Q: Only using **@JsonIgnore** during serialization, but not deserialization

Exactly how to do this depends on the version of Jackson that you're using. This changed around version 1.9, before that, you could do this by adding @JsonIgnore to the getter.

Which you've tried:

Add @JsonIgnore on the getter method only

Do this, and also add a specific @JsonProperty annotation for your JSON "password" field name to the setter method for the password on your object.

More recent versions of Jackson have added READ\_ONLY and WRITE\_ONLY annotation arguments for JsonProperty. So you could also do something like:

@JsonProperty(access = Access.WRITE\_ONLY)

private String password;

Q: How to forward a REST request to another resource?

@MatrixParam

@headerParam

@CookieParam

@Context uriInfo uriInfo

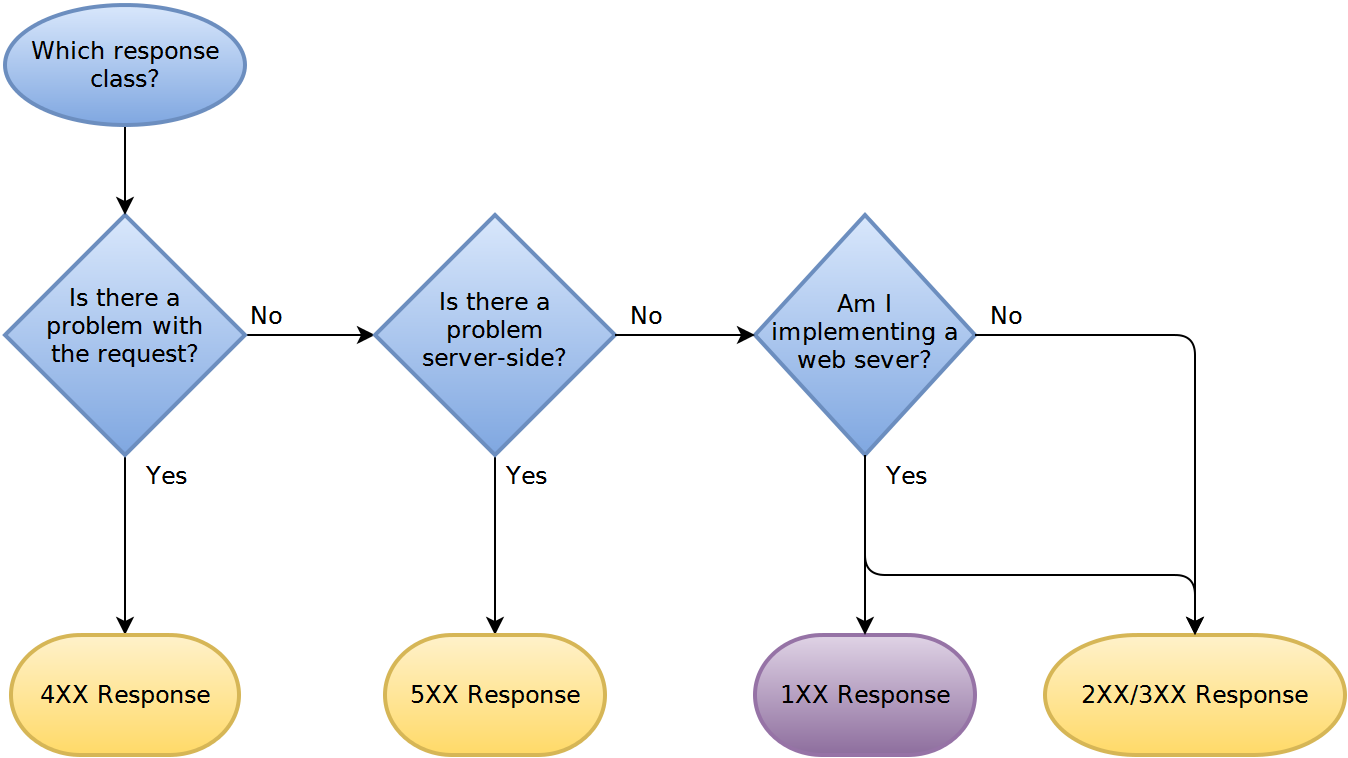
@Contect HttpHeaders header;

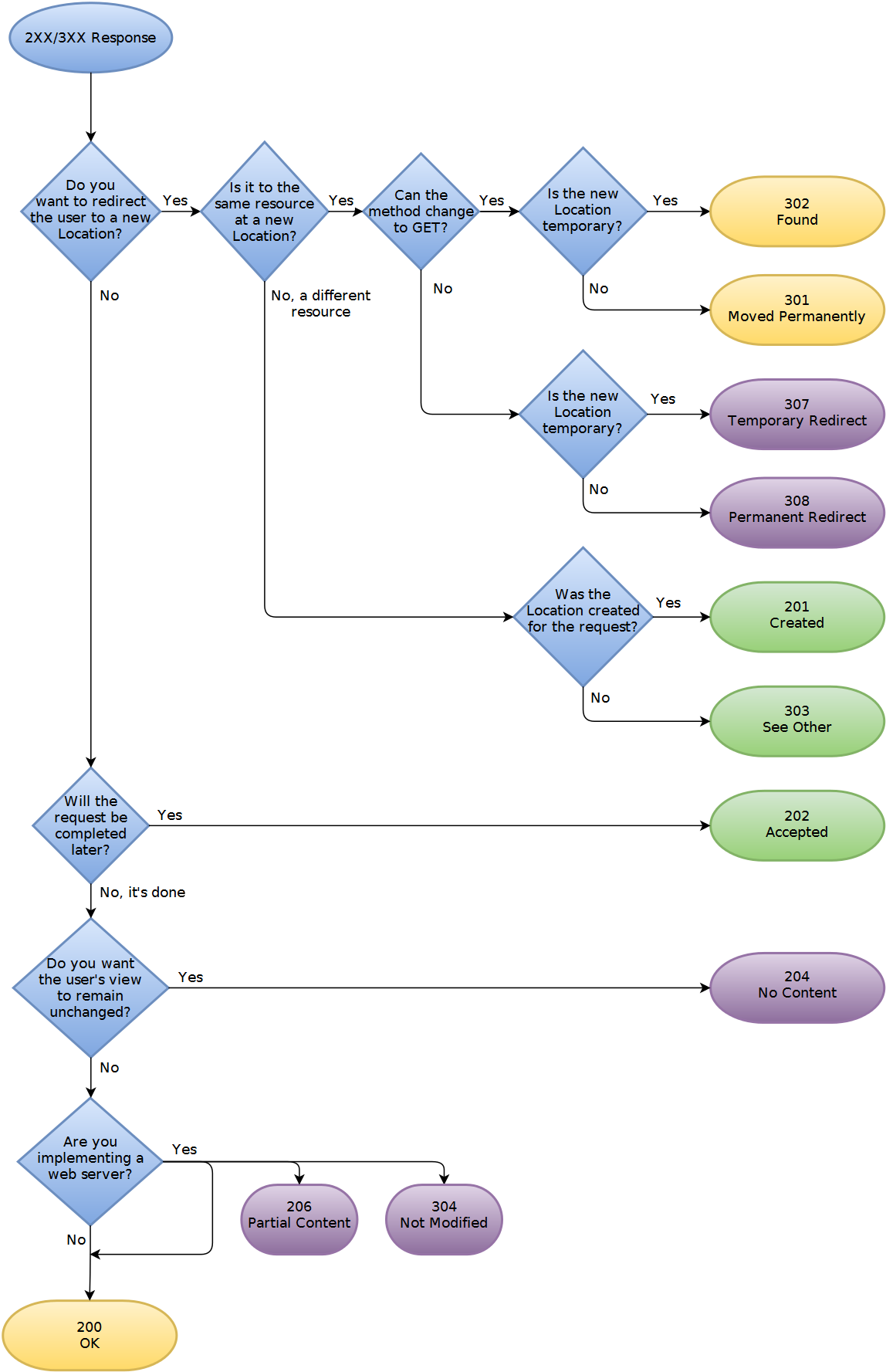
@Contect HttpHeaders header;

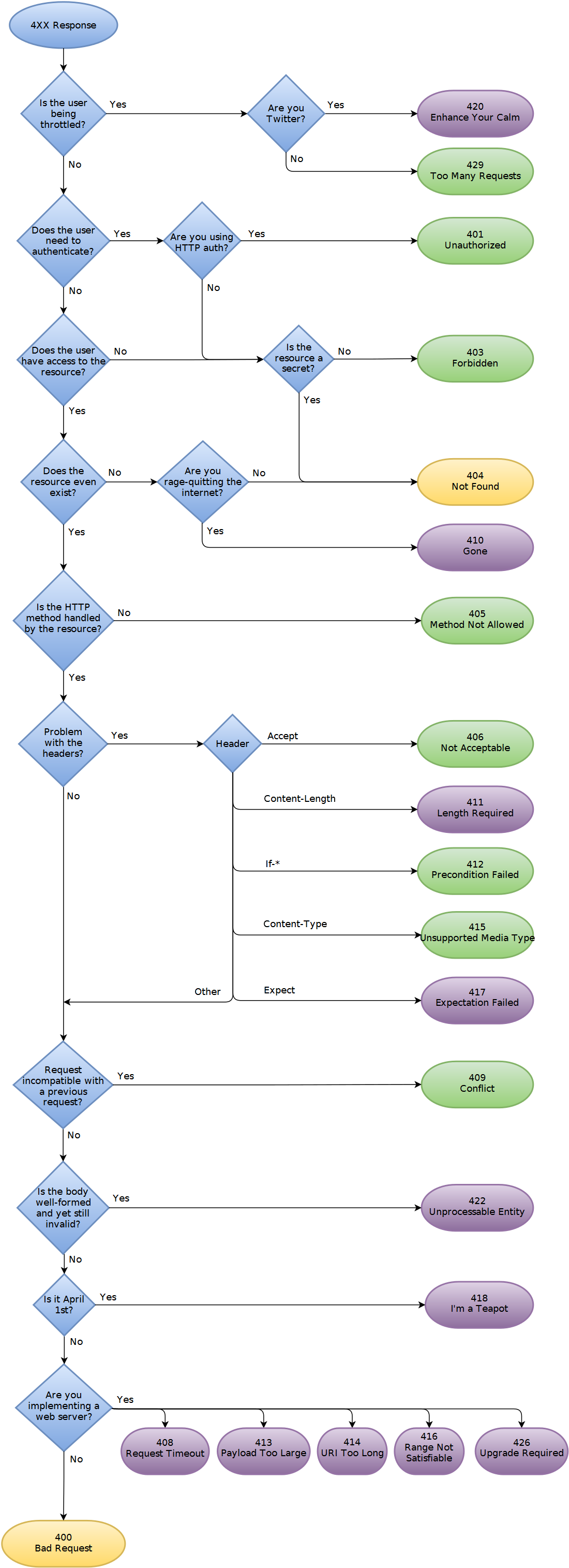
@BeanParam-> CustomBean🡪it should have properties with @MatrixParam and @headerParam

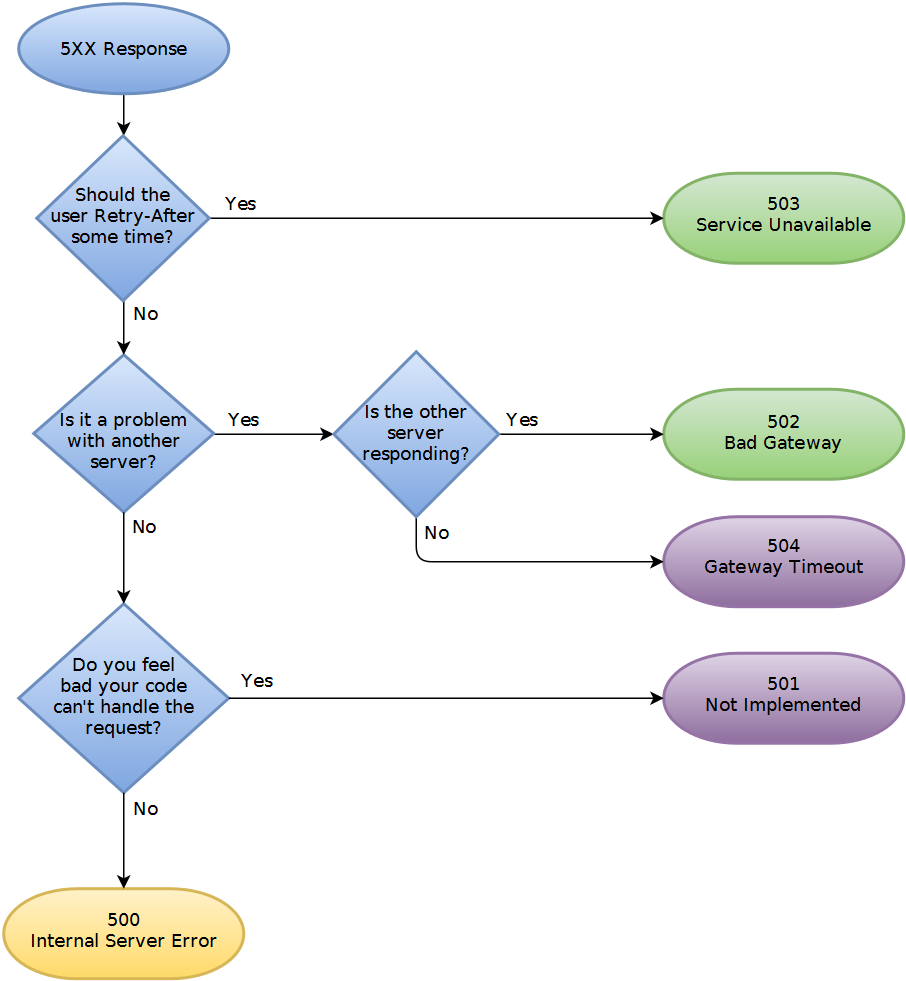
@path annotation is optional for Subresources

@XMLTransient









Q: How do I upload a file with metadata using a REST web service?

You basically have three choices:

1. Base64 encode the file, at the expense of increasing the data size by around 33%.
2. Send the file first in a multipart/form-data POST, and return an ID to the client. The client then sends the metadata with the ID, and the server re-associates the file and the metadata.
3. Send the metadata first, and return an ID to the client. The client then sends the file with the ID, and the server re-associates the file and the metadata.

I agree with Greg that a two phase approach is a reasonable solution, however I would do it the other way around. I would do:

POST http://server/data/media

body:

{

"Name": "Test",

"Latitude": 12.59817,

"Longitude": 52.12873

}

To create the metadata entry and return a response like:

201 Created

Location: http://server/data/media/21323

{

"Name": "Test",

"Latitude": 12.59817,

"Longitude": 52.12873,

"ContentUrl": "http://server/data/media/21323/content"

}

The client can then use this ContentUrl and do a PUT with the file data.

The nice thing about this approach is when your server starts get weighed down with immense volumes of data, the url that you return can just point to some other server with more space/capacity. Or you could implement some kind of round robin approach if bandwidth is an issue.

Q: Understanding REST: Verbs, error codes, and authentication

Question:

I am looking for a way to wrap APIs around default functions in my PHP-based web applications, databases and CMSs.

I have looked around and found several "skeleton" frameworks. In addition to the answers in my question, there is [Tonic](http://tonic.sourceforge.net/), a REST framework I like because it is very lightweight.

I like REST the best for its simplicity, and would like to create an API architecture based on it. I'm trying to get my head around the basic principles and have not fully understood it yet. Therefore, a number of questions.

**1. Am I understanding it right?**

Say I have a resource "users". I could set up a number of URIs like so:

/api/users when called with GET, lists users

/api/users when called with POST, creates user record

/api/users/1 when called with GET, shows user record

when called with PUT, updates user record

when called with DELETE, deletes user record

is this a correct representation of a RESTful architecture so far?

**2. I need more verbs**

Create, Update and Delete may be enough in theory, but in practice I will have the need for a lot more verbs. I realize these are things that could be embedded in an update request, but they are specific actions that can have specific return codes and I wouldn't want to throw them all into one action.

Some that come to mind in the user example are:

activate\_login

deactivate\_login

change\_password

add\_credit

how would I express actions such as those in a RESTful URL architecture?

My instinct would be to do a GET call to a URL like

/api/users/1/activate\_login

and expect a status code back.

That deviates from the idea of using HTTP verbs, though. What do you think?

**3. How to return error messages and codes**

A great part of REST's beauty stems from its use of standard HTTP methods. On an error, I emit a header with a 3xx,4xx or 5xx error status code. For a detailed error description, I can use the body (right?). So far so good. But what would be the way to transmit a **proprietary error code** that is more detailed in describing what went wrong (e.g. "failed to connect to database", or "database login wrong")? If I put it into the body along with the message, I have to parse it out afterwards. Is there a standard header for this kind of thing?

**4. How to do authentication**

* What would a API key based authentication following REST principles look like?
* Are there strong points against using sessions when authenticating a REST client, other than that it's a blatant violation of the REST principle? :) (only half kidding here, session based authentication would play well with my existing infrastructure.)

Ans:

I noticed this question a couple of days late, but I feel that I can add some insight. I hope this can be helpful towards your RESTful venture.

**Point 1: Am I understanding it right?**

You understood right. That is a correct representation of a RESTful architecture. You may find the following matrix from [Wikipedia](http://en.wikipedia.org/wiki/Representational_State_Transfer#Applied_to_web_services) very helpful in defining your nouns and verbs:

When dealing with a **Collection** URI like: **http://example.com/resources/**

* **GET**: List the members of the collection, complete with their member URIs for further navigation. For example, list all the cars for sale.
* **PUT**: Meaning defined as "replace the entire collection with another collection".
* **POST**: Create a new entry in the collection where the ID is assigned automatically by the collection. The ID created is usually included as part of the data returned by this operation.
* **DELETE**: Meaning defined as "delete the entire collection".

When dealing with a **Member** URI like: **http://example.com/resources/7HOU57Y**

* **GET**: Retrieve a representation of the addressed member of the collection expressed in an appropriate MIME type.
* **PUT**: Update the addressed member of the collection or create it with the specified ID.
* **POST**: Treats the addressed member as a collection in its own right and creates a new subordinate of it.
* **DELETE**: Delete the addressed member of the collection.

**Point 2: I need more verbs**

In general, when you think you need more verbs, it may actually mean that your resources need to be re-identified. Remember that in REST you are always acting on a resource, or on a collection of resources. What you choose as the resource is quite important for your API definition.

**Activate/Deactivate Login**: If you are creating a new session, then you may want to consider "the session" as the resource. To create a new session, use POST to http://example.com/sessions/with the credentials in the body. To expire it use PUT or a DELETE (maybe depending on whether you intend to keep a session history) to http://example.com/sessions/SESSION\_ID.

**Change Password:** This time the resource is "the user". You would need a PUT to http://example.com/users/USER\_ID with the old and new passwords in the body. You are acting on "the user" resource, and a change password is simply an update request. It's quite similar to the UPDATE statement in a relational database.

My instinct would be to do a GET call to a URL like /api/users/1/activate\_login

This goes against a very core REST principle: The correct usage of HTTP verbs. Any GET request should never leave any side effect.

For example, a GET request should never create a session on the database, return a cookie with a new Session ID, or leave any residue on the server. The GET verb is like the SELECT statement in a database engine. Remember that the response to any request with the GET verb should be cache-able when requested with the same parameters, just like when you request a static web page.

**Point 3: How to return error messages and codes**

Consider the 4xx or 5xx HTTP status codes as error categories. You can elaborate the error in the body.

**Failed to Connect to Database:** / **Incorrect Database Login**: In general you should use a 500 error for these types of errors. This is a server-side error. The client did nothing wrong. 500 errors are normally considered "retryable". i.e. the client can retry the same exact request, and expect it to succeed once the server's troubles are resolved. Specify the details in the body, so that the client will be able to provide some context to us humans.

The other category of errors would be the 4xx family, which in general indicate that the client did something wrong. In particular, this category of errors normally indicate to the client that there is no need to retry the request as it is, because it will continue to fail permanently. i.e. the client needs to change something before retrying this request. For example, "Resource not found" (HTTP 404) or "Malformed Request" (HTTP 400) errors would fall in this category.

**Point 4: How to do authentication**

As pointed out in point 1, instead of authenticating a user, you may want to think about creating a session. You will be returned a new "Session ID", along with the appropriate HTTP status code (200: Access Granted or 403: Access Denied).

You will then be asking your RESTful server: "Can you GET me the resource for this Session ID?".

There is no authenticated mode - REST is stateless: You create a session, you ask the server to give you resources using this Session ID as a parameter, and on logout you drop or expire the session.

Q: REST Content-Type: Should it be based on extension or Accept header?

Should the representation(html, xml, json) returned by a RESTful web service be determined by the url or by the Accept HTTP header?

Ans:

Both are valid. Quote from [xml.com](http://www.xml.com/pub/a/2004/08/11/rest.html):

A resource may have more than one representation. There are four frequently used ways of delivering the correct resource representation to consumers:

1. Server-driven negotiation. The service provider determines the right representation from prior knowledge of its clients or uses the information provided in HTTP headers like Accept, Accept-Charset, Accept-Encoding, Accept-Language, and User-Agent. The drawback of this approach is that the server may not have the best knowledge about what a client really wants.
2. Client-driven negotiation. A client initiates a request to a server. The server returns a list of available of representations. The client then selects the representation it wants and sends a second request to the server. The drawback is that a client needs to send two requests.
3. Proxy-driven negotiation. A client initiates a request to a server through a proxy. The proxy passes the request to the server and obtains a list of representations. The proxy selects one representation according to preferences set by the client and returns the representation back to the client.
4. URI-specified representation. A client specifies the representation it wants in the URI query string.

Q: SOAP vs REST (differences)

Question: I have read articles about the differences between SOAP and REST as a web service communication protocol, but I think that the biggest advantages for REST over SOAP are:

1. REST is more dynamic, no need for creating and updating UDDI.
2. REST is not restricted to XML format. REST web services can send plain text, JSON, and also XML.

But SOAP is more standardized (Ex; security).

So, am I correct in these points?

Ans1:

Unfortunately, there are a lot of misinformation and misconceptions around REST. Not only your question and the [answer by @cmd](https://stackoverflow.com/a/19884368/282110) reflect those, but most of the questions and answers related to the subject on Stack Overflow.

SOAP and REST can't be compared directly, since the first is a protocol (or at least tries to be) and the second is an architectural style. This is probably one of the sources of confusion around it, since people tend to call REST any HTTP API that isn't SOAP.

Pushing things a little and trying to establish a comparison, the main difference between SOAP and REST is the degree of coupling between client and server implementations. A SOAP client works like a custom desktop application, tightly coupled to the server. There's a rigid contract between client and server, and everything is expected to break if either side changes anything. You need constant updates following any change, but it's easier to ascertain if the contract is being followed.

A REST client is more like a browser. It's a generic client that knows how to use a protocol and standardized methods, and an application has to fit inside that. You don't violate the protocol standards by creating extra methods, you leverage on the standard methods and create the actions with them on your media type. If done right, there's less coupling, and changes can be dealt with more gracefully. A client is supposed to enter a REST service with zero knowledge of the API, except for the entry point and the media type. In SOAP, the client needs previous knowledge on everything it will be using, or it won't even begin the interaction. Additionally, a REST client can be extended by code-on-demand supplied by the server itself, the classical example being JavaScript code used to drive the interaction with another service on the client-side.

I think these are the crucial points to understand what REST is about, and how it differs from SOAP:

* REST is protocol independent. It's not coupled to HTTP. Pretty much like you can follow an ftp link on a website, a REST application can use any protocol for which there is a standardized URI scheme.
* REST is not a mapping of CRUD to HTTP methods. Read [this](https://stackoverflow.com/questions/19843480/s3-rest-api-and-post-method/19844272#19844272) answer for a detailed explanation on that.
* REST is as standardized as the parts you're using. Security and authentication in HTTP are standardized, so that's what you use when doing REST over HTTP.
* REST is not REST without [hypermedia](https://stackoverflow.com/a/29586455/1202421) and [HATEOAS](http://en.wikipedia.org/wiki/HATEOAS). This means that a client only knows the entry point URI and the resources are supposed to return links the client should follow. Those fancy documentation generators that give URI patterns for everything you can do in a REST API miss the point completely. They are not only documenting something that's supposed to be following the standard, but when you do that, you're coupling the client to one particular moment in the evolution of the API, and any changes on the API have to be documented and applied, or it will break.
* REST is the architectural style of the web itself. When you enter Stack Overflow, you know what a User, a Question and an Answer are, you know the media types, and the website provides you with the links to them. A REST API has to do the same. If we designed the web the way people think REST should be done, instead of having a home page with links to Questions and Answers, we'd have a static documentation explaining that in order to view a question, you have to take the URI stackoverflow.com/questions/<id>, replace id with the Question.id and paste that on your browser. That's nonsense, but that's what many people think REST is.

This last point can't be emphasized enough. If your clients are building URIs from templates in documentation and not getting links in the resource representations, that's not REST. Roy Fielding, the author of REST, made it clear on this blog post: [REST APIs must be hypertext-driven](http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven).

With the above in mind, you'll realize that while REST might not be restricted to XML, to do it correctly with any other format you'll have to design and standardize some format for your links. Hyperlinks are standard in XML, but not in JSON. There are draft standards for JSON, like [HAL](http://stateless.co/hal_specification.html).

Finally, REST isn't for everyone, and a proof of that is how most people solve their problems very well with the HTTP APIs they mistakenly called REST and never venture beyond that. REST is hard to do sometimes, especially in the beginning, but it pays over time with easier evolution on the server side, and client's resilience to changes. If you need something done quickly and easily, don't bother about getting REST right. It's probably not what you're looking for. If you need something that will have to stay online for years or even decades, then REST is for you.

Q: What exactly is RESTful programming?

Ans1:

An **architectural style** called [**REST (Representational State Transfer)**](https://en.m.wikipedia.org/wiki/Representational_state_transfer) advocates that web applications should use HTTP as it was **originally envisioned**. Lookups should use [GET](https://en.m.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods) requests. [PUT, POST, and DELETE requests](https://en.m.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods) should be used for **mutation, creation, and deletion respectively**.

REST proponents tend to favor URLs, such as

http://myserver.com/catalog/item/1729

but the REST architecture does not require these "pretty URLs". A GET request with a parameter

http://myserver.com/catalog?item=1729

is every bit as RESTful.

Keep in mind that GET requests should never be used for updating information. For example, a GET request for adding an item to a cart

http://myserver.com/addToCart?cart=314159&item=1729

would not be appropriate. GET requests should be [idempotent](https://en.m.wikipedia.org/wiki/Idempotence). That is, issuing a request twice should be no different from issuing it once. That's what makes the requests cacheable. An "add to cart" request is not idempotent—issuing it twice adds two copies of the item to the cart. A POST request is clearly appropriate in this context. Thus, even a **RESTful web application** needs its share of POST requests.

This is taken from the excellent book Core JavaServer faces book by David M. Geary.

Ans2:

REST is the underlying architectural principle of the web. The amazing thing about the web is the fact that clients (browsers) and servers can interact in complex ways without the client knowing anything beforehand about the server and the resources it hosts. The key constraint is that the server and client must both agree on the media used, which in the case of the web is HTML.

An API that adheres to the principles of REST does not require the client to know anything about the structure of the API. Rather, the server needs to provide whatever information the client needs to interact with the service. An HTML form is an example of this: The server specifies the location of the resource and the required fields. **The browser doesn't know in advance where to submit the information, and it doesn't know in advance what information to submit. Both forms of information are entirely supplied by the server.** (This principle is called [HATEOAS: Hypermedia As The Engine Of Application State](https://en.wikipedia.org/wiki/HATEOAS).)

**So, how does this apply to HTTP, and how can it be implemented in practice?** HTTP is oriented around verbs and resources. The two verbs in mainstream usage are GET and POST, which I think everyone will recognize. However, the HTTP standard defines several others such as PUT and DELETE. These verbs are then applied to resources, according to the instructions provided by the server.

For example, Let's imagine that we have a user database that is managed by a web service. Our service uses a custom hypermedia based on JSON, for which we assign the mimetype application/json+userdb (There might also be an application/xml+userdb and application/whatever+userdb - many media types may be supported). The client and the server have both been programmed to understand this format, but they don't know anything about each other. As [Roy Fielding](http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven) points out:

A REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media types.

A request for the base resource / might return something like this:

**Request**

GET /

Accept: application/json+userdb

**Response**

200 OK

Content-Type: application/json+userdb

{

"version": "1.0",

"links": [

{

"href": "/user",

"rel": "list",

"method": "GET"

},

{

"href": "/user",

"rel": "create",

"method": "POST"

}

]

}

We know from the description of our media that we can find information about related resources from sections called "links". This is called Hypermedia controls. In this case, we can tell from such a section that we can find a user list by making another request for /user:

**Request**

GET /user

Accept: application/json+userdb

**Response**

200 OK

Content-Type: application/json+userdb

{

"users": [

{

"id": 1,

"name": "Emil",

"country: "Sweden",

"links": [

{

"href": "/user/1",

"rel": "self",

"method": "GET"

},

{

"href": "/user/1",

"rel": "edit",

"method": "PUT"

},

{

"href": "/user/1",

"rel": "delete",

"method": "DELETE"

}

]

},

{

"id": 2,

"name": "Adam",

"country: "Scotland",

"links": [

{

"href": "/user/2",

"rel": "self",

"method": "GET"

},

{

"href": "/user/2",

"rel": "edit",

"method": "PUT"

},

{

"href": "/user/2",

"rel": "delete",

"method": "DELETE"

}

]

}

],

"links": [

{

"href": "/user",

"rel": "create",

"method": "POST"

}

]

}

We can tell a lot from this response. For instance, we now know we can create a new user by POSTing to /user:

**Request**

POST /user

Accept: application/json+userdb

Content-Type: application/json+userdb

{

"name": "Karl",

"country": "Austria"

}

**Response**

201 Created

Content-Type: application/json+userdb

{

"user": {

"id": 3,

"name": "Karl",

"country": "Austria",

"links": [

{

"href": "/user/3",

"rel": "self",

"method": "GET"

},

{

"href": "/user/3",

"rel": "edit",

"method": "PUT"

},

{

"href": "/user/3",

"rel": "delete",

"method": "DELETE"

}

]

},

"links": {

"href": "/user",

"rel": "list",

"method": "GET"

}

}

We also know that we can change existing data:

**Request**

PUT /user/1

Accept: application/json+userdb

Content-Type: application/json+userdb

{

"name": "Emil",

"country": "Bhutan"

}

**Response**

200 OK

Content-Type: application/json+userdb

{

"user": {

"id": 1,

"name": "Emil",

"country": "Bhutan",

"links": [

{

"href": "/user/1",

"rel": "self",

"method": "GET"

},

{

"href": "/user/1",

"rel": "edit",

"method": "PUT"

},

{

"href": "/user/1",

"rel": "delete",

"method": "DELETE"

}

]

},

"links": {

"href": "/user",

"rel": "list",

"method": "GET"

}

}

Notice that we are using different HTTP verbs (GET, PUT, POST, DELETE etc.) to manipulate these resources, and that the only knowledge we presume on the clients part is our media definition.

Further reading:

* The many much better answers on this very page.
* [How I explained REST to my wife](http://www.looah.com/source/view/2284)~~.~~
* [How I explained REST to my wife](http://web.archive.org/web/20130116005443/http:/tomayko.com/writings/rest-to-my-wife).
* [Martin Fowler's thoughts](http://martinfowler.com/articles/richardsonMaturityModel.html)
* [Paypal's API has hypermedia controls](https://developer.paypal.com/docs/api/)

(This answer has been the subject of a fair amount of criticism for missing the point. For the most part, that has been a fair critique. What I originally described was more in line with how REST was usually implemented a few years ago when I first wrote this, rather than its true meaning. I've revised the answer to better represent the real meaning.)