## 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
  + Models the application state as a single JS Object
  + Could be used independently from React too!
* 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
  + Action
    - A plain JS obj that must have a “type” key
  + Reducer
    - A function that accepts the state and an action and returns the new state.
  + Store
    - A big object, that represents the state of the entire application
    - The createStore() method accepts the root reducer as a parameter
  + dispatch()
    - in order to change the state, one must dispatch an action on the store
  + getState()
    - a method on the store to get the actual state
  + listen()
    - we can pass callback to the listen method of the store to subscribe and handle state 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
  + With a mock we can fake a function and pretend to run
    - jest.fn().mockReturnValue(**Promise.resolve()**)
      * .toBeCalledWith()
* **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
  + DRY is not a problem
    - More tests are better
    - If not dry it means it is more readable
    - It will not go to production
  + describe ()- > it()
  + async tests
    - (done) => {} OR return a promise
    - expect.assertions(1)
  + <https://github.com/sapegin/jest-cheat-sheet>
  + toMatchSnapshot()
  + beforeEach()
  + <https://medium.com/@eugenkiss/lean-testing-or-why-unit-tests-are-worse-than-you-think-b6500139a009>

CSR (SPA) vs SSR vs SSG

<https://developers.google.com/search/docs/crawling-indexing/javascript/javascript-seo-basics>

* React
  + Gatsby
  + Next
* **CSR** Client-Side Rendering
* **SSG** Static Site Generation
  + pre-rendering at build time
* **SSR** Server-Side Rendering
  + pre-render page at each request

## Security

<https://www.hacksplaining.com/lessons>

<https://watchyourhack.com/>

### Injection

* SQL injections
  + Parameterize Queries
    - Prevent statements
* What to do?
  + Sanitize input
  + ORMS (Object Relational Mapper)

### 3rd party libraries

* + npm audit
    - vulnerabilities
    - **deep nested dependency tree issue could arise!**

### Logging

* insufficient logging and monitoring allows attackers without noticing!
* if something is suspicious we have to log it
* aim: getting information about how users use your app / system
* **node**
  + **morgen**: middleware for **express**
  + **winston:** logger for everything…

### HTTPS

<https://code.tutsplus.com/tutorials/http-the-protocol-every-web-developer-must-know-part-1--net-31177>

<https://www.tutorialspoint.com/http/http_header_fields.htm>

* SSL/TLS Certificates
* Secured tunnel with encrypted messages

### XSS & CSRF

#### XSS (Cross Site Scripting)

* Solution: Sanitizing!
* [https://www.hacksplaining.com/exercises/xss-stored#](https://www.hacksplaining.com/exercises/xss-stored)
* <https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP>

#### CSRF (Cross-Site Request Forgery)

* <https://www.hacksplaining.com/exercises/csrf>
* Headers:
  + Content Security Policy: “script-src ‘self’”
  + Secure: prevent cookies accessed by other scripts
  + HTTP Only: Only HTTP Methods can access the cookies, no JavaScript can.
* Node middleware: **csurf**

### Code Secrets

#### Environment variables

…

#### Commit History

* Never commit an API key or password!

### Secure Headers

* node: **helmet**

### Access Control

* happens after successful authentication
* **CORS** – Cross-Origin Resource Sharing
  + allow API access only to specific domains
* **[Principal of Least Privilige]** – add only permissions which are absolutely necessary

### Data Management

* Always have backups preferably on different locations
* Not necessarily have to encrypt every column
* Hashing passwords: bcrypt, scrypt, Aragon2
  + Hash
  + Compare (forced delayed against brute force attacks)
* Encrypting databases: pgcrypto

### Don’t Trust Anyone

* node: ratelimiter

### Authentication

* make sure that they are who they say they are

## Code Analysis

* what to do with a codebase seen at the first time
* before even doing anything, look at the package.json
* always start with the big picture first!
* Always test our assumptions!

## Databases

<https://www.udemy.com/course/the-complete-web-developer-zero-to-mastery/learn/lecture/8853500?start=15#overview>

* **DBMS** – Database Management System
  + Collections of programs which allows us to access databases and work with data
* **General Principles**
  + It is preferred to have some consistency while storing the data, eg.: uppercase or lowercase first letter when saving names….
* **Relational** Databases
  + <https://www.khanacademy.org/computing/computer-programming/sql>
  + **Types**
    - **PostgreSQL**
      * Data Types: <https://www.postgresql.org/docs/current/datatype.html>
      * Install on WSL
        + GUI: DBeaver
      * Start on WSL
        + `sudo systemctl start postgresql` will not work!
        + `sudo service postgresql` status will do the job!
      * Create a database (on WSL)
        + psql

-l: list databases

\d: list tables

* + - * + Created
        + When it's first installed, PostgreSQL just has the 'postgres' user, and the way to initially enter PostgreSQL is by typing sudo su - postgres , and then psql . After Andrei creates the 'test' database, we can create a user with the same name as our current logged in user, to be a database administrator. This way we can just type in psql 'test' from the command line and enter the database without the need of logging in as the 'postgres' user, just like Andrei does in the lecture. This can be done with CREATE USER your-user-name-here WITH SUPERUSER; , and we can verify that he was created with \du . Now we can exit by typing \q and then exit , and enter our database just like Andrei does, with psql 'test' .
        + CREATE TABLE users (name text, age smallint, birthday date);
      * Use GUI Client on WSL
        + <https://stackoverflow.com/questions/14588212/postgresql-resetting-password-of-postgresql-on-ubuntu/14588440#14588440>
      * SQL commands / functions
        + **CREATE TABLE** users (name text, age smallint, birthday date);
        + **INSERT INTO** users (name, age, birthday) **VALUES** ('Andrei', 31, '1989-12-07');
        + **SELECT** name, age, birthday **FROM** users;
        + SELECT \* FROM users;
        + **ALTER** TABLE users ADD score smallint;
        + **UPDATE** users **SET** score = 50 **WHERE** name='Andrei';
        + UPDATE users SET score = 99 WHERE age = 37 OR age = 45;
        + SELECT \* FROM users WHERE name **LIKE** 'B%';
        + SELECT \* FROM users **ORDER BY** birthday **ASC**;
        + SELECT **AVG**(age) FROM users;
        + SELECT **SUM**(age) FROM users;
        + create table login(
        + ID **serial** **not null primary key,**
        + secret **VARCHAR (100)** not null,
        + name text **unique** not null,
        + );
        + SELECT \* FROM users **JOIN** login **ON** users.name = login.name;
        + **DELETE FROM** users WHERE name = 'Sally';
        + **DROP TABLE** login;
        + **DEFAULT**
        + **TIMESTAMP**
    - Two or more tables with columns and rows
    - Each row represents an entry
    - Each column store a very specific type of information
  + The relation between tables and field is called schema
  + Primary key: identifies each row in a table
  + Foreign key:
  + With SQL we can communicate with the databases
  + nodejs: knex.js / increment()
  + Transactions
    - Transactions are an important feature of relational databases, as they allow correct recovery from failures and keep a database consistent even in cases of system failure.
    - All queries within a transaction are executed on the same database connection, and run the entire set of queries as a single unit of work.
    - Any failure will mean the database will rollback any queries executed on that connection to the pre-transaction state.
* **NoSQL**
  + Types
    - MongoDB
      * Document Oriented
      * Has its own MongoDB query language
    - Redis
  + We do not need to have a pre-defined schema
  + Allow greater flexibility

## Docker

The problem: Each service or container may have its own requirements. From different node versions, two conflicting library dependencies, and when you add in the fact that every developer's machine and environment is different.

**Docker bundles your application into an image, a standalone executable package, and executes this image in a container and the environment inside of the container is completely isolated from the host machine.**

of the host machine.

### Containers

We need a way for us to be able to run our programs and our apps in all the environments possible. And this is where containers come in.

Web apps are built around the idea of containers, the nice small boxes that can be run anywhere.

**The environment inside of the container is completely isolated from the host machine!**

#### Image

How it works? We create a container with Docker and inside of it, we hav e an image.

An image is what Docker uses to bundle your application into a standalone package that can live inside of a container.

So within this image, I can say I want a node server.

### Microservices Architecture

Products composed of multiple layers. These layers can be considered services.

Each their own container, doing its own thing and communicating with each other to make the whole system work.

### History

#### Virtual Machines

We had tools like VMware or virtual box and virtual machines are what we call **sandbox environments**, which contains a full fledged computer with its virtual hardware operating system, kernel software.

Problem: Sometimes it took a few minutes to even boot up these applications!

#### Containers

They are designed, unlike virtual machines, for running just single applications on each container because they're so small and thin. Containers use the hosts operating system. They don't have their own OS like we have with VMs.

### DockerHub

Like npm in javascript world

### Dockerfile

* FROM node:8.11.1 //tells Docker to use the node image from Dockerhub (this is the parent image of our image)
* CMD [“/bin/bash”] // tells what to run in the container
* docker build -t supercoolcontainer . *// creates an image based on the Dockerfile with the given tagname*
* docker run -it supercoolcontainer
  + the -it flag allows us to enter the docker container
* docker run -it -**d** supercoolcontainer // running in the background
* docker ps // lists currently running containers
* docker exec -it [ID] bash // access the container
* docker stop [ID]
* WORKDIR /usr/some/cool/path // defining the working directory
* COPY ./ ./ // copies everything from the WD to the container
* RUN npm install
  + CMD vs RUN: a Dockerfile can only have one CMD, usually at the end of the file
* docker run -it -p 3000:3000 supercoolcontainer
  + exposing a port / port binding / port forwarding
  + localhost refers to our computer and our computer does not know about our containers

### Docker compose

Docker Compose launches multiple containers. (Separate services under one command.)

docker-compose.yml

* docker-compose build
* docker-compose up
* docker-compose down

## Misc

### Progressive Tooling

tools that can be used to improve page performance

<https://progressivetooling.com/>

### State of JavaScript

<https://stateofjs.com/en-us/>

### Stack Overflow Annual Developer Survey

<https://insights.stackoverflow.com/survey/>

### A Developers Morning Routine

* <https://www.producthunt.com/>
  + Check out cool products for inspiration and developer tools with over 200 upvotes on Product Hunt from the previous day.

### The Real Web Developer Roadmap

* Part 1: <https://youtu.be/57GuRoJ5Bfw>
* Part 2: <https://youtu.be/aeKQy_08fpk>
* Diagram: <https://coggle.it/diagram/XgtihGj7x4Fvucp6/t/%F0%9F%9A%80%F0%9F%91%A9%E2%80%8D%F0%9F%92%BB-web-development-%F0%9F%91%A8%E2%80%8D%F0%9F%92%BB%F0%9F%9A%80/24016189368f9b6c68d536238aa1e5d26260a76147667cfa043fec9e613d129f>
* npm update!