



# Implementing HATEOAS in Jakarta REST: A Comprehensive Developer's Guide

# Contents

Guide Updated: **October 2024**

<b>Introduction</b>	<b>1</b>
<b>Key HATEOSAS Concepts</b>	<b>1</b>
<b>The Link Class and LinkBuilder</b>	<b>2</b>
Link Class	2
LinkBuilder	2
<b>Implementing HATEOAS in Jakarta REST</b>	<b>2</b>
Basic JAX-RS Resource	3
Adding HATEOAS Support	4
Key Changes for HATEOAS Support	6
Example JSON Response	7
<b>Benefits of HATEOAS</b>	<b>8</b>
<b>Implementation Considerations</b>	<b>8</b>
<b>Conclusions</b>	<b>9</b>

## Introduction

HATEOAS, short for Hypermedia as the Engine of Application State, is a key concept in the design of RESTful APIs. It enables API clients to dynamically interact with a service by discovering available actions and resources through hypermedia links embedded in the API's responses. This makes the API more adaptable to changes and easier to evolve over time. This approach enhances the self-descriptiveness of the API, making it more adaptable to changes and easier to evolve over time.

This guide is designed for Java and Jakarta EE developers who are familiar with RESTful API development and Jakarta REST (formerly JAX-RS), and are looking to enhance their APIs by implementing HATEOAS principles. By the end of this guide, you will be able to:

- Understand the core concepts of HATEOAS and its benefits in API design
- Refactor a basic JAX-RS resource into a HATEOAS-compliant endpoint
- Create and include hypermedia links in API responses
- Address common implementation considerations

Whether you're building a new API or improving an existing one, this guide will help you to create more flexible, discoverable, futureproof and easy to evolve RESTful services.

## Key HATEOAS Concepts

Before diving into the implementation of HATEOAS in Jakarta REST, it's essential to understand the foundational principles that drive this approach

1. **Hypermedia Links:** These are URLs or URI templates embedded within the API's responses (typically in JSON or XML format). They guide the client on how to interact with the API, indicating available actions (e.g., 'create', 'update', 'delete') and the resources they operate on.
2. **Self-Descriptiveness:** A HATEOAS-compliant API provides enough information within its responses to guide the client on possible next steps. The client doesn't need out-of-band knowledge about the API's structure or available actions.
3. **Dynamic Client Interaction:** The client interacts with the API based on the hypermedia links it receives, making it adaptable to changes in the API's structure or available actions.

## The Link Class and LinkBuilder

Before we look at implementing HATEOAS, it's important to understand two key classes that are core to a HATEOAS implementation in Jakarta REST: the `Link` class and its associated `LinkBuilder`.

### Link Class

The `Link` class represents a hypermedia link that can be included in API responses. It comprises the following properties:

1. **URI:** The target URI of the link.
2. **Relation Type (rel):** Describes the relationship between the current resource and the linked resource (e.g., "self", "next", "previous").
3. **Media Type:** Indicates the expected media type of the linked resource.
4. **Title:** An optional title for the link.

### LinkBuilder

`LinkBuilder` is a builder class for the creation of `Link` objects. It provides a fluent API for constructing links with various properties. Some of its methods include:

1. `Link.Builder fromUri(URI uri)`: Creates a builder for a link with the given URI.
2. `Link.Builder rel(String rel)`: Sets the relation type of the link.
3. `Link.Builder type(String type)`: Sets the media type of the linked resource.
4. `Link.Builder title(String title)`: Sets the title of the link.
5. `Link build()`: Constructs the final `Link` object.

## Implementing HATEOAS in Jakarta REST

While Jakarta REST (formerly JAX-RS) does not natively support HATEOAS, it provides all the necessary constructs needed to implement HATEOAS in your APIs. Let's explore how to use these constructs to create hypermedia services.

## Basic JAX-RS Resource

We'll start with a basic JAX-RS resource for managing orders in an e-commerce system:

```
@Path("/orders")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class OrderResource {

    @Inject
    private OrderService orderService;

    @GET
    @Path("/{id}")
    public Response getOrder(@PathParam("id") Long id) {
        Order order = orderService.findOrder(id);
        if (order == null) {
            return Response.status(Response.Status.NOT_FOUND).build();
        }
        return Response.ok(order).build();
    }

    @POST
    public Response createOrder(Order order) {
        Order createdOrder = orderService.createOrder(order);
        return Response.status(Response.Status.CREATED)
            .entity(createdOrder)
            .build();
    }

    @PUT
    @Path("/{id}")
    public Response updateOrder(@PathParam("id") Long id, Order order) {
        order.setId(id);
        Order updatedOrder = orderService.updateOrder(order);
        return Response.ok(updatedOrder).build();
    }
}
```

```
@DELETE
@Path("/{id}")
public Response deleteOrder(@PathParam("id") Long id) {
    orderService.deleteOrder(id);
    return Response.noContent().build();
}
```

This `OrderResource` class provides basic CRUD (Create, Read, Update, Delete) operations for orders. However, it does not implement HATEOAS principles yet.

## Adding HATEOAS Support

To add HATEOAS support, let's create a wrapper record called `OrderRepresentation` for our `Order` entity that includes hypermedia links:

```
public record OrderRepresentation(Order order, List<Link> links) {
    public OrderRepresentation(Order order) {
        this(order, new ArrayList<>());
    }

    public void addLink(Link link) {
        this.links.add(link);
    }
}
```

Now, let's update our `OrderResource` to use this new `OrderRepresentation` class and include hypermedia links:

```
@Path("/orders")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class OrderResource {

    @Inject
    private OrderService orderService;

    @Inject
```

```
private UriInfo uriInfo;

@GET
@Path("/{id}")
public Response getOrder(@PathParam("id") Long id) {
    Order order = orderService.findOrder(id);
    if (order == null) {
        return Response.status(Response.Status.NOT_FOUND).build();
    }

    OrderRepresentation orderRep = new OrderRepresentation(order);

    // Add self link
    Link selfLink = Link.fromUriBuilder(uriInfo.getAbsolutePathBuilder())
        .rel("self")
        .type(MediaType.APPLICATION_JSON)
        .build();
    orderRep.addLink(selfLink);

    // Add update link
    Link updateLink = Link.fromUriBuilder(uriInfo.getAbsolutePathBuilder().
path("update"))
        .rel("update")
        .type(MediaType.APPLICATION_JSON)
        .build();
    orderRep.addLink(updateLink);

    // Add cancel link (assuming there's a cancel endpoint)
    Link cancelLink = Link.fromUriBuilder(uriInfo.getAbsolutePathBuilder().
path("cancel"))
        .rel("cancel")
        .type(MediaType.APPLICATION_JSON)
        .build();
    orderRep.addLink(cancelLink);

    return Response.ok(orderRep).build();
}

@POST
public Response createOrder(Order order) {
    Order createdOrder = orderService.createOrder(order);
    OrderRepresentation orderRep = new OrderRepresentation(createdOrder);
```

```
// Add self link for the newly created order
URI location = uriInfo.getAbsolutePathBuilder().path(createdOrder.
getId()).toString().build();
Link selfLink = Link.fromUri(location)
    .rel("self")
    .type(MediaType.APPLICATION_JSON)
    .build();
orderRep.addLink(selfLink);

return Response.status(Response.Status.CREATED)
    .entity(orderRep)
    .location(location)
    .build();
}

// Rest of CRUD methods would be similar to the create method above...
}
```

## Key Changes for HATEOAS Support

1. **OrderRepresentation Record:** We use this wrapper Java record to include hypermedia links along with the Order object.
2. **UriInfo Usage:** We inject the UriInfo interface to build URIs based on the current request context. This ensures our links are always correct, even if the base URL of our API changes.
3. **Adding Links:** In the getOrder method, we add several links:
  - A "self" link, pointing to the current resource
  - An "update" link, also pointing to the current resource
  - A "cancel" link, pointing to a hypothetical cancel endpoint
4. **Creating Resources:** In the createOrder method, we include a "self" link to the newly created order in the response.
5. **Link Class:** We use the Link class from Jakarta REST to create our hypermedia links. Each link includes a relation type ("rel") which describes its purpose, and a media type.



## Example JSON Response

Here's an example of what the JSON response for a GET request to `/orders/123` might look like:

```
{
  "order": {
    "id": 123,
    "customerName": "John Doe",
    "totalAmount": 99.99,
    "status": "PROCESSING"
  },
  "links": [
    {
      "rel": "self",
      "href": "<http://api.example.com/orders/123>",
      "type": "application/json"
    },
    {
      "rel": "update",
      "href": "<http://api.example.com/orders/123>",
      "type": "application/json"
    },
    {
      "rel": "cancel",
      "href": "<http://api.example.com/orders/123/cancel>",
      "type": "application/json"
    }
  ]
}
```

This response provides the order details along with links to related actions. A client can use these links to navigate the API and perform actions without needing to know the exact URL structure in advance.

## Benefits of HATEOAS

1. **Improved Evolvability:** You can modify your API's structure or add new features without breaking existing clients. The clients discover new capabilities through the hypermedia links.
2. **Reduced Client-Side Logic:** The client doesn't need to hardcode URIs or understand the API's internal structure. It navigates the API based on the provided links.
3. **Enhanced Discoverability:** HATEOAS makes your API easy to explore, allowing clients (and developers) to understand its capabilities by simply following the links.
4. **Flexibility:** The set of available actions can change based on the state of the resource or the permissions of the user, and this is communicated dynamically through the provided links.

## Implementation Considerations

1. **Performance:** Adding hypermedia links to your responses increases the payload size. Consider the trade-off between the benefits of HATEOAS and the increased data transfer.
2. **Versioning:** HATEOAS can reduce the need for API versioning, as clients are guided by the links provided rather than hard coded expectations about the API structure.
3. **Documentation:** While HATEOAS improves API discoverability, it doesn't eliminate the need for good documentation. Ensure you document the meaning of different link relations and any expected workflows.
4. **Client Implementation:** Clients need to be built to understand and use the hypermedia controls. This may require more sophisticated client-side logic.


## Conclusions

Implementing HATEOAS in your Jakarta REST APIs leads to more flexible, adaptable and user-friendly APIs. Using hypermedia links and self-descriptiveness, you allow your clients to interact with your API dynamically, resulting in a more resilient and future-proof integration.

While implementing HATEOAS requires more upfront work in designing and implementing the API, it results in a more reliable and evolvable system in the long run. It's particularly useful for complex systems where the set of possible actions might vary based on the state of the resource or the permissions of the user.

Through the brief principles and examples outlined in this guide, you can create RESTful APIs that are not just functional, but also self-describing and discoverable, providing a superior experience for API consumers and setting a strong foundation for the long-term evolution of your services. Download Payara Enterprise trial today and start building your next API on our production supported Jakarta EE runtime today! Happy Coding!

### Interested in Payara? Try Before You Buy

A promotional banner for Payara trials. It features two laptops. The left laptop displays the Payara Enterprise logo and a code editor interface. The right laptop displays the Payara Cloud logo and a dashboard interface. Between the laptops is a large orange button with the text 'FREE TRIAL'. Below each laptop is an orange button with the text 'PAYARA SERVER FREE TRIAL' and 'PAYARA CLOUD FREE TRIAL' respectively. The background is a dark blue gradient with orange fish icons and arrows pointing towards the laptops.

**FREE TRIAL**

**PAYARA SERVER  
FREE TRIAL**

**PAYARA CLOUD  
FREE TRIAL**



[sales@payara.fish](mailto:sales@payara.fish)



UK: +44 800 538 5490  
Intl: +1 888 239 8941



[www.payara.fish](http://www.payara.fish)

Payara Services Ltd 2024 All Rights Reserved. Registered in England and Wales; Registration Number 09998946  
Registered Office: Malvern Hills Science Park, Geraldine Road, Malvern, United Kingdom, WR14 3SZ