Web Services, HTTP and REST

Reading / References

- Restful Web Services[1], chapters 1-4, especially chapter 4. The appendices contain information about HTTP.
- HTTP on MDN[2]
- Browser Networking[3] free online book, optional
- HTTP specification (more a reference than a read)[4]
- HTTP the definitive guide[5] not free, not required

Practice questions on the reading

- What kind of information can we send over the HTTP protocol?
- What are the basic parts of a HTTP request and an HTTP response?
- List 5 important HTTP Response Codes and their meaning.
- List 8 important HTTP headers and their meaning. At least 3 headers should be request headers, and at least 3 should be response headers.
- What is the *method information*? What are different ways it may be specified? What are their advantages/limitations? What are the drawbacks of using a system other than the HTTP verbs to communicate method information?
- What is the *scoping information*? What are various ways it can be communicated?
- Does the HTTP protocol specify what can go into the body part of requests or responses?
- How does REST differ from other web service approaches? What characterizes RESTful design?
- Describe what Amazon's S3 is and what its main components and terms are. What are the URI/URL schemes one would use to access S3 resources?

Notes

We will discuss Web Services and related concepts in this section. We start with some key terminology:

Web Service A service provided over the internet via HTTP, and usually targetted at automatic consumption via other computers or programs. In that way it differs from a personal or company website.

Web API API in general stands for Application Program Interface. In this particular instance it describes how the Web Service expects clients to interact with it.

[1] http://learning.acm.org/books/book_detail.cfm?id=1406352&type=safari
[2] https://developer.mozilla.org/en-US/docs/Web/HTTP
[3] https://hpbn.co/
[4] http://www.w3.org/Protocols/rfc2616/rfc2616.html
[5] http://shop.oreilly.com/product/9781565925090.do

Since its primary clients are computers themselves, the API needs to be formally described.

HTTP A simple protocol for transfer of data over an internet. HTTP establishes a simple way to describe resources and how to request and receive them, and we will discuss this extensively in the future.

REST A particular way of describing and serving the resources via HTTP that uses some of the strengths of HTTP. Fairly accepted nowadays as a standard way to build a web service. Almost every programming language offers a framework that implements REST, and many web services follow a RESTful API.

HTTP

HTTP stands for HyperText Transfer Protocol. It is what is known as an "Application Layer Networking protocol". This means that it does not concern itself with how network packets will traverse the internet, but instead aims at a higher-level description of the information in those packets. It is the primary mode of communication between web browsers and web servers. Every time you request a webpage, HTTP is the protocol that arranges for that webpage to find its way to you.

There are some key characteristics that determine the behavior of HTTP:

addressable Every resource you may want to access has a corresponding Uniform Resource Identifier (URI), what we all affectionately know as a URL. A simple, clear, uniquely identifiable access point for the resource.

stateless The protocol is "stateless". In other words, each request that the client sends to the server has no memory of prior requests and replies.

This is an important feature of the protocol, and something that actually lent to its popularity. It makes the implementation of it on both the server and client side easier, and makes the overall protocol simpler, as neither side needs to maintain any information from previous requests. Other protocols had been proposed around the same time, but HTTP won in the end as the standard, in large part due to its simplicity.

One of the consequences of course is that in situations where we need to maintain the history of what has occured, both browser and server need to agree on a way to do that (e.g. keeping someone "logged in").

client/server The HTTP protocol is characterized by the uneven description of the two parties involved. One party is the *client* and the other is the *server*. The client sends **requests** to the server, and the server sends back **responses**.

session The typical interaction between client and server, called a *session* would go something like this:

1. Client establishes a network connection with server. This is typically done via TCP/IP and requires some amount of initial setup.
2. Client sends a request packet and waits for the answer.

3. Server receives the packet, then prepares and sends a response.
4. In HTTP 1.1 and later, the client may send further requests and receive responses.
5. When the client has no more requests, they close the connection.

Clients and servers specify themselves via their IP addresses (and port numbers). This is taken care of at the TCP/IP layer.

request The client sends an "HTTP request" to the server over the network. That request includes the **request method**, followed by the **resource path** as well as the protocol version, typically HTTP/1.1.

It will be followed a series of **headers**, that can identify various aspects of the request, like the content type, the host name, the content length, the accepted languages for the reply and so on.

Some request methods allow content, which can be found following the headers. Here is an example from the MDN page:

```
POST /contact_form.php HTTP/1.1
Host: developer.mozilla.org
Content-Length: 64
Content-Type: application/x-www-form-urlencoded

name=Joe%20User&request=Send%20me%20one%20of%20your%20catalogue
```

This says that the request uses the POST method and the resource path is `/contact_form.php`. There are 3 headers, one specifying the host, and two more specifying the details about the content. After that and a following empty line we find the content (`name...`).

Here are the main request methods. Think of these as "function calls". A web browser can typically only use the first two, but the client of a web service could use more.

GET A GET request asks for some resource and is not meant to cause any changes to the server.

POST A POST request is used when submitting forms for example. It is typically expected to be used for updating some server information.

PUT Used for changing information of some server resource. Often performed via a POST instead, but that violates the REST principles we will discuss later.

HEAD The reply will contain just the header information, without any content body. It is useful for the client to know what kind of headers to provide along with the "true" request.

DELETE Used for deleting server resources (if allowed).

response Server responses have a special first line containing the protocol followed by the **response status code**. This is followed by response headers, and finally the content body of the response. Here is an example:

```
HTTP/1.1 200 OK
Date: Sat, 09 Oct 2010 14:28:02 GMT
Server: Apache
Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT
ETag: "51142bc1-7449-479b075b2891b"
Accept-Ranges: bytes
Content-Length: 29769
Content-Type: text/html

<!DOCTYPE html... (here comes the 29769 bytes of the requested web page)
```

There are many different request and response headers, look at the documentation for more information.

Here are some typical response status codes, there are many more:

200 *OK*. The resource was available and is returned.
301 *Moved Permanently*. The resource has been moved to a different location. A location header gives the new location.
304 *Not Modified*. The resource hasn't been modified since the last time we asked for it. Our cached version will do just fine.
404 *Not Found*. The requested resource cannot be found. Typically the result of typos in the request.
500 *Internal Server Error*. Usually indicates problems with the server's configuration/availability.

You typically don't have to directly create these requests and responses as text, there are libraries that do that for you and allow you to talk about these responses on the level of objects.

REST

> The Internet and the Web did not have to exist. They come to us courtesy of misallocated defense money, skunkworks engineering projects, worse-is-better engineering practices, big science, naive liberal idealism, cranky libertarian politics, techno-fetishism, and the sweat and capital of programmers and investors who thought they'd found an easy way to strike it rich.
>
> The result is, amazingly, a simple, open (for now), almost universal platform for networked applications. This platform contains much of human knowledge and supports most fields of human endeavor. We think it's time to seriously start applying its rules to distributed programming, to open up that information and those processes to automatic clients. If you agree, this book will show you how to do it.

REST stands for "REpresentational State Transfer", which is a bit of a mouthful, and is best described in terms of the goals it proposes for a Web API design. A key underlying principle is that REST attempts to use as much of the HTTP protocol as possible to communicate information. This is in contrast to other standards that place all pertinent information about the request mostly in the request's body.

URIs Scoping Information should be communicated via the URI/URL. That should contain all the information needed to determine which data is being requested or should change. The server should not need to read the body of the request to determine what data the request is about. And the client should not need to learn how to form the request's body to obtain the information it needs. This uses the **addressability** of HTTP. This way all the server needs to do to teach the client how to ask for something is to provide the URI for that resource.

HTTP verbs Method Information, namely what action should be taken on the data, should be communicated via the choice of HTTP verb/method (i.e. GET, POST, PUT, DELETE etc). This way there is a clear understanding on what should happen when a GET request is made, for instance such an action should never delete anything.

Response codes The response codes of the request should be as precise as possible in describing what went wrong with the request, if anything. That information should not be hidden somewhere in the response body.

Chapter 4 of Restful Web Services discusses an approach to achieve RESTfullness, that th authors term "Resource Oriented Architecture". It emphasizes the concept of **resource**, i.e. any part of the service important enough to be addressed by itself.

- Any resource should be addressable via at least one URI (though it is possible for multiple URIs to refer to the same resource). This is a key feature afforded to us by the use of the web. Statelessness is another key part of this: That URI should be all the information you need to get to the resource. The server should not have to remember anything from your prior activities on the site.

Inline activity: Consider in our online evaluations application what resources we have and what URIs we might use for them.

WORK IN PROGRESS