<https://javabrains.io/courses/javaee_jaxrs/lessons/Rest-Response>

Notes

We've looked at requests so far, and understood resource URIs and HTTP methods. Let's switch to responses now.

From requests to responses

We've learned where to make requests (resource URIs) and how to make requests (HTTP methods). Time to look at responses now. When a request comes in, what should the REST web service respond with? Knowing what the client will get back from the server is an important part of the API, because the client needs to write code to handle the response.

If it were a web application, we know the response is usually an HTML page. With styling, formatting and also, of course, the actual data in a presentable format. But when it comes to RESTful web services, you don't need to do all the styling and formatting anymore. You just need to send the actual data. How do you send it?

We discussed about various standard formats that responses can be sent in, like XML and JSON. JSON has been growing in popularity, because it is much more compact and less verbose when compared to XML, especially when large data is involved. Also, more often than not, a client to a RESTful API is client side Javascript code, and sending back data in JSON means it can easily convert it to a Javascript object. Considering these advantages, we'll choose JSON as response for our social media application in this course. However, note that you do not typically need to settle for just one format. You can write APIs to support multiple response formats, and we will implement one such API endpoint later in this course to illustrate that.

Formats

Let's say our Message entity class has these four member variables: the id, the text of the message, when it was created and who created it.

public class MessageEntity {

private long id;

private String message;

private Date created;

private String author;

...

}

When a GET request is made for a specific message, say message ID 10, the JSON that you would return would look something like this:

{

"id":"10",

"message":"Hello world",

"created":"2014-06-01T18:06:36.902",

"author":"koushik"

}

But the response doesn't *have* to be JSON. You could return XML as well, if the client asks for it in XML format. We haven't yet covered how a client can *ask* for a specific format. We'll be looking at that later. But yes, a client can say "I need a JSON response" or "Give me an XML response". Here is a possible XML response for the same message ID 10.

<messageEntity>

<id>10</id>

<message>Hello world</message>

<created>2014-06-01T18:06:36.902</created>

<author>koushik</author>

</messageEntity>

Clearly the JSON response and the XML response are different. But they represent the same resource: message ID 10. So, in other words both these responses are different representations of the same resource. This is a very important thing to remember. When you make REST API calls you are sending or receiving *representations* of the resource. Different representations could have different formats, even though the underlying resource is the same. This is actually how REST gets its name. Representational State Transfer. You are transferring the representational state.

When you make REST API calls you are sending or receiving *representations* of the resource.

Message Headers

Ok, so it's great that a REST web service can return data in XML or JSON. But that brings up a problem. How does the client know what format the response is in? The client can of course request data in a particular format, but there's no guarantee that the service responds in that format. Say, a client request asks for XML. But if the REST service knows only JSON, it does return JSON ignoring the client's preference for XML. How does the client know the format then?

The answer is using HTTP headers. The HTTP protocol has a concept of request and response headers. Every HTTP request or response has a body, which is the message itself, and certain header values that contain metadata about the message. The header data could be stuff like the content length and date. One such possible header is Content-Type. The response could contain the Content-Type header with the value for JSON or XML. There are special values for JSON and XML, and we'll learn more about that when we implement this, but for now, know that the type of content is being sent back as a response header. The client can then examine this header value and then parse the response body content accordingly.

Status codes

Think about error messages in a web application. Whens something goes wrong, the application typically returns a page with an error message, maybe in bold red text. Even if it isn't in red, the message itself would give the user an idea that it's an error. But in the case of REST APIs, since the consumer is not a human, we need to provide some set of codes to the consumer to help them identify error scenarios.

HTTP specification requires the very first line of any response to be a *status line*. This line will have a numerical code and a short phrase explaining what the code means. This is not just for errors. Every HTTP response needs to have this line. If the response is successful, the very first line of the response will be: 200 OK

Let's take the familiar 404 error code. If a request is made on a URI, for example /messages/101 and there is no message available with ID 101, the first line of the response should be: 404 Not Found

Again, the code 404 is for the client code to read and act. The phrase Not Found is an aid to the programmer, in case they forget what the code means. Not that any programmer would ever forget what 404 means. I mean, come on!

There are a bunch of codes that are important for us to remember and use when developing a REST API. The error codes start from 100 and go up to 599. Not all of them are valid error codes though, so you don't have 500 different possible error codes. There are 5 classes of status codes and the first digit indicates what class the code belongs to: 1 to 5.

1XX Codes - Informational

The codes starting with 1XX are informational, like acknowledgement responses. We'll not be using this set of codes in this course.

2XX Codes - Success

The codes starting with 2XX are success codes. This indicates that the server received the request from the client and processed it successfully. Some examples:

200 OK

Indicates successful response. You'd return this for any request that you can successfully respond to.

201 Created

Indicates successful resource creation. Say you get a POST request for a collection URI like /messages and you successfully create a new message. You could return 200 OK to indicate success, but a better response code would be 201 Created.

204 No Content

Sometimes the server receives requests that need it to do something, but it doesn't need to return any content back. Like DELETE requests, for example. In this case, you could either return 200 OK with no response content. Or return 204 No Content, which makes it obvious that the server really intends to send nothing back.

3XX Codes - Redirection

The server sends these codes to ask the client to do further action to complete the request. For example, it could be a redirect, asking the client to send the request somewhere else.

302 Found and 307 Temporary Redirect

One of these two error codes are returned by the server if it wants the client to request elsewhere. It's a redirect.

304 Not Modified

When a client tries to get a resource that it has already got before, the server can send this status code to say "I've already given you this resource a little while back, and nothing has changed since then."

4XX Codes - Client error

These error codes are returned if the client makes an error in the request. The request syntax could have been incorrect, or the client is requesting something that it's not supposed to see.

400 Bad Request

This is a client error. The server is not able to understand the request

401 Unauthorized

The request needs the client to *sign in* or authorize themselves.

403 Forbidden

The client may have authorized, but they are still not allowed to make the request. (Maybe they don't have the right access rights).

404 Not Found

No description required. :)

415 Unsupported Media Type

The client is speaking in a language that the server cannot understand

5XX Codes - Server error

The 4XX codes are when the client screws up when sending the request. The 5XX codes are when the server screws up when sending the response. It's basically the server saying, Ok, I got your request, and it looked like a valid one, but something went wrong when I tried to process it.

500 Internal Server Error

This is a generic error code. The server gets a request. The resource exists (or you'd send a 404 instead) but something went wrong when processing the request. In such cases, the standard practice is to send the error code 500, along with error details in the body of the request.

There are a bunch of other codes, but these are the important ones to remember. We'll look at more when we start implementing some of these APIs. But let me remind you again. These error codes are for you, as a web service developer to use. The clients know what it means when they see one of these error codes. So, it's up to you to send the right error codes when these events happen. For example, let's say you get a runtime exception when processing a request. You need to send back error code 500. Because it means server error. And the client will then know what's happened.

Scenarios

Let's look at the same CRUD use cases we saw in the previous tutorial, and identify what the status codes should be for the message resource

| **Operation** | **URI** | **Method** | **Success / Failure** | **Status code** |
| --- | --- | --- | --- | --- |
| Get message | /messages/{messageId} | GET | Success | 200 |
|  |  |  | Not found | 404 |
|  |  |  | Failure | 500 |
| Delete message | /messages/{messageId} | DELETE | Success | 200 or 204 |
|  |  |  | Not found | 404 |
|  |  |  | Failure | 500 |
| Edit message | /messages/{messageId} | PUT | Success | 200 |
|  |  |  | Wrong format / data | 400 or 415 |
|  |  |  | Not found | 404 |
|  |  |  | Failure | 500 |
| Create message | /messages | POST | Success | 201 |
|  |  |  | Wrong format / data | 400 or 415 |
|  |  |  | Failure | 500 |

Hope this gives you a better idea of the status codes to be returned. Responses for other resources would mostly follow the same pattern. Again, this is just a small subset of the HTTP status codes, and we'll look at more when we start implementing APIs.

In this tutorial, we learnt about:

1. Resource representations
2. Message headers in HTTP and
3. HTTP status codes

Next tutorial::

<https://javabrains.io/courses/javaee_jaxrs/lessons/HATEOAS>

#### Notes

That's not a typo. This is actually an acronym. HATEOAS. In the running for the worst acronym ever. It stands for Hypermedia as the Engine of Application State. I know. Worst acronym ever, huh? But bear with me, and you'll soon understand what that means.

## Hyperlinking

Maybe you remember, I mentioned in the first video of this series that there's no service definition specification for REST APIs. There's no formal document that really documents the API. Most REST APIs have "help" pages that explain what the API URIs are and what operations are supported. I also mentioned in the first video that the best RESTful APIs don't even need any documentation. I'll now explain what I mean by that.

So, we visit websites online all the time. When was the last time you looked up any documentation to use a website? Well, never, I hope. You don't need documentation to use web sites. You go to the home page, and you'll find links to other pages. You click on one such link, and you'll get that page, with more links. You don't need to read a document to know where to go. You just remember the website home address, and any other links you need to navigate will be provided to you in the response.

This is basically the advantage of using HTTP. Remember that HTTP is HyperText Transfer Protocol. We've discussed that hypertext is text that has links to other text. These links, which are called hyperlinks, are what's really handy to navigate your way through any site. Let's think about the response we return in our REST API. What if we implement the same concept there too? Let's say you receive a GET request from a client for a message ID. We return the message information in JSON or XML, yes. But what you could also do is send links to comment resource URIs. And likes and shares resource URIs. It's the server saying "Hey client, I know you asked for message ID 20. Here's the contents of message #20. I'm also throwing in collection resource URIs for comments, shares and likes. If you want to get a list of all the comments for message ID 20, this is the URI to use. Oh, and here's the profile resource URI for the author of the message, if you want to get the profile information of the author of this message".

So, the web service is being super-helpful to the client by providing all these links in the response. Similar to hyperlinks in web sites. Whether the client wants to use it or not doesn't matter. But if they want it, it's there. And just like that, you've eliminated the need for documentation for all these APIs. The client developer just picks up the value of the right URIs from a previous response and makes subsequent calls to those URIs.

If you do this, you don't let the client programmer have to know and hard-code the URIs in order to interact with the resources and the application state. You basically let the hypertext you send in the response drive the client's interaction with the application state. So, you could say that hypertext, or hypermedia as it is sometimes called, is being the driver or engine of application state. Hypermedia as the Engine of Application State. HATEOAS. Whew. Does that make sense now? It's still a bad choice for a name. But it should at least make some sense now.

## A Scenario

Let's walk through a scenario so that this concept becomes clearer. Let's start with the /messages collection URI. Accessing /messages should give you a list of messages in the system. Let's say a message representation has the following fields:

1. Message ID
2. Message Content
3. Message Author
4. Posted Date

Four simple properties. A JSON representation for a sample message would look something like this:

{

"id": "01",

"content": "Hello World!",

"author": "koushik",

"postedDate": "03-01-2014"

}

Now when you access the /messages collection URI, you'd basically get a collection of such message resources. To keep it simple, let's say there are just 3 messages in the system. Accessing /messages would give something similar to this:

[

{

"id": "1",

"content": "Hello World!",

"author": "koushik",

"postedDate": "03-01-2014"

},

{

"id": "2",

"content": "Yo!",

"author": "sid",

"postedDate": "04-01-2014"

},

{

"id": "3",

"content": "What's up?",

"author": "jane",

"postedDate": "04-02-2014"

}

]

Now that the client has the list of messages, let's say they want the details of the first message - message ID 1. We've already designed the resource URI for message to be /messages/{messageId}. So, to get the URI, they'll have to take the value of the ID field of the message they are interested in, and append it to the string /messages/ and there they have the resource URI. But this means that the client will have to know this beforehand. They need to know that they need to pick up the ID property from the response, and they need to know what to append it to. Now, here's a question. As a API service implementer, why not send that to the client yourself? Since we are sending the message resource details anyway, why not just construct the URI fully and send it to the client?

Consider a sample response for a single message like this:

{

"id": "1",

"content": "Hello World!",

"author": "koushik",

"postedDate": "03-01-2014",

"href": "/messages/1"

}

If this were to be the kind of response for every message in /messages, then the client wouldn't really have to do any URI construction. The resource URI is one of the properties of the resource. If you were to design your API so that every resource has the instance resource URI to itself, it makes it really convenient for the client to use it.

Notice that the name of the link property is href. That must be familiar. That's exactly how you specify links in HTML. href is a property of the <a> tag. It serves a similar purpose here.

## Link relations

We are on our way to implementing some HATEOAS concepts. We are not fully there yet. There are still some more things you'll need to learn. Let's look at the concept of links, and how you can apply them to the resources in the Messenger API. We've looked at adding the resource URI to every resource. So, a profile resource, or a comment resource, well, pretty much every resource could have a href attribute that has the value of the instance resource URI. Bu that's not the only link you can provide. For instance, a message resource could also have links to get all the comments for that message. And all the likes and shares for that message. You could even have links for the client to post a new comment to that message. Keep extending this, and it gets a bit messy.

{

"id": "1",

"content": "Hello World!",

"author": "koushik",

"postedDate": "03-01-2014",

"href": "/messages/1",

"comments-href": "/messages/1/comments",

"likes-href": "/messages/1/likes",

"shares-href": "/messages/1/shares",

"profile-href": "/profiles/koushik",

"comment-post-href": "/messages/1/comments"

}

If you do this, the client doesn't need to remember the URIs, yes, but they now have to remember the property names for these URIs and you basically have then same problem. There needs to be a better way to manage these links. And there is! You can use the relattribute.

If you've used the anchor tags when writing HTML, you might have encountered this relattribute before. It's basically an attribute that you can add to any link to specify the relationship between the current document and the linked document.

The most common example of rel is in stylesheet links. You'd have seen stylesheet links in HTML head tags like this:

<link rel="stylesheet" href="path/to/some.css"/>

Here href provides the actual URL being linked, and the rel attribute describes the relation of that link to the main document. Here the relation is that the link is a stylesheet of the main document.

We can use the rel attribute to add extra information in the links in our REST response. Here's the original href response modified with the rel attribute addition:

{

"id": "1",

"content": "Hello World!",

"author": "koushik",

"postedDate": "03-01-2014",

"links" : [

{

"href": "/messages/1",

"rel": "self"

}

]

}

What's different here is that we've introduced this new property called links which is an array. This is going to contain all the links that you'd want to embed in the response. However, you add the rel attribute to make it clear what the link points to. Notice the rel value selfwhich indicates that the link in the resource points to itself.

This could be extended by adding new links and assigning the appropriate rel values for each:

{

"id": "1",

"content": "Hello World!",

"author": "koushik",

"postedDate": "03-01-2014",

"links" : [

{

"href": "/messages/1",

"rel": "self"

},

{

"href": "/messages/1/comments",

"rel": "comments"

},

{

"href": "/messages/1/likes",

"rel": "likes"

},

{

"href": "/messages/1/shares",

"rel": "shares"

},

{

"href": "/profiles/koushik",

"rel": "author"

}

]

}

Now the client doesn't need to remember the link property values. They just have to find the link with the right rel value for the resource they want and then look up the href value from that link.

A couple of things to note here. While the concept of having the URIs in the response to achieve HATEOAS is something that's well understood and mostly agreed upon by all, the way to do this could vary differently among implementations. The format of JSON that I've outlined here is just one of the multitude of ways you could structure links. Again, there's no right or wrong. You can choose to tweak how you want to present the links in the JSON response of your API depending on your preference. Secondly, the rel attribute is a part of the HTTP specification, so there are only certain standard values that are allowed for it. [This link](http://www.iana.org/assignments/link-relations/link-relations.xml) lists the available values. And obviously, the rel values here like "comments" and "likes" are not valid. But we'll still use it. Like I mentioned before, the idea is to have an API that's easy for the clients to use, and easy for you to maintain. You don't want to focus too much on getting things right and going by the book. At least, not at the cost of complicating the API too much.

In summary, HATEOAS is a way to provide links to resources in the API response, so that the client doesn't have to deal with URI construction and business flow. They make a request, and the next steps, along with the URIs are handed to them in the response. When you write APIs, you can choose to add URIs in the response using the href attribute. You can also provide more information about the relationship of the linked resource using the rel attribute.

Next tutorial::

<https://javabrains.io/courses/javaee_jaxrs/lessons/The-Richardson-Maturity-Model>

#### Notes

This is the final tutorial in section 1 API design. We've looked at different factors when building the RESTful API for the Messenger application. In this tutorial, we'll wrap up and take an overall look at how far we've come and what that means.

Below is the API documentation summary of what we have so far. I hope the choices and the design approach for this API is clear to you now. If you are unsure of why any part of the API is a particular way, I encourage you to revisit the relevant tutorial.

## Messages

| **Operation** | **URI** | **Method** | **Success / Failure** | **Status code** |
| --- | --- | --- | --- | --- |
| Get message | /messages/{messageId} | GET | Success | 200 |
|  |  |  | Not found | 404 |
|  |  |  | Failure | 500 |
| Delete message | /messages/{messageId} | DELETE | Success | 200 or 204 |
|  |  |  | Not found | 404 |
|  |  |  | Failure | 500 |
| Edit message | /messages/{messageId} | PUT | Success | 200 |
|  |  |  | Wrong format / data | 400 or 415 |
|  |  |  | Not found | 404 |
|  |  |  | Failure | 500 |
| Create message | /messages | POST | Success | 201 |
|  |  |  | Wrong format / data | 400 or 415 |
|  |  |  | Failure | 500 |

## Profiles

| **Operation** | **URI** | **Method** | **Success / Failure** | **Status code** |
| --- | --- | --- | --- | --- |
| Get profile | /profiles/{profileName} | GET | Success | 200 |
|  |  |  | Not found | 404 |
|  |  |  | Failure | 500 |
| Delete profile | /profiles/{profileName} | DELETE | Success | 200 or 204 |
|  |  |  | Not found | 404 |
|  |  |  | Failure | 500 |
| Edit profile | /profiles/{profileName} | PUT | Success | 200 |
|  |  |  | Wrong format / data | 400 or 415 |
|  |  |  | Not found | 404 |
|  |  |  | Failure | 500 |
| Create profile | /profiles | POST | Success | 201 |
|  |  |  | Wrong format / data | 400 or 415 |
|  |  |  | Failure | 500 |

## Comments (and similarly Likes and Shares)

| **Operation** | **URI** | **Method** | **Success / Failure** | **Status code** |
| --- | --- | --- | --- | --- |
| Get comment | /messages/{messageId}/comments/{commentId} | GET | Success | 200 |
|  |  |  | Not found | 404 |
|  |  |  | Failure | 500 |
| Delete comment | /messages/{messageId}/comments/{commentId} | DELETE | Success | 200 or 204 |
|  |  |  | Not found | 404 |
|  |  |  | Failure | 500 |
| Edit comment | /messages/{messageId}/comments/{commentId} | PUT | Success | 200 |
|  |  |  | Wrong format / data | 400 or 415 |
|  |  |  | Not found | 404 |
|  |  |  | Failure | 500 |
| Create comment | /messages/{messageId}/comments | POST | Success | 201 |
|  |  |  | Wrong format / data | 400 or 415 |
|  |  |  | Failure | 500 |

Now that we have designed the API this way, let's look at what this means. Are we in a position to say this API is "fully RESTful"? Remember, in the first tutorial, I mentioned that this isn't a yes or no question, and that there is a spectrum of anywhere from "not fully RESTful" to "almost RESTful" to "not RESTful at all". These terms are hard to work with. How do you know how RESTful an API is? Well, there is one way to know, and that's using a model developed by Leonard Richardson. It's called the Richardson Maturity Model, and it breaks down all the concepts we've discussed into 3 levels. Every REST API belongs to one of these 3 levels. The model also defines a Level 0 which is not a RESTful API. It is not necessary that every API score highly as per this model. But it helps to understand this model when designing any RESTful API so that you at least know where you stand. And try to make it better if possible.

Let's start with Level 0. I hope you are familiar with some of the basics of a SOAP web service. The way a SOAP web service generally works is that there is a URL called the endpoint where the service is exposed. One URL. That URL receives all requests from the client. If you were to write the Messenger API as a SOAP web service, you'd probably have one URI at /messenger. This URL receives all requests. How does it know what to do? How does the client tell it to do different stuff, like look up messages or delete a comment? Well, that happens in the message that's sent to this common URL. The message contains both the operation that needs to be performed, and the data that's needed for that operation. For example, the XML below could create a new message:

<create-message>

<message-content>Hello World!</message-content>

<message-author>koushik</message-author>

</create-message>

And a delete comment request (sent to the same URL) could look like this.

<delete-comment>

<message-id>30</message-id>

<comment-id>2</comment-id>

</delete-comment>

Notice that the operation that needs to be performed is a part of the request that's sent. This is how the same URL can be used for different operations. In fact, the same HTTP method can be used for each operation, because, all the details are in the request body. In fact, that's what SOAP does. The requests are always POST, with the POST body containing all the information.

This is Level 0 in the Richardson Maturity Model. This is often called the **The swamp of POX**. This refers to the common use of Plain Old XML (or POX) to define everything that an operation needs. No HTTP concepts are leveraged for communicating information between the server and client.

This design approach is obviously not something we want to do in this course. If you were to refine this model to introduce the concept of resource URIs, you will reach level 1 in the RMM. This is the starting level for RESTful APIs. The earlier level isn't even considered REST. We designed resource URIs for messages (/messages), profiles (/profiles) and so on. If you did just this, and nothing else, you stand at level 1. Now you have message requests going to one URI and all comments requests going to another URI. There would still be information about the operation in the requests, because the message URI needs to handle adding deleting or updating messages.

If you take the next step and use different HTTP methods for these different operations, then you've reached Level 2 in RMM. An API on Level 2 uses standard HTTP methods like GET, POST, PUT and DELETE to do different operations, on the resource URI. The URI specifies what resource is operated upon, and the HTTP method specifies what the operation is. There also needs to be better use of HTTP status codes, and the right use of idempotent and non-idempotent methods for an API to be at Level 2.

Finally, Level 3 is when you implement HATEOAS. That is, the responses have links that control the application state for the client. The client doesn't need to be aware of the different API URIs. All the URIs that the client would need is a part of the response that the server sends. If an API implements this, it is said to be at Level 3 of RMM, and is considered fully RESTful.

And there you go. Now you can look at any REST API design and easily identify which level in RMM it belongs to. Again, this is not supposed to be a strict rule. I encourage you to use this model as a guideline when designing your REST APIs, as a tool for learning and understanding, rather than a scorecard to measure with. You may not choose to make every API achieve Level 3 of RMM, but it helps to understand what the theoretical ideal is.

In the next section, we'll start looking at JAX-RS and start implementing this API. There's a lot to REST API design, and these tutorials have only scratched the surface. So, I do encourage you to explore more about REST APIs, and treat these tutorials as a starting point, rather than as complete learning.