Understand REST by exploring  
a working RESTful API

STUDENT WORKBOOK

Created by Richard Pawson

[Introduction 3](#_Toc510197735)

[Exercise 1. Exploring the application from a user perspective 4](#_Toc510197736)

[Exercise 2. Viewing JSON 7](#_Toc510197737)

[Exercise 3. Following links 11](#_Toc510197738)

[Exercise 4. Exploring the metadata about a link 15](#_Toc510197739)

[Exercise 5. Invoking an action with parameters 17](#_Toc510197740)

[Exercise 6. Exploring alternative HTTP methods 20](#_Toc510197741)

[Exercise 7. Using the REST Client plug in 23](#_Toc510197742)

[Exercise 8. Exploring how the application communicates with the RESTful API 26](#_Toc510197743)

[Exercise 9. Test questions 27](#_Toc510197744)

# Introduction

In this workbook we are going to learn about ‘RESTful APIs’. REST stands for REpresentational State Transfer, but that information probably doesn’t help your understanding one little bit! By the end of this workbook you might *still* not understand exactly why it was named that way (few people do) - but you *will* understand what exactly a RESTful API *is*, and why it is so important.

The ‘API’ part of the term may already be familiar to you: it stands for Application Programming Interface. An API is a software technique whereby one program can access data and/or functionality from another program, without needing to understand the internals of that other program, and without requiring that other program to be modified or re-compiled. APIs are used whenever an application program makes use of ready-made functionality from a library, or when an application program accesses a database, say. One important category of API is used to allow one program to access the data/functionality of another *over a network*. This might arise when two different programs need to send messages to each other - for example in an electronic mail system - but most commonly arises when you have a ‘client’ program, running on a PC or mobile phone, say, that needs to communicate with shared resources running on a dedicated server machine in your server room, or on a ‘virtual’ machine ‘in the cloud’.

APIs designed to support inter-program communication over a network may be built using many different technologies: RPC, SOAP, Corba, .NET remoting, Windows Communication Foundation, WebSockets, Signal R, to name but a few. A *RESTful API* is one that specifically communicates using a ‘hypertext protocol’ - in practice that almost always means HTTP, which was one of the enabling technologies for the World Wide Web. However, any API that communicates over HTTP is not *necessarily* RESTful: REST imposes some additional constraints. Those constraints are not very complex, but to be RESTful, an API must follow them rigorously. In this workbook, we will specify those constraints by exploring a real RESTful API.

But what’s the payoff? Why does is matter whether an API is RESTful, or not? The answer is that if an API that follows all the rules of REST, then the system as a whole can full take advantage of the infrastructure of the World Wide Web. ‘Infrastructure’, here, does not just refer to the physical infrastructure of the Internet, but to the software infrastructure that handles: security; temporary caching of data to reduce network traffic; sharing the load between multiple servers; fail-safe redundancy; temporary (or permanent) re-routing of requests to a different URL; and on, and on …

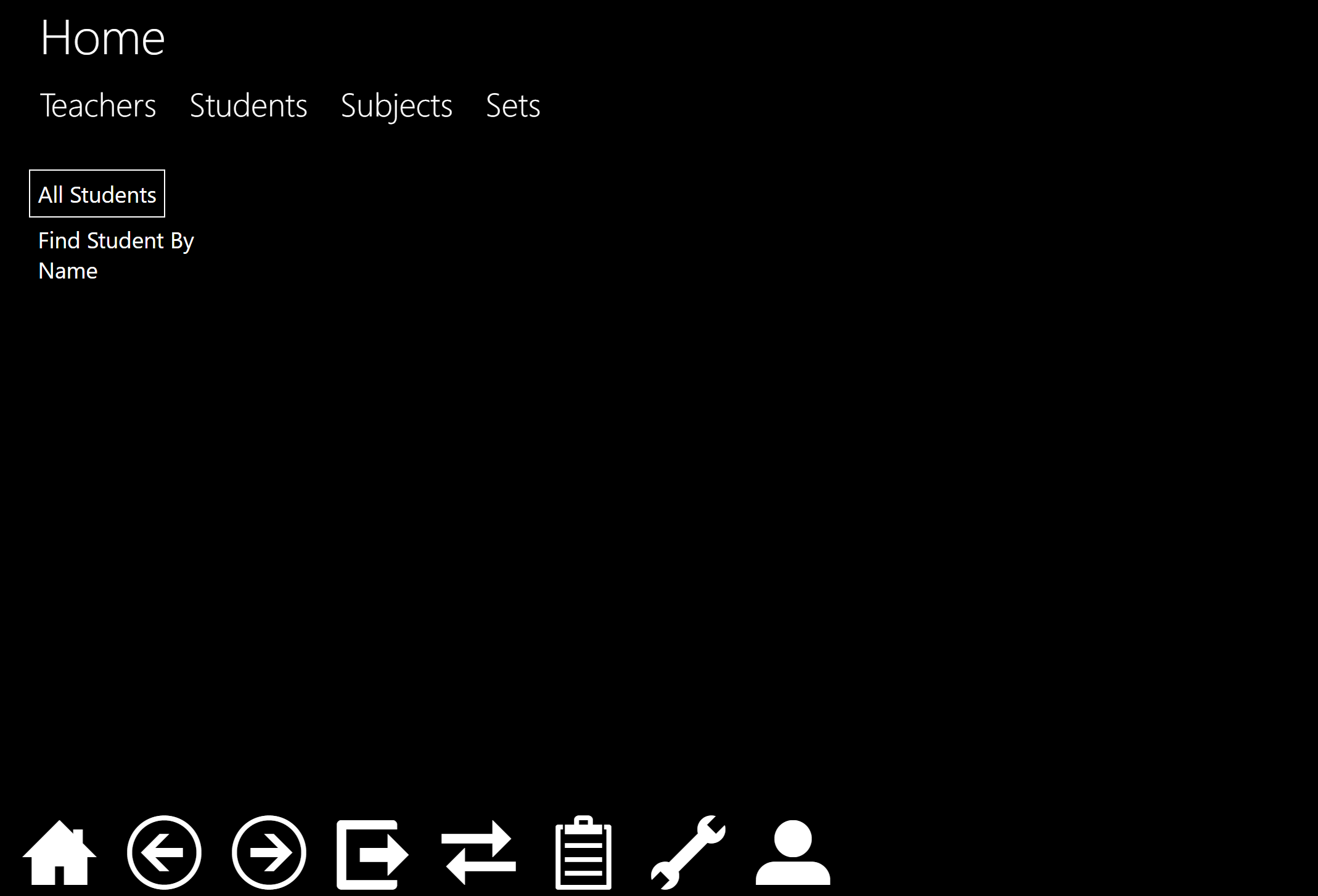
Since the very beginnings of the World Wide Web in the early 1990s, sceptics were predicting that it would not scale up. Since then it has scaled up faster than any system in the history of technology.

This is, in large part, because the original team that designed HTTP did such a brilliant job of it, anticipating issues that few people could have foreseen. RESTful APIs capitalise on that extraordinary work, even though their application has little in common with the original conception of the World Wide Web.

We are going to explore a working RESTful API built on top of a *very simplified* school administration system, called Academy. Before looking at the RESTful API, however, we will briefly explore the application from a user’s perspective, to understand what it does.

# Exploring the application from a user perspective

The Academy application can be reached on this URL: <http://academyspa.azurewebsites.net>, which takes you to the home screen, showing four ‘main menus’, each of which will expand when clicked:

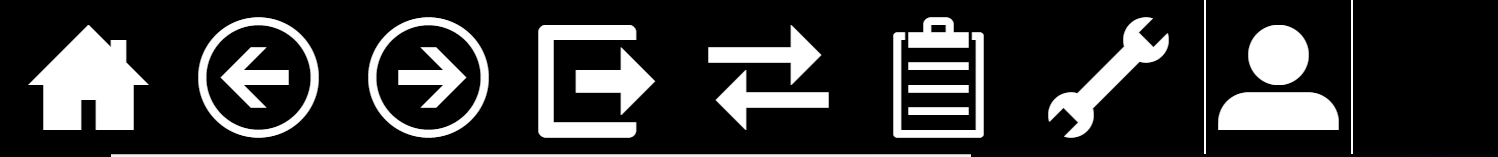


Click on **Students > All Students** to return the (short) list of example students, then **right-click** on one of those students, which will split the screen:



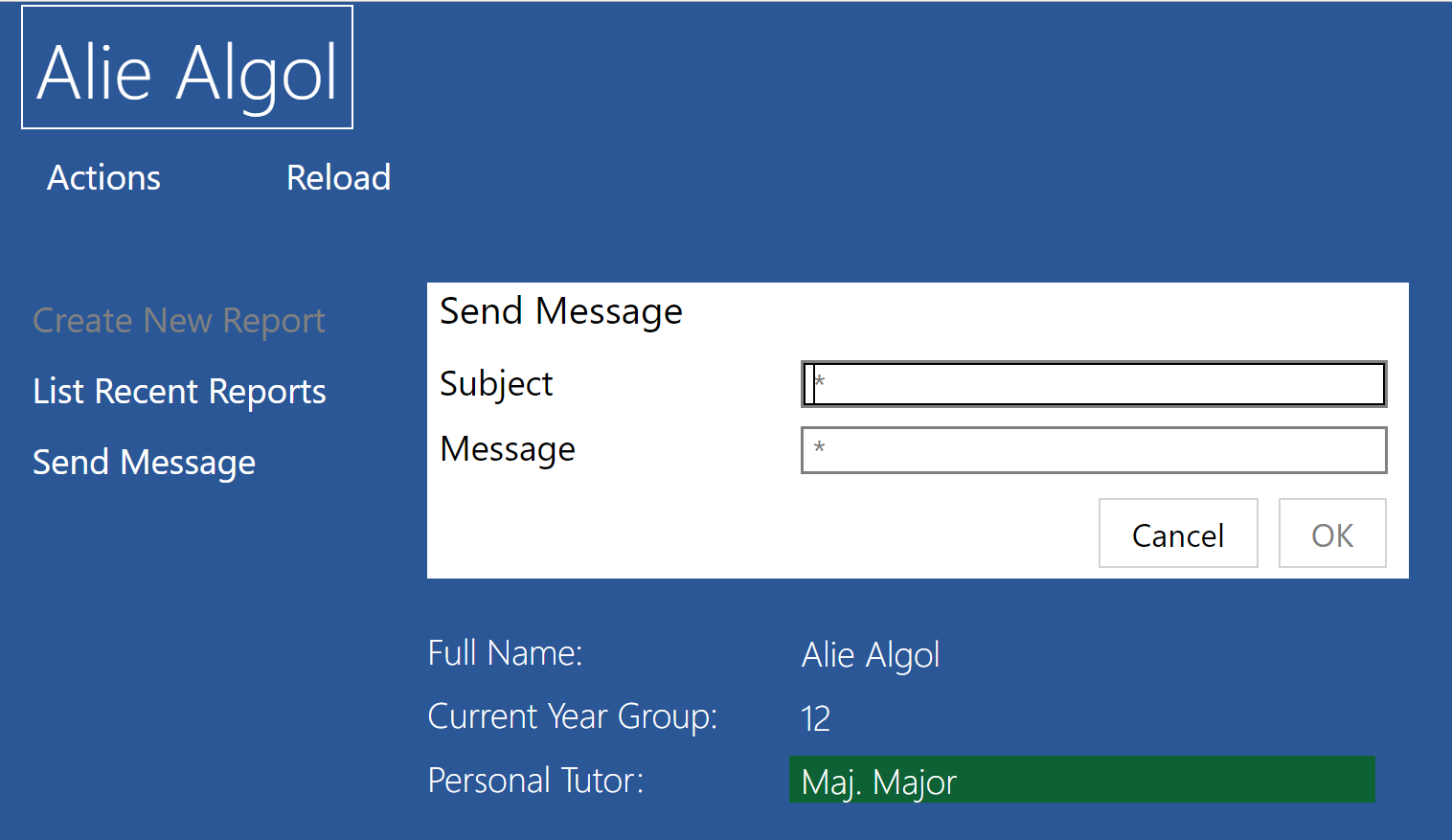
Try navigating around the system, just by left-clicking and right-clicking on the various links (coloured bars). You’ll quickly realise that each colour represents a different kind object - this system being built using the object-oriented programming (OOP) paradigm.

You should also explore the role of the icons at the bottom of the screen. Hovering on each icon will give you a tooltip, and you can then click it. The most important are the first three: Home, Back, and Forward, but you are welcome to explore the others. (We’ll come back to the Spanner icon shortly).



Finally notice that certain objects display an **Actions** button, which will expand to show actions that can be applied *to that object*. If you already understand OOP: these actions correspond to *instance methods* on that object; if you don’t yet understand OOP: it is not necessary for this workbook.

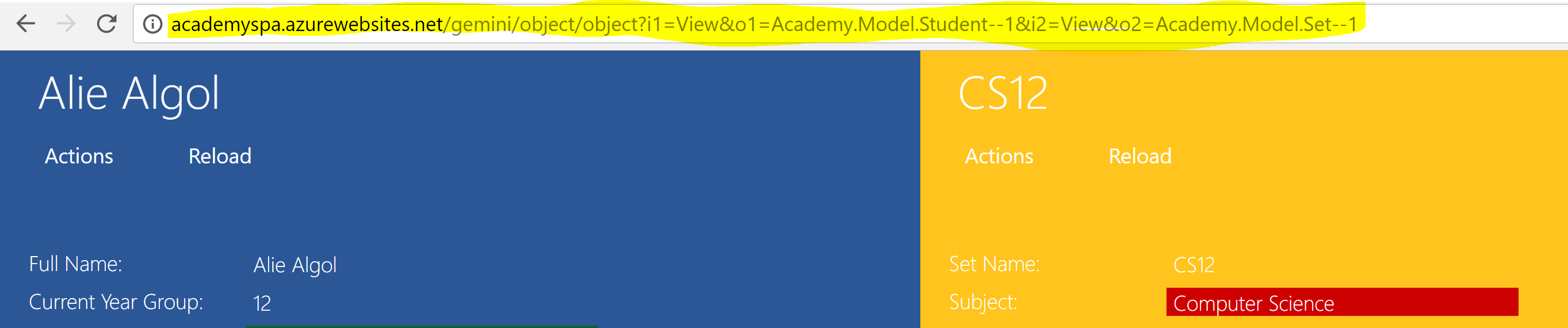
Some of these actions will be greyed-out, because, as a non-registered user, you are not authorized to create new objects. You can try using the Send Message action on Student, by completing the dialog, and hitting **OK**:



Don’t worry - these are fictitious students, and your message won’t actually be transmitted, but you can see how actions work in principle.

### URLs

As you explore the application you will notice that the URL shown at the top of the browser is changing as the contents of the screen change - just as it does when you browse the web. For example:

 It would seem reasonable to infer that these are the URLs on the server where the application is getting its data from. *But, surprisingly, that inference would be wrong.*

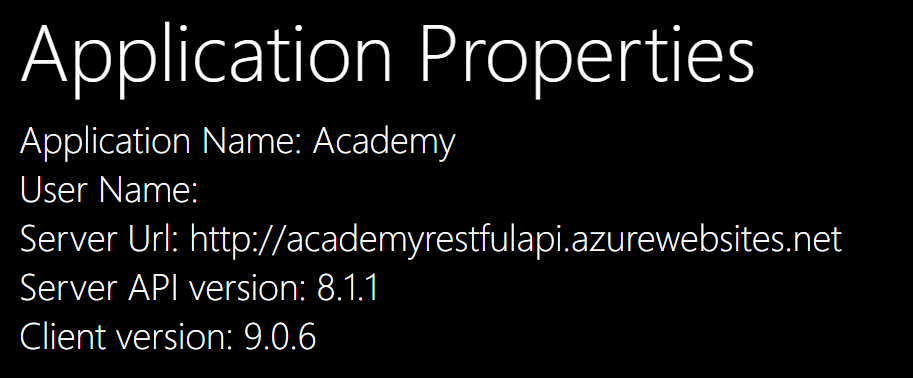
The Academy application that you are using is designed according to a modern software architecture known as a ‘Single Page Application’, or SPA.. When you accessed the application originally, via <http://academyspa.azurewebsites.net> your browser downloaded an Html file that contains a minimal amount of Html, plus a very substantial JavaScript file that incorporates all of the *client* application logic. As you navigate around the application, in the background the JavaScript is sending lots of requests (to retrieve data, and to execute functionality) to *another* server, and then processing any returned data to create the HTML that is displayed for each screen.

Once you’ve started the application, the changing URLs that you see *at the top of the screen* are never requested of any server (unless you do a Refresh on the browser, which is neither necessary nor recommended). Rather, that URL is just used locally by the JavaScript as a kind of shorthand way to specify what is being displayed on screen at any point.

In fact, if a particular user action does not require new data to be fetched, nor changes to be uploaded, nor actions that involve server logic, then the action is executed purely locally, within the browser, by the JavaScript. For example, if you are just navigating around objects that you have recently viewed, there is no network communication involved at all. (In the final exercise of this workbook you’ll be able to prove that for yourself).

### Finding the ‘real’ application server

If you click on the spanner icon, you will be shown a list of Application Properties:



(There is no User Name, because you were not required to log on to this rather limited version of the application).

Notice the **Server URL** property. *This* is the server that the client’s JavaScript is accessing in the background, to retrieve data and invoke server functionality. Both this, and the server from which your browser originally downloaded the client application, are hosted on Azure, so both URLs end in azurewebsites.net, but they have different server names: academyspa, and academyrestfulapi, respectively. As the name suggests, the second server is the one that provides the RESTful Api to be ‘consumed’ by the client (SPA).

We are now going to explore the RESTful API *directly* – seeing it the way the JavaScript on the client would be ‘seeing’ it. Be prepared for it not looking like a nice, user-friendly, presentation: the RESTful API is not intended to be seen by any end-user!

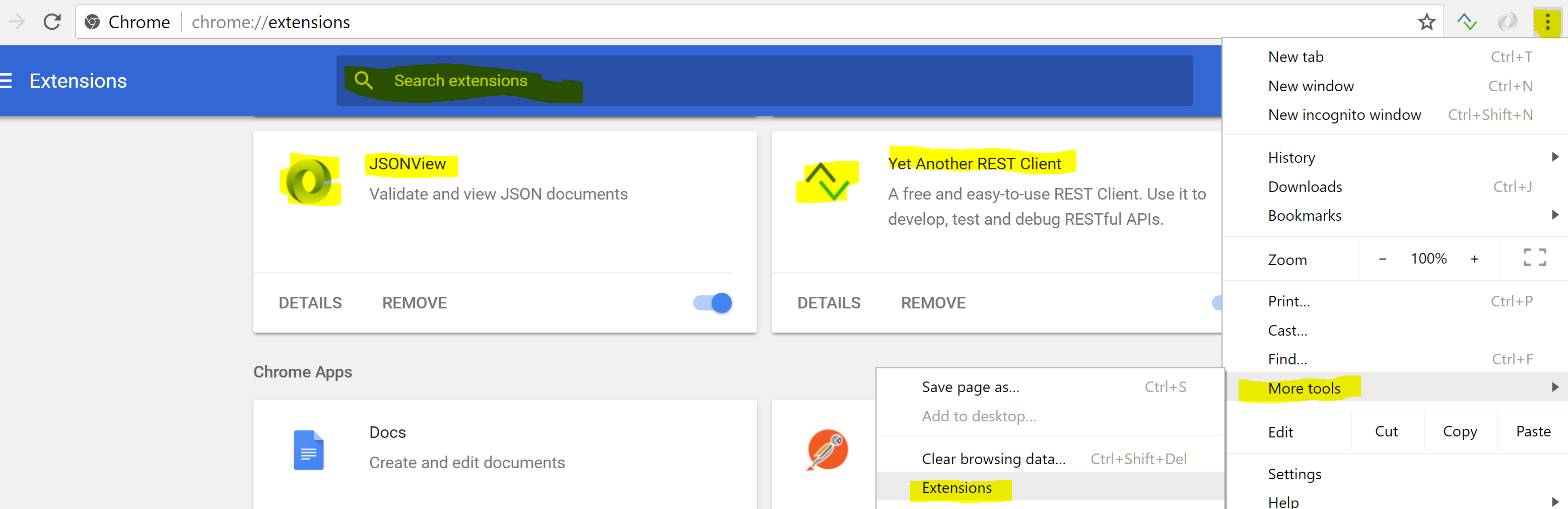
# Viewing JSON

For the next exercise you will need to use the Chrome browser, and to install two plug-ins. (It is possible to use an alternative browser with equivalent capabilities, but without such capabilities you may not be able to view or use the RESTful API properly).

In Chrome, go to the **Extensions** menu, then use the search bar to find and install these two extensions:

* **JSONView**
* **Yet Another REST Client** (we won’t use this until a later exercise, but you might as well install it now).

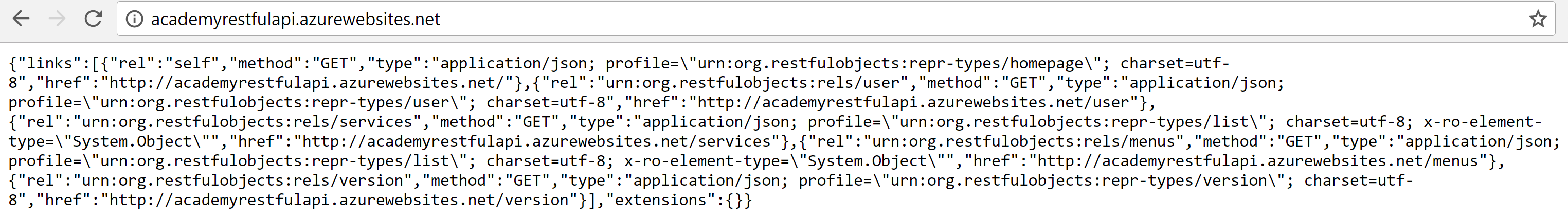
(Be careful: there are a number of different extensions with fairly similar names. Check the logos as shown below).



Then navigate directly to the URL for the RESTful Api that was shown on the Application Properties screen in the previous exercise. The browser should display this:



If your display looks like this:



It means that your browser is not set up to format JSON: either JSONView was not installed, or it is not switched on.

## So what is JSON, exactly?

JSON is a specific standard for representing structured data. An alternative, somewhat similar standard, is known as XML, which stands for eXtensible Markup Language. Here is the same data as shown in the JSON above, rendered as XML:

<?xml version="1.0" encoding="UTF-8"?>

<root>

<extensions />

<links>

<element>

<href>http://academyrestfulapi.azurewebsites.net/</href>

<method>GET</method>

<rel>self</rel>

<type>application/json; profile='urn:org.restfulobjects:repr-types/homepage'; charset=utf-8</type>

</element>

<element>

<href>http://academyrestfulapi.azurewebsites.net/user</href>

<method>GET</method>

<rel>urn:org.restfulobjects:rels/user</rel>

<type>application/json; profile='urn:org.restfulobjects:repr-types/user'; charset=utf-8</type>

</element>

<element>

<href>http://academyrestfulapi.azurewebsites.net/services</href>

<method>GET</method>

<rel>urn:org.restfulobjects:rels/services</rel>

<type>application/json; profile='urn:org.restfulobjects:repr-types/list'; charset=utf-8; x-ro-element-type='System.Object'</type>

</element>

<element>

<href>http://academyrestfulapi.azurewebsites.net/menus</href>

<method>GET</method>

<rel>urn:org.restfulobjects:rels/menus</rel>

<type>application/json; profile='urn:org.restfulobjects:repr-types/list'; charset=utf-8; x-ro-element-type='System.Object'</type>

</element>

<element>

<href>http://academyrestfulapi.azurewebsites.net/version</href>

<method>GET</method>

<rel>urn:org.restfulobjects:rels/version</rel>

<type>application/json; profile='urn:org.restfulobjects:repr-types/version'; charset=utf-8</type>

</element>

</links>

</root>

XML consists of data *elements,* each surrounded by start- and end- tags, such as <method>GET</method> and <rel>self</rel> above. Elements may contain other elements, so that the whole structure forms a hierarchy.

The basic data elements consist of *name/value* pairs, such as method: "GET" and rel: "self" above. These are organised into a hierarchy using (curly) *braces*, where the contents of each pair of braces defines an *object,* and each object can form the value part of a higher-level name/value pair. The name/value pairs *inside* any braces are said to be that object’s *properties.*

JSON and XML are both used in communication protocols, but also potentially for storing data in files.

*Most* RESTful APIs use either JSON or XML, but JSON is the more popular, and growing, as it has the following advantages over

* The total amount of text required is slightly smaller, making it *slightly* more efficient for network communication.
* Many people find JSON slightly easier on the eye, though note that this is only important to developers. An end-user of an application should never see, let alone have to read, either JSON or XML.
* JSON works well with JavaScript - indeed it stands for *JavaScript Object Notation*. JavaScript is an object-oriented programming language (OOPL). In most OOPLs every object can be converted into a string representation; in JavaScript, there is a function to convert the state of any object into a JSON representation. More significantly, because JavaScript is a dynamically-typed language, any file of JSON may be turned into a JavaScript object with a single function call. JavaScript is the most common way to run functionality in a browser, so JSON is an ideal way to send data to a JavaScript application running in a browser. But JavaScript is also increasingly used to write server-side programs also.

When browsing JSON using the JSON View plug-in, notice that it is possible to expand and collapse regions of the JSON using the plus and minus symbols. These **+,*-*** symbols do not form part of the JSON itself – and if you select some JSON to copy to the clipboard, the **+,-** symbols will not be included:



Having gotten the idea of looking at JSON, now we need to look at the content and meaning of the JSON returned by our RESTful API.

We’ll end this exercise with a new bit of terminology, though.

In a RESTful API, every URL defines a *resource*, and when we access that URL, it returns a *representation*, which in the current example is in the form of JSON.

So the JSON above is the ‘Home’ representation returned by the ‘Home’ resource.

# Following links

Looking at the Home the representation, we can see that it consists of a collection of ‘links’ - five of them in this case. Each of these links has four properties (rel, method, type, and href), which we will look at later. For the moment notice that each of the five links includes an underlined URL (in its href property), and that by clicking on this link you will be taken to a new resource, returning a new JSON representation.

From the home representation, follow this sequence of links (each one can be found within the returned representation):

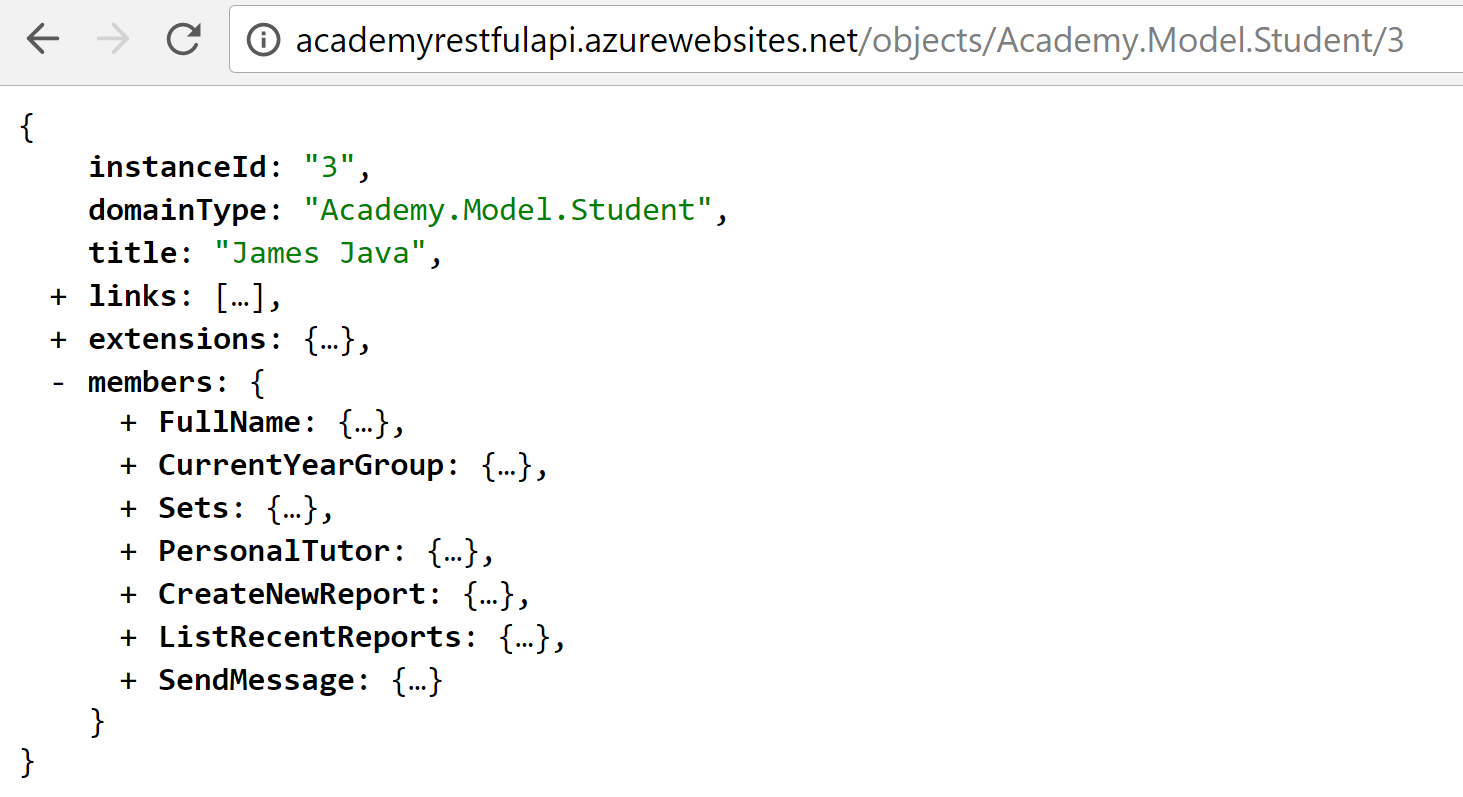
<http://academyrestfulapi.azurewebsites.net/services>

<http://academyrestfulapi.azurewebsites.net/services/Academy.Model.StudentRepository>

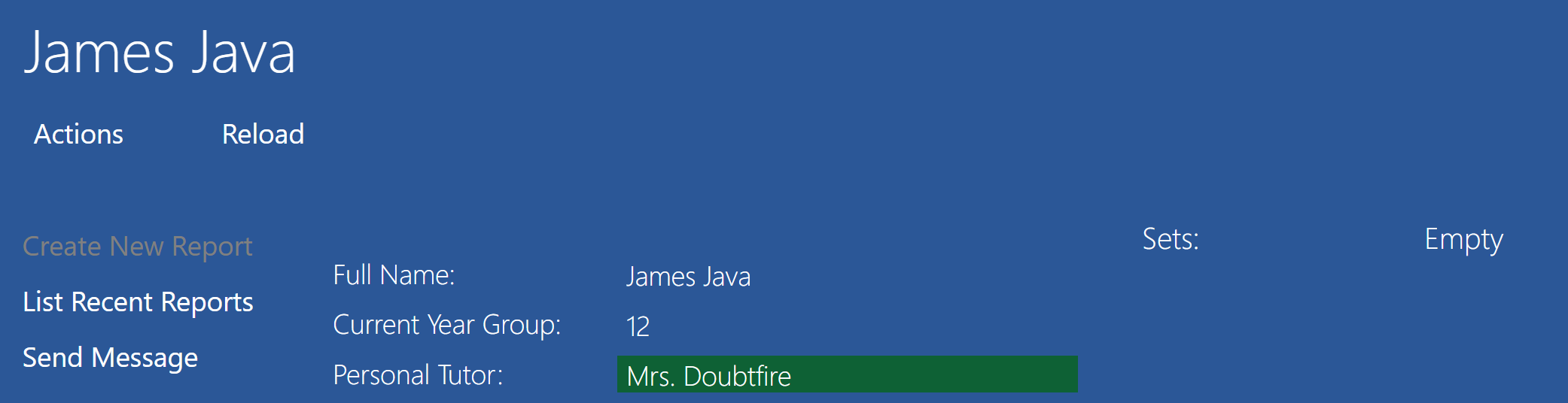
<http://academyrestfulapi.azurewebsites.net/services/Academy.Model.StudentRepository/actions/AllStudents/invoke>

<http://academyrestfulapi.azurewebsites.net/objects/Academy.Model.Student/3>

You should now be seeing the following (some elements within the JSON have been collapsed in this view and are shown with the ‘+’ icon):



Scanning the collapsed content of this JSON representation you might correctly infer that we are looking at a representation of the student James Java. The members represent the properties of that Student object and the actions that can be applied to it. These match up to the properties and actions in the user’s view:



Expand the CurrentYearGroup and PersonalTutor members as shown below:



Notice that CurrentYearGroup has a value property containing a simple numeric value: 12. PersonaTutor also has a value property, but this contains a link to another object as highlighted above. It also has a title property, Mrs. Doubtfire, which *may* be used on an application (and *is* on the application screen we looked at above) to render a user-friendly link without the user (or the JavaScript) program having to follow the link first to get hold of the linked teacher’s name.

When the *application* user clicked on the link for James Java, behind the scenes the JavaScript accessed the appropriate resource via the RESTful API. Then from the returned representation it extracted the data needed, perhaps processed that data (e.g. perhaps combining with data from other objects in memory), and then created some HTML to display in friendly form to the user.

But how did the JavaScript know how to construct the URL to call?

Look URL at the top of the browser/tab that is displaying the JSON:



The URL has a clear structure, which an informed programmer can read. The first part of the URL (academyrestfulapi.azurewebsites.net) does not change: it is propertly called the *authority*, but many people refer to it as the ‘server’ or ‘domain name’ (actually, only azurewebsites.net is the domain name). The rest of the URLs shown above consists of the *path* - and this can be considered to be like a directory path in a file-based operating system.

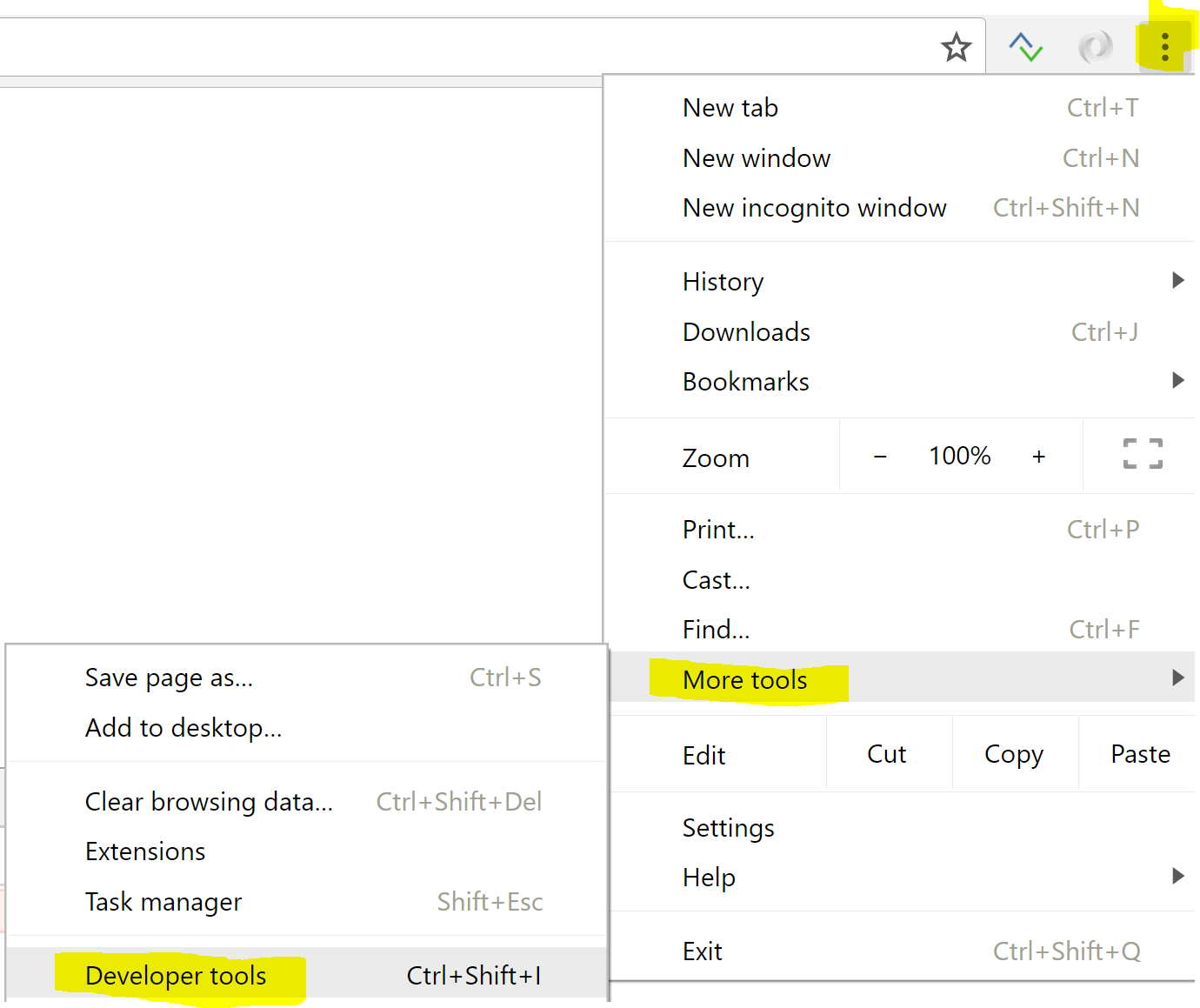
The path in the URL above specifies that the resource is an object, of the type Student, and with the Id of 3. (In the Academy application, this corresponds to the row in the Students table of the underlying database with primary key 3. This is quite a common pattern, but a RESTful API does not have to be built on top of a database). Academy.Model is just the namespace within which the Student class is defined, and Academy.Model.Student is said to be the ‘fully-qualified type name’ of the Student object - if you don’t understand exactly what that means, don’t worry: it’s not very important, just an implementation detail.

Once we are familiar with this structure we *could* navigate directly to that Student, or another one, if we know the Id for that student. And therefore a JavaScript program could construct and then call the appropriate URL if it knows the type of object and its Id.

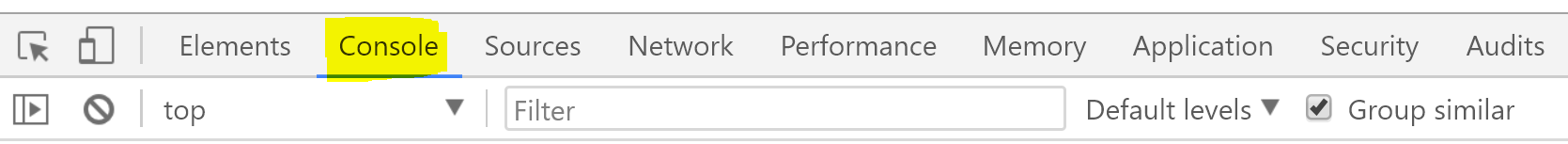
Prove this by editing the URL, then hitting **Enter**, to retrieve the details of the Student with Id of 2, and then the Teacher with Id of 4.

1. Paste in screenshots for those two objects.

What happens if you try to find a type of object that doesn’t exist (Sport, say); or you specify a valid type of object, but forget to include the fully-qualified name of the type; or if you get the type correct, but you specify an Id in the wrong format (‘three’, say); or you specify an Id in the correct format but out of range (try any number over 100, say)? Try it and you will find that you get a blank screen. This does not seem very helpful, but now open Chrome’s Developer Tools:



Then click on the **Console** tab:



Below this, you should now see an error message in red. (If not, hit the Refresh icon on the browser).

1. What is the Http error code in the message? You have probably encountered this message before, when browsing the web. What does it mean?

An important part of REST conformance, is that the API should always return the correct HTTP status code, indicating whether a request has succeeded, and, if not, indicating the nature of the error. We’ll see some other types of error in due course. Meantime you can a look up the list of HTTP status codes here: <https://en.wikipedia.org/wiki/List_of_HTTP_status_codes>

### Well-structured URLs

It is commonly stated that the URLs (resources) in a RESTful API must have well-structured *paths*, like the one shown above, making it easier both for a programmer to understand the nature of the resource, and for a program to construct the URL for a resource of interest. Indeed, most RESTful APIs do this. But actually, this idea is not definitional to REST. It is perfectly possible to create a fully REST-compliant API where the path is *opaque* (incomprehensible to human eyes), such as:

<http://academyrestfulapi.azurewebsites.net/5a2539328fd222729b959229a36e988898fddaae>

Although this is not common, some systems adopt the latter approach, in which the structure of the path has been encrypted, specifically to prevent hackers from guessing at URLs and hence exploring the database directly. (There are, however, other mechanisms to prevent that).

Much more important than having human-understandable URLs is that in a true RESTful API, every resource can be reached just *by following the links from a single home resource*, without having to understand, or even know, the structure of the URLs. Indeed that is how we got to this Student representation.

An additional rule that RESTful APIs must follow is that each request-response, initiated by following links, is completely self-contained. The interpretation of that action does not depend on the previous request that you sent, and not does any request set up an expectation that it will be followed by another request to complete an action. Another way of saying this is that REST does not permit any higher-level protocol to operate over and above HTTP. Still another way of saying it is that the server that provides the RESTful API is ‘stateless’: it does not does not remember previous requests in order to provide a context or an interpretation for them.

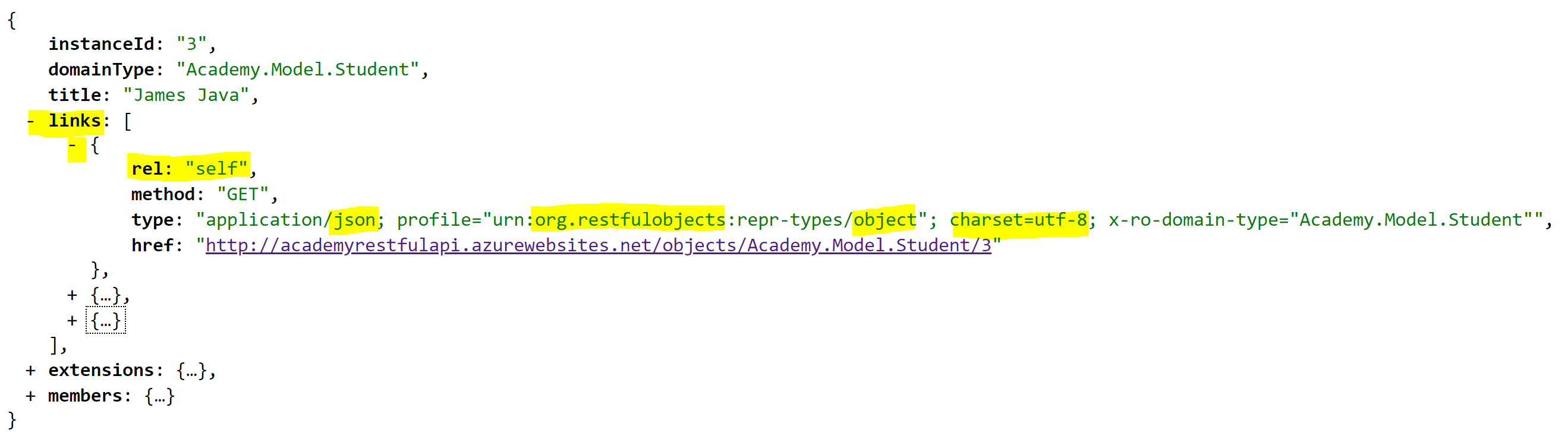
Close the **Developer Tools** pane, and return to the JSON representation of that Student number 3. Do this by following the links from the home resource as you did the first time.

Now we’ll look at the make-up of this representation: how do we know what all this means?

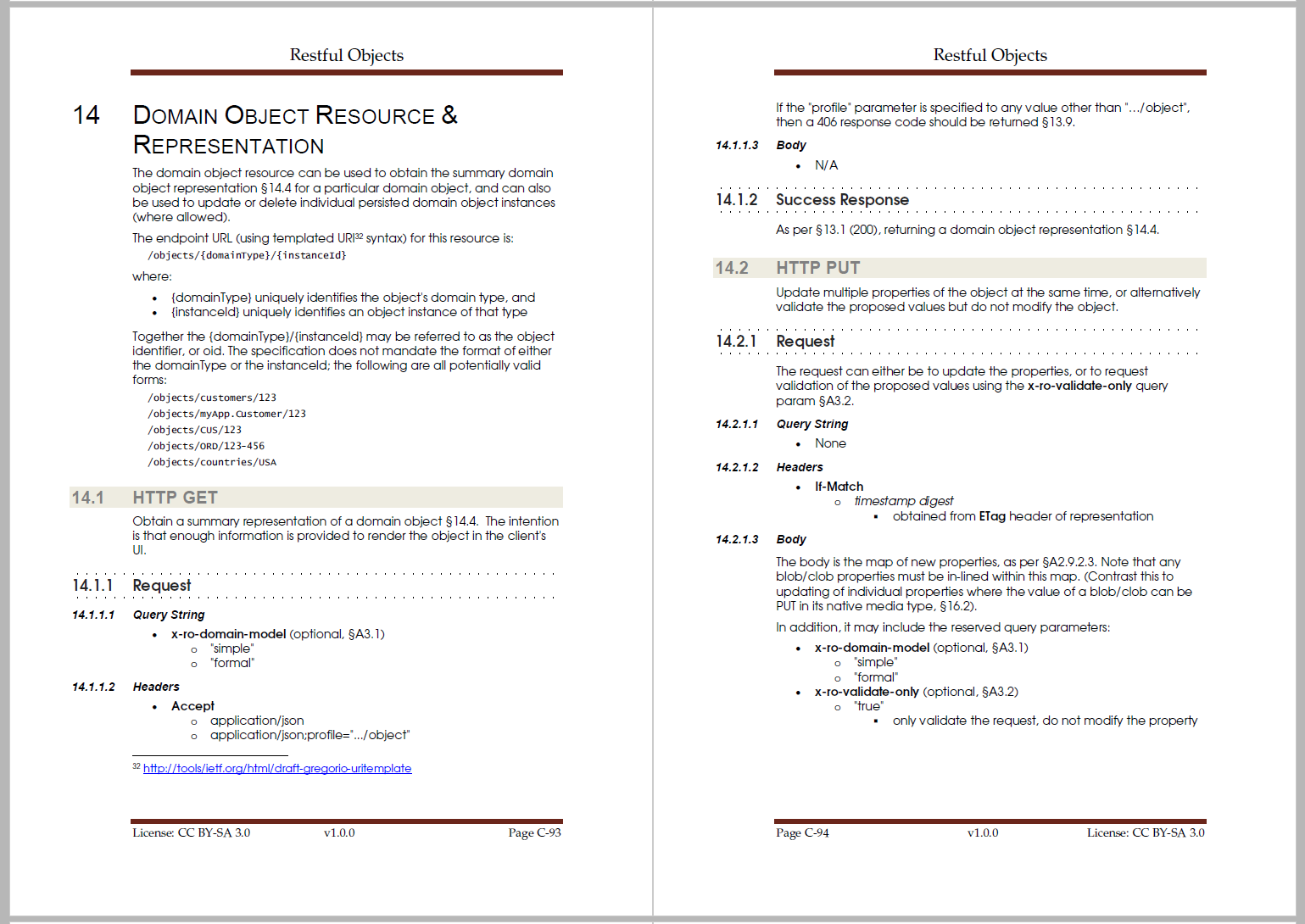
# Exploring the metadata about a link

So far we have thought of a ‘link’ as just being a URL that we can click on (or that a JavaScript program could invoke). But to be RESTful, an API needs to provide additional information *about* the link. We call data that is *about* other data: *metadata.* This metadata tells us about the relationship of the linked resource to this one, how to follow the link, and what form of representation we can expect to get back, *without having to follow the link.*

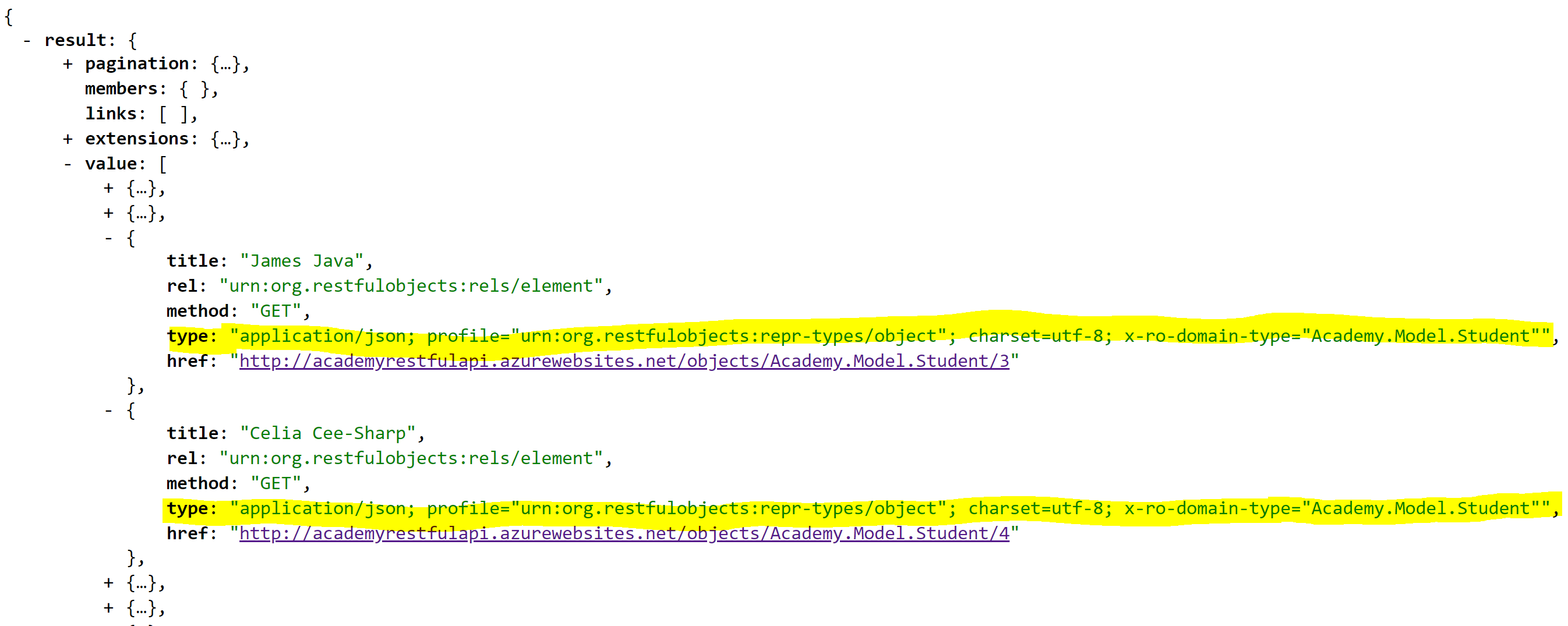
Looking at the representation of Student 3, notice has a property called links (as do all the representations, including the Home representation that we looked at first), and that in this case, it contains three links, each of which is a JSON object (it has curly braces and other properties inside it). The topmost link object looks like this:



Here rel: "self" means that this link (unlike the others) actually points to the same resource that created this representation - you can see that URL in the href property (short for *hypertext reference*). All representations in this system have a self link. This might seem redundant! But actually, it is giving us some useful information in the type property, including the fact that the representation is formatted as JSON (application/json), and that it uses the UTF-8 character set (charset=utf-8), but most importantly that the structure of this representation conforms to the *Restful Objects* standard for representing an object. It tells us that we (as developers) may look up the definition of this standard on <http://restfulobjects.org> (urn:org.restfulobjects). You don’t need to look this up now, but if you do you will find that you are downloading a 200-page public specification that defines resources and representations in detail. Here’s a snapshot of the first two pages of that specification for the ‘Domain Object Resource & Representation’:



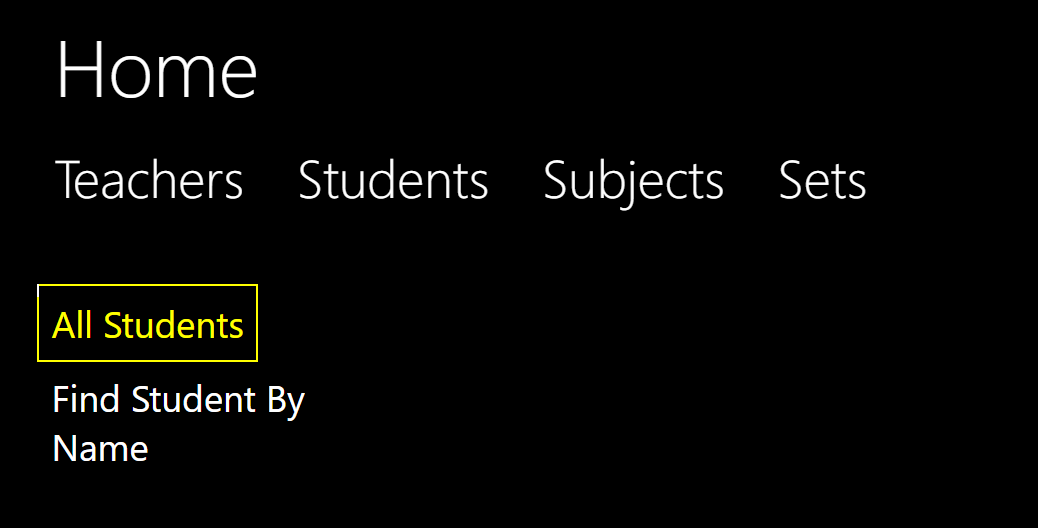
In fact, if you press the back button on the browser, to go back to the list of links to all students, you can see that that the link to Student 3, and indeed to each of the students, contains that same information:



This is important: not only can we navigate the application just by following links from the home resource, but each link provides us *in advance* with a specification for the template that the returned representation will adopt.

# Invoking an action with parameters

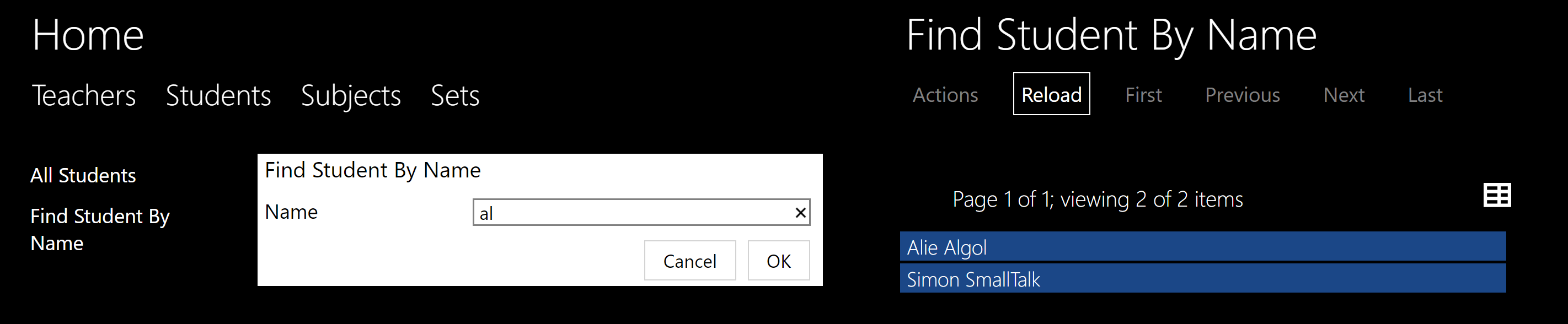
When you explored the application, you will have invoked the **All Students** action at some point:



As you’ve probably now guessed, in the background the JavaScript was accessing the following URL on the server (which you also followed directly at one point in Exercise 3):

<http://academyrestfulapi.azurewebsites.net/services/Academy.Model.StudentRepository/actions/AllStudents/invoke>

But you might also have used the **Find Student By Name** action, and specified the name or part-name in the dialog (in this case we’ve right-clicked on OK to show the results of the action in the right-hand pane):



How would a program specify that it wanted to retrieve Students matching ‘al’ via the RESTful API? More generally, how would the program pass data back into the API?

There are two ways this is done. We’ll look at the simplest one first, but be warned that this can be used only in limited circumstances. Start by following these links, starting from the home resource:

<http://academyrestfulapi.azurewebsites.net/services>

<http://academyrestfulapi.azurewebsites.net/services/Academy.Model.StudentRepository>

and then expand the FindStudentByName member:



Notice that within the extensions section there is a property: hasParams: true. This indicates to the program using this API that the FindStudentByName action will require one or more parameters to be provided. Notice, also, that the link, unlike the AllStudents action, does not end with /invoke. Following this link will not invoke the action, it will return us more details about how to invoke the action:



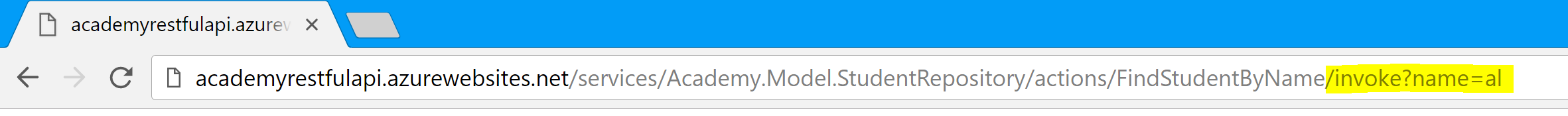
Notice the following:

* There is a section of JSON called parameters, and this contains just one property, indicating that the action takes one parameter called name.
* That parameter has a type of string, which has a maximum permitted length of 0 (meaning ‘unlimited length’ here).
* Within the links section, there is now a link with the relationship of invoke.

Open the Developer Tools > Console (as you did in Exercise 3) and then click on the invoke link above.

1. Paste in a screenshot that shows the status code, and the JSON that is returned. What is official meaning of that status code?

Now, in the URL bar of the browser, append ?name=al onto the end of the current URL:



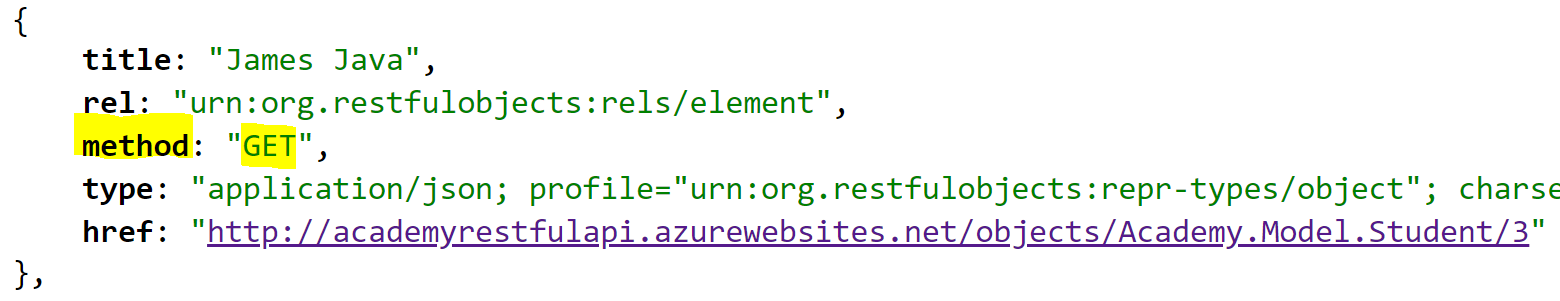
This is standard HTTP syntax: the part after the question-mark is known as the ‘query’, or the ‘query string’ because its principal use is for specifying details of a query, as we are doing here.

Hitting Enter on this extended URL should now produce the correct results.

This simple query-string syntax may only be used if the parameters we must provide are simple values such as strings, Booleans, or numbers. (You may use spaces an punctuation in the string, but these symbols must be URL encoded – see <https://www.w3schools.com/tags/ref_urlencode.asp> ).

If the parameters that you need to specify are other objects, then you must instead provide them formatter as JSON, in the form of a ‘payload’. We will look at how to do this in the next exercise, because another constraint on the use of the query-string syntax is that it may only be used when accessing a resource with the HTTP GET method. So far, every link we have followed has used the GET method, but it is now time to look at others.

# Exploring alternative HTTP methods

So far, every link that we have followed has had the value GET for its method property e.g.:

This refers to the HTTP method that must be used to access the linked resource. GET is the method that your browser *automatically* applies when a user click on any link on the web - and means that the resource is just providing the client with data, it is not changing anything on the server. It is ‘read only’, if you like. But GET is only one of several methods used within HTTP. The most commonly-used of the other methods are PUT, POST, and DELETE.

When JavaScript or other code creates an HTTP request within a program (i.e. one not directly initiated by the user clicking on a link), it must specify which of the HTTP methods it is using for that request, and this must match the method (or, sometimes, one of several permissible methods) that the resource is expecting. So having it clearly defined in the information *about* the link is important if the JavaScript application is just going to follow links in the RESTful API.

Now it is commonly observed that there is a connection between those four most common HTTP methods, the general database concepts of Create, Read, Update and Delete (collectively known as ‘CRUD’), and specific SQL keyword, as shown in this table:

|  |  |  |
| --- | --- | --- |
| **CRUD concept** | **HTTP method** | **SQL keyword** |
| Create | POST | INSERT |
| Read | GET | SELECT |
| Update | PUT | UPDATE |
| Delete | DELETE | DELETE |

However, while there is a conceptual correspondence here, they are not directly equivalent. The SQL keyword are actual instructions in their own right, whereas the HTTP method is not itself executed. Rather the HTTP method provides a piece of information *about* an instruction – the instruction being the call to a specific URL. Thus you can say that the HTTP *method* is a piece of *metadata* about the instruction being called.

The authors of HTTP were effectively saying: in a large, complex, publicly accessible system, we need to *warn* users, or programs using the resources, of the possible consequences of following any given link, by providing them with a simple set of classifications:

* Links invoked with a GET method retrieve data but do not alter the state of anything on the server. Such links are said to be ‘side effect free’. (If you are familiar with the *Functional Programming* paradigm, you will have encountered this phrase before).
* Links invoked with a PUT method will change the state of the server, possibly by updating data on a database. However, making an identical PUT request repeatedly has the same effect as making the request it once. For example, if a particular request is to ‘change the Date Of Birth to 17th June 2001’ - it does not matter how many times we call it. In mathematics, functions that behave this way are said to be *idempotent*.
* Links invoked with a POST method will, like a PUT, change the state of the server, but making an identical request using POST repeatedly, may result in multiple changes - for example if the request is to ‘Increase the score by 1 goal’. This is why, when you are browsing the web, especially when using an online-shopping system, and you hit the browser’s Back or Refresh button, you sometimes see a pop-up dialog box warning you that you are about to submit a form again. It means that the previous request used the POST method, and you might end up accidentally ordering the same item twice, for example.
* Links invoked with a DELETE method will delete a resource: after following this link, the resource will no longer be there for you to follow again. (Some systems use it to mean just that ‘requesting this resource may result in *some data* being deleted, but the correct meaning is that the resource itself will be deleted).

Let’s find a link that doesn’t use the GET method. Explore the members property of Student 3 (or any student) and expand the member representing the SendMessage action:



Now looking at the SendMessage action you it has a method of GET, but remember that this is not the link that *executes* the action - it is a link to a resource that provides details of the action. So follow that link to view the action representation:



Now we can see that this representation contains a link where the relationship is defined as invoke, and that the HTTP method for this ‘action invoke’ link is indeed POST.

1. Try clicking on the invoke link. What HTTP status code do you get back? What is the meaning of this HTTP status code (look it up on the web)?

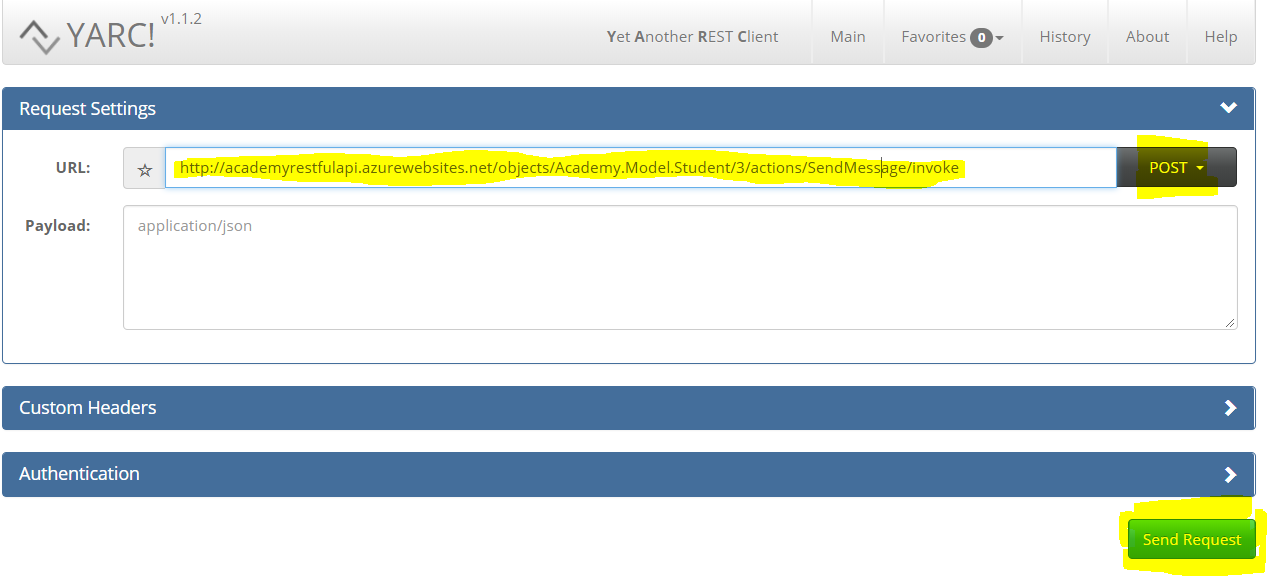
Remember that when you just click on a link, your browser is setting the method to GET automatically, and this method isn’t accepted by this resource. When the JavaScript application makes the request it will have read the fact that the link requires a POST and specified that method when making the request. How can we do the same from the browser? We now need to make use of the ‘REST Client’ plug in.

# Using the REST Client plug in

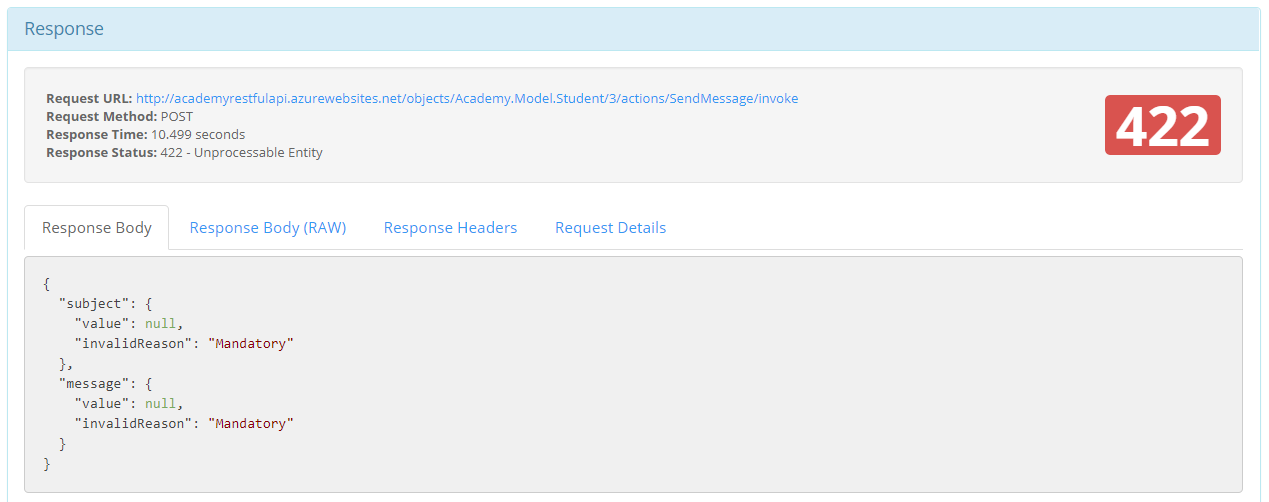
First, make a copy of the URL for the action invoke link:

<http://academyrestfulapi.azurewebsites.net/objects/Academy.Model.Student/3/actions/SendMessage/invoke>

Then open the REST client extension (‘YARC - Yet Another REST Client’) - via the  icon - which should open the tool in another tab. Paste in the URL, set the method to POST, and click **Send Request**.



1. What response do you get back - paste in a screenshot showing just the response.

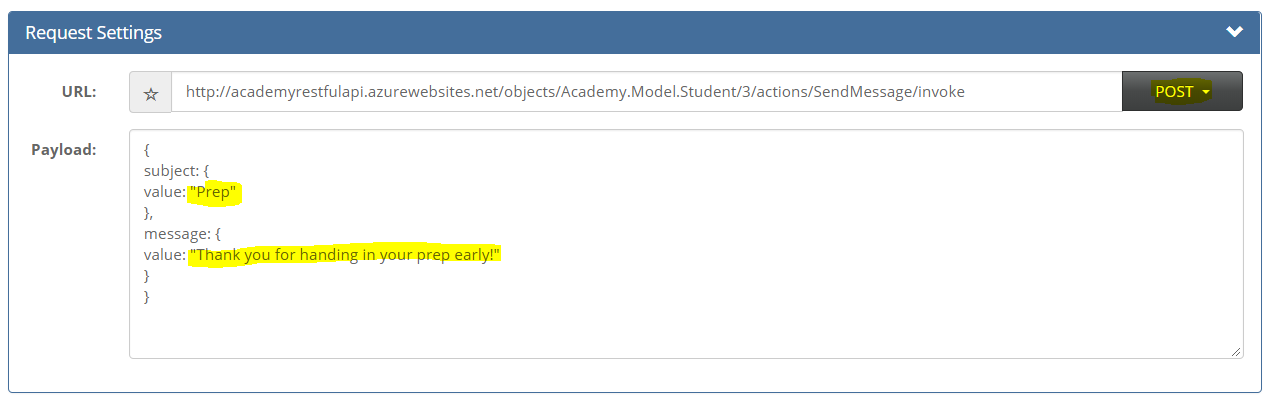


The response is telling us, as we saw in the previous exercice, that the request could we did not provide the values for the subject and message parameters, both of which are mandatory. (Compare this to invoking the **Send Message** action via the user interface). For a POST action the parameters must be specified as a ‘payload’, formatted as JSON. The YARC plug-in provides a field for specifying the payload, but, as we can see in the screenshot above, we left that field blank in the request. But how do we, or more realistically, how would the JavaScript application using this RESTful API know what payload it had to provide, and in what form? The answer is that the action details representation provided all that information, if we expand the arguments property of the invoke-action link:



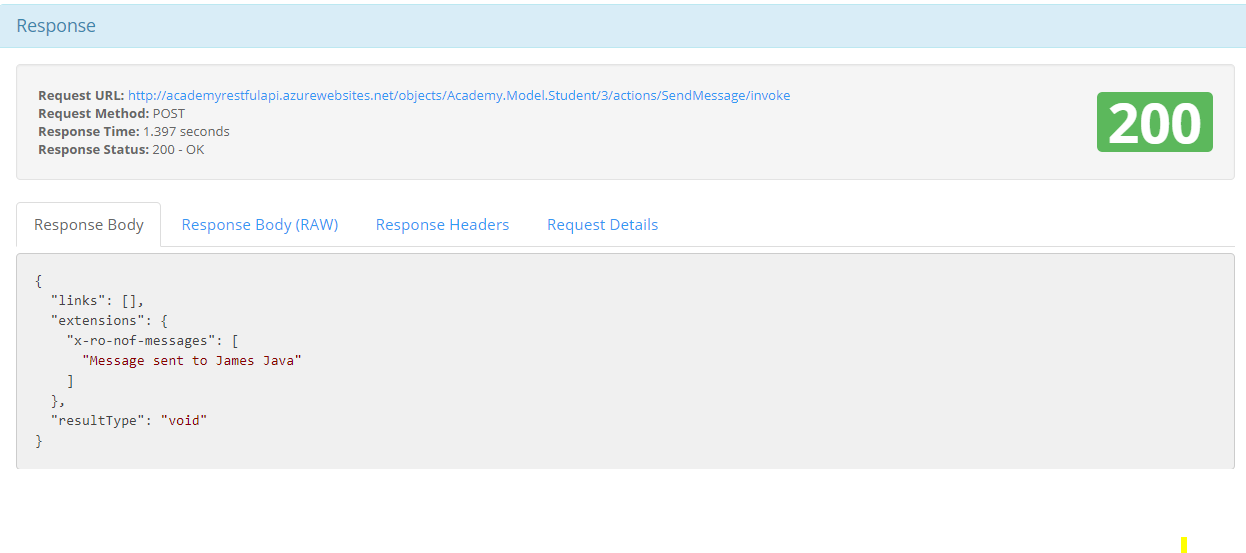
The link for the action invoke includes a blank template for the actual JSON format of the payload, as highlighted above:

Copy and paste that (highlighted) blank template into the payload field on YARC and then edit it to specify the subject and message (in quotes) e.g.:



Make sure that you have the correct number of open and close braces. (Line breaks are not important: you could format this all on one line). Then hit submit. If you did everything correctly, you should get back a successful response - status code 200.

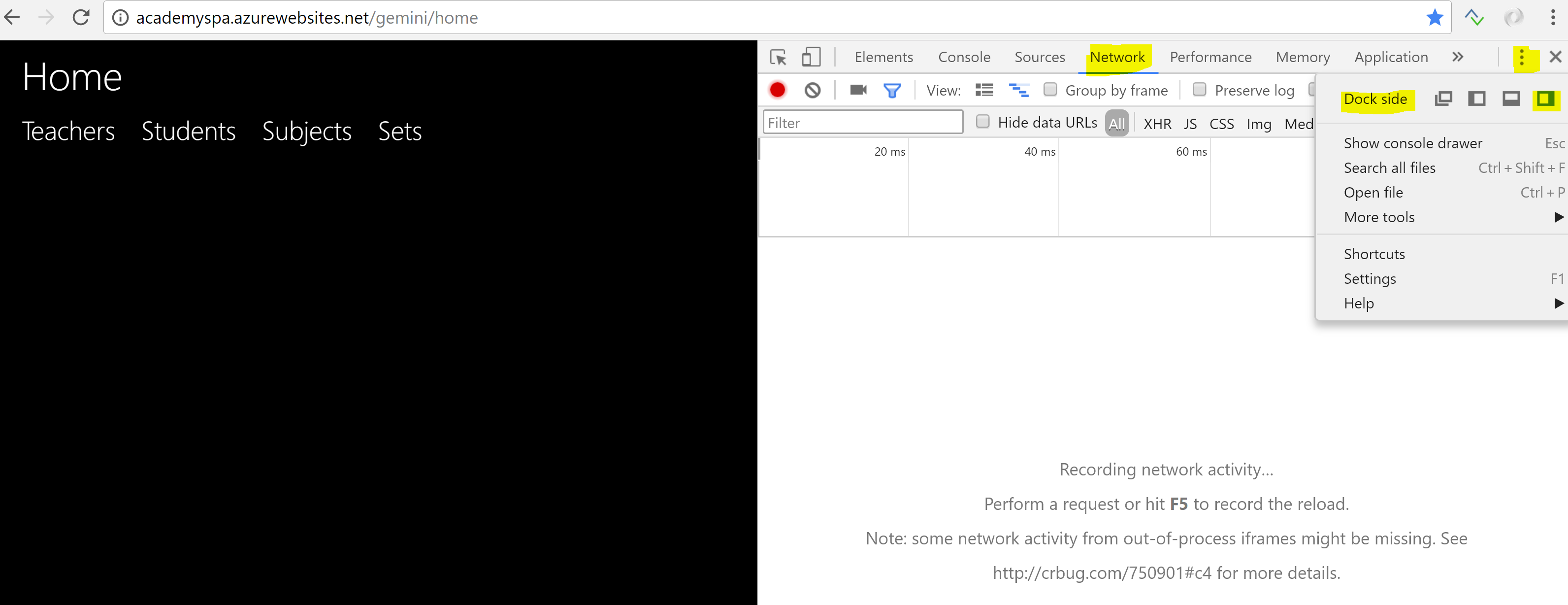
1. What information is returned in the ‘body’ of the response. How does this correspond to what you see on the user interface (the application) when you use the Send Message action



So, as well as using a POST method, we’ve now seen how to include a ‘payload’ - containing parameters to be passed to an action, or (not seen) property values when modifying an existing object.

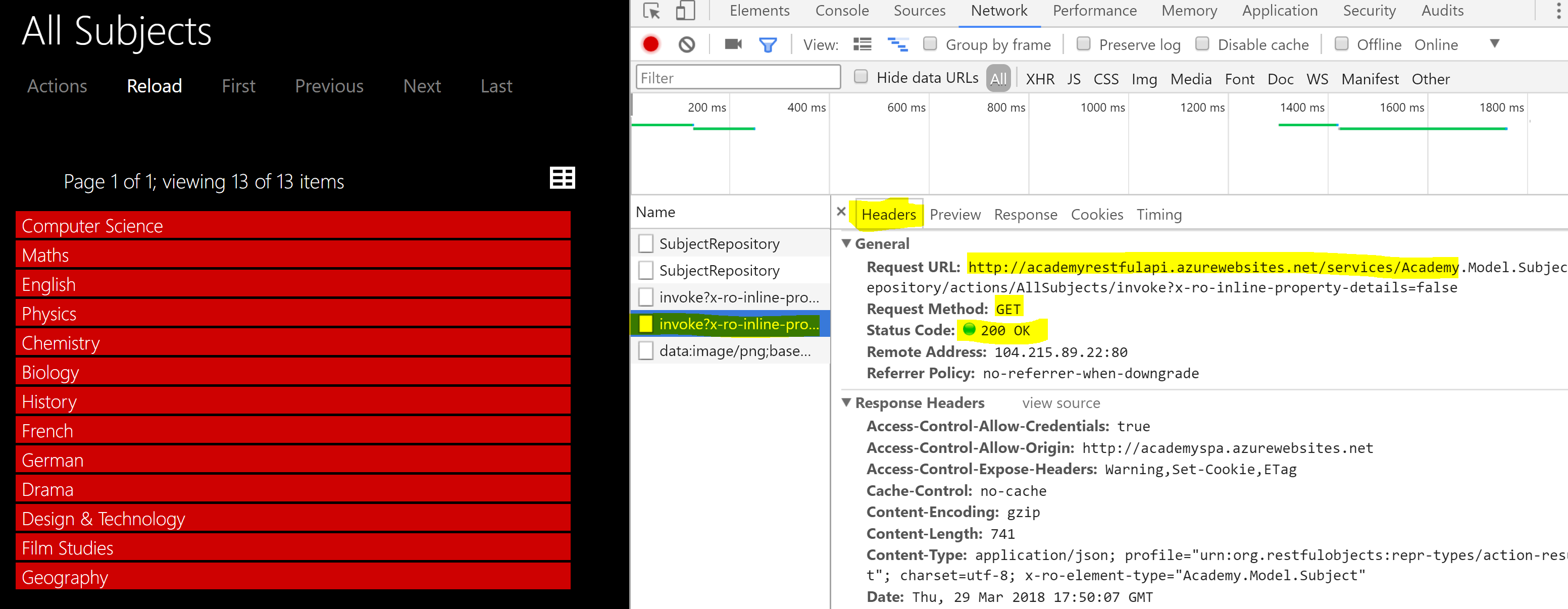
# Exploring how the application communicates with the RESTful API

Go back to the user’s application URL: <http://academyspa.azurewebsites.net> . Then open Chrome’s **Developer tools** and click on the **Network** tab. (To position the tools to the side of the main screen, select the **Dock side** from the **Customize** menu as shown).



Now start to use the application, and notice that in the Network pane you see a build-up of entries, each corresponding to a single request-response over the network. Some user actions will generate *multiple* such entries; others will add no new entries because the user’s action could be completed entirely by local processing within the browser.

Clicking on one of those entries, and on the **Headers** tab, you can see more details of the request, as in the example below:



Highlighted areas in the Headers tab show the **URL** (resource) being requested from the RESTful API, the **Request Method**, and the resulting **Status Code**.

If you switch from the Headers tab to the **Response** tab, you can view the JSON Representation returned in each case, though, sadly, Chrome Developer Tools formats this JSON as a single line - very long in some cases!

# Test questions

1. What might a RESTful API be used for?
2. A typical RESTful API defines a large number of resources - what form do they take? And what does each resource return?
3. What are the similarities and differences between JSON and XML?
4. A truly RESTful API may be navigated entirely by following links from a home resource. Each link will include a URL for the resource, but what other information will it need to provide?
5. Almost all RESTful APIs operate over HTTP, but not all APIs that operate over HTTP are RESTful. What would make an API that operates over HTTP RESTful?
6. A RESTful API operating over HTTP must make correct use of the HTTP methods, of which the most common are GET, PUT, POST and DELETE. How do they differ? In what ways are they similar to, and different from, the SQL keywords SELECT, UPDATE, INSERT and DELETE.
7. Name two ways in which parameter values may be sent over HTTP by the requestor to the server, specifying any restrictions on how they may be used.
8. A RESTful API operating over HTTP must make correct use of HTTP status codes. List three status codes and describe, in your own words, what each means.