

The Power of HTTP

In this relaxing and RESTful exercise we are going to take a look at the often forgotten potential of HTTP to manage resources on the web. By looking at the basic verbs and status codes we are going to work our way through a series of simple operations for browsing and managing content on the web.

The HTTP Verbs

GET:	Request a representation of the specification resource. A safe operation; should NEVER be used to make permanent changes to resources.
HEAD:	Like GET, except this request only asks for the response headers containing information about the resource, such as the time it was last modified.
POST:	Enables the client to send data to the server for processing, e.g. create a new comment. Where the server creates the new resource is entirely its own decision. The created resources can also be modified to be different from that supplied by the client.
PUT:	Like POST except that the server must accept the clients exact data and create a resource at the location they request. A GET request should return exactly the same resource that was PUT by the client.
DELETE:	Deletes a resource such that is not available to be interacted with by anything but PUT.
OPTIONS:	Returns the supported HTTP verbs for a specified URL. Note, this is not widely supported.

Note: This is not a complete list of HTTP verbs.

HTTP Status Codes

All HTTP status codes consist of a three digit number. Codes are grouped into sets of a hundred.

1XX	2XX	3XX	4XX	5XX
Information	Successful	Redirection	Client Error	Server Error
<i>Continue (100)</i>	<i>OK (200)</i>	<i>Multiple (300)</i>	<i>Bad Request (400)</i>	<i>Error (500)</i>
<i>Switch Protocols (101)</i>	<i>Created (201)</i>	<i>Moved (302)</i>	<i>Unauthorized (401)</i>	<i>Not Implemented (501)</i>
	<i>Accepted (202)</i>	<i>See Other (303)</i>	<i>Not Found (404)</i>	<i>Unavailable (503)</i>
		<i>Use Proxy (305)</i>	<i>Method Not Allowed (405)</i>	<i>Timeout (504)</i>

Required Software

In order to get the most out of this exercise we recommend that you install the following software:

Terminal Client



Putty

A free telnet and ssh client for connecting to socket servers.

or for Mac OSX user and Linux users



Terminal

Powerful application for development, already installed.

Web Browser and Development Tools



Google Chrome + Add-ons

Googles Web Browser



Rest Console

Available in the Google Web Store in Chrome.



Postman Client

Available in the Google Web Store in Chrome.

In each exercise the required software will be referred to by its icon.

Putty Overview (if using it)

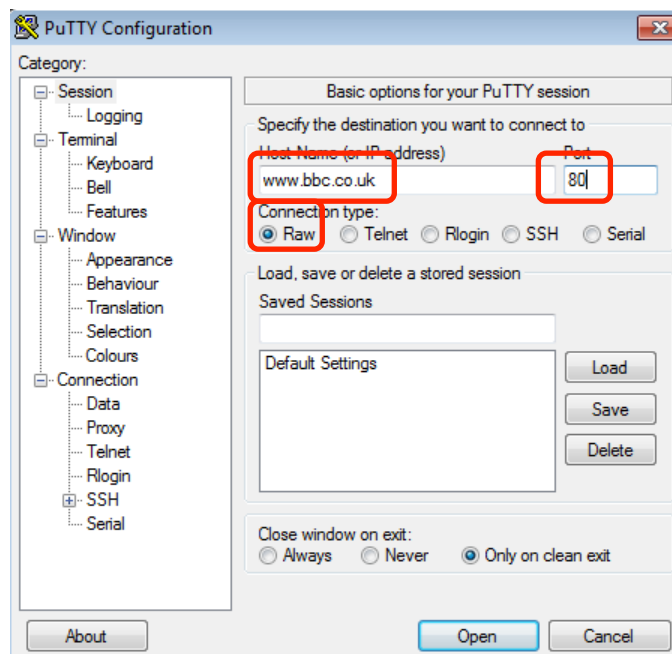
In all the exercises we are going to make use of Putty's ability to connect to a server using the telnet protocol. This is a raw socket level protocol compatible with that used to make requests over the web.

The following example shows how to interpret the instructions in this exercise into the data you enter into putty.

From **Exercise 1**:

```
> telnet www.bbc.co.uk 80
```

In **Putty**:



Note: Although in a terminal we use telnet, in putty we need to select the Raw protocol.

Once connected, you will just get a blank terminal window that you can type in. Note that you have a **limited** time to enter your data in this terminal before the server will close your connection and the terminal will just disappear.

We recommend you use the available examples in order to copy and paste the commands into your terminal to avoid running out of time. Remember you are pretending to be a machine, and machines are faster at this stuff than you.

1. Exercise 1 – Redirected

In this exercise we are going to look at how many sites in fact redirect the browser to other pages in order to retrieve content. This is something we cannot investigate using the browser.

This is of critical importance to data as it shows how you can redirect a user to their requested content dependant on the headers in their request. For example the user might have requested a JSON serialisation of the data about a URI and we can use redirections to tell them the URL where they can access this JSON.



1.1 A Simple Redirection

In order to simply demonstrate this we can connect to `www.google.com`. If you are in the UK then this *should* redirect you to `www.google.co.uk`. So lets try:

```
> telnet www.google.com 80
```

In the box that appears you want to do a GET request the root page of `www.google.com` `/` by entering the following and press the enter key twice (a blank new line ends a request):

```
GET / HTTP/1.1  
Host: www.google.com
```

You will notice that the first time we specify that we are talking HTTP is in the first line of the request and not before. Prior to this point we have only specified the address of the machine on the web we want to talk to and the port.

Once we are connected to the machine on the internet, we then request the root page `/` of the **host** that goes by the name of **www.google.com**. Thus the two pieces of information although the same are completely unrelated. Equally valid is to request the root page of `www.google.com` from another random machine on the internet.

For those who have heard about Domain Name Server (DNS) attacks, it is these attacks that try to move servers addresses on the internet so when a computer looks up the address of `www.google.com` it actually connects to a malicious server. This server will still respond to the request and the user has no way of knowing this has happened unless they are using a secure (`https`) connection on port 443, like your bank does.

Once you have run your request you should see something like the following output in your terminal:

```
HTTP/1.1 302 Found  
Location: http://www.google.co.uk/  
Cache-Control: private  
Content-Type: text/html; charset=UTF-8  
...
```

If you got an HTTP/1.1 400 Bad request, then you typed the request wrong and you will need to close your session and try again.

To close a telnet session in **terminal** hold **CTRL** and press the “**J**” key, then type **quit** and press **enter**. To get back to your previous command, press the **up** arrow.

In **putty** you just close the window and start again, so you might want to save the session information in putty to avoid having to type the same information lots of times.



1.2 Content Negotiation

In this exercise we are going to expand our request slightly and do some content negotiation on a URI to request some data about an object from our server. Our resource is notice 4090 (/notice/4090) on the “id.ecs.soton.ac.uk” server, making our URI “http://id.ecs.soton.ac.uk/notice/4090”, something you can try in a web browser first to see what it is.

```
> telnet id.ecs.soton.ac.uk 80
```

This time we are going to add an extra header to our request to say that we would like some RDF data about our resource.

```
GET /notice/4090 HTTP/1.1
Host: id.ecs.soton.ac.uk
Accept: application/rdf+xml
```

This time we see that the redirection is taking us to a new URL which contains the RDF data we want.

```
HTTP/1.1 303 See Other
...
Location: http://rdf.ecs.soton.ac.uk/notice/4090
...
```

What happens if you don't specify the Accept header?

1.3 Discussion

Redirection via content negotiation is a clear way of separating a URI “which represents a resource” from a serialised form of that resource.

Q: If a URI is actually identifying a person, and you do a GET request on that URI, should you be able to download that person?

A: When we invent teleporters, maybe.

For now we have to accept that we get a representation of that thing, rather than the thing itself. Logically, each representation is a resource in its own right, thus by redirecting to URLs we are allowing each representation to stand alone.

Not all sites enable content negotiation however, and some that do don't separate the different serialisations of the resource.

The BBC (music and programmes sites) don't separate serialisations from the URI and allow content negotiation directly on the URI:

Try asking from **text/xml** from:



<http://www.bbc.co.uk/music/artists/24f1766e-9635-4d58-a4d4-9413f9f98a4c>

OK, I slightly lied when I said that. To get the XML version you can just put a “.xml” on then end of that URI to make it a URL. Why they don't use redirection when they clearly have the capability is a little confusing.

Another example is dbpedia, the website currently at the centre of the linked-data web. You can also ask for various forms of their data by simply putting an extension on the end of the URI to turn it to a URL. You can try this with the following URIs to see what happens:



http://dbpedia.org/resource/Johann_Sebastian_Bach

VS. the URL



http://dbpedia.org/data/Johann_Sebastian_Bach.json

Many different sites expose their data in different ways, and those explained here are not the only ones out there. However the ones explained here are the ways support by the web and w3c standards.



2. Head to Browser Land

For the rest of these exercises we are going to move back over to the familiar environment of the browser. From now on, although we will be doing many of the same things we have been doing already, using the browser you will not be able to see any 3XX status codes, the browser annoyingly handles these transparently and we don't get to see them.

The example below shows how we connect to <http://id.ecs.soton.ac.uk/notice/4090> to first retrieve html by entering the correct data and pressing "send":

The screenshot shows the Postman interface with the following elements highlighted by red boxes:

- The URL input field containing `http://id.ecs.soton.ac.uk/notice/4090`.
- The method dropdown menu set to `GET`.
- The `Headers (0)` button.
- The `Send` button.

This time we are going to request the RDF version using an accept header, you will need to click the headers button to show this section.

The screenshot shows the Postman interface with the following elements highlighted by red boxes:

- The URL input field containing `http://id.ecs.soton.ac.uk/notice/4090`.
- The method dropdown menu set to `GET`.
- The `Headers (1)` button.
- The header input field containing `Accept` and the value `application/rdf+xml`.
- The `Send` button.

Before continuing, ensure you have familiarised yourself with the interface of POSTMAN, try looking at the various options under body and headers for each request.

The REST Console is much prettier and easier to use however does not allow us to do every type of request, thus why we have chosen to use POSTMAN in these exercises.



2.1 Heading our resources


In this exercise we are going to come away from the **GET** request and look at what the **HEAD** request does. Remember that you can specify all the same headers for a **HEAD** request that you can for a **GET** request. The difference is that a **HEAD** request will not actually return the resource, just the metadata about the resource.

Try a HEAD request on the following URIs and URLs

A BBC News RSS Feed:

 http://news.bbc.co.uk/rss/newsonline_world_edition/front_page/rss.xml

The Open Data Institute banner logo:

 <http://www.theodi.org/sites/default/files/logo.svg>

You will want to look at the returned headers of each request to find any useful information and discuss what it means for efficiency on the web.

3. Having a REST

Having now understood how to retrieve web based resources over HTTP we are now going to look at the other verbs, PUT, POST and DELETE. These three verbs make the web writable and allow the construction of powerful Application Programming Interfaces (APIs).

In the majority of cases, such APIs only allow GET, HEAD and POST requests in order to retrieve and manipulate existing objects. You don't start to see PUT and DELETE being made available unless it is your own personal content you are managing, e.g. via the flickr or Google APIs.

If such an API is used to manage representations of resources, using the HTTP verbs without applying any other limits, then this service is said to be a RESTful one. REST (Representational State Transfer) architectures consist of clients and servers. Clients initiate requests to servers; servers process requests and return appropriate responses. Requests and responses are built around the transfer of representations of resources. A resource can be essentially any coherent and meaningful concept that may be addressed. A representation of a resource is typically a binary file or document that captures the current or intended state of a resource.

With such APIs being able to Create, Retrieve, Update and Delete resources they are often also referred to as being CRUD APIs.

In this next exercise we are going to take a look at a simple picture collections management system and look at how such a system can manage data in both text and binary form.

In order to do this exercise you will need to be able to access a copy of the collections management server. Instructions on how to install or access this environment are provided separately to this exercise.

Once you have your collections management system running, you will need to know its location on the web, e.g. `pictures.example.org:3000`. In this exercise we shall refer to this as the `base_url`.

3.1 Exploring the Collections Management System



To first test that you can access the pictures management system, we recommend that you open the `base_url` in your normal web browser.

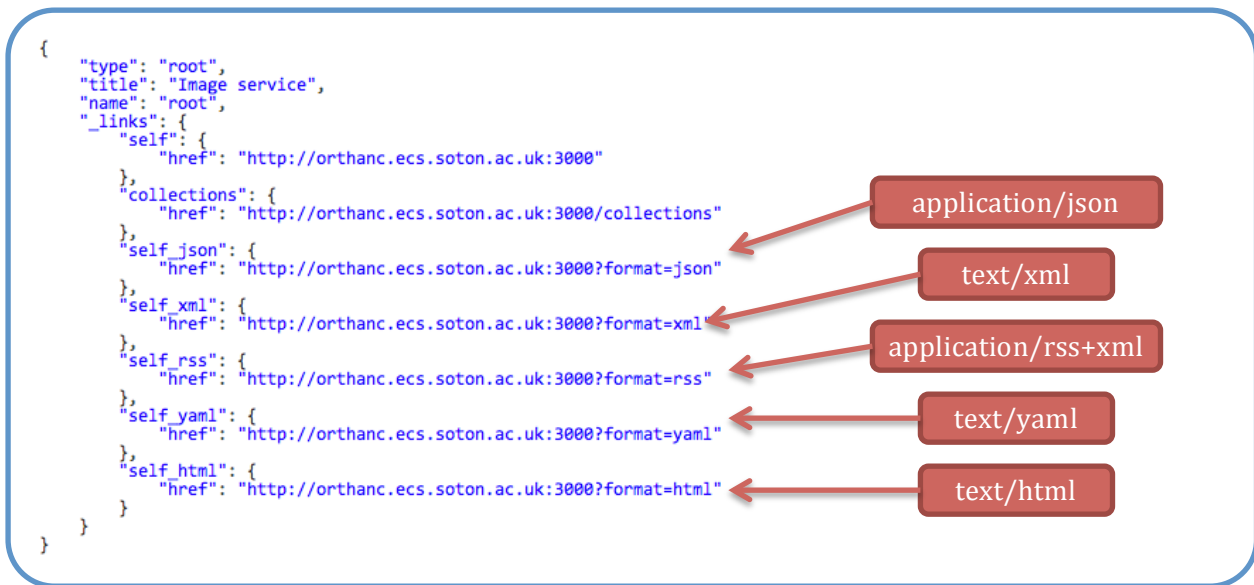


You should then be able to explore a number of collections of images and see related comments and metadata. Note how the URLs change while you are browsing, see if you can identify which ones are URIs and which are URLs.

While you are browsing you will also note that you can create new collections, change existing collections and upload new images. While this can be done via the web browser we are going to focus on how this can be done using a RESTful API.

Before we do this however, try browsing the system using the POSTman client or REST console, starting again at your `base_url`. You will note that because your browser did not ask for the text/html content type then you should get back a JSON object.

Looking at this JSON object you should also see a number of links to other format URLs. Unlike previous examples, to change the format of the response this time we add a query to the end of the URI.



In the diagram above, the content-types have been added. This can be used in combination with an **accept** header to perform content-negotiation on the objects in the system (as per the second example in section 2).

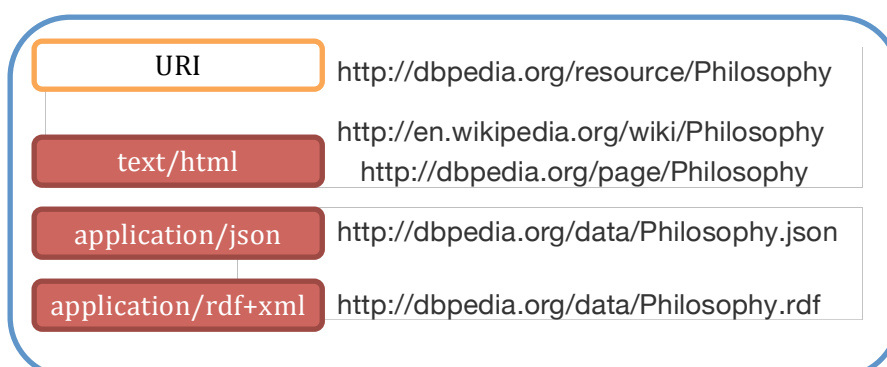
As you browse around the system, try and answer the following questions:

- How are the names, titles, descriptions, comments and other metadata represented?
- Why have a `_links` section? How does it change dependant on the resource you are browsing?
- What other data is available via the API which is not exposed in the web interface?
- What are the potential uses of each of the serialisations (JSON, RSS etc)?

In answering some of these questions it should become clear that there is a further separation of data from a serialised form of the data. We don't know the "master" form of the data in the collections management system, neither should we care, so long as the API exposes some useful serialisations.

This modular approach allows further serialisations to be added in the future, without needing to fundamentally change the data structure.

Wikipedia / dbpedia example (where the URI always 302s)



3.2 Creating a new collection

Using the **POST** verb, we can simply create a collection from the `base_url/collections`, this can be done both in a terminal or from a web browser client.



Using a terminal client:

```
> telnet base_url 3000
```

```
POST /collections HTTP/1.1
Host: base_url
```



Using the POSTman client:



In either case you should hopefully see a response telling you that the collection has been created (a 201 created HTTP status code) and a block of JSON (or whatever you requested) representing that object.

```
HTTP/1.1 201 Created
Content-Type: application/json
Content-Length: 595
Date: Sat, 30 Mar 2013 15:43:11 GMT
Connection: keep-alive

{
  "title": "Untitled",
  "name": "collection",
  "type": "item",
  "id": 34,
  "_links": {
    "self": {
      "href": "http://base_url:3000/collections/34"
    },
    "metadata": {
      "href": "http://base_url:3000/collections/34/metadata"
    },
    "parent": {
      "href": "http://base_url:3000/collections"
    },
    "collection": {
      "href": "http://base_url:3000/collections"
    },
    "comments": {
      "href": "http://base_url:3000/collections/undefined/comments"
    },
    "images": {
      "href": "http://base_url:3000/collections/undefined/images"
    }
  },
  "description": "",
  "created": 1364658318,
  "author": "Unknown"
}
```

One thing missing from this response is the content-location header in the response to tell the client where the resource was created. This is bad practice and needs to be corrected. In this case the resource was created at `http://base_url:3000/collections/34` as stated in the metadata's "self" link.

3.2 Manipulating a Collection

In the previous section we managed to create a new collection, however all the metadata is set to the default values, not very useful. In order to fix this we are going to perform a **PUT** operation on our collection URI using the following block of JSON data:

```
{
  "title": "ODI Collection",
  "name": "My cool collection of ODI related things",
  "type": "items",
  "description": "This collection is awesome!",
  "author": "Davetaz"
}
```

Although we can do this request using a terminal, I would recommend that you don't due to the complications in calculating the length of the object that you are sending to the server. The example below only works for the content defined above! Note all that your collection number must have already been created.

```
> telnet base_url 3000
```



```
PUT /collections/34 HTTP/1.1
Host: base_url
Content-Type: application/json
Content-Length: 186

{
  "title": "ODI Collection",
  "name": "My cool collection of ODI related things",
  "type": "items",
  "description": "This collection is awesome!",
  "author": "Davetaz"
}
```

Going via the browser, it will calculate the content-length for you (remember to change the collection number to the one you created previously):



The screenshot shows a web browser's developer tools interface. The URL bar shows `http://base_url:3000/collections/34`. The method dropdown is set to `PUT`. The `Content-Type` header is set to `application/json`. The `Headers` tab is selected, showing `Headers (1)`. The `raw` tab is selected, showing the JSON content:

```
1 {
2   "title": "ODI Collection",
3   "name": "My cool collection of ODI related things",
4   "type": "items",
5   "description": "This collection is awesome!",
6   "author": "Davetaz"
7 }
```

Try changing this request and updating the object to see how the response changes each time.

3.2 Deleting a Collection

One of the simplest operations available, as long as you remember your `base_url` and collection number.



Via a terminal:

```
> telnet base_url 3000
```

```
DELETE /collections/34 HTTP/1.1
Host: base_url
```



Or browser client:

The screenshot shows a browser client interface. The URL field contains `http://base_url:3000/collections/34`. The method dropdown is set to `DELETE`. There are buttons for `URL params` and `Headers (0)`.

Questions:

- What HTTP status code is returned in response of a delete operation?
- What happens if you try and delete the same collection again?
- Is this different than if you try and delete a collection that has never existed?

3.3 Adding a picture

This is the most complex operation here, and involves first creating a new image container, in the same way we created a collection, and then uploading an image.

Creating the image container can be done as follows, again ensure you change your `base_url` and collection number to ones that already exist.

The screenshot shows a browser client interface. The URL field contains `http://base_url:3000/collections/31/images`. The method dropdown is set to `POST`. There are buttons for `URL params` and `Headers (0)`.

Once you have created your image container you can upload your image to a specific URI. Note that in this example you will need your `base_url`, collection number and the image number returned to you in the previous response. Note that this URI we require is not listed in the response you just received, and again this is perhaps not the expected behavior.

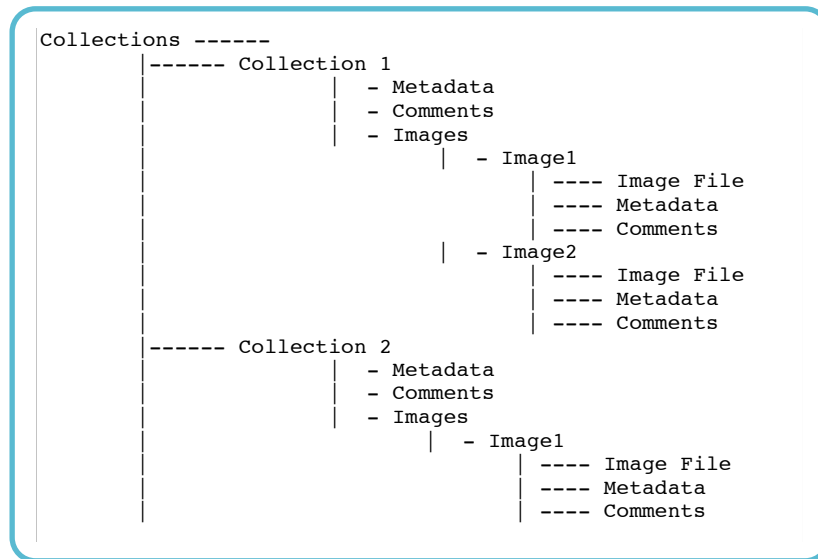
The screenshot shows a browser client interface. The URL field contains `http://base_url:3000/collections/31/images/29`. The method dropdown is set to `POST`. There are buttons for `URL params` and `Headers (0)`. Below the URL field, there are tabs for `form-data`, `x-www-form-urlencoded`, and `raw`. The `form-data` tab is selected. Under this tab, there is a `file` section with a `Choose Files` button and a file named `logo.png`. There is also a `Text` section with a `Text` dropdown.

Questions:

- What happens if you try to replace an image using PUT?
- Can you think of a better way to handle image uploads (hint, this way is not the simplest)?
- Are there any other issues with handling images using this API that seem inconsistent?

The Collection Management System Structure

The following describes the structure of resources in the collections management system you have just been experimenting with. In addition to handling collections and related images, the collection management system also allows for the inclusion of comments and other related metadata. Feel free to experiment with the system to see if you can discover the difference between each type of resource and the methods of interaction and serialisation.



API Overview

GET	/	Indexing of available services and formats
GET	/collections	List of all collections
POST	/collections	Create a collection
GET	/collections/:collectionId	Get a specific collection
PUT	/collections/:collectionId	Change a specific collection
DELETE	/collections/:collectionId	Delete a specific collection
GET	/collections/:collectionId/metadata	Get just the metadata for a specific collection
GET	/collections/:collectionId/comments	Get all comments for a collection
POST	/collections/:collectionId/comments	Create a specific comment about a collection
GET	/collections/:collectionId/comments/:commentId	Get a specific comment about a collection
PUT	/collections/:collectionId/comments/:commentId	Change a specific comment about a collection
DELETE	/collections/:collectionId/comments/:commentId	Delete a specific comment about a collection
GET	/collections/:collectionId/images	Get all images in a collection
POST	/collections/:collectionId/images	Create an image in a collection

GET	/collections/:collectionId/images/:imageId Get a specific image in a collection
PUT	/collections/:collectionId/images/:imageId Change a specific image in a collection
DELETE	/collections/:collectionId/images/:imageId Delete a specific image in a collection
GET	/collections/:collectionId/images/:imageId/metadata Get just the metadata for an image in a collection
POST	/collections/:collectionId/images/:imageId/image Set the image file for an image in a collection
PUT	/collections/:collectionId/images/:imageId/image Set the image file for an image in a collection (same as above)
GET	/collections/:collectionId/images/:imageId/image Get the image file for an image in a collection (redirects to actual file location)
GET	/collections/:collectionId/images/:imageId/comments Get all the comments for an image in a collection
POST	/collections/:collectionId/images/:imageId/comments Create a comments for an image in a collection
GET	/collections/:collectionId/images/:imageId/comments/:commentId Get a specific comment for an image in a collection
PUT	/collections/:collectionId/images/:imageId/comments/:commentId Change a specific comment for an image in a collection
DELETE	/collections/:collectionId/images/:imageId/comments/:commentId Delete a specific comment for an image in a collection

Status Codes

200 OK	Normal response (everything is fine)
201 Created	Response when POST has been handled and model is created
204 No Content	No content (after deletion)
301 Redirect	Redirect. This is used to redirect images accessed via REST to the publicly available image files e.g. /collections/1/images/20/image -> /images/1/20.jpg
400 Bad Request	Shown when an invalid ID is provided (i.e. not a number)
304 Not Modified	Provided when a browsers cached version is suitable
404 Not found	Response when item with provided ID does not exist
406 Not Acceptable	Cannot provide a data in a format that satisfies Accept headers or override format
410 Gone	Response when item with provided ID has been deleted
500 Internal Server Error	Response when something in the server has failed (e.g., database file fails to load)

Credits

I would like to thank Peter West, Alaa Al Marshedi and Xiaohu Zhou, students of the University of Southampton for kindly helping create the collection management system.