IIT Madras

ONLINE DEGREE

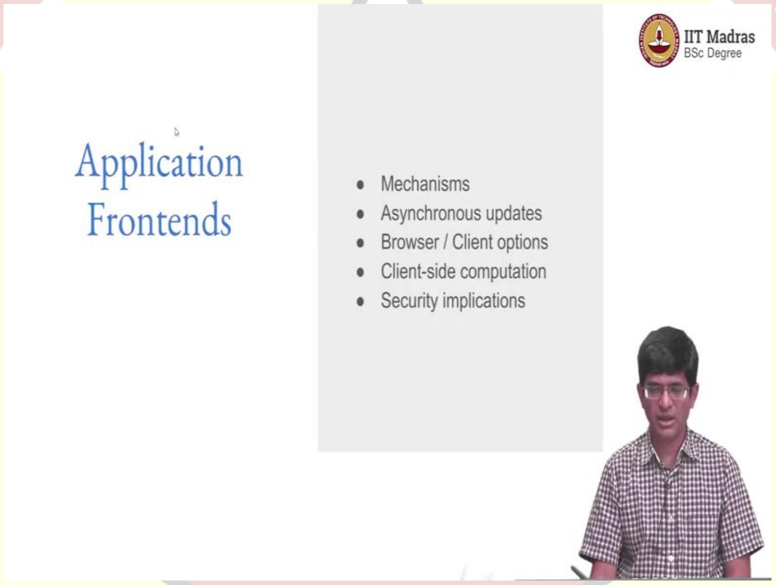
Modern Application Development - I

Professor. Nitin Chandrachoodan
Department of Electrical Engineering
Indian Institute of Technology, Madras

Application Frontends

Hello everyone, and welcome to this course on Modern Application Development. Hello, everyone. Now we are going to look at the frontend, or the browser side that is going to be seen by the user of an application. What are the aspects that you need to keep in mind and what is it that we basically mean by the frontend.

(Refer Slide Time: 00:33)

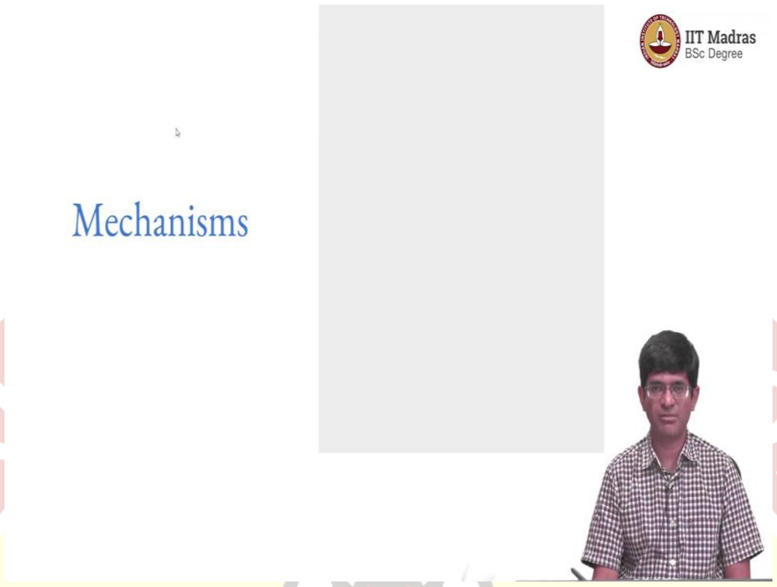


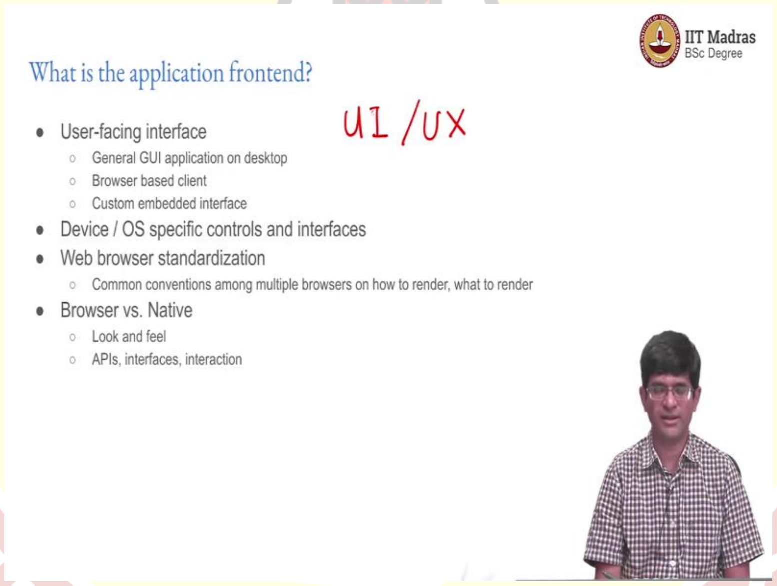
Application Frontends

- Mechanisms
- Asynchronous updates
- Browser / Client options
- Client-side computation
- Security implications

So, we look at various aspects over here, what are different mechanisms that are used. What do we mean by asynchronous updates, different options of clients and browsers, something called client side computation. And finally, once again, security implications. So just like we did in the case of the backend, we also need to understand security implications at the frontend of a browser.

(Refer Slide Time: 01:00)


Mechanisms


What is the application frontend?

- User-facing interface
  - General GUI application on desktop
  - Browser based client
  - Custom embedded interface
- Device / OS specific controls and interfaces
- Web browser standardization
  - Common conventions among multiple browsers on how to render, what to render
- Browser vs. Native
  - Look and feel
  - APIs, interfaces, interaction

UI /UX

So, let us start by looking at the different mechanisms by which frontends can be implemented. So, first of all, what is the application frontend. And this is the user facing interface, meaning that you could be talking about the general GUI application on a desktop, or in our case, it is a browser based client. So, there is some kind of a web browser, a client software that is running. Or you could also actually have some kind of a custom embedded interface. And what I mean by a custom embedded interface is that there can be a sort of specialized application, which is running on specialized hardware, which does not even have a screen.

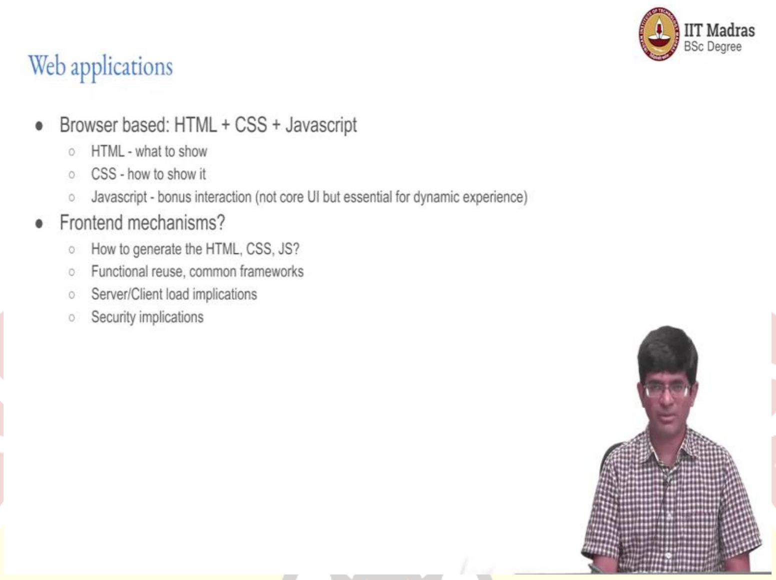
But yet uses let us say, web protocols HTTP, and so on, in order to communicate with some server and get you some kind of a response. An example of that is actually something like, even your Google Assistant, the Google Home devices, and so on, they do not have a screen, they do not have anything to display. But when you turn them on, they connect through your phone, or whichever way it is that you use to set them up. But finally, the device just directly communicates with a server that's sitting at Google data centers, in order to capture certain kinds of information, make certain requests, get certain responses, and so on.

Now, what this means is that there are device or operating system specific controls, the ones that we are traditionally aware of, the keyboard, mouse, and maybe touchscreen, and things of that sort. But there could be others, like, for example, the voice input in a Google Assistant. Now, among all of these things, since we are primarily looking at web based applications, we will be focusing the most on web browsers. And there are many things that need to be understood in the context of the standardization of web browsers.

In particular, there are many common conventions that are followed by web browsers in terms of how to render something what to render, and how exactly a page should look. There is also finally one notion of so called native application. And how exactly do browser based applications compare with native applications, in particular how did they look and feel, what kind of API's do they have access to, are they able to interact directly with the hardware that's available on the device that you have or are there restrictions, all of those are things that the frontend needs to take care of.

Because finally, the user is interacting only with the application frontend. So, what are all the things that you can do in order to provide a better experience. So, pretty much all of this comes down to this term, which is sometimes, of course we know what UI means, its User Interface. But there is also UX, which is the user experience. And the user experience is ultimately one of the most important things when you are trying to design a frontend.

(Refer Slide Time: 03:58)



Web applications

- Browser based: HTML + CSS + Javascript
  - HTML - what to show
  - CSS - how to show it
  - Javascript - bonus interaction (not core UI but essential for dynamic experience)
- Frontend mechanisms?
  - How to generate the HTML, CSS, JS?
  - Functional reuse, common frameworks
  - Server/Client load implications
  - Security implications

So, coming back to our focus, which is web applications, we are looking at browser-based applications, which means that what comes back from the server is going to be primarily HTML text, plus CSS, which is used for the styling, and some amount of JavaScript. And the JavaScript is something which we have not touched upon much so far. The main focus of our course over here is web applications, which means that it is browser based.

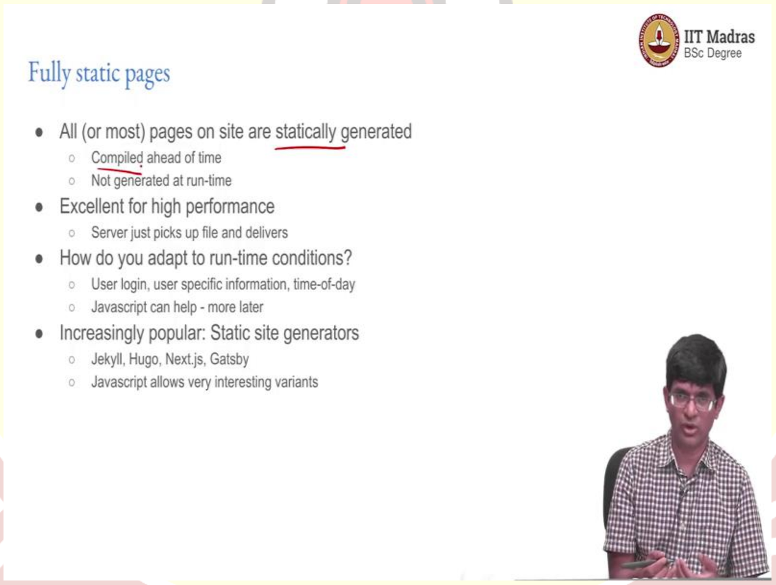
And what I mean by browser based is that what is being sent back from the server to the client is usually HTML. And that is the main, the important part, the rest of it could be CSS which is used for styling, and some amount of JavaScript which is something we have not discussed in detail so far, but which we will be looking at in a little bit more detail in this set of lectures. Now, the HTML tells you basically the content what to show. CSS tells you how to show it, basically in other words, how to make it look nice, like to put it simply.

The JavaScript is, in some sense, or at least was originally brought in as a kind of bonus. It allowed for sort of smoother interactions. It is not part of the core user interface. But it is nowadays it has become essential, especially for a dynamic experience. And suitable use of JavaScript can result in fantastic looking web pages, fantastic feeling webpages, more importantly.

They may not look fancy, but the way they respond to the kinds of things that you are trying to do can be really things that, make you really want to use that website. But there is also a danger that JavaScript can be misused quite easily. So, what are the things that can be done with it, we will look at that to some extent.

Now, what are the mechanisms by which we can generate this, the frontend mechanisms. How do we generate this HTML, CSS, JavaScript? There are these concepts of functional reuse, how do we reuse some types of code, some common frameworks, you have already seen bootstrap, which is a set of CSS styles, which we try to reuse as far as possible simply because they have, over the years been found to be aesthetically pleasing. And using those makes it easy to sort of make your website look nice. So, we also have many implications on the, all of these choices also have implications on the load, both on the server side as well as the client side. And, of course, finally, security. So, all of those are things that we will be looking at as we move forward.

(Refer Slide Time: 06:40)



Fully static pages

- All (or most) pages on site are statically generated
  - Compiled ahead of time
  - Not generated at run-time
- Excellent for high performance
  - Server just picks up file and delivers
- How do you adapt to run-time conditions?
  - User login, user specific information, time-of-day
  - Javascript can help - more later
- Increasingly popular: Static site generators
  - Jekyll, Hugo, Next.js, Gatsby
  - Javascript allows very interesting variants

So, the first mechanism, in some sense that I can use in order to deliver a website is to use something called fully static web pages, where all or at least most of the pages on the site are statically generated. Now, what does statically generated, it may mean, it means that at the time that a user makes a request, the server does not need to do any processing, it just pretty much can take content that is already available, usually in a file and deliver it to the end user.

So, no processing needs to be done. Which means that the pages are compiled ahead of time and not generated at runtime. Now, this is actually excellent if you want to get high performance simply because it is very, relatively very easy to make a server whose only job is to pick up a file and deliver it.

So, if the only job that a web server had was, look at the request, look at the path, go look for a file in the file system with that same name, and deliver it to the client, you can make servers that can do this extremely fast, you can pretty much max out the line speed, what I mean by line speed is the speed of the network, how many bytes per second, can the network accommodate, the server will be able to give you that many, that much data.

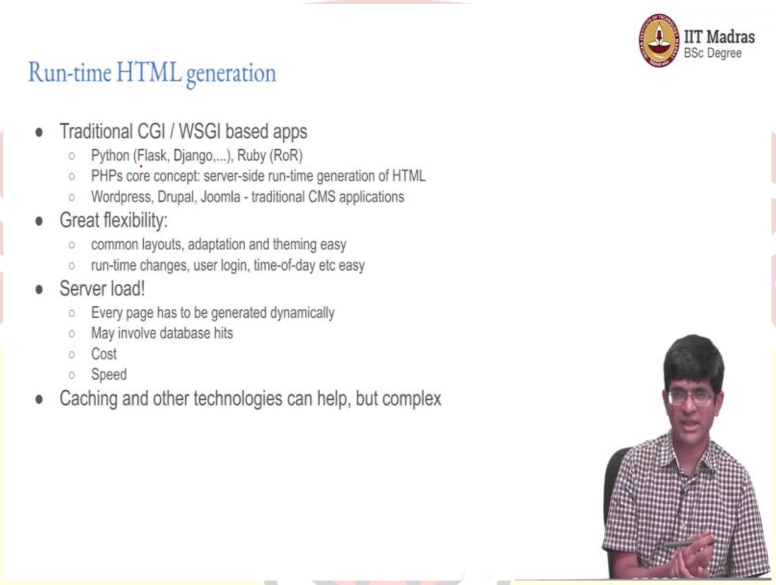
Now, the problem is how do you adapt to runtime conditions, you want people to be able to log in, which means that every time a person logs in, you can not just go and retrieve the same file and send it back to them. It has to have something which says, hello ABC, whatever your name is. Some user specific information, this is Gmail, this is a set of emails that you have got, it can just show you a blank page and expect you to be happy about it.

And depending on the time of day, you might have different kinds of pages that you want to show. Now, in some specific cases, JavaScript can help with this, it can actually be run on the client side and can change what is displayed, by making separate requests back to the server. And all of those requests could have been compiled ahead of time and stored somewhere, it is always a trade off, you do more work at compilation time, less work at runtime. And this kind of static page, it is an interesting throwback to the beginning of the web, because if you look at it, the original notion of the internet or of the web, essentially had this kind of it had only static web pages.

And what static web pages do is all that the server needs to do is pick up a file and send it back to the client. Now, what happens, what is happening nowadays is that there are many so called static site generators, you might have come across names like Jekyll or Hugo, or Next.js. in the case of JavaScript, which are geared towards this notion of building a site where most of the pages are just stored as static HTML and can be delivered on demand to the client.

So, in some sense, 40 years or 30 years down the line, we have come full circle and, after all the dynamic pages that were built up over time, we are increasingly seeing a return to the case where applications can also be built using static pages. Now, obviously, in order for an application to be interesting, you need some interactivity. JavaScript plays a big role over there. Now, a lot of what can be done with something like this is out of the scope of what we can do in this course. So, we will not be going further into too much detail on it. But it is definitely something that is of interest as you move forward.

(Refer Slide Time: 10:32)



Run-time HTML generation

- Traditional CGI / WSGI based apps
  - Python (Flask, Django,...), Ruby (RoR)
  - PHPs core concept: server-side run-time generation of HTML
  - Wordpress, Drupal, Joomla - traditional CMS applications
- Great flexibility:
  - common layouts, adaptation and theming easy
  - run-time changes, user login, time-of-day etc easy
- Server load!
  - Every page has to be generated dynamically
  - May involve database hits
  - Cost
  - Speed
- Caching and other technologies can help, but complex

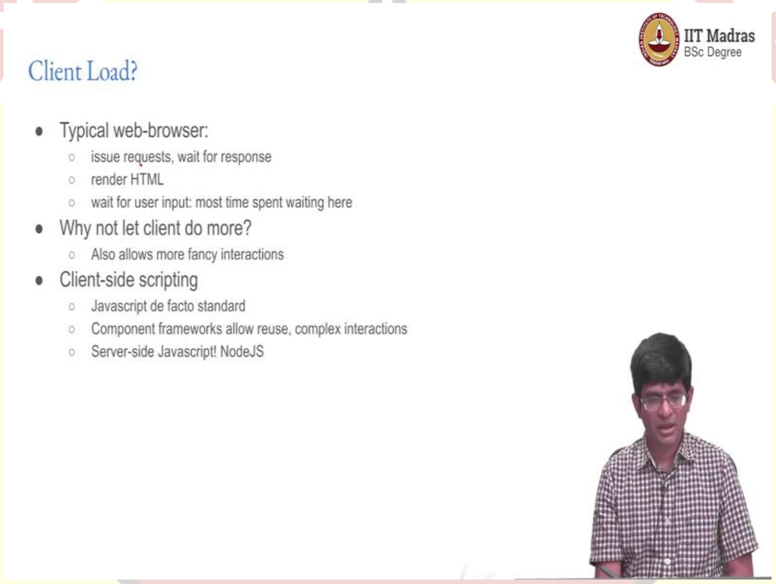
On the other hand, even the framework that we are using in this course flask, what does it do, it is basically a web services gateway interface, WSGI type of application. And every single request, that comes from a client is processed at runtime. So, the server has to go, run some script, come up with a response, render a template and send it back to the user. A lot of well-known sites and site management mechanisms, usually what are called content management systems, or CMS, are built around this kind of approach. So, you have WordPress, Drupal, Joomla, all of those are generating the pages for you at runtime.

Great flexibility, there was a reason why they became popular, they allowed you to have sort of a common template, but at the same time dynamic pages. Runtime changes or making a small change anything was very easy. The problem is, they require much more powerful servers. Every page is generated dynamically. And there is a good chance that many of those dynamic pages involved hit store database.

So, not only do you have a problem of getting a suitably powerful database server, you also have the problem that you can not have multiple sort of, server frontends, which accept requests from clients, because then they would all need to talk to the same database and the database might become the bottleneck. Cost is going to be a huge issue. Speed also can be a concern.

And why is that because basically, the number of requests per second that a server can handle is severely limited if it has to generate each page dynamically at runtime, compared to just pulling out a file and sending it. Now, of course, there are techniques like caching and so on, that can help in parts of it, but they are complex, not easy to implement, and definitely do not solve the problem completely.

(Refer Slide Time: 12:37)


Client Load?

- Typical web-browser:
  - issue requests, wait for response
  - render HTML
  - wait for user input: most time spent waiting here
- Why not let client do more?
  - Also allows more fancy interactions
- Client-side scripting
  - Javascript de facto standard
  - Component frameworks allow reuse, complex interactions
  - Server-side Javascript! NodeJS

Now, server load is obviously an issue with this kind of runtime generation of pages. So then came the question, what is the client doing? A typical web browser basically issues a request, it waits for a response, renders the HTML shows you something on the screen, and then waits for the user to click somewhere. So, it is pretty much doing nothing, most of the time, most of the time on a browser is spent being idle. So, the question basically became, why not let the client do more.

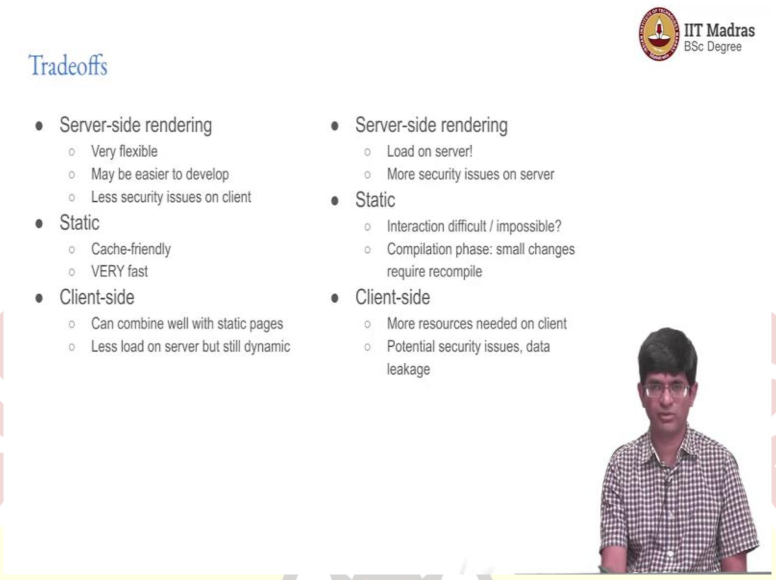
And it started out with something simple, I mean, at least let us say some kind of fancy zooming or animations and things of that sort on the client side, can we have a script or some other way by which the client basically makes the interface more appealing to the user. But then over time, it also came about that look, why not let the client actually do part of the processing there. And the requests that it sends to the server can be much more sort of compact and compressed, and requesting only small parts of what needs to be updated.

This is where JavaScript has its origins, JavaScript basically came about as some kind of scripting mechanism that allowed you to make small modifications to webpages, or small amounts of interactivity. But in the process, it became pretty much the de facto standard, de facto meaning. I mean, it is there, whether or not you like it, it is the standard. It is not that anybody just decided JavaScript is going to be the thing, it was the first one that came up, it was the most popular, it succeeded, and therefore it is the standard.

Having said that, I do not have anything against JavaScript. I think it has very nice features in some ways. So, it is entirely possible to do good things with it. It is also possible to write really bad code with JavaScript, but that's true, pretty much of any language. Now, what happens, what happened eventually, after many years of using JavaScript for the frontend is that people started doing component reuse. You started building components, like the navigation bars, or certain other kinds of things that come up on a part of the screen and say, maybe I can reuse this entire thing as a small widget that sits on part of the screen and others can use it also.

So maybe, they just have to take the JavaScript code that I am using, and they can use it in another context. And eventually, it got to the point where people said, look, JavaScript is a fully capable language, why not also use it for the backend processing, meaning that the part that actually implements controllers, and eventually also for the model and the interaction with the database. And that is what you might have heard of as NodeJS. NodeJS essentially says, let us take all of this developments that have come up at the frontend, and use that knowledge in order to implement good back end computations as well.

(Refer Slide Time: 15:24)



Tradeoffs

- Server-side rendering
  - Very flexible
  - May be easier to develop
  - Less security issues on client
- Static
  - Cache-friendly
  - VERY fast
- Client-side
  - Can combine well with static pages
  - Less load on server but still dynamic

- Server-side rendering
  - Load on server!
  - More security issues on server
- Static
  - Interaction difficult / impossible?
  - Compilation phase: small changes require recompile
- Client-side
  - More resources needed on client
  - Potential security issues, data leakage

Now, there are obviously trade offs. So, what do we have, we have sort of different modes in which we could operate one of them is so called server-side rendering, which is very flexible, can be somewhat easy to develop, simply because, what you are doing or there is, you have some standardized templates. And all that you need to do is then add the extra data, which is required for each page that you want, potentially less work in the client end, which potentially means less security issues.

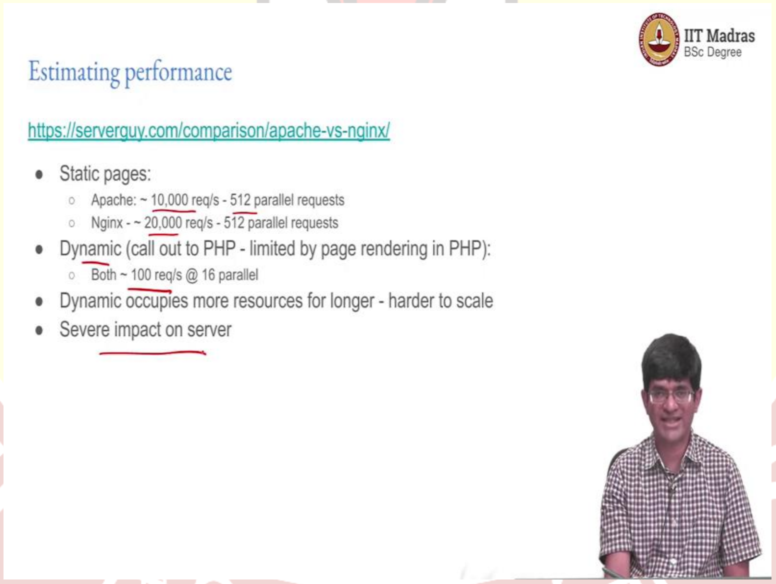
So, this is of course, debatable but, there is simply less work being done on the client side. The problem, of course, is just there is so much load on the server, the server has to do everything, and the client is pretty much sitting idle. It also means that you have to be extra careful about security issues on the server, because it has to be capable of handling all the different kinds of inputs that the client can throw at it, it has to be capable of handling load related issues, and various other things, which you might be able to sort of break off a little bit more.

The other extreme is fully static, which is of course, extremely cache friendly, static files can be cached anywhere in proxies, or whatever it is. And it is going to be very fast, the server just needs to pull a file out of the file system and send it over the network. The problem is interaction with the user becomes difficult. Every time I have to click on something and the server, I have to click on a link, go back, get some data from the server, it is difficult to really have any kind of interaction.

And, unless you have some mechanism for something dynamic on the server side, you pretty much have only static pages. And the other thing about static is that, even a small change in the page, or one page might require recompiling, the entire website. There are ways of making this fast, of course, but it is something to be considered. And finally, you have sort of a combination of both, where you have predominantly static pages. So, the client side, it combines well, with static pages. There is much less load on the server, but it is still a dynamic. And why is it dynamic, because you are able to have some part of it, the server is capable of making some very specific kinds of responses.

But those are kept highly limited. More work is done on the client, which means more resources are needed on the client side. Potentially, you could have security issues on the client side, you are now running, a fairly complex programming language on the client browser, that can open up, data leakage or various other kinds of issues that you might not have thought about before.

(Refer Slide Time: 18:10)



Estimating performance

https://serverguy.com/comparison/apache-vs-nginx/

- Static pages:
  - Apache: ~ 10,000 req/s - 512 parallel requests
  - Nginx - ~ 20,000 req/s - 512 parallel requests
- Dynamic (call out to PHP - limited by page rendering in PHP):
  - Both ~ 100 req/s @ 16 parallel
- Dynamic occupies more resources for longer - harder to scale
- Severe impact on server

So now, this is actually not in relation to all of this, there is this website, I came across, where there was actually a comparison done between two different popular web servers, Apache, and Nginx, web server. And essentially, what they were doing was sort of trying to see, how well they perform under different conditions. Now, I am not using this as an advertisement for either Apache or Nginx. Apache is a fantastic web server I have used it for many applications. And it is like very, it is rock solid, it is like very well designed, it has lots of great features.

But nowadays, Nginx is also very good. And there are places where it is able to do better than Apache, simply because it is sort of been better tuned, people understood certain problems with Apache and then tried to solve them better, in some ways. So as a result, what they observed at least was that in terms of static pages, and Apache web server could handle something like 10,000 requests per second, a single machine handling 500 or parallel requests.

So of course, think about this, what does 10,000 requests per second mean? Remember that you have to send 10,000 responses. So, even if each response is something like 1 kilobyte in size, it means that now we are basically sending out 10 megabytes per second of data, or that is 10 megabytes is roughly 100 megabits per second. So, it is quite possible that, you are already close to sort of saturating the network. But that is not really the case over here, because it turned out that on the same machine, the Nginx server was able to hit 20,000 requests per second.

So clearly, I would say that, what is happening is that the size of the response is not very large, probably like not even 100 bytes or so. But this was clearly an artificial example just to test server performance. And in fact, there are tools that allow you to test server performance, you just create artificial requests and see how fast your server is able to keep up. So good either way, we are talking about like, order of 10,000 requests per second, whichever server you use.

Now dynamic, where essentially, they are calling out to a PHP rendering engine. Look at it 100x less. So, what could be done with 10,000 requests per second has now come down to something like 100 requests per second, simply because every page is now being rendered in PHP. There is a script that is being run in order to generate the output corresponding to each page. This is obviously a much more severe impact on the server. And it also occupies more resources for longer and is hard to scale.