty) {
    }

    static List<Topic> filter(List<Topic> topics, String category, String difficulty) {
        Predicate<Topic> byCategory = Optional.ofNullable(category)
                .<Predicate<Topic>>map(c -> t -> t.category().equals(c))
                .orElse(t -> true);
        Predicate<Topic> byDifficulty = Optional.ofNullable(difficulty)
                .<Predicate<Topic>>map(d -> t -> t.difficulty().equals(d))
                .orElse(t -> true);

        return topics.stream().filter(byCategory.and(byDifficulty)).toList();
    }

    public static void main(String[] args) {
        List<Topic> topics = List.of(
                new Topic("advanced-spring-mvc", "spring-mvc", "ADVANCED"),
                new Topic("spring-mvc-fundamentals", "spring-mvc", "INTERMEDIATE"),
                new Topic("threads", "concurrency", "ADVANCED"));

        System.out.println(filter(topics, "spring-mvc", null));
        // [Topic[advanced-spring-mvc,...], Topic[spring-mvc-fundamentals,...]] -- category
only

        System.out.println(filter(topics, "spring-mvc", "ADVANCED"));
        // [Topic[advanced-spring-mvc,...]] -- both filters applied

        System.out.println(filter(topics, null, null));
        // all three -- no filters means every predicate defaults to "true"
    }
}
```

Every filter parameter is optional: if present, it narrows the result; if absent, it has no effect at all (via a predicate that defaults to `true`). A real repository usually pushes this logic down into the database, into a `WHERE` clause or (for many optional fields) a JPA `Specification` -- but the core idea is the same: every filter falls back to "exclude nothing" when it isn't supplied.

The Shape of a Paginated Response: content, totalElements, totalPages

Returning a `Page<T>` directly from a controller works, but Spring Data itself advises against it -- `PageImpl`'s internal fields aren't a documented, stable contract, and its default JSON shape has changed across versions:

```
import org.springframework.data.domain.Page;

import java.util.List;

// Returning a Page<T> directly from a @RestController works, but Spring Data itself
// warns against it: PageImpl's internal fields aren't a stable, documented API
// contract, and its default JSON shape has changed across Spring Data versions.
// The recommended fix is the same idea as the DTO pattern -- wrap the page in a
// shape YOU control and document, not one an internal class happens to produce.
class PagedResponseShapeExample {

    record TopicSummary(String slug, String title) {
    }

    // A stable, project-owned response shape -- pulls only what a client actually
    // needs out of Page<T>, in field names this project's own API docs can commit to.
    record PagedResponse<T>(List<T> content, int page, int size, long totalElements, int
totalPages) {
        static <T> PagedResponse<T> from(Page<T> springDataPage) {
            return new PagedResponse<>(
                    springDataPage.getContent(),
                    springDataPage.getNumber(),
                    springDataPage.getSize(),
                    springDataPage.getTotalElements(),
                    springDataPage.getTotalPages());
        }
    }

    public static void main(String[] args) {
        Page<TopicSummary> springDataPage = new org.springframework.data.domain.PageImpl<>(
                List.of(new TopicSummary("advanced-spring-mvc", "Advanced Spring MVC")),
                org.springframework.data.domain.PageRequest.of(0, 2),
                5);

        PagedResponse<TopicSummary> response = PagedResponse.from(springDataPage);
        System.out.println(response);
        // PagedResponse[content=[TopicSummary[slug=advanced-spring-mvc, ...]], page=0,
        //   size=2, totalElements=5, totalPages=3]

        // Whatever Page<T>'s own serialization looks like in a given Spring Data
        // version, this record's shape doesn't change unless this project changes it.
    }
}
```

The fix is the same idea as the DTO pattern: wrap `Page<T>` in a `PagedResponse<T>` this project can document and control the field names of. Whatever `Page<T>`'s internal

serialization looks like in a given Spring Data version, this record's shape only changes if this project changes it.

API Versioning: URI Versioning vs. Header Versioning

An API changes over time -- two common ways to offer a new shape without breaking existing clients:

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RestController;

// Two common ways to version a REST API. Neither needs a new mechanism -- both
// reuse tools this project already knows: @GetMapping's path (Mapping Annotations
// and HTTP Methods) for URI versioning, @RequestHeader (Path Variables and Request
// Parameters) for header versioning.
@RestController
class ApiVersioningExample {

    // URI versioning: the version is part of the path itself -- impossible to miss,
    // easy to route differently, but "v1"/"v2" leak into every client's URLs forever.
    @GetMapping("/api/v1/topics/{slug}")
    public String getTopicV1(String slug) {
        return "{\"slug\":\"" + slug + "\"}"; // v1 shape: flat
    }

    @GetMapping("/api/v2/topics/{slug}")
    public String getTopicV2(String slug) {
        return "{\"slug\":\"" + slug + "\",\"links\":{}}"; // v2 shape: adds a field
    }

    // Header versioning: the URL never changes -- one @GetMapping, the version comes
    // from a request header instead.
    @GetMapping("/api/topics/{slug}")
    public String getTopic(String slug, @RequestHeader(name = "Api-Version", defaultValue =
"1") int apiVersion) {
        return apiVersion >= 2
                ? "{\"slug\":\"" + slug + "\",\"links\":{}}"
                : "{\"slug\":\"" + slug + "\"}";
    }

    public static void main(String[] args) {
        ApiVersioningExample controller = new ApiVersioningExample();

        System.out.println(controller.getTopicV1("advanced-spring-mvc"));
        // {"slug":"advanced-spring-mvc"}
        System.out.println(controller.getTopicV2("advanced-spring-mvc"));
        // {"slug":"advanced-spring-mvc","links":{}}

        System.out.println(controller.getTopic("advanced-spring-mvc", 1));
        // {"slug":"advanced-spring-mvc"}
        System.out.println(controller.getTopic("advanced-spring-mvc", 2));
        // {"slug":"advanced-spring-mvc","links":{}}
    }
}
```

URI versioning (`/api/v1/...` vs. `/api/v2/...`) is part of `@GetMapping`'s path, the same mechanism from Mapping Annotations and HTTP Methods -- impossible to miss, but "v1"/" v2" leaks into every one of a client's URLs forever. Header versioning (`Api-Version: 2`) uses `@RequestHeader` from Path Variables and Request Parameters -- the URL never changes, but the version is no longer visible just by looking at the URL, and becomes dependent on documentation.

What Is Idempotency? Naturally Idempotent Methods

An operation is idempotent when calling it once produces **the same result** as calling it N times:

```
import java.util.HashMap;
import java.util.Map;

// An operation is idempotent when calling it once has the same effect as calling it
// N times. Mapping Annotations and HTTP Methods already introduced idempotent
// as a property of GET/PUT/DELETE -- this example proves it by actually calling each
// operation twice and checking the store ends up in the same state either way.
class IdempotentMethodsExample {

    static final Map<String, String> store = new HashMap<>();

    static void put(String key, String value) {
        store.put(key, value); // PUT: replaces whatever was there -- same result every time
    }

    static void delete(String key) {
        store.remove(key); // DELETE: removing something already gone is still "gone" --
same result
    }

    static String post(String value) {
        // POST: creates a NEW resource every time -- calling it twice is NOT the same
        // as calling it once.
        String id = "id-" + (store.size() + 1);
        store.put(id, value);
        return id;
    }

    public static void main(String[] args) {
        store.clear();

        put("topic-1", "Advanced Spring MVC");
        put("topic-1", "Advanced Spring MVC"); // calling PUT again
        System.out.println(store);
        // {topic-1=Advanced Spring MVC} -- calling it twice left the exact same state

        delete("topic-1");
        delete("topic-1"); // calling DELETE on something already gone
        System.out.println(store.containsKey("topic-1"));
        // false either way -- both calls end in the same state

        store.clear();
        String firstId = post("REST API Design");
        String secondId = post("REST API Design"); // calling POST again
        System.out.println(firstId + " != " + secondId + " -> " + store);
        // id-1 != id-2 -> {id-1=REST API Design, id-2=REST API Design}
        // two calls created two resources -- POST is NOT idempotent by default
    }
}
```

`PUT` and `DELETE` are idempotent by nature -- sending the same `PUT` twice leaves the resource in the same final state; sending the same `DELETE` twice leaves the resource "gone" either way (the second call changes nothing). `POST` is **not** -- by definition, every call creates

a new resource. This distinction was introduced in Mapping Annotations and HTTP Methods' "HTTP Methods: The Safe and Idempotent Concepts" section; here we actually run it and confirm it.

Making POST Idempotent with the Idempotency-Key Header

`POST` not being idempotent creates a real problem: when a client retries a request after a timeout, it's unclear whether the server already processed the first attempt. The fix is a key the client generates, so the server can say "I've already seen this one":

```
import java.util.HashMap;
import java.util.Map;

// IdempotentMethodsExample showed POST creating a new resource on every call -- a
// real problem when a client retries a request after a timeout, unsure whether the
// first attempt actually succeeded. The fix: the client generates a unique
// Idempotency-Key per logical operation and sends it with every retry; the server
// remembers which keys it has already processed and returns the SAME result instead
// of creating a duplicate.
class IdempotencyKeyExample {

    record OrderResult(String orderId, String status) {
    }

    static final Map<String, OrderResult> processedKeys = new HashMap<>();
    static int nextOrderNumber = 1;

    static OrderResult createOrder(String idempotencyKey, String item) {
        OrderResult existing = processedKeys.get(idempotencyKey);
        if (existing != null) {
            return existing; // same key seen before -- return the original result, create
nothing
        }

        OrderResult result = new OrderResult("order-" + nextOrderNumber++, "CREATED: " +
item);
        processedKeys.put(idempotencyKey, result);
        return result;
    }

    public static void main(String[] args) {
        String key = "a1b2c3-client-generated-uuid";

        OrderResult first = createOrder(key, "Java Mug");
        System.out.println(first);
        // OrderResult[orderId=order-1, status=CREATED: Java Mug]

        // The client didn't get a response in time (network blip) and retries with
        // the SAME key:
        OrderResult retry = createOrder(key, "Java Mug");
        System.out.println(retry);
        // OrderResult[orderId=order-1, status=CREATED: Java Mug] -- identical, no
duplicate order

        // A genuinely new order uses a fresh key, and does create a new resource:
        OrderResult secondOrder = createOrder("d4e5f6-different-uuid", "Mechanical
Keyboard");
        System.out.println(secondOrder);
        // OrderResult[orderId=order-2, status=CREATED: Mechanical Keyboard]
    }
}
```

The client generates a single `Idempotency-Key` (usually a UUID) for one logical operation and sends the same key on every retry. If the server has already processed that key, it returns the original result **without creating a new resource** -- the second call has exactly the same effect as the first, so `POST` becomes effectively idempotent.

What Is HATEOAS? (A Quick Look)

HATEOAS means a response carries not just data, but the client's **next steps**:

```
import java.util.LinkedHashMap;
import java.util.Map;

// HATEOAS (Hypermedia as the Engine of Application State): a response includes not
// just data, but the LINKS a client can follow next -- the API guides the client,
// instead of the client having to hard-code every URL it might ever need. This
// project doesn't use the real `spring-hateoas` library (it isn't a dependency
// here), so this example hand-builds the same shape a real HATEOAS response has,
// to show the idea without adding a library this project doesn't otherwise need.
class HateoasConceptExample {

    record TopicResponse(String slug, String title, Map<String, String> links) {
    }

    static TopicResponse toResponseWithLinks(String slug, String title, String
previousSlug, String nextSlug) {
        Map<String, String> links = new LinkedHashMap<>();
        links.put("self", "/api/topics/" + slug);
        if (previousSlug != null) {
            links.put("previous", "/api/topics/" + previousSlug);
        }
        if (nextSlug != null) {
            links.put("next", "/api/topics/" + nextSlug);
        }
        return new TopicResponse(slug, title, links);
    }

    public static void main(String[] args) {
        TopicResponse response = toResponseWithLinks(
                "advanced-spring-mvc", "Advanced Spring MVC",
                "spring-mvc-views-thymeleaf", "rest-api-design");

        System.out.println(response);
        // TopicResponse[slug=advanced-spring-mvc, title=Advanced Spring MVC,
        //   links={self=/api/topics/advanced-spring-mvc,
        //          previous=/api/topics/spring-mvc-views-thymeleaf,
        //          next=/api/topics/rest-api-design}]

        // A client following "next" never needs to know this project's URL scheme
        // (/api/topics/{slug}) -- it just follows the link the server gave it. This
        // project's own topic.html does the conceptual equivalent server-side
        // (previousTopic/nextTopic in TopicController), just rendered as HTML
        // <a> tags instead of a JSON "links" map.
    }
}
```

A client following the `next` link never needs to know this project's URL scheme (`/api/topics/{slug}`) at all -- it just follows the link the server gave it. This project doesn't use the real `spring-hateoas` library (it isn't a project dependency), so the example above hand-builds a `links` map -- but the idea is the JSON equivalent of what this project's own `topic.html` already does with `previousTopic` / `nextTopic` (see Spring MVC Views and Thymeleaf): the server knows where "previous"/"next" are, the client doesn't need to.

Best Practices

- **Explicitly filter out fields a consuming client should never see (password hashes, internal notes, internal IDs) with a DTO** -- returning an entity directly makes it easy to forget one (see "The Risks of Returning an Entity Directly: Why a DTO?").
- **Wrap paginated responses in a DTO you control, don't return `Page<T>` directly** -- Spring Data itself recommends this (see "The Shape of a Paginated Response: content, totalElements, totalPages").
- **Pick a versioning strategy from the start (or at least before the first breaking change) and apply it consistently** -- switching from URI to header versioning (or back) partway through breaks every existing client (see "API Versioning: URI Versioning vs. Header Versioning").
- **Take `Idempotency-Key` seriously on `POST` endpoints with side effects (payments, order creation)** -- network timeouts are real and common; without this pattern, a retry can cause concrete user-facing harm like double charges (see "Making POST Idempotent with the Idempotency-Key Header").

Common Mistakes

1. Returning an entity directly "for now" and pushing the DTO to later. Once clients depend on the entity's shape, inserting a DTO later becomes a backward-incompatible change -- setting up the DTO from the start is much cheaper than adding one afterward (see "The Risks of Returning an Entity Directly: Why a DTO?").

2. Reading pagination parameters (`page` , `size` , `sort`) by hand with `@RequestParam` instead of using `Pageable` . This re-solves a problem Spring already solves -- a `Pageable` parameter does the same job, with validation and defaults, in a single line (see "Pagination: Pageable and Page").

3. Forgetting a filter parameter can be `null` and calling `.equals(...)` on it directly. Code like `category.equals(t.category())` throws a `NullPointerException` when `category` isn't supplied -- every optional filter needs to explicitly express "no effect when not supplied" (see "Filtering: Dynamic Queries from Query Parameters").

4. Mixing URI versioning and header versioning within the same API. If some endpoints use `/api/v1/...` and others use an `Api-Version` header, it becomes hard for a client to guess which strategy applies where -- an API should stay consistent with one strategy (see "API Versioning: URI Versioning vs. Header Versioning").

5. Storing `Idempotency-Key`s on the server forever. In a real application, keys should expire after some window (24 hours, say) -- otherwise memory/storage grows without bound; the `Map` in this example only shows the idea, with no expiry logic (see "Making POST Idempotent with the Idempotency-Key Header").

6. Documenting HATEOAS links but never actually putting them in the response. The whole point of HATEOAS is that a client can find its next step by looking at the **response itself**, not the documentation -- a link that's only documented but never appears in the response isn't HATEOAS, it's just an ordinary API contract (see "What Is HATEOAS? (A Quick Look)").

Summary, Cheat Sheet, and Glossary

REST API design goes beyond a single endpoint working correctly -- it's about an entire API staying consistent, predictable, and backward compatible. Key points:

- DTO: a pattern that separates the API contract from an entity's internal structure, with distinct shapes for request and response
- `Pageable` / `Page<T>` / `Sort`: Spring Data's counterpart for pagination and sorting, resolved automatically from query parameters
- Filtering: every query parameter is optional, a predicate that has no effect when not supplied
- `PagedResponse<T>`: a project-controlled pagination shape, instead of `Page<T>`'s unstable internals
- URI versioning: the version lives in the path (`/api/v1/...`) -- visible but permanent
- Header versioning: the version lives in a header (`Api-Version`) -- the URL stays fixed, but the version becomes invisible
- Idempotent: an operation that produces the same result whether called once or N times (`GET` / `PUT` / `DELETE` naturally, `POST` not)
- `Idempotency-Key`: a client-generated header that lets the server say "I've already processed this request"
- HATEOAS: a response carrying the client's next steps (links) alongside its data

Quick reference:

```
@RestController
class TopicApiController {

    @GetMapping("/api/v1/topics")
    PagedResponse<TopicSummary> list(
            @RequestParam(required = false) String category,
            Pageable pageable) {
        // filter -> paginate -> wrap in a DTO
        return PagedResponse.from(repository.findAll(pageable));
    }

    @PostMapping("/api/v1/orders")
    ResponseEntity<OrderResponse> createOrder(
            @RequestHeader("Idempotency-Key") String key,
            @RequestBody CreateOrderRequest request) {
        OrderResponse existing = seenKeys.get(key);
        if (existing != null) return ResponseEntity.ok(existing);
        // ... create the new resource, add it to seenKeys ...
        return ResponseEntity.status(HttpStatus.CREATED).body(created);
    }
}
```

Glossary

DTO (Data Transfer Object) — A data shape designed for the API contract, independent of the database entity.

`Pageable` — A Spring Data interface carrying page number, size, and sorting, resolved automatically from query parameters.

`Page<T>` — A Spring Data interface carrying a page's content along with total element/page counts.

`Sort` — A Spring Data type defining sorting by one or more fields with a direction (`ASC` / `DESC`).

URI versioning — A versioning strategy where the API version is part of the URL path.

Header versioning — A versioning strategy where the API version is specified through an HTTP header, keeping the URL fixed.

Idempotent — An operation that produces the same result whether called once or N times.

`Idempotency-Key` — A client-generated HTTP header that lets the server determine whether it has already processed a given request.

HATEOAS (Hypermedia as the Engine of Application State) — The REST principle that a response should carry links a client can follow, alongside its data.

Appendix: Mini Project — A Paginated and Filtered Topic Catalog API

Bringing this lesson's three data-shaping mechanics (filtering, pagination/sorting, and a stable response shape) together in a single catalog endpoint:

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageImpl;
import org.springframework.data.domain.Pageable;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;
import java.util.Optional;
import java.util.function.Predicate;

// Mini project, part 1/2: combines this lesson's three data-shaping mechanics --
// filtering (?category=), pagination and sorting (Pageable), and a stable response
// shape (PagedResponseShapeExample's pattern) -- into a single catalog endpoint.
@RestController
class PaginatedCatalogController {

    record TopicSummary(String slug, String title, String category, String difficulty) {
    }

    record PagedResponse<T>(List<T> content, int page, int size, long totalElements, int
totalPages) {
        static <T> PagedResponse<T> from(Page<T> springDataPage) {
            return new PagedResponse<>(springDataPage.getContent(), springDataPage.
getNumber(),
                    springDataPage.getSize(), springDataPage.getTotalElements(),
springDataPage.getTotalPages());
        }
    }

    private final List<TopicSummary> allTopics;

    PaginatedCatalogController(List<TopicSummary> allTopics) {
        this.allTopics = allTopics;
    }

    @GetMapping("/api/topics")
    public PagedResponse<TopicSummary> listTopics(
            @RequestParam(required = false) String category,
            Pageable pageable) {

        Predicate<TopicSummary> matchesCategory = Optional.ofNullable(category)
                .<Predicate<TopicSummary>>map(c -> t -> t.category().equals(c))
                .orElse(t -> true);

        List<TopicSummary> filtered = allTopics.stream().filter(matchesCategory).toList();

        int start = Math.min((int) pageable.getOffset(), filtered.size());
        int end = Math.min(start + pageable.getPageSize(), filtered.size());

        Page<TopicSummary> page = new PageImpl<>(filtered.subList(start, end), pageable,
filtered.size());
        return PagedResponse.from(page);
    }
}
```

```
import org.springframework.data.domain.PageRequest;

import java.util.List;

// Mini project, part 2/2: drives PaginatedCatalogController with a small in-memory
// catalog -- one call with just paging, one adding a category filter, showing the
// filter narrows the total BEFORE paging is applied (totalElements reflects the
// filtered count, not the full catalog).
class PaginatedCatalogDemo {

    public static void main(String[] args) {
        List<PaginatedCatalogController.TopicSummary> catalog = List.of(
                new PaginatedCatalogController.TopicSummary(
                        "spring-mvc-fundamentals", "Spring MVC Fundamentals", "spring-mvc",
"INTERMEDIATE"),
                new PaginatedCatalogController.TopicSummary(
                        "advanced-spring-mvc", "Advanced Spring MVC", "spring-mvc",
"ADVANCED"),
                new PaginatedCatalogController.TopicSummary(
                        "threads", "Threads", "concurrency", "ADVANCED"));

        PaginatedCatalogController controller = new PaginatedCatalogController(catalog);

        System.out.println(controller.listTopics(null, PageRequest.of(0, 2)));
        // PagedResponse[content=[...2 topics...], page=0, size=2, totalElements=3,
totalPages=2]

        System.out.println(controller.listTopics("spring-mvc", PageRequest.of(0, 2)));
        // PagedResponse[content=[...2 spring-mvc topics...], page=0, size=2,
totalElements=2, totalPages=1]
        // -- totalElements is 2, not 3: it reflects the filtered set, not the whole catalog
    }
}
```

`listTopics` takes an optional filter with `@RequestParam(required = false) String category`, pagination/sorting with `Pageable`, and wraps the result in a stable `PagedResponse<T>`, the same pattern as `PagedResponseShapeExample`. `PaginatedCatalogDemo` makes two calls -- one unfiltered, one with a `category` filter -- to show that `totalElements` reflects the filtered set, not the whole catalog.

Appendix: Mini Project — Idempotency-Key-Backed Order Creation

The last mini project brings the DTO pattern together with the `Idempotency-Key` mechanism on a real `@PostMapping` / `ResponseEntity`:

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RestController;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

// Mini project, part 1/2: combines the DTO pattern (a request/response shape
// separate from any entity) with IdempotencyKeyExample's mechanism, wired through a
// real @PostMapping/@RequestHeader/ResponseEntity -- the same building blocks from
// Request ve Response Handling, applied to this lesson's idempotency problem.
@RestController
class IdempotentOrderController {

    record CreateOrderRequest(String item) {
    }

    record OrderResponse(String orderId, String item) {
    }

    private final Map<String, OrderResponse> processedKeys = new ConcurrentHashMap<>();
    private int nextOrderNumber = 1;

    @PostMapping("/api/orders")
    public ResponseEntity<OrderResponse> createOrder(
            @RequestHeader("Idempotency-Key") String idempotencyKey,
            @RequestBody CreateOrderRequest request) {

        OrderResponse existing = processedKeys.get(idempotencyKey);
        if (existing != null) {
            return ResponseEntity.ok(existing); // already processed -- 200, not a new 201
        }

        OrderResponse created = new OrderResponse("order-" + nextOrderNumber++, request.
item());
        processedKeys.put(idempotencyKey, created);
        return ResponseEntity.status(HttpStatus.CREATED).body(created);
    }
}
```

```
// Mini project, part 2/2: calls IdempotentOrderController.createOrder directly --
// once, then a "retry" with the same Idempotency-Key -- and shows the status code
// difference (201 vs. 200) alongside the identical order id.
class IdempotentOrderDemo {

    public static void main(String[] args) {
        IdempotentOrderController controller = new IdempotentOrderController();
        var request = new IdempotentOrderController.CreateOrderRequest("Java Mug");

        var first = controller.createOrder("a1b2c3-client-generated-uuid", request);
        System.out.println(first.getStatusCode() + " " + first.getBody());
        // 201 CREATED OrderResponse[orderId=order-1, item=Java Mug]

        var retry = controller.createOrder("a1b2c3-client-generated-uuid", request);
        System.out.println(retry.getStatusCode() + " " + retry.getBody());
        // 200 OK OrderResponse[orderId=order-1, item=Java Mug]  -- same order, not a
duplicate

        var secondOrder = controller.createOrder("d4e5f6-different-uuid",
                new IdempotentOrderController.CreateOrderRequest("Mechanical Keyboard"));
        System.out.println(secondOrder.getStatusCode() + " " + secondOrder.getBody());
        // 201 CREATED OrderResponse[orderId=order-2, item=Mechanical Keyboard]
    }
}
```

`createOrder` uses the `CreateOrderRequest` / `OrderResponse` DTO pair, reads the client's key with `@RequestHeader("Idempotency-Key")`, and returns `201 Created` for a key it hasn't seen before, `200 OK` (with the same body) for one it has. `IdempotentOrderDemo` shows a "retry" with the same key returning the same order id, while a different key genuinely creates a new order.