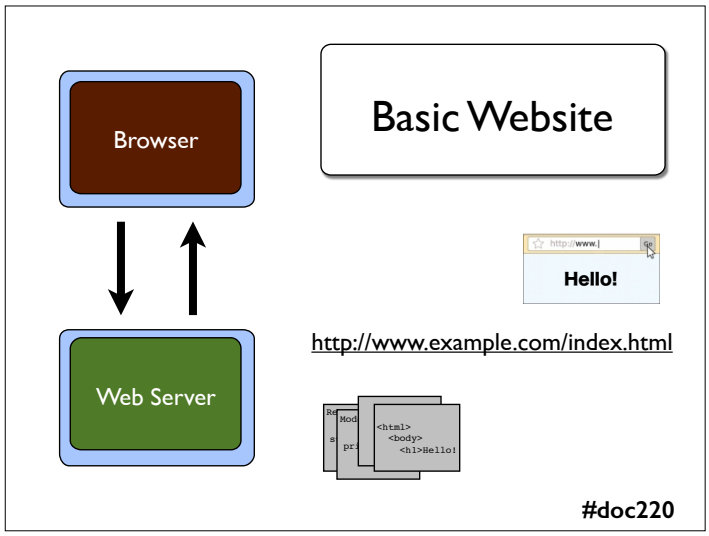
220

Web Applications

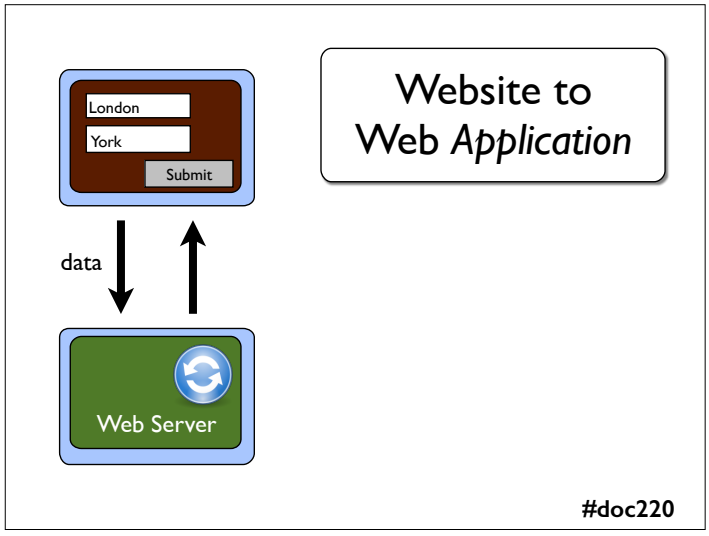
Dr Robert Chatley - rbc@imperial.ac.uk

@rchatley #doc220

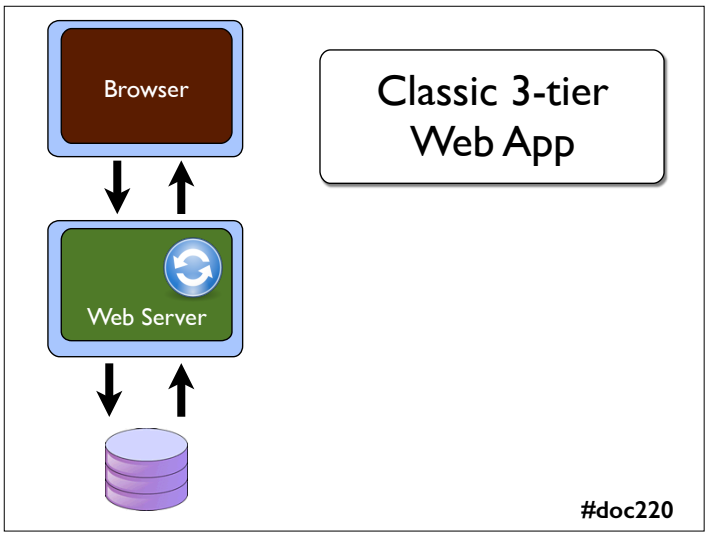


In this lecture we look at the design of web applications, how they are structured, and the similarities and differences with desktop interactive applications. We start with a little history and look at the evolution of web applications of the last 15 years or so.

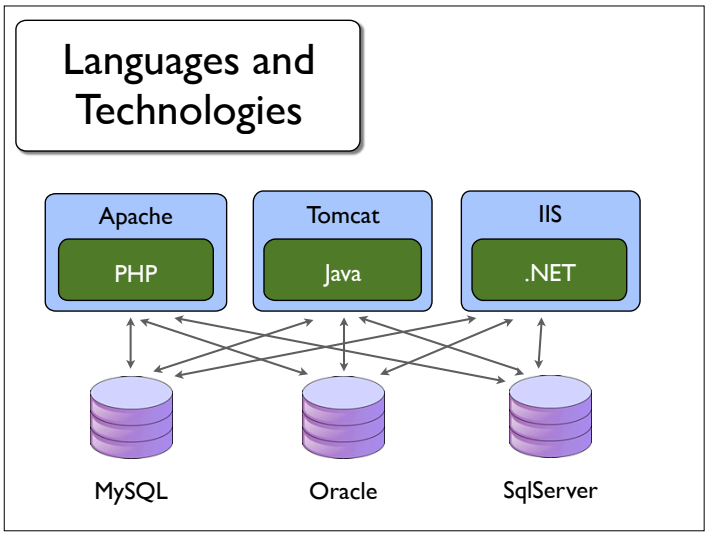
The basic way that the web works hasn't really changed since the first websites were set up. The basic mechanism is one of requests and responses. The application and computation is split between the server and the browser. The user types an address into the browser, which is resolved to an IP of a server. The server processes the request and returns a response, often as HTML, which is rendered by the browser.



Moving on from a simple static website, we have the web application, which provides us with something much more interactive. Web applications follow the same sort of request/response mechanic, but there is much more computation - the server must run a custom program to process the request, rather than just serving a file. The user's input is sent in the request to be processed by the server application in order to form a custom response.



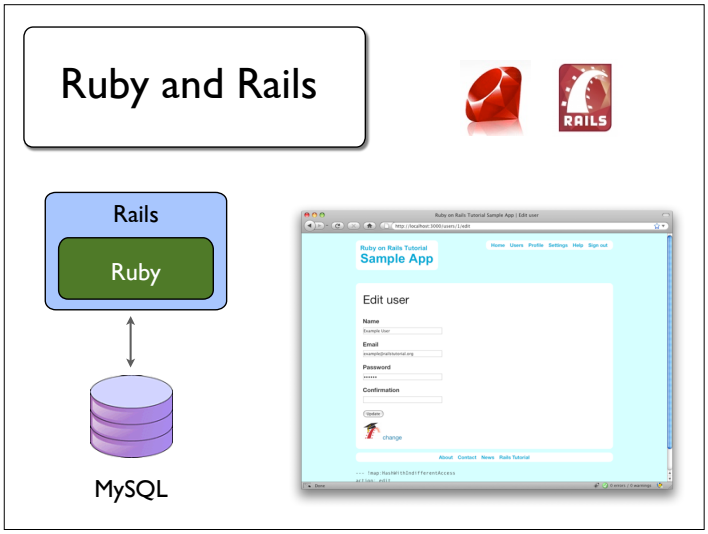
A common design for web applications is the 3-tier architecture. The processing is split between the browser, the web server, and the database. This is very common for sites that are shops, allow you to make bookings, have personalised user accounts, or allow you to look up "current" data in some form.



There is a wide variety of technologies that can be used in each layer. Typical technology choices for a 3-tier architecture might be: LAMP Linux/Apache/PHP/MySQL (plus some common variations, e.g. Postgres instead of MySQL).

More enterprise setups might include Java/Tomcat/Oracle or IIS/ASP.NET/SqlServer

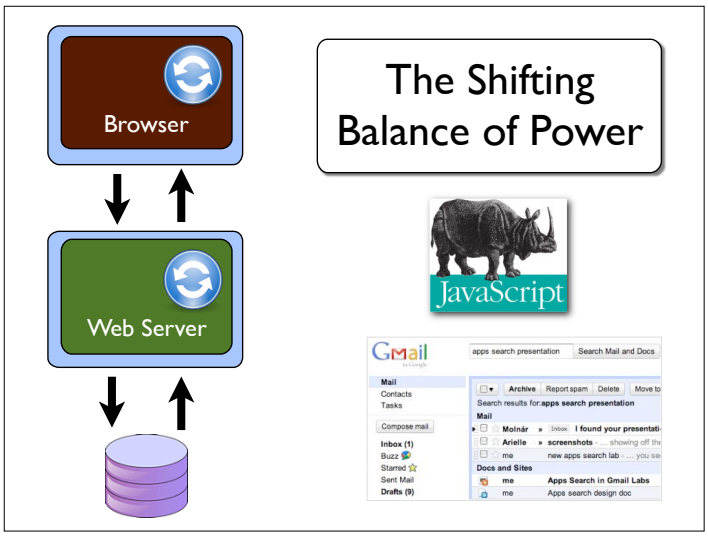
Many different combinations are possible.



A certain type of web development was made easier with the introduction of Rails.

Rails helps developers to be more productive - it provides a lot of boilerplate code that makes it easy to get started and makes it easier to build an application, if you want to follow the structure that it favours.

Rails is not a good fit for all types of site - it's not a magic bullet. It uses a lot of conventions, which means that it is often quite difficult to work out what is actually going on in your application, and under the covers is very complicated.

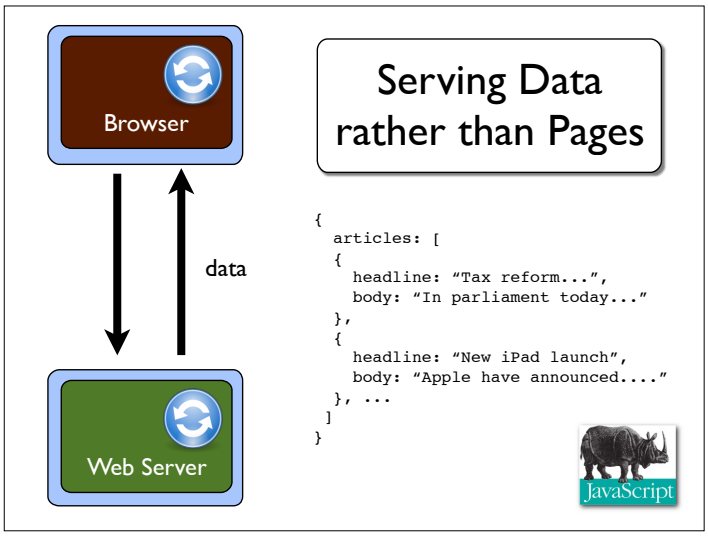


As client devices became more powerful, and browsers more capable, processing shifted more to the client.

JavaScript became more prominent, and a lot more of the code of the application was executed in the browser, enabling a richer user experience.

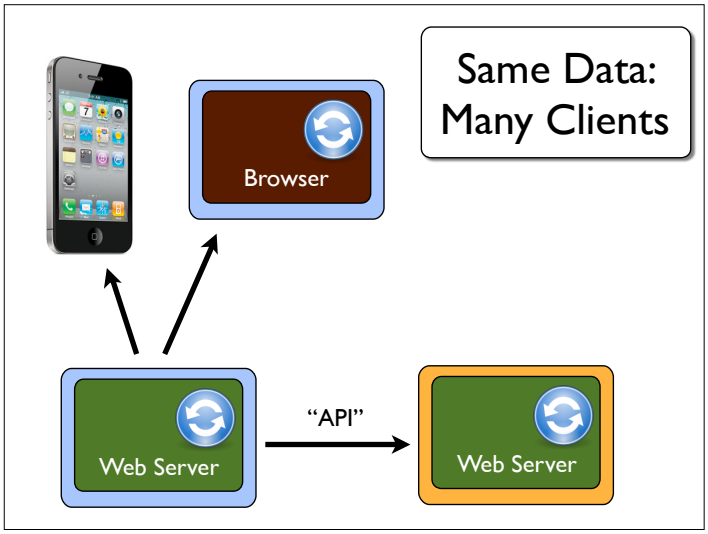
It is now no longer the case that we make one request to the server per click. Some clicks may be processed entirely in the UI. Data for different parts of the page may be loaded independently.

Ajax (Asynchronous JavaScript and XML) became a popular model for transferring data between the browser and the server without causing a full page refresh. This gave the appearance of browser based "applications" rather than just "websites". GMail was a forerunner and showed what could be done.



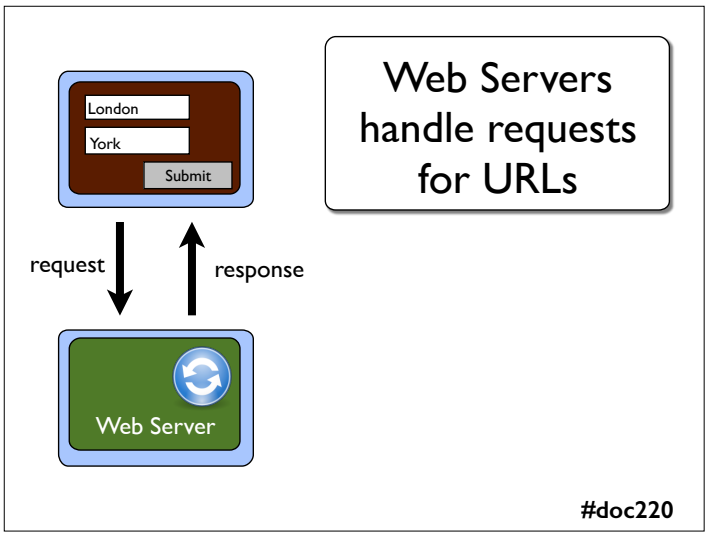
With the browsers taking more responsibility for the presentation layer, it is now common for the server to send structured data rather than whole pages formatted as HTML. JavaScript in the browser then processes this data and renders it in a suitable form.

Popular data transfer formats are JSON or XML.

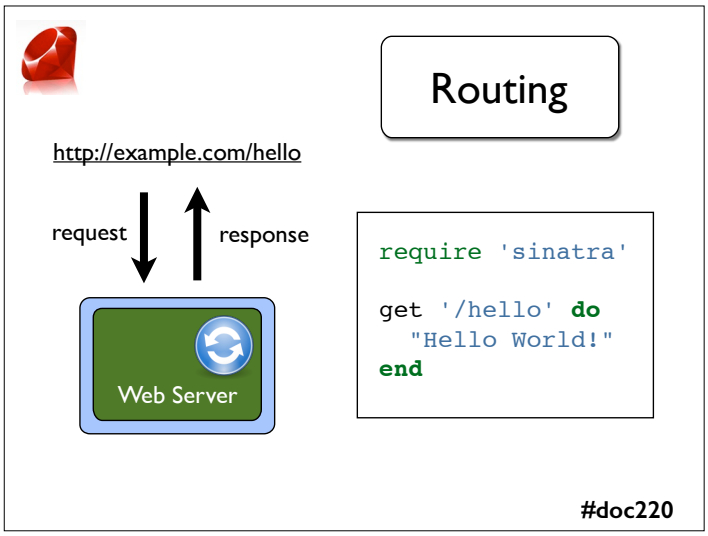


It is common now to need to support different clients (e.g. a mobile app in addition to a desktop browser). If we design our application appropriately, with the correct separation of concerns, different clients can process the same data and present it in different ways, but query the same server.

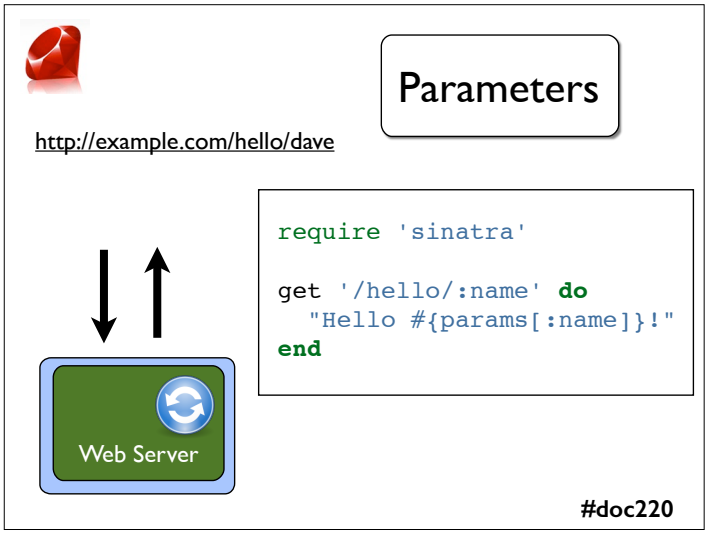
Additionally, other servers can query your server's data to use in their applications - an API - a website for computers to read.



Now let's look in more detail at the mechanics of implementing a web application. For this example we'll use Ruby to implement our server-side component, using the Sinatra library. There are libraries and frameworks available for almost all popular languages to allow you build a web application without starting from scratch. Most of them have similar concepts, but may implement them in different ways.



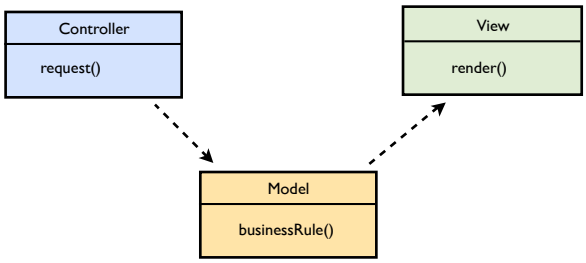
The first thing we need to do is create a server, and map a particular URL to trigger a particular bit of controller code. In Sinatra this requires very little code. Here we map the URL /hello and return a plain text response, a simple string. Not very interesting but at least it allows us to check that our server is working.



To allow us to return more varied responses, we need to be able to process the input request in some way. Here we show using a URL template to match and extract a portion of the URL so that we can use its value to customise the response.

We can do something similar to capture query parameters that are specified as name value pairs after a ? in the URL. http://example.com/hello/dave?formality=casual

Model-View-Controller



MVC separates interactive apps into 3 parts: data; display; user input

#doc220

We have the basic mechanism, but for anything more complex it is typical to create a design based on the Model-View-Controller pattern that we saw when we were looking at native desktop applications.

We divide our server-side code into the controller, which reacts to requests routed to different URLs, and the model, which models the data and business rules for the domain we are working in. We could also write code to render the model as an appropriate view, but commonly we use a templating language to do this.

Views - Templates

views/hello.haml

```
%html
  %head
    %title Hello
  %body
    %h1 Hello #{name}!
```

#doc220

Here is an example of using HAML to create a template for an HTML page that will form our view. There are lots of different templating technologies to choose from, each of which has there own quirks. You just need to find one that you like.

In HAML we use % to create HTML elements, and the indentation dictates the nesting. We use the Ruby string interpolation syntax #{...} to include a value from the model.

By default with Sinatra we save our HAML templates into a folder called *views*.

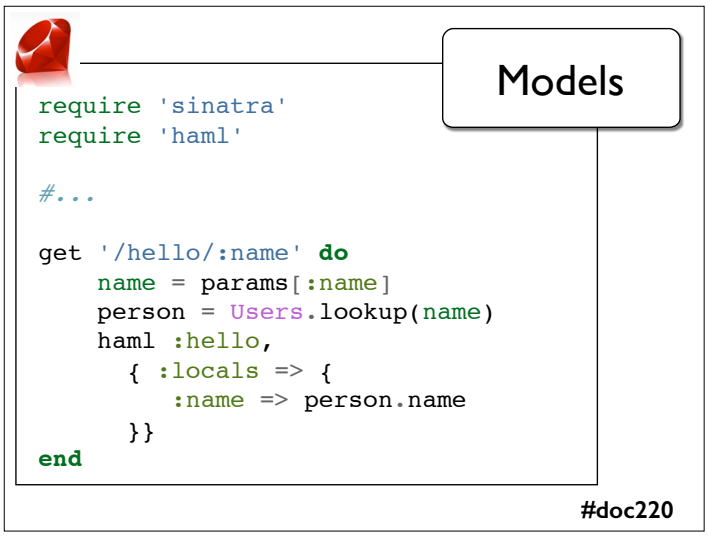
Controller Selects View

```
require 'sinatra'
require 'haml'

get '/hello/:name' do
    haml :hello,
      { :locals => {
          :name => params[:name]
      }}
end
```

#doc220

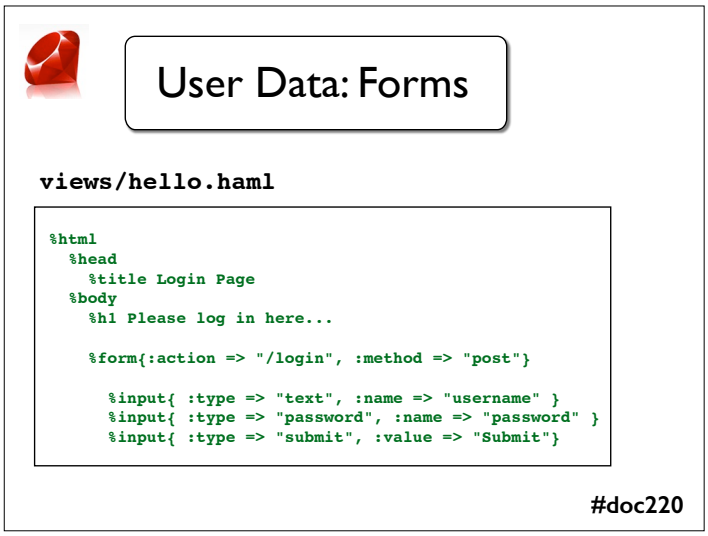
Here is the controller code that maps to this view. The haml method is defined when we require the haml library at the top of the code, and we pass it a *symbol* (marked with a colon) that matches the name of the template file. For the second parameter we pass a *hash* (which is like a map or a dictionary, mapping keys to values using the => notation. The => symbol is known as a *hashrocket*). The hash sets up some data (locals) as (another) hash of keys to values. We can use these locals in the view template, by referring to the keys using the #{...} notation.



When we make the application a little bit richer, we probably want to separate the domain logic out into a Model class, leaving only the code that decodes and dispatches HTTP requests in the Controller.

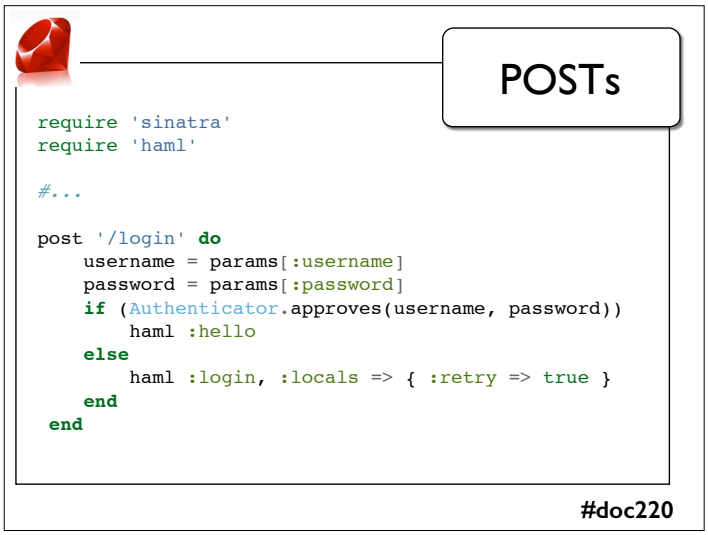
Here we use a class method on Users (which we assume has been initialised in some appropriate way) to access our store of users. The controller doesn't know or care how this is implemented. It could be in memory, in a file, in a database, or something else entirely.

Here we pull the name our of the Person object before pushing to the view template, as we only need the name. We could send the whole Person object, and destructure it in the view if we needed more than one property (displaying a full user record etc).



To build a more useful application, we will likely want to take input from the user, and use this to modify the state of the model in some way before rendering a response.

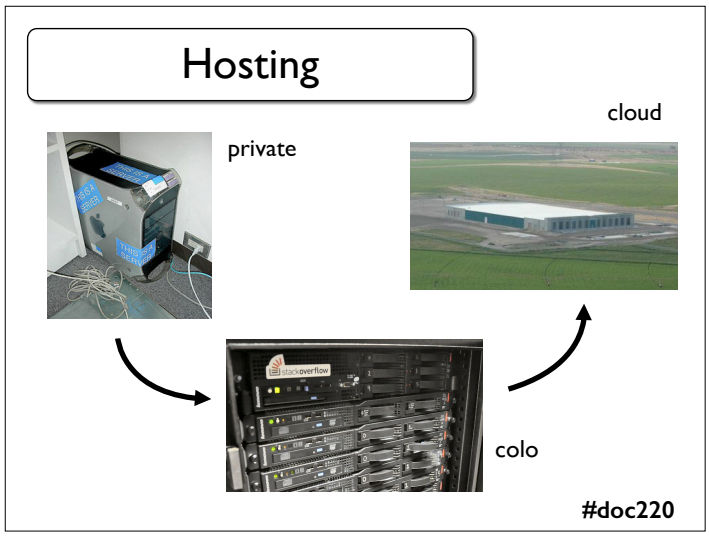
This is done by using a different type of HTTP request (typically a POST or sometimes a PUT, more on that later...) to transfer the parameter data. We may trigger the POST by implementing an HTML form in the view template, and setting its action and method parameters to send data to the server.



When we make the application a little bit richer, we probably want to separate the domain logic out into a Model class, leaving only the code that decodes and dispatches HTTP requests in the Controller.

Here we use a class method on Authenticator (which we assume has been initialised in some appropriate way) to access our store of users and security credentials. The controller doesn't know or care how this is implemented. It could be in memory, in a file, in a database, or something else entirely.

In this case we use the result of the authentication to decide which view to render for the user, but in reality we would probably want to do something more sophisticated than this to actually make a useful application.



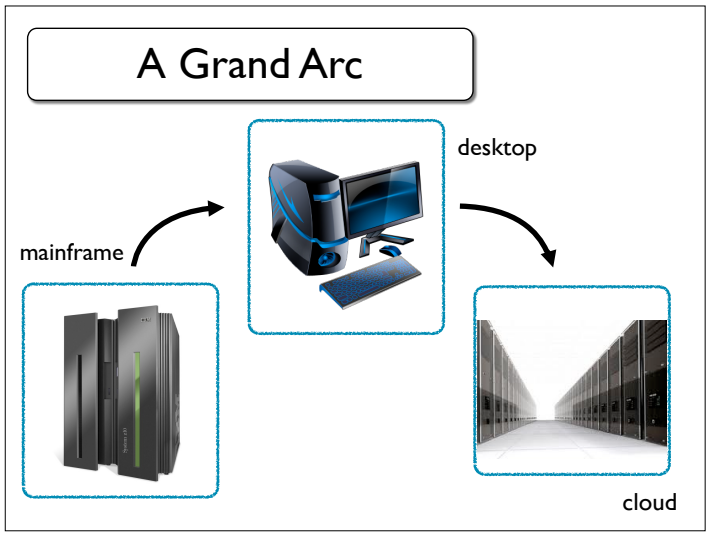
To run a web application, we need a server, otherwise the site will go down when we close our laptop.

Originally a business might host its website on machine in its office, but this conveys a high maintenance cost and responsibility - you need to make sure your server is up and working, has a good internet and power connection etc.

When you get a bit more “serious” you might rent some servers in a colocation facility. These are still your physical machines, but they run in a dedicated datacentre. There can be long lead times on provisioning servers in a datacentre.

Now cloud services (e.g. Amazon, Heroku) are very popular. Provisioning of a new server takes minutes, all you need is a credit card. This makes adapting to demand

---



Back in the 1940s, T.J. Watson is alleged to have said “I think there is a world market for maybe 5 computers” - although this quote is misattributed. There were a few mainframe computers, used through remote dumb terminals. In the early 2000s we saw PCs spread to every desktop. Now with cloud, computation goes on in relatively few, factory-like, data centres, accessed remotely through web-browsers that we use as clients. Interesting.
