Course:How to use Linked Data/Building Linked Data Applications/Using Web APIs

0.1 5.16 HTTP communication

If a Linked Data application is built on the web then it may use Web APIs to either provide data or consume data from other sources. The HTTP protocol is the fundamental technology on which Web APIs are built. The serving of documents on the Web (for example the serving of an HTML page to a web browser) is carried out using the HTTP protocol.

HTTP communication is based on an interaction that involves a series of requests and responses. A client sends a request to a server. The server sends back a response to the client.

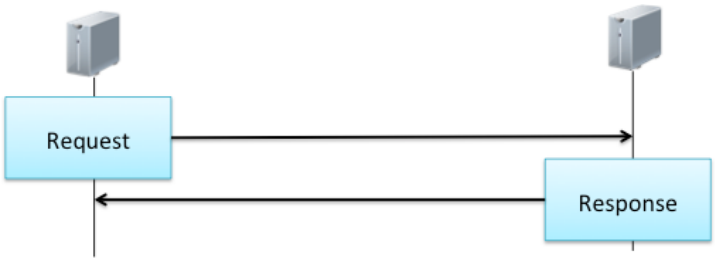

Figure 34: Request-Response interaction.

0.1.1 5.16.1 HTTP Request

Each HTTP request contains a method, URI, Header and optionally a body. The *method* indicates the type of the request that the server should perform. The most familiar types of HTTP request are *GET* and *POST*. A GET request means the client wants to retrieve content. The URI sent with the GET request is the resource from which the content should be retrieved. A POST request is used to send data. The accompanying URI indicates where the data should be sent. Web forms generally use POST requests to send data to the server.

The other types of HTTP request are used more broadly in web-based client-server communication but not necessarily important when using a web-browser to retrieve and send content. A *PUT* request to used to store data at the specified URI. A *DELETE* request is used to delete the specified URI.

Other types of HTTP request include HEAD, TRACE, CONNECT, OPTIONS and PATCH. For example, a *HEAD* request is used to retrieve header information.

A *TRACE* request is used to allow the client to see what the server is receiving. This is generally used for diagnostics.

As well as the method and URI a HTTP request contains *header* information that gives additional detail about the HTTP request. A *body* is required with POST and PUT methods and expresses the data that is to be sent to, or stored at the specified URI. For example, if you submit a web form, then the data entered into the form can be represented in the body of the HTTP request.

0.1.2 5.16.2 HTTP Response

A response to a HTTP request contains a numerical response code, a header and optionally a body. The *response code* gives overall status information on how the request is being handled. The response code is a three digit number beginning with a 1, 2, 3, 4 or 5. Response codes beginning with 1 are provisional responses indicating that the request has been received and is being acted upon.

Requests starting with 2 indicate that the request has been successfully received, understood and accepted by the server. Codes beginning with 3 indicate that further action needs to be taken by the client that issued the initial request. Codes beginning with 4 indicate that the request is erroneous and cannot be met by the server. The most commonly seen response code beginning with 4 is the 404 response code. The 404 response code informs the client that the request was erroneous because it asked for a resource that does not exist on the server. Finally, codes beginning with 5 indicate that there is an error, but in this case the problem is with the server and it is unable to fulfil the valid request.

0.1.3 5.16.3 HTTP Request Response pattern

We can now put the request and response together to illustrate the HTTP request-response pattern, in which a client requests and then receives web page. The client issues a GET request with the URI for the Wikipedia page about The Beatles. This tells the server that the client wants to retrieve information from the provided URI. The final response from the server has a response code 200 (indicating that the request has succeeded) and returns the HTML page about The Beatles.

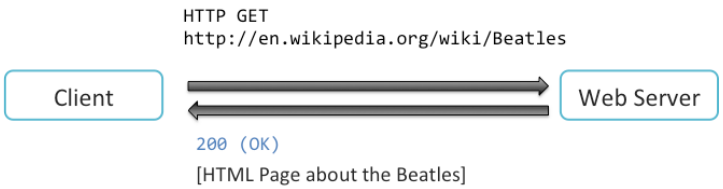

Figure 35: The HTTP request-response pattern.

0.1.4 5.16.4 HTTP content negotiation

When requesting data in a particular media from DBpedia rather than Wikipedia, the communication pattern between client and server is more complex. In this

case the URI refers to the concept of The Beatles rather than any particular document describing The Beatles. As shown below, the request uses the method GET and declares that a response is required in HTML format. The URI refers to the concept of The Beatles, which is not a resource in HTML format. The server responds with a code of 303 and another URI. This tells the client to instead make a request for HTML from this alternative URI. The client then makes a second request (not shown) for HTML using the new URI and receives a HTML page about The Beatles with a response code of 200.

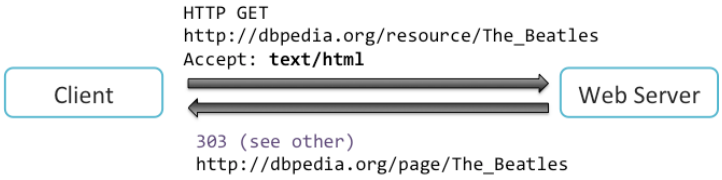

Figure 36: Content negotiation when requesting HTML using a Linked Data URI.

In the above HTTP conversation the client requested information in a certain format using a particular URI. The server responded with an alternative URI. The client then made a second request with the new URI. The server responded with the requested information. This conversation between client and server to determine the correct resource is called *content negotiation* and is often abbreviated to *conneg*.

The reason this conversation to determine the appropriate content is required is that different types of content can potentially be returned associated with the same resource. If information is being accessed via a web browser, then HTML is likely to be the preferred format. If the client is a Linked Data application consuming data, then the requested information about The Beatles will be preferred in an RDF format such as Turtle (see chapter 1 for more information about the Turtle format). In the figure below we have a GET request using the same URI as in the HTML example above. However, this time the client requests text/turtle rather than text/html. The server responds with the status code 303 (i.e. see other) and a URI where the information can be accessed in Turtle format.

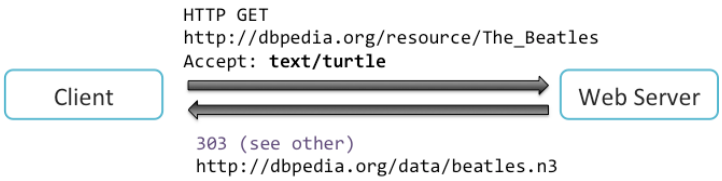

Figure 37: Requesting data in Turtle format

The client then issues a second request and this time retrieves the data in Turtle format along with a status code of 200. This approach is routinely used to publish Linked Data, in which a series of URIs name concepts used in the Linked Data application, such as The Beatles and Paul McCartney. Requests for data to a Linked Data URI are directed to another URI depending on the data format requested. Multiple RDF formats of the same data may be made available such as RDF/XML and Turtle.

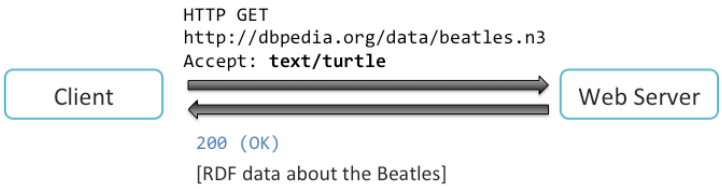

Figure 38: Retrieving data in Turtle format.

0.2 5.17 Web APIs

The HTTP conversations we have seen above provide the foundation for Web APIs. Web APIs are particularly important when a Linked Data application needs access to data that is being dynamically created. If a Linked Data application is using data about music artists and releases from the 1960s then few updates to the stored data will be required. If dynamically changing data is being used such as current weather conditions or traffic levels, then web APIs can be used to provide access to new data. A Web API can provide a range of functionalities, giving access to views on that data and also transforming the data in ways of use to other applications.

Over the past few years there has been a huge growth in the number of Web APIs, though most of these are not Linked Data Web APIs. The most common form of API is the REST (Representational State Transfer) API. REST is an architectural style that uses the HTTP protocol for communication between client and server. The Programmable Web is a general directory of Web APIs. This allows providers to register their API and other application developers to search for available APIs. The vast majority of Web APIs registered with the Programmable Web use the REST model.

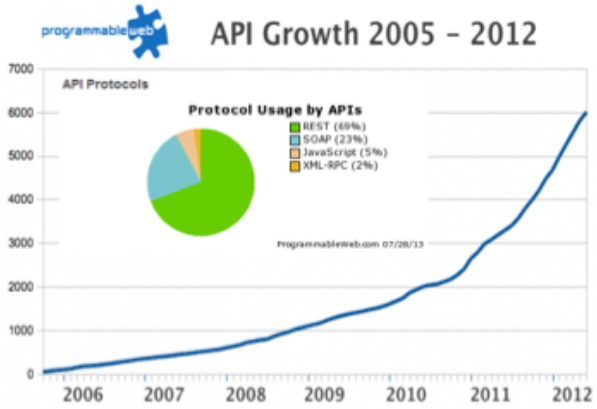

Figure 39. Growth in WEP APIs [28].

0.3 5.18 Richardson Maturity model for Web Services

The Richardson Maturity Model provides a way of thinking about the main elements that make up a REST architecture. The model is divided into a number of layers, each layer being a precondition of the one above. All of these layers can be

seen as a necessary requirement of a REST architecture. Starting from the lowest level, we have resources and their URIs. For example, as we saw above, The Beatles resource is identified by the URI http://dbpedia.org/resource/The_Beatles. The second level is HTTP verbs. These are essentially the HTTP methods (such as GET and POST) we saw earlier. These define actions to be carried out such as sending or retrieving data. The third level is known by the acronym HATEOAS (Hypertext As The Engine Of Application State). This describes the Hypermedia controls, in other words, the higher-level functions provided by the Web API such as inspecting and modifying the music releases associated with a music artist.

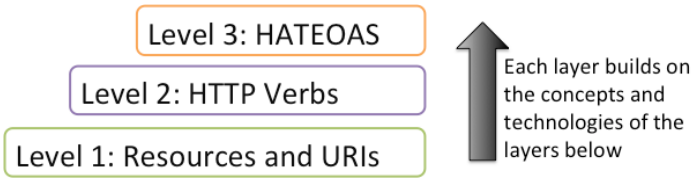

Figure 40: The Richardson Maturity Model [29]

On the lower Resource level, the Web API just makes available URIs that identify resources. As we saw in the previous section, a Linked Data URI may direct the client to alternative representations of that resource in, for example, Turtle or RDF/XML. On the second level, the HTTP verbs are methods that can act on those resources. Different methods can be used such as GET, POST and DELETE. The METHOD option is used by a client to get information about the types of request currently available. The server responds requests with an appropriate code such as 200 (OK), 303 (see other) or 404 (not found). The methods and their associated response codes give us a standardized form of communication in terms of the HTTP protocol. This therefore defines the different types of request that a client can make and the ways in which a server can respond to those requests.

HTTP verbs can be characterised as to whether they are safe and whether they are idempotent. A verb is characterised as safe if it cannot change the resource addressed on the server. For example, GET is safe because it merely retrieves information from the resource. It does not attempt to modify the resource. A HTTP verb is characterised as idempotent if the effect of sending one request is identical to sending multiple identical requests. The method DELETE is idempotent. Sending a single request to DELETE a resource will remove it form the server. Sending the DELETE request multiple times results in the same state. The resource stays deleted no matter how many times the request is sent. The only difference when sending multiple DELETE requests is that once the resource has ben deleted the server will respond with the code 404 (not found) rather than 200 (OK) as the resource is no longer available for deletion.

HTTP Verb	Effect	Characteristic
GET	retrieve the representation of a resource identified with a URI	safe, idempotent
PUT	create or overwrite a resource identified by a client-generated URI	idempotent
POST	create a resource identified by a server-generated URI	
DELETE	delete a resource (or its representation) identified with a URI	idempotent
OPTIONS	request for information about the available communication options	safe, idempotent

Figure 41: HTTP verbs

On the third level, HATEAOS describes how we use resources to drive the application through a series of dynamic states. For example, a client may send an order for a music album to a Web API. In this case the request will need to identify the album to be purchased, such as Revolver by The Beatles. The Web API creates a new resource to identify the order and sends a response to the client. The response indicates the resource identifying the order and the options available to the client. In this case, the server indicates that the order is awaiting payment and the price to be paid. The client could then respond by sending payment details such as a credit card number of the resource created for the order. If payment was accepted, then the Web API may trigger a physical shipment of the album or send the customer details of where a digital copy of the music can be accessed.

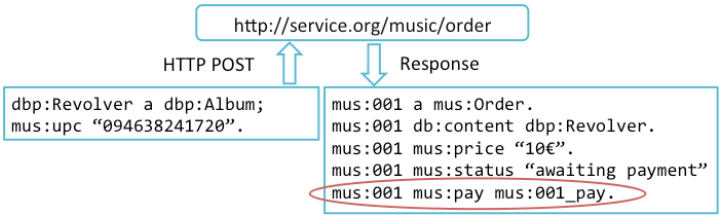

Figure 42: Transitions through states in a music ordering process.

1 Text and image sources, contributors, and licenses

1.1 Text

- **Course:How to use Linked Data/Building Linked Data Applications/Using Web APIs** *Source:* https://en.wikitolearn.org/Course%3AHow_to_use_Linked_Data/Building_Linked_Data_Applications/Using_Web_APIs?oldid=4596 *Contributors:* WikiToBot and Move page script

1.2 Images

- **File:Howtouselinkeddata_chapter5_image067.png** *Source:* http://pool.wikitolearn.org/images/pool/2/2b/Howtouselinkeddata_chapter5_image067.png *License:* ? *Contributors:* ? *Original artist:* ?
- **File:Howtouselinkeddata_chapter5_image069.png** *Source:* http://pool.wikitolearn.org/images/pool/f/f1/Howtouselinkeddata_chapter5_image069.png *License:* ? *Contributors:* ? *Original artist:* ?
- **File:Howtouselinkeddata_chapter5_image071.png** *Source:* http://pool.wikitolearn.org/images/pool/9/96/Howtouselinkeddata_chapter5_image071.png *License:* ? *Contributors:* ? *Original artist:* ?
- **File:Howtouselinkeddata_chapter5_image073.png** *Source:* http://pool.wikitolearn.org/images/pool/1/15/Howtouselinkeddata_chapter5_image073.png *License:* ? *Contributors:* ? *Original artist:* ?
- **File:Howtouselinkeddata_chapter5_image075.png** *Source:* http://pool.wikitolearn.org/images/pool/4/44/Howtouselinkeddata_chapter5_image075.png *License:* ? *Contributors:* ? *Original artist:* ?
- **File:Howtouselinkeddata_chapter5_image077.png** *Source:* http://pool.wikitolearn.org/images/pool/e/ec/Howtouselinkeddata_chapter5_image077.png *License:* ? *Contributors:* ? *Original artist:* ?
- **File:Howtouselinkeddata_chapter5_image079.png** *Source:* http://pool.wikitolearn.org/images/pool/0/0e/Howtouselinkeddata_chapter5_image079.png *License:* ? *Contributors:* ? *Original artist:* ?
- **File:Howtouselinkeddata_chapter5_image081.png** *Source:* http://pool.wikitolearn.org/images/pool/2/2e/Howtouselinkeddata_chapter5_image081.png *License:* ? *Contributors:* ? *Original artist:* ?
- **File:Howtouselinkeddata_chapter5_image083.png** *Source:* http://pool.wikitolearn.org/images/pool/d/d6/Howtouselinkeddata_chapter5_image083.png *License:* ? *Contributors:* ? *Original artist:* ?

1.3 Content license

- [Project:Copyright Creative Commons Attribution Share Alike 3.0 & GNU FDL]
- Creative Commons Attribution-Share Alike 3.0