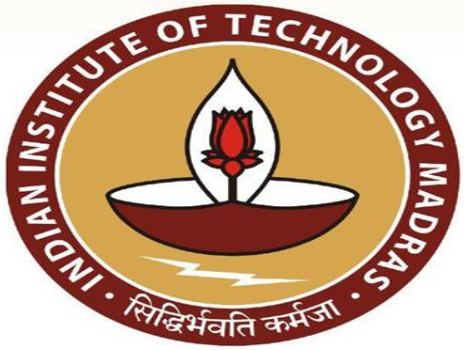

IIT Madras
BSc Degree

Frontend

Application Frontends

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

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

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

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

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

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

Tradeoffs

- Server-side rendering
  - Very flexible
  - May be easier to develop
  - Less security issues on client

- Server-side rendering
  - Load on server!
  - More security issues on server

Tradeoffs

- Server-side rendering
  - Very flexible
  - May be easier to develop
  - Less security issues on client
- Static
  - Cache-friendly
  - VERY fast

- Server-side rendering
  - Load on server!
  - More security issues on server
- Static
  - Interaction difficult / impossible?
  - Compilation phase: small changes require recompile

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

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

Asynchronous Updates

Original web

- Client sends request; Server responds; Client displays
- For any update of page:
  - new request sent from client to server
  - server has to respond with complete page, HTML, styling etc
  - client has to render the page again from scratch
- Potential issues
  - Server load: lots of redundant data to be sent each time
  - server-rendering -> more work
  - slow updates: load full page, re-render

Asynchronous Updates

- Update only part of the page
  - Load extra data in the background after the main page has been loaded and rendered
- Quick response on main page: better user experience
- Request for update can ask for just minimal data to refresh part of a page
  - Example: show user a form to select animal
  - request data about animal alone from server - no need for HTML or other styling
  - refresh only one in the page with text about the animal
- Originally seen as AJAX, now many variants

Core idea: refresh part of the document based on asynchronous (background) queries to server

DOM

Document Object Model

- Programming interface for web documents
- What is a web-page?
  - HTML source? Rendered image?
- DOM is an abstract model (tree structure) of the document
- Object-oriented allows manipulation like known objects
- Tightly coupled with JavaScript in most cases
  - Can also be manipulated from other languages (Python has xml DOM interface for example)

Example usage

https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

```
const paragraphs = document.querySelectorAll("p");
// paragraphs[0] is the first <p> element
// paragraphs[1] is the second <p> element, etc.
alert(paragraphs[0].nodeName);
```

Manipulating the DOM

```
<html>
  <head>
    <script>
       // run this function when the document is loaded
       window.onload = function() {

         // create a couple of elements in an otherwise empty HTML page
         const heading = document.createElement("h1");
         const heading_text = document.createTextNode("Big Head!");
         heading.appendChild(heading_text);
         document.body.appendChild(heading);
      }
    </script>
  </head>
  <body>
  </body>
</html>
```

Component building

- DOM is manipulatable through programs
- Bring concepts of programming into DOM manipulation:
  - Objects
  - Composition of objects, inheritance
  - Loops, iterators, programmable placement
- Lot more flexibility in front-end
- Lot more *complexity* in front-end

Summary

- Asynchronous updates opened up front-end development
- Many new frameworks, technologies
- Beyond scope of this course
    - But essential knowledge for someone interested in app development

Browsers and Clients

Minimal requirements

- Render (display) HTML
- Cookie interaction: accept, return cookies from server to allow sessions
- Text-mode browsers (lynx, elinks etc) may not do anything more!

Text-mode and Accessibility

- Browse from command line - only text displayed
- No images, limited styling

Accessibility:

- Page should not rely on colours or font sizes/styles to convey meaning
- W3 accessibility guidelines

Page styling

- Cascading Style Sheets (CSS) most popular now
- Difficult in text, accessible browsers
  - But has many features to help even with those!
- Proper separation of HTML and styling gives best freedom to browser, user

Interactivity

- Some form of client-side programmability needed
- JavaScript most popular - de facto standard
- Can interact with basic HTML elements (buttons, links, forms etc.)
- Can also be used independently to create more complex forms

Performance of JS depends on browser and choice of scripting engine

JavaScript engines

- Chrome/Chromium/Brave/Edge: V8
- Firefox: SpiderMonkey
- Safari, older versions of IE use their own

Impact

- Performance: V8 generally best at present
- JS standardization means differences in engines less important

Client load

- JS engines also use client CPU power
  - Complex page layouts require computation
- Can also use GPU: extensive graphics support
  - Images
  - Video
- Potential to load CPU
  - Wasteful - block useful computations
  - Energy drain! - https://www.websitecarbon.com/

Machine clients

- Client may not always be a human!
- Machine end-points: typically access APIs
- Embedded devices: post sensor information to data collection sites
  - Especially for monitoring, time series analysis etc.
- Typically cannot handle JS - only HTTP endpoints

Alternative scripting languages

Python inside a browser? - Brython!

https://brython.info/

Problems with alternatives

- JS already included with browsers - why alternative?
- Usual approach: **transpilation**
  - Translation - Compilation
- Some older browsers tried directly including custom languages - now mostly all convert

WASM

- WebAssembly
- Binary instruction format
- Targets a stack based virtual machine (similar to Java)
- Sandboxed with controlled access to APIs
- “Executable format for the Web”
- Handles high performance execution - can translate graphics to OpenGL etc.

Emscripten

- Compiler framework: compile C or C++ (or any other language that can target LLVM) to WebAssembly
- Potential for creating high performance code that runs inside browser
- Limited usage so far

https://emscripten.org/index.html

Native Mode

- File system
- Phone, SMS
- Camera object detection
- Web payments

Functionality can be exposed through suitable APIs: requires platform support

- Adds additional security concerns!

Client-side computation

Validation

- Server-side validation essential
  - No guarantees that request actually came from a given front-end!
- But some client-side validation can reduce hits on server
- Example: email, date range, sanitization (no invalid characters) etc.
- Similar validation to backend, but now in front-end script
- Extra work, but better user experience

https://developer.mozilla.org/en-US/docs/Learn/Forms/Form_validation

Inbuilt HTML5 form controls

- Partial validation added by HTML5 standard
- `required`: mandatory field
- `minlength, maxlength`: for text fields
- `min, max`: for numeric values
- `type`: for some specific predefined types
- `pattern`: regular expression pattern match

Important: older browsers may not support all features.

Is backward compatibility essential for your app?

JavaScript validation

Constraint Validation API:

https://developer.mozilla.org/en-US/docs/Web/API/Constraint_validation

- Supported by most browsers
- Much more complex validation possible

Remember: not a substitute for server-side validation!

```
<form>
  <label for="mail">I would like you to provide me with an e-mail address:</label>
  <input type="email" id="mail" name="mail">
  <button>Submit</button>
</form>
```

```
const email = document.getElementById("mail");

email.addEventListener("input", function (event) {
  if (email.validity.typeMismatch) {
    email.setCustomValidity("I am expecting an e-mail address!");
  } else {
    email.setCustomValidity("");
  }
});
```

```
const email = document.getElementById("mail");

email.addEventListener("input", function (event) {
  if (email.validity.typeMismatch) {
    email.setCustomValidity("I am expecting an e-mail address!");
  } else {
    email.setCustomValidity("");
  }
});
```

Captcha

- Problem: scripts that try to automate web-pages
- Can generate large number of requests in short time - server load
- Railway Tatkal, CoWin appointments etc.

Solution

- Prove that you are a human
- Limited number of clicks possible per unit time
- Script on page will generate some token - server will reject requests without the token

Crypto-mining ?

- Javascript is a “complete” language
- Can implement any computation with Javascript
- Modern JS engines very powerful, fast
  - Can even access system graphics processor (GPU) for rendering etc.
- Run a simple page that loads and runs a JS script
- Script will send results back to server through async calls
- Client may not even be aware!

Security Implications

Sandboxing

- Should JS be run automatically on every page?
  - Yes: provides significant capabilities
  - No: what if the page tries to load local files and send them out to server?
- Sandbox: secure area that JS engine has access to
- Cannot access files, network resources, local storage
- Similar to a Virtual Machine, but at higher level (JS interpreter)

Overload and DoS

- DoS: Denial of Service
- Run a script that takes over the browser engine and runs at high load
- Difficult to even navigate away from the page, or close the page
- Potentially exploit bugs in browser
- Server attack:
  - Replace some popular JS file with a bad version
  - Will be loaded by large number of sites, users - can write script to access some other site
  - Target site will be hit by huge number of requests from several sources, very difficult to control

Access to native resources

- Can JS be used to write fully native applications?
- Access to resources like local storage, sensors (tilt, magneto, camera?)
- Can be permitted explicitly through browser

Can also be compiled directly to native resources!

- Reduce browser overheads
- Smoother interaction with system

Summary

- Frontend experience determined by browser capabilities
  - Basic HTML + CSS rendering - styling
  - Javascript / client-side scripting for user interaction, smoother integration
- Native clients possible
- Potentially serious security implications!
- Always validate data again at server, do not assume client validation
  - HTTP is stateless: server cannot assume client was in a particular state!