**RESTful Architecture and APIs (Application Programming Interfaces)**

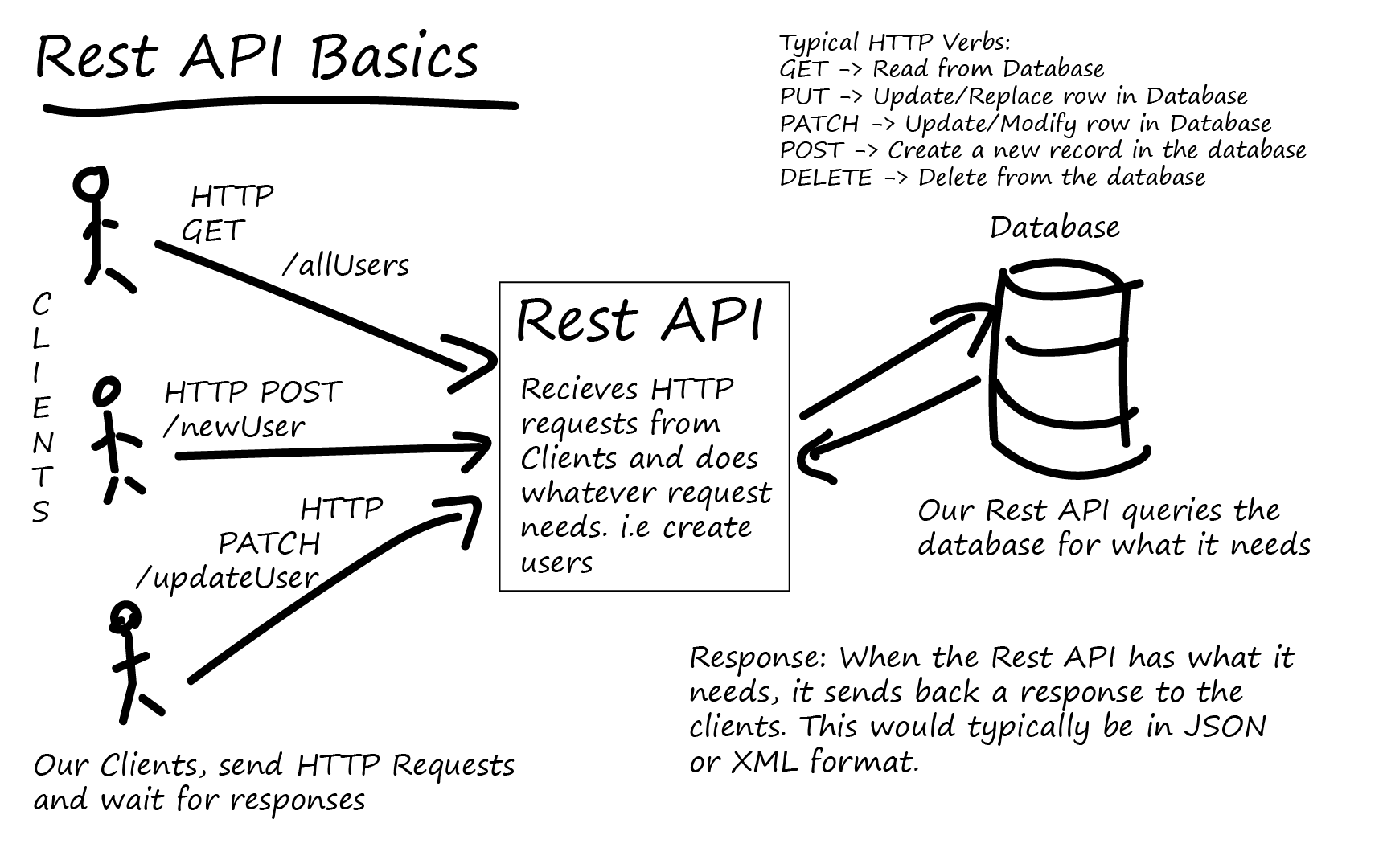
In our last lesson we covered how Internet pages are created and hosted so that we, the user, can browse them. We found out how to write basic HTML, CSS and how to interact with a hosting provider.

However, the Internet is a complicated place.

Alongside human traffic, there is a whole world of machine-to-machine communication going on, all the time. Whenever you navigate to Google, for example, your browser needs to communicate with a DNS server to resolve [www.google.com](http://www.google.com) to an IP address. When you search Google, your search query gets broken down into a handful of terms, appropriate pages found, organised into a list and returned to your server. Services and sites communicate with each other all the time – if you search for package holidays on Google, then go and browse some news sites, chances are you’ll see adverts for package holidays in the advertising banners thanks to behind-the-scenes communication between Google, its ad networks, and the pages you are visiting.

Before we understand how machines communicate, we need to understand RESTful architecture.

**REST** – **RE**presentational **S**tate **T**ransfer

[](https://www.google.com/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=2ahUKEwi6lreJr6fgAhVmBGMBHcraB8oQjRx6BAgBEAU&url=http%3A%2F%2Ftotal-qa.com%2Frest-services%2F&psig=AOvVaw1rqWTEARDREjb4CcKDBBDv&ust=1549551334574863)

This is a framework for all communications across the Internet, invented by Roy Fielding in 2000 at the time when web development was just getting started.

RESTful services have six guiding principles:

* **Client-server architecture**
  + Also known as the *separation of concerns*. The server serves – it hands out information, and consumes requests. It doesn’t concern itself with client-side matters like rendering webpages. Multiple servers can be chained up in client-server models, so it scales well.
* **Statelessness**
  + This constraint is sometimes broken but it’s a good, solid principle. In general, information about the state of a session (a connection to e.g. a website) should be held on the *client* side, not on the server side. In other words, the server should hold *no* information about the client that the client can’t wrap up and send. However, this doesn’t always work – what about the timeout during online banking?
* **Cacheability**
  + The ability for responses from a web server to be cached (or declare as non-cacheable) so that data integrity is maintained between calls. What’s a cache? Simply put, a local repository for fetched information that we can re-use instead of making another call to the server. If we didn’t have the ability to mark pages as cacheable (or not) then the Internet would be a much busier place – and probably not able to sustain as much traffic as it does!
* **Layerability** 
  + The client should not know (or be able to discriminate) between different instances or layers of the server that it is talking to. Why is this important? Well, for busy websites like Facebook, there will be, at any time, hundreds of concurrent web application servers up and running to service requests. If clients (web browsers) behaved differently depending on the application server servicing the request, the user experience would be incoherent. Likewise, servers should be able to forward requests to other servers for scalability.
* **Code-on-demand**
  + Servers should be able to send code to the browser to be executed – in other words, servers should be able to extend the core capabilities of the browser. The application code that underpins Facebook isn’t installed in your browser – it’s served from Facebook, and Facebook is able to send through client-side code (e.g. JavaScript) to be executed in your browser.
* **Uniform interface**
  + RESTful services should be able to communicate using a standard interface.
    - Request-based architecture – information is communicated in individual requests. The request *payload* is separate from information about the request.
    - Representation-based resource manipulation – if the client has a copy of a resource (e.g. a customer record), the client should be able to manipulate the ‘master copy’ of the record using the copy e.g. delete or amend the record on the server.
    - Self-description – Each request should identify how it should be processed. E.g. HTML requests to be processed through the browser, or requests containing XML through a particular application.
    - HATEOAS (Hypermedia as the Engine of Application State) – A little complicated, but essentially once the application/client has accessed the RESTful API, it should be able to discover more information through *relative* links, not absolute ones. E.g. navigating from <https://www.derekcolley.co.uk>, to navigate to cydney.htm the client should be able to reference ./cydney.htm and not the full link, <https://www.derekcolley.co.uk/cydney.htm>. This is important as the client shouldn’t have to be ‘programmed’ with a full list of all resources each time.

Remember – APIs are not the same as web browsing, although the concepts are the same!

APIs use URI (**Uniform Resource Identifiers**) – very similar (not quite the same) as URLs. Effectively, links to a resource that also contain, in the link, the protocol for what to do with the resource –

<https://www.google.com> – the protocol is https://, telling the browser it’s a web page, and the resource is [www.google.com](http://www.google.com), the address itself.

**Activities**

The best way to learn how APIs work is to get hands-on – so go to <https://www.getpostman.com> and we will build our first API. Once it’s installed, use <https://jsonplaceholder.typicode.com/> as a test platform.

Review <https://www.w3schools.com/tags/ref_httpmethods.asp> for some information on GET and POST requests.

Watch <https://www.youtube.com/watch?v=t5n07Ybz7yI> for a quick (10 min) tutorial on API testing with Postman.