## SSH

~ secure shell (protocol)

It is a protocol, like

* HTTP
* FTP
* HTTPS

It encrypts data. Communication between devices (client and host).

**ssh {user}@{host}**

You will be granted a remote terminal window.

ssh key (github, bitbucket…)

### How SSH works

Encryption is a way to “have secrets”.

1. **Symmetric Encryption**
   1. A secret **key** for encryption and decryption.
   2. Problem: anyone who has the key can decrypt messages!
   3. Solution: Key Exchange Algorithm
      1. The key is never shared between the client and server!
2. **Asymmetrical Encryption**
   1. Public vs. private keys
      1. Two separate keys for encryption (public) and decryption (private)
         1. One way relationship
         2. **The private key never relieved!**
      2. They linked together in terms of functionality
         1. But the private key can not mathematically compute from its public counterpart.
   2. Difiie Hellman Key Exchange Algorithm
      1. It makes it possible for each party to combine their own private data with public data are from other systems to arrive in an identical secret session key.
3. **Hashing**
   1. Solves the Middle man problem.
4. **Authentication**
   1. Password
   2. Rsa (public-keygen)

<https://www.udemy.com/course/the-complete-junior-to-senior-web-developer-roadmap/learn/lecture/10197518#overview>

## Performance

JTS – Section 3

**Each second of web page load delay cost a heavy amount of money!**

<https://www.udemy.com/course/the-complete-junior-to-senior-web-developer-roadmap/learn/lecture/10202472#content>

### 3 keys to performance

1. **Client side (rendering)**
   1. Critical Render Path
      1. The files arrive at the browser
         1. HTML (parsing) -> DOM
         2. Style link -> CSS
            1. CSSOM
         3. Script tag -> DOM -> CSSOM
         4. DOM + CSSOM => Render Tree => Layout => **Paint**
      2. Optimization opportunities
         1. Put the script tag at the end of the .html file this way it is not going to block the page rendering
            1. Except for special cases, eg.: google analytics
         2. CSS is render blocking!
            1. Load only what is needed!
            2. Above the fold loading
            3. link tag media attributes
            4. less specificity
            5. +1: internal CSS (no load)
            6. +2: inline CSS
         3. JS is parses blocking!
            1. It can affect both CSSOM and the DOM!
            2. Use **async** and **defer** attrs

async: Download during the renders phase and execute as soon as downloaded

use it only if the script does not care about the DOM at all…

defer: Download during the parsing phase and execute code only after the page is completely parsed

* + - * 1. Minimize DOM manipulation

It causes redraw

* + - * 1. Inline scripts (in HTML) block rendering
  1. Optimized Code
     1. Phases
        1. Loading (files…)
        2. Scripting (V8 Parse + Compile)
           1. Execute
        3. Rendering
        4. Painting
        5. Other
        6. Idle
     2. **AoT** vs **JIT**
        1. Browsers do JIT, Angular came out with AoT
           1. AoT lowers “Scripting” time
     3. Performance budget
        1. Periodically test site performance on various devices (hardwares)
     4. Holy Grails
        1. Time to Meaningful Paint
        2. Time to Interactive
     5. **Code Splitting (Progressive Bootstrapping)**
        1. In HTTP/1 times there was one big bundle js file
        2. With HTTP/2 we can use small js chunks
           1. A minimally functional page +
           2. More resources for **lazy loading** or extra features
        3. **Production build**s (minified, uglified, splitted)
        4. **Dynamic import**
        5. **Route based chunking** and/or **Component based chunking!**
     6. **Tree Shaking**
        1. **During build time unused code gets removed**
        2. <https://web.dev/reduce-javascript-payloads-with-tree-shaking/>
  2. PWA (Progressive Web App)
     1. <https://appsco.pe/>
     2. Lighthouse
     3. 3 pillars
        1. https
        2. app manifest
           1. <https://developer.chrome.com/docs/lighthouse/pwa/viewport/>
           2. <https://realfavicongenerator.net/>
        3. service worker
           1. programmable proxy
           2. it enables us to work our pwa-s offline
           3. it intercepts any request made to the network and checks to see if you really need to communicate to the network

then the service works tries to access the cache API

* + 1. **Github Pages**

1. **Network latency**
   1. Minimize files
      1. Text
         1. Minify / Uglify
      2. Images
         1. @media queries
         2. Proper file format
            1. PNG
            2. GIF
            3. JPG
            4. SVG
         3. Webp, etc..
         4. CDN like imigx
         5. Remove metadata (exif)
            1. <https://www.verexif.com/en/>
   2. Minimize Delivery
      1. Less trips
         1. Limit the number of files
            1. Bundling
      2. HTTP/2
         1. <https://developers.google.com/web/fundamentals/performance/http2/>
      3. HTTP/3
   3. How to debug?
      1. Slow 3G throttling!
2. Server side (backend processing)
   1. CDNs
   2. Caching
   3. Load Balancing
   4. DB Scaling
   5. GZIP

PageSpeed Insights

<https://pagespeed.web.dev/>

WebPageTest

<https://www.webpagetest.org/>

React (Frameworks)

(jQuery was very imperative)

* Components
* One-way data flow
* Virtual DOM
  + Can work outside of the browser too (native)

State Management

* State is kind of a memory, the app has to remember things in order to be able to be interactive
* State describes how our app should look
* Keeping state in a store
* Redux: components don’t have state, only get props from a central state
* State management is inspired by databases (CQRS, etc.)
* It is useful for sharing data between containers
* 3 principles:
  + Single source of truth
  + State is read only!
  + Changes only via pure functions
* Action -> Reducer -> Store -> Make changes
* **Flux Pattern**: Action -> (Middleware) -> Dispatcher -> Store -> View | **One-Way Data-Flow**
* Before Flux**:** 
  + **MVC:** Action -> Controller -> Model -> View
    - Problem: View change can trigger model change!

Testing

* Testing is a method in software development where individual units of source code, assets or programs are tested to see whether they work properly.
* TDD: Test Driven Development
* Categories
  + **Unit Tests**
    - Most important, easiest and cheapest to implement
    - Test individual functions or classes
    - Functional Programming, Pure Functions, very easy to test
      * Deterministic
      * Pure Function Component: Props -> View
    - Write separated code
    - Contracts: **unit tests do not test the so called contracts!**
    - Run on save, or commit.
  + **Integration Tests**
    - Testing different parts of the application working together
    - Cross-communication between units
    - Stubs, mocking db calls
    - Impossible to cover everything
  + **Automation (UI) Tests**
    - Testing real life behaviour
    - Humans could test this too
    - They always run in a browser(-like) environment
    - The hardest to set up (different environments, edge cases)
* **Testing** Libraries
  + Jasmine
  + Jest
  + Mocha
* **Assertion** Libraries
  + Jasmine
  + Jest
  + Chai
* Test **Runner**
  + …
  + Karma (allows tests to run in the browser)
    - Puppeteer
    - jsdom – DOM-like API
* **Mocks** (fakes functions), **Spies** (information about functions), **Stubs** (replaces functions)
  + …
  + Sinon.js
* **Code Coverage** (what percentage of the source code is tested)
  + Istanbul
* **BDD**: Behaviour Driven Development (we expect some function from our function)
* Jest is maybe better then Jasmine and created by Facebook
* Misc

Misc

**Progressive Tooling**

tools that can be used to improve page performance

<https://progressivetooling.com/>

* npm update!