[1. General 2](#_Toc329627)

[2. Terminology 3](#_Toc329628)

[3. Internet 4](#_Toc329629)

[3.1 Ports: 4](#_Toc329630)

[3.2 HTTP 4](#_Toc329631)

[4. Design 5](#_Toc329632)

[4.1 Full App Design 5](#_Toc329633)

[4.2 Single Page Design 5](#_Toc329634)

[4.3 Structure and Naming Convention 6](#_Toc329635)

[4.3.1 CSS File Naming Conventions 7](#_Toc329636)

[5. HTML 8](#_Toc329637)

[5.1 Security 8](#_Toc329638)

[5.2 Testing Your Web Page 8](#_Toc329639)

[5.2.1 Caching 8](#_Toc329640)

[5.2.2 Android 8](#_Toc329641)

[5.2.3 iOS 8](#_Toc329642)

[5.3 Security 9](#_Toc329643)

[6. HTML and CSS Frameworks 10](#_Toc329644)

[6.1 Bootstrap – 10](#_Toc329645)

[7. Testing 11](#_Toc329646)

[8. Security 12](#_Toc329647)

[8.1 Security in HTML 12](#_Toc329648)

[8.2 Security in Python Script 12](#_Toc329649)

[8.3 Security in Java Script 12](#_Toc329650)

[9. Frameworks 13](#_Toc329651)

[10. Web Servers 14](#_Toc329652)

[10.1 Session 14](#_Toc329653)

[10.2 BaseHTTPServer 14](#_Toc329654)

[11. Rest (REpresentational State Transfer) 15](#_Toc329655)

[11.1 RESTFul Web Services 15](#_Toc329656)

[11.2 15](#_Toc329657)

[12. Internet of Things (IoT) 16](#_Toc329658)

[12.1 The IoT Protocols 16](#_Toc329659)

[12.2 XMPP 16](#_Toc329660)

[12.3 MQTT Protocol 17](#_Toc329661)

[12.4 CoAP (Constrained Application) Protocol 18](#_Toc329662)

[12.5 Comparison 19](#_Toc329663)

[12.5.1 Comparing Web and IoT Protocols 19](#_Toc329664)

[13. Web Interface for Embedded Devices 21](#_Toc329665)

[13.1 Embedded Web Servers 22](#_Toc329666)

[13.2 Scripts Languages Recommended for Embedded Systems 22](#_Toc329667)

[13.3 Security 22](#_Toc329668)

[14. Tools and Utils 24](#_Toc329669)

[14.1 Yahoo Query Language 24](#_Toc329670)

# **General**

<https://www.youtube.com/watch?v=gVXcqO9A1vo>

<https://coggle.it/diagram/WMMEvSoNyAABBX2w/t/web-development-in-2018/b97ca171d59ba2ab3b7ea8da244a8ed3a154ffa067568635fe2676068a1d44d0>

# **Terminology**

|  |  |
| --- | --- |
| **Term** | **Explanation** |
| **AJAX** | **A**synchronous **Ja**vascript and **X**ML - allows updating a webpage without reloading all of it. Uses a browser built-in XMLHttpRequest object to request data from the server and Javascript to display it. |
| **CSS** | Cascading Style Sheets  style sheet language used for describing the look and formatting of a document written in a markup language. |
| **CGI** – Common Gateway Interface | a standard way for [web servers](https://en.wikipedia.org/wiki/Web_server) to interface with executable [programs](https://en.wikipedia.org/wiki/Program_%28programming%29) installed on a server that [generate web pages dynamically](https://en.wikipedia.org/wiki/Dynamic_web_page). Such programs are known as *CGI scripts* or simply *CGIs*; they are usually written in a [scripting language](https://en.wikipedia.org/wiki/Scripting_language), but can be written in any [programming language](https://en.wikipedia.org/wiki/Programming_language). |
| **CRUD** – | Create (INSERT INTO in DB)  Read (SELECT in DB)  Update (UPDATE in DB)  Delete (DELETE in DB)– The main functions that are used in a data  driven web sites.  Want to learn more about the critical rendering path? Check out [Website Performance Optimization](https://www.udacity.com/course/ud884), taught by Cameron and Ilya Grigorik, a performance engineer at Google. |
| **DOM** | Document Object Model  a cross  platform and language  independent interface that allows programs to dynamically access and update the content, structure and style of objects in HTML (and other markup languages). The nodes of every document are organized in a tree structure, called the DOM tree.  The DOM Tree is what you see in the developer tools in the browser. This is the HTML + any inheritance + any scripts processing/changes of the HTML. |
| **HTML** | HyperText Markup Language  the standard markup language used to create web pages. The documents of the Web  content and structure of web pages. |
| **Modal Window** – | any box that pops up and interrupts an action you are attempting to complete (like a toast with ok/close buttons) |
| **Toast** – | a box that shows up to notify user of something and then fades away on its own, without needing any user interaction. |

# Internet

## Ports:

Most computer supports ports 0-65,536

Ports 0-10,000 are usually assigned to operating system for specific use

Port 80 is the most common for web servers. Port 8080 is also commonly used.

When we have the client and the server application on the same machine, we use the term localhost with IP 127.0.0.1.

## HTTP

### Caching Control

Through various iterations of the HTTP protocol, a few different cache-focused headers have arisen with varying levels of sophistication. The ones you probably still need to pay attention to are below:

1. **Expires**: The Expires header is very straight-forward, although fairly limited in scope. Basically, it sets a time in the future when the content will expire. At this point, any requests for the same content will have to go back to the origin server. This header is probably best used only as a fall back.
2. **Cache-Control**: This is the more modern replacement for the Expires header. It is well supported and implements a much more flexible design. In almost all cases, this is preferable to Expires, but it may not hurt to set both values. We will discuss the specifics of the options you can set with Cache-Control a bit later.
3. **Etag**: The Etag header is used with cache validation. The origin can provide a unique Etag for an item when it initially serves the content. When a cache needs to validate the content it has on-hand upon expiration, it can send back the Etag it has for the content. The origin will either tell the cache that the content is the same, or send the updated content (with the new Etag).
4. **Last-Modified**: This header specifies the last time that the item was modified. This may be used as part of the validation strategy to ensure fresh content.
5. **Content-Length**: While not specifically involved in caching, the Content-Length header is important to set when defining caching policies. Certain software will refuse to cache content if it does not know in advanced the size of the content it will need to reserve space for.
6. **Vary**: A cache typically uses the requested host and the path to the resource as the key with which to store the cache item. The Vary header can be used to tell caches to pay attention to an additional header when deciding whether a request is for the same item. This is most commonly used to tell caches to key by the Accept-Encoding header as well, so that the cache will know to differentiate between compressed and uncompressed content.

# Design

## Full App Design

1. What pages does this app need (pen and paper)?
2. How will every page will look like (pen and paper)?
3. Choose URL for every page

## Single Page Design

1. Look for natural boxes – boxify.  
   You can use ‘\*{outline: 1px solid red !important;’ } in CSS to see the boxes.
2. Look for repeated styles & semantic elements – what should be headers/divs etc.
3. Write your HTML. Make sure to create it with:
   1. Responsive design (think mobile)!
   2. Accessibility
4. Apply Styles (from Biggest to smallest)
5. Fix things
6. Check on other browsers, window sizes etc.  
   To test differenct browsers and devices you can use the developer tools on Chrome or cloud-based services such as: <https://www.browserstack.com/>
   1. Make sure your page looks good at all sizes (phone, tablet, computer)
   2. Make sure your touch targets are easy to hit (see tap targets)
7. Validate your HTML and CSS:  
   To verify HTML: <http://validator.w3.org/#validate_by_input>  
   To verify CSS: <http://jigsaw.w3.org/css-validator/#validate_by_input>

**Important Notes:**

1. **Start Small:**  
   Create your 1st design for the smallest screen it should work on (phone?)  
   After it is done, move to the 2nd smallest screen until there is no need to create any bigger designs.  
   This will force you to prioritize what content is actually important to your users and has to appear in all designs!! It will also force you to think about performance right from the start!  
   It is much harder to re-format your big screen design to a smaller screen.
2. **Accessibility:**

Make sure that you design your web pages to be accessible:

* + 1. have well-organized code that uses appropriate markup
    2. ensure text alternatives exist for non-text and visual content
    3. create an easily-navigated page that's keyboard-friendly

A great resource for your projects going forward is the W3 Consortium's Web Content Accessibility Guidelines (WCAG). They set the international standard for accessibility and provide a number of criteria you can use to check your work: <https://www.w3.org/WAI/standards-guidelines/wcag/>

If you'd like to learn more about responsive web design, please check out the Udacity [Responsive Web Design](https://www.udacity.com/course/responsive-web-design-fundamentals--ud893) or [Responsive Images](https://www.udacity.com/course/responsive-images--ud882) courses.

## Structure and Naming Convention

#### Project Structure

I've noticed a lot of frontend developers are moving away from css and js in favor of styles and scripts because there is generally other stuff in there, such as .less, .styl, and .sass as well as, for some, .coffee. Fact is, using specific technology selections in your choice of folder organization is a bad idea even if everyone does it. I'll continue to use the standard I see from these highly respected developers:

1. assets/html
2. assets/images
3. assets/styles
4. assets/styles/fonts
5. assets/scripts
6. assets/data ; some xml files and other stuff
7. assets/media ; for music etc
8. assets/lib (or 3rd-party); this one is intended for code you don't make or modify, the libraries as you get them
9. assets/lib-modded (or 3rd-party-modified); this one is intended for code you weren't expected to modify, but had to, like applying a workaround/fix in the meantime the library provider releases it
10. assets-server (or assets-server or assets-local); this one is intended for content used server side, not to be used by the client, like libraries in languages like PHP or server scripts, like bash files
    1. assets-server**/fonts**
    2. assets-server**/lib**
    3. assets-server/lib-modded

And their destination build equivalents, which are sometimes prefixed with dest depending on what they are building:

1. ./
2. images
3. styles
4. styles/fonts
5. scripts

This allows those that want to put all files together (rather than breaking out a src directory) to keep that and keeps things clearly associated for those that do break out.

I actually go a bit futher and add

1. scripts/source
2. scripts/after

Which get smooshed into two main-source.min.js and main-after.min.js scripts, one for the header (with essential elements of normalize and modernizr that have to run early, for example) and after for last thing in the body since that javascript can wait. These are not intended for reading, much like Google's main page.

If there are scripts and style sheets that make sense to minify and leave linked alone because of a particular cache management approach that is taken care of in the build rules.

These days, if you are not using a build process of some kind, like [gulp](http://gulpjs.com) or [grunt](http://gruntjs.com), you likely are not reaching most of the mobile-centric performance goals you should probably be considering.

#### Files Naming

Use BEM: <http://getbem.com/introduction/>

For large sites where css might define a lot of background images, a file naming convention for those assets comes in really handy for making changes later on.

For example:

**[component].[function-description].[filetype]**

**footer.bkg-image.png**

footer.copyright-gradient.png

according to google style-guide it is **not recommended to use type selectors** as part of the name (e.g. div.error) since it makes it fragile if the type change (e.g. article.error).

#### CSS Files Naming

### CSS File Naming Conventions

If you only will ever have one CSS file on the site, you can name it whatever you like. One of the following is preferable:

**styles.css or default.css**

If your website will use multiple CSS files, name the style sheets after their function so it is clear exactly what the purpose of each file is. Since a webpage can have multiple style sheets attached to them, it helps to divide your styles into different sheets depending upon the function of that sheet and the styles within it. For example:

1. **Layout vs. design**

layout.css design.css

1. **Page Sections**

main.css nav.css

1. **Whole site with sub-sections**

mainstyles.css subpage.css

If your website uses a framework of some kind, you will likely notice that it uses multiple CSS files, each dedicated to different portions of the pages or aspects of the site ([typography](https://www.lifewire.com/what-is-typography-3467428), color, layout, etc.).

# HTML

Mozilla Developer Network has a great [article](https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Introduction) on HTML and the [DOM](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model).

<div> - dividing the page into boxes.

## Security

1. Never send data using ‘GET’ (all parameters are sent in clear!)

## Testing Your Web Page

It’s very important to test your web page on different devices

### Caching

If you change your design, a lot of browser won’t refresh because they cache the page.

To do a hard-refresh for the page in most Windows and Linux browsers:

1. Hold down Ctrl and press F5.

In Apple Safari:

1. Hold down ⇧ Shift and click the Reload toolbar button.

In Chrome and Firefox for Mac:

1. Hold down both ⌘ Cmd+⇧ Shift and press R.

### Android

To set up a test and debugging environment on your Android device:

* + - 1. [Download and install Chrome Canary](http://www.google.com/intl/en/chrome/browser/canary.html) (it will not interfere with your regular Chrome)

On Linux, the [Chromium Dev channel](http://www.chromium.org/getting-involved/dev-channel) is similar to Canary.

* + - 1. Learn about [Remote Debugging on Android with Chrome](https://developer.chrome.com/devtools/docs/remote-debugging)

Chrome Canary is the developer version of Chrome. It looks and acts like the regular Chrome browser, but it includes new and experimental features that haven't been released yet. We recommend analyzing websites with Canary to take advantage of the latest tech. However, be warned that Canary isn't guaranteed to be stable, so expect crashes and occasional bugs.

### iOS

iOS WebKit Debug Proxy: <https://github.com/google/ios-webkit-debug-proxy>

Please note that on the forums, there is a discussion continuing about ios-webkit-debug-proxy. Depending on your version of canary, if you're using it, it might take a lot of time and some students suggest trying Safari Dev Tools and point to links like this:

<https://www.smashingmagazine.com/2014/09/testing-mobile-emulators-simulators-remote-debugging/2/>

Remember you can run in simulator mode in Chrome Dev Tools.

## Security

# HTML and CSS Frameworks

**Full Featured Frameworks:**

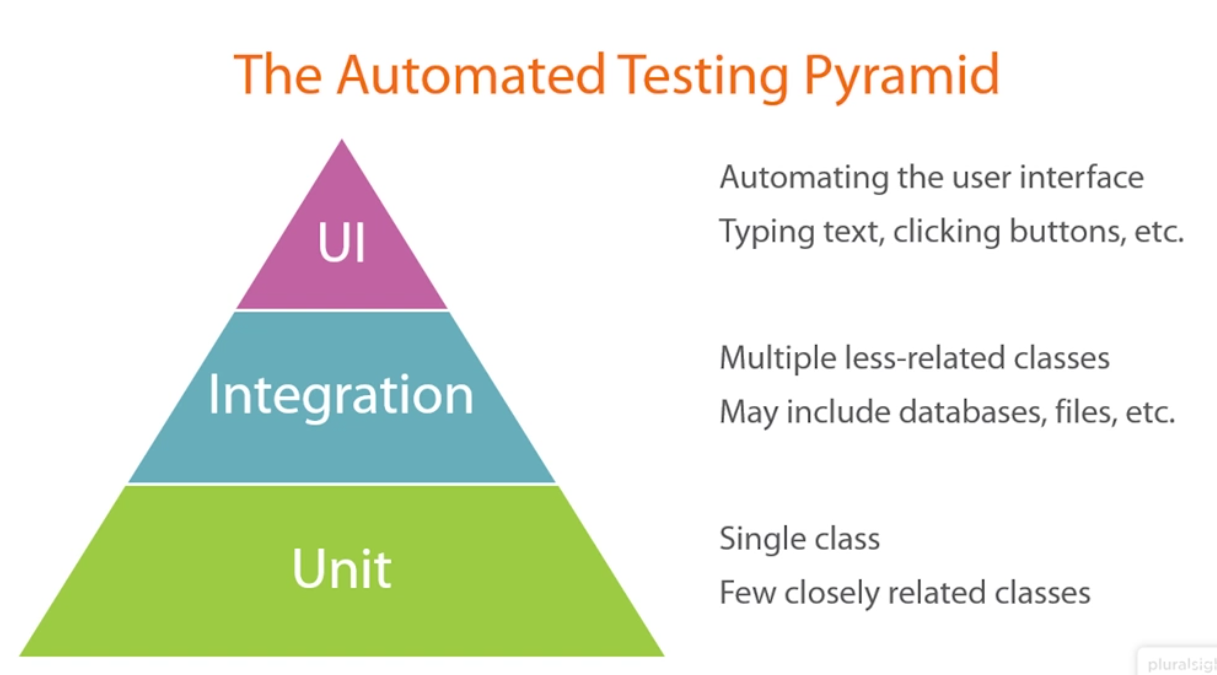
Bootstrap: <http://getbootstrap.com/>   
Foundation: <http://foundation.zurb.com/>   
Yaml: <http://www.yaml.de/>

**Lightweight Frameworks:**

960 Grid: <http://960.gs/>   
Suzy: <http://susy.oddbird.net/>

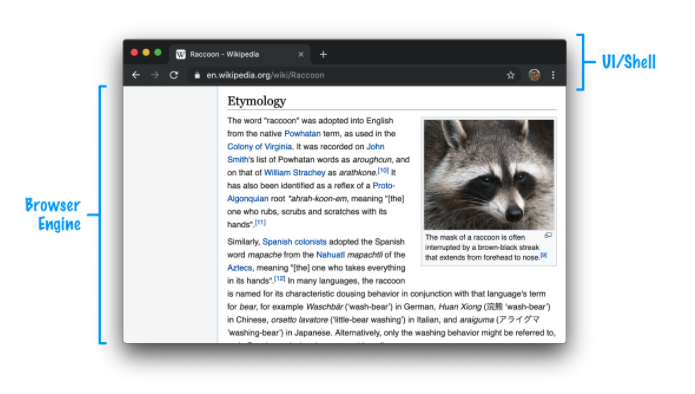
## Bootstrap –

# Testing



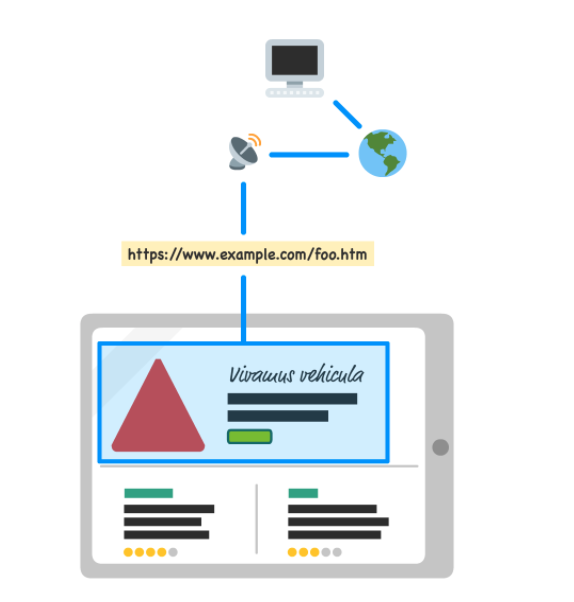
# Webview

**A WebView is an embeddable browser that a native application can use to display web content.**   
Note: native apps are not only on Android and iOS. windows/mac applications are also native apps.



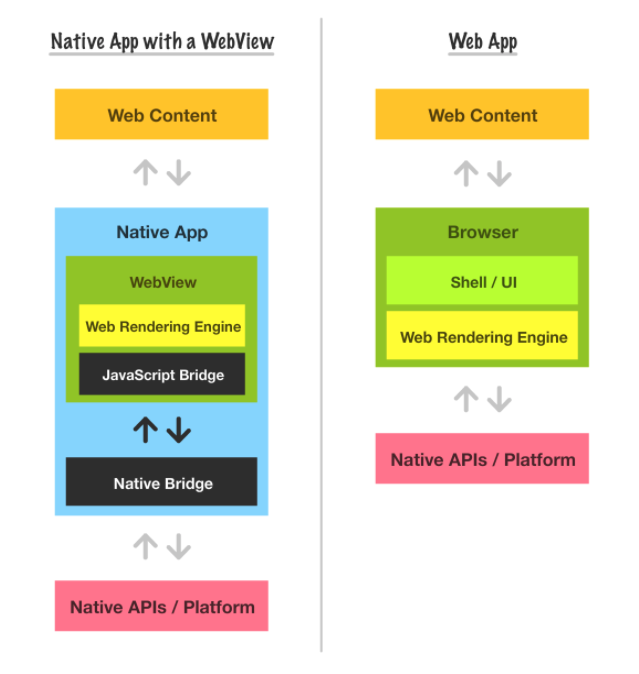
A **WebView** is just the **browser engine** part that you can insert sort of like an iframe into your native app and programmatically tell it what web content to load.

Your WebView is almost like a web-friendly island inside a large ocean of nativeness. The contents of this island don't have to be local to your app. Your WebView will commonly load web content remotely from a http:// or https:// location. This means you can take parts (or all) of your web app that lives on your server and rely on the WebView to display it inside your native app:



This flexibility opens up a whole world of code reuse between your browser-focused web app and the parts of your web app that you want to display inside a native app.

...your JavaScript running inside your WebView **has the ability to call native system APIs**. This means you aren't limited by the traditional browser security sandbox that your web code normally has to adhere by. The following diagram explains the architectural differences that make this possible:



the same JavaScript you write for the web will not only work inside your WebView, it can also call into native APIs and help your app more deeply integrate with cool system-level functionality like sensors, storage, calendar/contacts, and more.

For most purposes, you don't have to specially test your web app inside a WebView unless you are calling native APIs. Otherwise, the functionality between what you see inside a WebView is the same as what you would see in the browser, especially if you match the rendering engines:

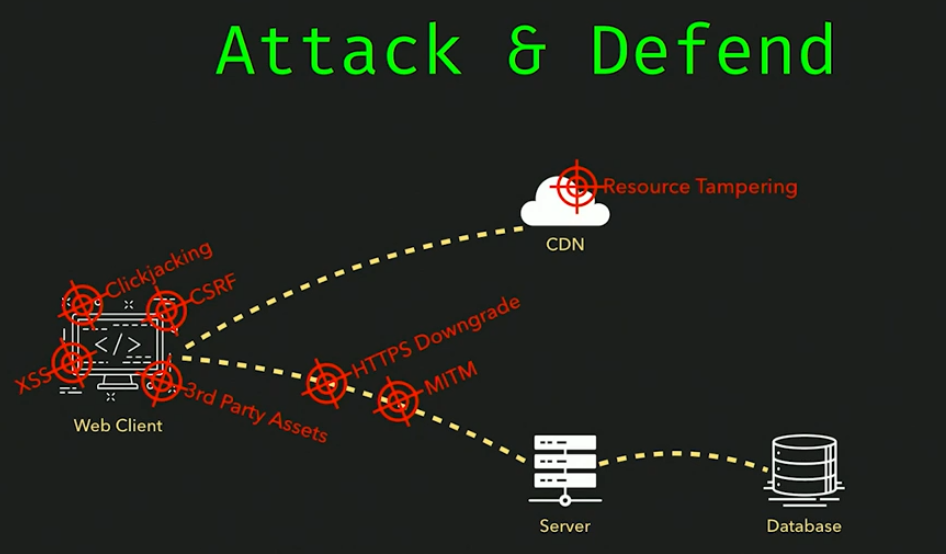
1. On iOS the web rendering engine is **always** WebKit, the same one that powers Safari...and Chrome. Yes, you read that correctly. Chrome on iOS actually uses WebKit under the covers.
2. On Android, the rendering engine under the covers is ***usually*** Blink, the same one that powers Chrome.
3. On Windows, Linux, and macOS, since these are the more permissive desktop platforms, there is a lot of flexibility in choosing the WebView flavor and rendering engine used under the covers. The popular rendering engines you see will be Blink (Chrome) and Trident (Internet Explorer), but there is no one engine that you can rely on. It all depends on the app and what WebView implementation it is using.

# Security

## Testing

You should test your security on a modern browser but also with an old browser that doesn’t protect your site from attacks. A recommended method is to use ie9 for testing on a virtualBox (expires after 90 days) from: <https://developer.microsoft.com/en-us/microsoft-edge/tools/vms/>

## Attacks & Defence



### Cross-Site Scripting: XSS

1. An injection attack where we inject code into text-fields and trick the application into executing it for us. Very prevalent.
2. Allow attacker to read data or perform operations on a user’s behalf.

### Phonegap Volnurability

## Defences

1. Always work only with https. No http at all (sensitive to MITM attacks for example)

## Security in HTML

1. Never send data using ‘GET’ (all parameters are sent in clear!)

## Security in Python Script

1. Validate all the input parameters you receive:
   1. The best solution is to use an existing library:

Import cgi

def escape\_html(s):

return cgi.escape(s,quote=True)

* 1. Manually:

def my\_escape\_html(s):

s = s.replace("&", r"&amp;")

s = s.replace("\"", r"&quot;")

s = s.replace(">", r"&gt;")

s = s.replace("<", r"&lt;")

return s

## Security in Java Script

Validate all the input data:

1. Manually:

newHTML = newHTML.replace(/</g,"&lt");

newHTML = newHTML.replace(/>/g,"&gt");

### JWT

A **JSON web token** (JWT) is JSON Object which is used to securely transfer information over the web (between two parties).

1. Course: <https://frontendmasters.com/courses/secure-auth-jw>
   1. Code sample for React: <https://github.com/chenkie/react-user-authentication>
   2. Code sample for Angular: <https://github.com/chenkie/angular2-user-authentication>
   3. Authentication api: <https://github.com/chenkie/user-authentication-api>

# Frameworks

Web frameworks:

1. Django,
2. Ruby on Rails,
3. Flask .  
   Flask does not force you to use any particular design pattern (e.g. MVC). However, setting up the data base is the Model.

# Web Servers

## Session

A way for a server to save information over multiple Web Pages to create a more personalized user experience.

## BaseHTTPServer

Python library for creating web server.

## Services

Services are used to abstract external API — in many cases server API like the one [provided](https://www.reddit.com/dev/api/) by Reddit. The benefit of this abstraction layer is that API’s change and we want to decouple our code as much as possible from them. If in the future Reddit decides to rename endpoints or change field names, we can hopefully contain the impact on our app to the service alone.

**Rule: Services must be completely stateless.**

# Rest (REpresentational State Transfer)

**REST has become the standard for most Web and Mobile apps.**

REST is a way of sending/getting information from sites without all the heavy HTML and CSS parsing. It uses light-weight formats such as text, JSON and XML instead.

REST is web standards based architecture and uses HTTP Protocol for data communication. It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods. REST was first introduced by Roy Fielding in 2000.

In REST architecture, a REST Server simply provides access to resources and REST client accesses and presents the resources. Here each resource is identified by URIs/ global IDs. REST uses various representations to represent a resource like text, JSON and XML. Nowadays JSON is the most popular format being used in web services.

## RESTFul Web Services

While REST is an architectural style for networked hypermedia applications, it is primarily used to build **Web services that are lightweight, maintainable, and scalable**.

**A service based on REST is called a RESTful service**.

REST is **not dependent on any protocol, but almost every RESTful service uses HTTP** as its underlying protocol.

## 

# GraphQL

1. Frontendmasters course: <https://github.com/FrontendMasters/intro-to-graphql>
   1. <https://github.com/FrontendMasters/intro-to-graphql>
2. Strongly typed (think typescript for your data) spec for a query language and runtime for your data.
3. It can be combined with any API since it’s only a query language. So for example, it can sit in front of the rest api. doesn’t necessarily replaces it.
4. GraphQL is just a Spec
   1. Has many different implementations of it.
   2. GraphQL-JS – js implementation. Open source, created by Facebook
5. Allow the client to specify exactly which fields they want to get back from the server so they don’t have to get all the possible fields in the rest API. For example: a client just want to get the 1st and last name of the users.
6. Enables excellent dev tools and experiences.
7. Great ecosystem.
8. Apollo is a great library for both Server and Client side: <https://www.apollographql.com/docs>

## GraphQL vs Rest

|  |  |  |
| --- | --- | --- |
|  | REST | GraphQL |
| URL | A different URL path for every request (e.g. ../resource/verb/id/..) | One URL. All the request details are in the POST (or GET) body. |
| Returned data structure | Constant and defined by the server’s API definition | Dynamic - determined by the query (request) – depending on which fields the client asked for. |
| Relational Data | Client side code need to make multiple calls and combine the data | GraphQL query language support complex relational query that are resolved by GraphQL |
| Query to controller relationship | Every query is resolved by one controller | Every query can be resolved by many different controllers on the server side. |

### When to use GraphQL

1. When different clients/use cases require different subsets of the data (not all requests require all the data)
2. Unifying multiple sources in the backend and presenting them on the same GraphQL interface to the front end.
3. Resolve performance issues by reducing the amount of data being transferred from the server
4. Rapid prototyping – you can have all your data on the server but play around with what you ask for from the client side.

### When to use Rest

1. Depending on the queries – if they ask for too much data or cyclic data, they can break your server.
2. You can choose which fields to return in REST by using URL parameters:  
   GET /books/1492030716?fields=title,pageCount
3. GraphQL might be an overkill for simple applications with a few fields.
4. Caching graphQL network queries is harder since they all go to the same endpoint.

## Apollo Server

An open source GraphQL server that’s compatible with any GraphQL client.

<https://www.apollographql.com/docs/apollo-server/>

1. Apollo Server also automatically create GraphQL Playground which is a web interface to our GrpahQL server (like coral diver)

## Server Side

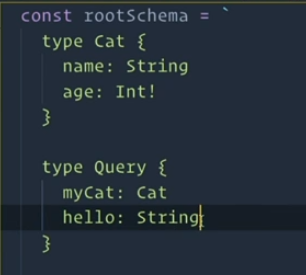
### Schema

This is GraphQL schema – not to be confused with the DB schema:

1. DB schema defines what data the DB holds and how it’s saved on it.
2. GraphQL scheme defines the user interface - which resources are available for querying, how they relate to each other and how you can query them.
3. GraphQL schema sits in front of the DB queries, it validates incoming request queries.
4. Some GraphQL tools create GraphQL APIs based off your DB schema and vice versa.  
   Note: these are helper tools. You shouldn’t blindly expose everything your DB return to the client, just what is actually needed to be exposed.

Schema definition on the server side:

No , between types/fields.



#### SDL (Schema Definition Language)

1. That’s the preferred way to write the GraphQL schema
2. Supported by all GraphQL tools
3. Can be composable – spread over different filed.   
   Note, when expending over different files, use ‘extend’ reserved word before you Query and Mutation definitions so it will combine them together instead of overriding them.
4. Defined in .gql files.

GraphQL support the following types:

1. Scalar types – built in primitives:
   1. String
   2. Int
   3. Float
   4. Bool
   5. ID
2. Enum – like enum in other languages
3. Custom-Scalar type – you can define new scalar types (e.g. Date, Url etc) using the ‘scalar’ reserved word.
4. Object Types (with the ‘type’ reserved word) are user-defined types that can be made out of other object types or scalar types.

Validation on the queries:

1. ! - required field/type

### Query

1. A type that defines all the queries that your API support.

type <name. usually Query> {  
 …  
}

### Mutation

1. Defines all the queries on your API that mutate your data. Usually, mutations will take arguments.

type <name. usually Mutation> {  
 …  
}

### Input types

In order to support inputs for queries/mutations, we need to define the input types to use:

Input <name> {  
 .. the fields of the input. Defined in the same way as in   
}

### Resolvers

1. The resolvers are the end points that actually retrieve the data back to the client (like controllers in a REST API)
2. Every query and mutation must have a matching resolver (with the same name)
3. Types and types’ fields often has resolvers as well (it’s resolvers all the way down…)
4. The incoming query determine which resolvers will be called and in which order

resolvers: {

      Query: {

        myCat() {

          return { name: 'Garfield', age: 5 }

        },

        hello() {

          return "miawoo"

        }

      }

    },

1. If an optional field (without a !) is not returned – it will return null.
2. If a required field (with !) is not returned – it will return an error

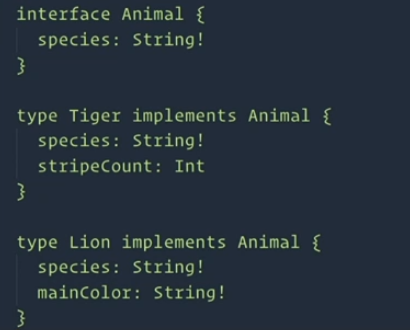
#### Creating Resolvers

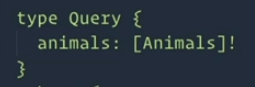
1. Resolver has to return the type as defined in the query/mutation and/or delegate to another resolver
   1. To delegate to another resolver, you need to:
      1. Return {} (empty object)
      2. Create additional resolvers for the types returned by this object
      3. GraphQL will search for the additional resolvers to resolve the query’s required fields.
2. Resolver parameters:
   1. Starting object - what the parent resolver returned or starting value from server.
   2. args – the arguments as appear in the Query schema
   3. context – a shared state object that is passed to all resolvers. It’s created when your server starts (new ApolloServer(..)). It’s a great place to put things that all resolvers need like: user authentication, shared logic, cache storage mechanism, DN model etc.
   4. info – the AST (Abstract Syntax Tree) of the query. This field is used very rarely.
3. Error Handling:
   1. All errors that are thrown from the GraphQL server will return to the client as errors in the query.
   2. Note: your html call will still return 0x200 but in the data itself, you’ll get back the “errors” field. Usually in this case, the “data” field will be null but not all the time.
   3. You should define your own Error objects and translate any thrown exception to your defined client-facing errors that will be both more user friendly and not expose secret information.
4. Can be async

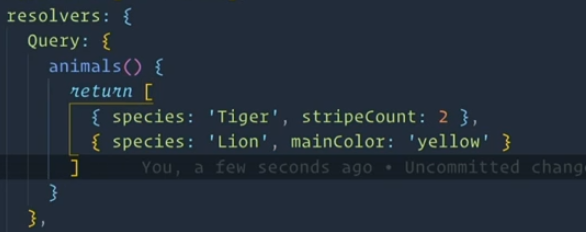
### Interfaces and Fragments

1. Interface – allow defining a common interface that can be implemented by different types. This allows us to return different types in queries as long as they inherit from the same interface.
2. Fragment – the … on syntax n query that allows querying on fragments of the returned object if they exist.

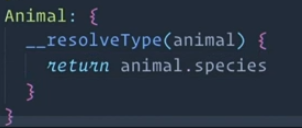
Example:







\_\_resolveType – returns the actual type of the interface so that we can use fragments in our queries:



So now the query:



Returns:



### Unions

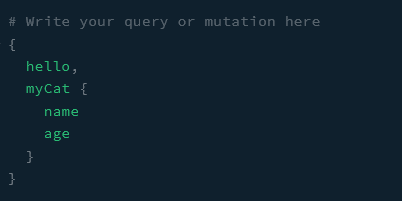
1. Unions allow us to create a type that is composed of many different types where any of them may be fulfilled but they may not have any relation to each other (like union in C)
2. Great for search queries

## Client Side

### Query:

{  
 <query name> {  
 … which fields do I want to get back. No , between fields.   
 }   
 … additional queries. No , between them  
}

For example:



1. We can also run multiple queries/mutations at the same time. They will all run in parallel on the server:  
     
   

### Response:

{  
 “data” : {  
 <query name>: {  
 … returned fields for this query  
 }  
 }  
}

For example:

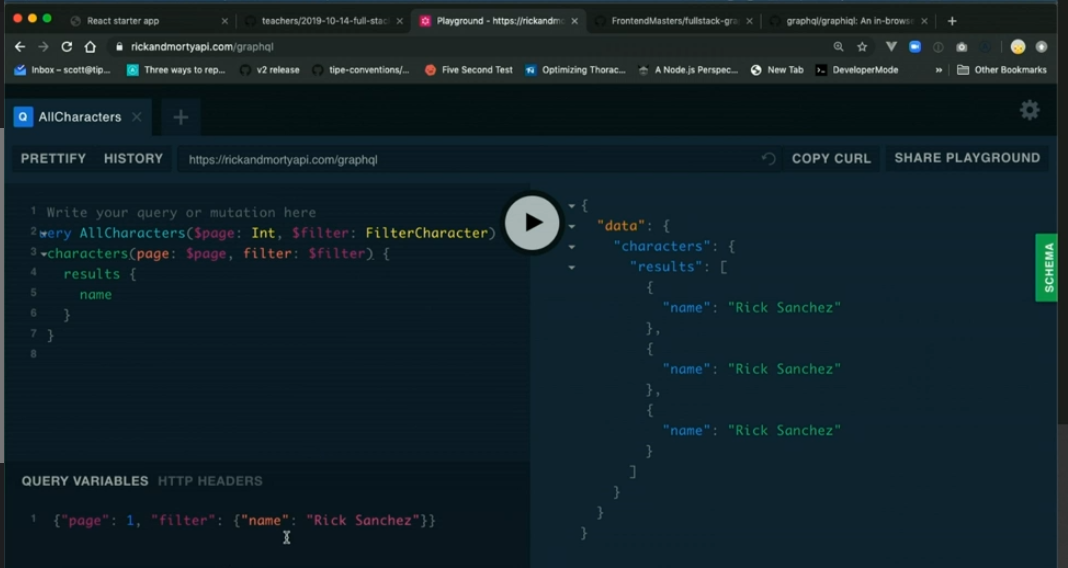


### Operation Names

1. Unique names for your client side Query and Mutation operations.
2. Used for client-side caching, indexing inside of tools like GraphQL playground etc.
3. Like naming functions in JS vs keeping them anonymous.

To define an operation:

query/mutation <QueryName>(<optional parameters with ‘$’>) {  
 <query parameters like before>  
}



### Aliasing

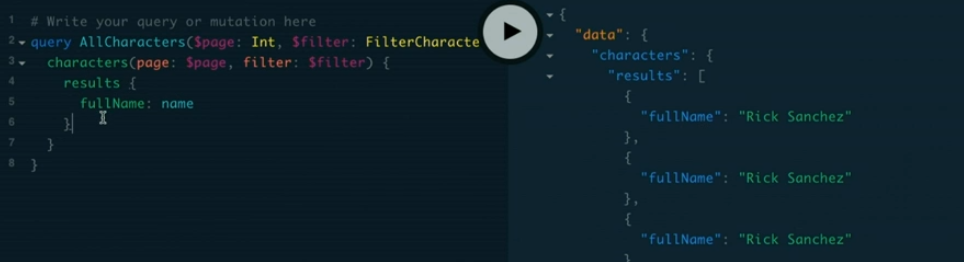
As part of our query/mutation, we can use aliases to get the returned data renames as we define it to be.

Syntax in the query:

Instead of writing <query field> , we’ll write:

<new Name>: <query field>

For example:



Useful for:

1. Unifying your results so that you can always find them with the same structure (e.g. data.results….
2. When running multiple queries in parallel, you might need to rename one (all) of them to prevent collision that will fail or remove one of them.
3. Readability of your results.

### Apollo Client

1. Encapsulate HTTP logic used to interact with GraphQL API.
2. Support client side state management, instead of (and then you can query your state using GraphQL as well) or side-by-side with Redux.
   1. You can have Redux for managing your local state and Apollo to manage your GraphQL state.
3. Have a plug in mechanism for extending its capabilities
4. Framework independent. Can run with any framework as well as vanilla JS.

#### Apollo Cache

Apollo store all the returned nodes (object) from your queries **flat** by a **unique ID**

1. Apollo generate this unique ID from the returned .\_id or .id combined with the \_\_typename
2. Make sure that you always return .id or .\_id from every query so that Apollo will be able to save them.
3. Always return the same fields from queries and mutations:  
   In order to help Apollo optimise its performance, we want to reuse the same fields in our queries so that whatever Apollo cache on one command, it can use on another and it doesn’t actually have to call the server over and over again for every query. This will also allow apollo to update an existing entity in its cache when you mutate it. So for example, if you have a list of all pets with id,name,type and image and you call an updatePet mutation with the same fields, Apollo client will know to update this pet in all its lists. However, if your mutation only return id and name, it will not be able to update its cache.
4. Apollo can’t update its lists (getAllPets) when you create/delete an entity. In order to tell Apollo that it change, we need to do this manually:
   1. We can refetch the matching queries – latency issues.
   2. We can manually update Apollo’s cache after the mutation:   
      <https://www.apollographql.com/docs/react/data/mutations/#the-update-function> – this is the standard way to do this.

### Optimistic UI

Definition: Your UI does not wait until after a mutation operation to update itself. Instead, it anticipates the response from the API and proceeds as if the API call was sync. The API response replaces the generated one. This gives the illusion of your app being really fast.

Apollo provides a simple hook that allows us to write to the local cache as soon as we send the mutation request, without waiting for the server to response (as opposed to the update function that’s called after the server responded): <https://www.apollographql.com/docs/react/performance/optimistic-ui/#gatsby-focus-wrapper>

Note: once the server returns the response, our app will update with the correct value so if we were wrong in our prediction, it will update at this time without an error.

Syntax:

1. Add optimitsticResponse as a parameter to your mutation function call (the one returned from useMutation). Note: you can also put this in the definition of the function (in useMutation call) but your don’t have all the parameters of the call then.
   1. optimisticResponse should include all the fields returned from the mutation request + the \_\_typename field (so that Apollo Client can cache it as an optimistic version of the element).
2. Remove the <Loader /> return for this mutation’s loading stage.

Example:

const [createPet, newPet] = useMutation(CREATE\_PET, {

    update(cache, {data: {addPet}}) {…});

// NOTE: you can also add the optimisticResponse here and then you

// might not need it in createPet call.   
 // or it can serve as a fallback for createPet here.  
 // this depends on your use-case.   
 // Note that when you call createPet, you have the input to pass to the

// Server. So your optimisticResponse will be more realistic

});

const onSubmit = input => {

    setModal(false)

    createPet({variables: {newPet: input},

      optimisticResponse: {

        addPet: {

          \_\_typename: "Pet",

          id: "temp-id",

          name: input.name + "-temp",

          type: input.type,

          img: ""

        }

      }});

}

…

if(newPet.loading) {

// don’t do anything so the app will use the optimisticResponse until

// the server actually answers

//return <Loader />

  }

### Client Side Schema

Apollo Client support adding client-side-only fields/data types into your schema to be handled by GraphQL in the same way it handles the server-side schema.

This is actually a replacement for using Redux with React.

In order to add/expand your schema on the client side:

1. Add the additional types/fields using the gql`` syntax
2. Add the additional resolvers
3. Pass the new types and resolvers to the client when you create it
4. Whenever you access these fields/types, you need to tell Apollo Client that these are client-only fields, using the @client directive

Example:

import gql from "graphql-tag";

/\*\*

 \* Adding local fields to our Schema that will be client-side only,

 \* without updating the server:

\*/

const typeDefs = gql`

    extend type User {

        age: Int

    }

`;

const resolvers = {

    User: {

        age() {

            return 35;

        }

    }

};

…

const client = new ApolloClient({

    link,

    cache,

    typeDefs,

    resolvers

});

…

// Using the new fields:

const GET\_ALL\_PETS = gql`

  query GetPets {

    pets {

      id

      owner {

        id

        age @client

      }

    }

  }

`;

### Fragments

<https://www.apollographql.com/docs/react/data/fragments/#gatsby-focus-wrapper>

Fragments allow us to use re-usable fields that we can use in different queries (similar to the server-side interfaces).

To define a fragment:

* + - 1. Define the fragment using the fragment syntax (see below)
      2. Wherever you want to use the fragment, add it will …fragmentName and add ${FRAGMENT\_CONST} for every query that uses it

Example:

const PET\_DETAILS  = gql`

  fragment PetDetails on Pet {

    id

    name

    type

    img

    owner {

      id

      age @client

    }

  }

`;

const GET\_ALL\_PETS = gql`

  query GetPets {

    pets {

      ...PetDetails

    }

  }

  ${PET\_DETAILS}

`;

### React Integration

In order to integrate GraphQL

## Security

1. Common attacks and how to defend against them: <https://cheatsheetseries.owasp.org/cheatsheets/GraphQL_Cheat_Sheet.html>

### Authentication

There are many ways to authenticate with GraphQL:

1. Lock down the entire API by checking auth outside of GraphQL or when creating the context object
2. If only some of your resolvers need authentication (or need special authentication), you can pass everything they need to authenticate in the context object and have every resolver authenticate independently.
3. Use custom directives in your SDL
   1. This allows defining customer directive on specific fields (like:  
       @auth(role: “Admin”) which can lock down specific fields. Note: you will need to implement any custom-directive you use.
4. Use resolver middleware (in some frameworks that support this like GraphQL yoga). This means that you can have custom code that runs before/after every resolver. You can use this code to do authentication.
5. Every method has it’s benefits and costs. Choose according to your use case.

## Caching

### Caching DB Results on the Server

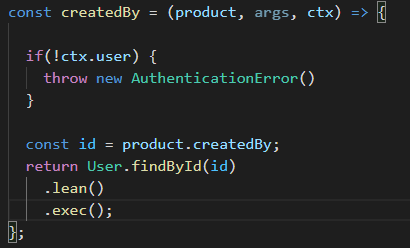
1. DataLoader - created by Facebook for GraphQL – in memory store for caching DB results.
2. Reddis
3. Mongo

### Caching Results at the Network Layer

1. If the variance on the queries coming in is not big, you can cache them yourself using your CDN if you use one.
2. If the variance is too big, the hit/miss ratio might not be worth the additional processing required for caching.
3. You can also use Apollo Engine which is Apollo-specific solution for caching

### Performance

When you have a lot of DB queries at different levels, you might see performance degradation because of the repeated DB access. The way to improve this (where is matters) it to implement everything in the one high-level resolver. This required this resolver to look into the query structure and to create a unique DB query to return everything it needs instead of letting the GraphQL do its thing and automatically call different resolvers to return different pieces of data.

For example, this code is called after the product resolver was called (and called the DB) and this code call the DB again. Instead, the product resolver could have been able to resolve this query as well using query parameters:  
  


## Tools

1. GraphQL Playground – web-interface for your GraphQL. Included with ApolloServer by default. In the browser, just put in the GraphQL URL and it will automatically open the playground for you.
2. Apollo Dev Tools for your browser – allow looking into local Apollo cache etc.
3. Prisma – open source - they create GraphQL for existing DB schemas.
4. Hasura – open source - create GraphQL from existing Postgres DB
5. GraphQL-anywhere – allow you to run GraphQL anywhere, including on the client. For example, for querying the DOM. Note: this has a performance price so think before you use!
6. GraphQL-cli –
7. Graphql-lodash – a directive library you can add to your GraphQL to support many directives inside of your queries on the client side.
8. To simulate a delay in GraphQL requests with Apollo client:

// This delayLink is for testing how our app works when we have // delay on the server for every server request (query/mutation):

// It will receive the request and will delay it by 800ms before actually sending it to the server

const delayLink = setContext(

    request =>

        new Promise((success, fail) => {

            setTimeout(() => {success()}, 800);

            })

);

// a link is a network interface to interface a graphQL server:

const serverLink = new HttpLink({uri: PETS\_SERVER});

// This link is a stack of links that will runn in the order the appear in.

// In this case, delayLink will run before the serverLink

// the reason we do this like this and not through the network dev-tools in the browser

// is that we don't want all the operations to have delays, only the GraphQL requests.

const link = ApolloLink.from([

    delayLink,

    serverLink

]);

// we initiate the cache and client as usual:

const cache = new InMemoryCache();

const client = new ApolloClient({

    link,

    cache

});

1. GraphQL Voyager – give you a graph of your schema. Can be useful to investigate schemas
2. One graph – connect all your APIs and create a GraphQL to use with them
3. Fly.io – javaScript on the CDN. Similar to lambda @ Edge
4. Awesome-graphql list on github as a list of good tools for GraphQL.

# Server->Client Updates

Usually, the client initiate commands to the server and update the UI with the response. But what if we want our server to notify our client that something changes and they need to update?

The main options from <https://medium.com/serverless-transformation/asynchronous-client-interaction-in-aws-serverless-polling-websocket-server-sent-events-or-acf10167cc67> :

**(Long) Polling:**

The client will query the server in a loop every X time to see if there is any new data.

Pros:

1. HTTP: robust and easy to implement

Cons:

1. Wastes bandwidth and server resources for redundant calls if the rate of changes is < than the rate of queries.
2. Doesn’t update the client immediately when there are changes (if critical)
3. If implementing with Lambdas – this can be a deal breaker due to cost.

**WebSocket:**

Your front-end opens a long-lasting bi-directional communication with your back end through a webSocket protocol. Thus, the back end can push a message as soon as necessary.

**Pros**

1. Implementation with Lambdas: WebSocket APIs is an official solution from AWS API Gateway. It handles the connections for you, and only pings your Lambdas on messages.
2. Cost with Lambdas: cheap, using [AWS API Gateway pricing source](https://aws.amazon.com/api-gateway/pricing/#WebSocket_APIs) we calculated that a small app with around 25,000 messages per day, triggering 25,000 Lambdas behind, would cost below $1 per month.

**Cons**

1. WebSocket is a different protocol for delivering data, you need to deal with two different paradigms in your app, HTTP and WS.
2. WebSocket is not automatically multiplexed (compared to HTTP/2). Implementing multiplexing both on the server and the client is a bit complicated.

**Server-Sent Events:**

Your front-end opens a long-lasting, uni-directional communication from your back-end through the HTTP protocol. Here as well, the back-end can push a message as soon as necessary.

**Pros**

1. Simple implementation and data efficient with HTTP.
2. It is automatically multiplexed over HTTP/2 out of the box.

**Cons**

1. Implementation with Lambdas: requires to keep a live connection between the client and your Lambdas in the back-end. Lambdas are not meant to be kept alive for a whole client session. Going around this limit is a hassle.
2. Cost with Lambdas: too costly because of the necessity to keep live Lambdas up and running. This con is a **deal-breaker**.

**AWS AppSync:**

AppSync is a fully managed GraphQL API layer. It handles the parsing and resolution of requests connecting them to different data sources such as Lambdas, DynamoDB or HTTP APIs.

It, out of the box, includes real-time GraphQL subscriptions. To which you can connect with your favourite GraphQL front-end framework such as Apollo or Relay.

**Pros**

1. AppSync is serverless by design.
2. Easy to use, providing GraphQL schema and resolver templates is enough to generate an available GraphQL endpoint which supports GraphQL subscription.
3. Implementation with Lambdas: out of the box.
4. Cost with Lambdas: can be cheaper than WebSocket API as AppSync cost per million connection minutes is $0.08, whereas it’s $0.29 for WebSocket. However the price per million messages is higher ($1.14 for WS against $2 for AppSync).

**Cons**

1. GraphQL only.
2. Vendor (AWS) lock-in.
3. There is [no proper way to implement custom authorizers](https://github.com/aws/aws-appsync-community/issues/2) (only accepting API KEY, AWS IAM, OpenID Connect and AWS Cognito).
4. Hard to pair with an Event-Driven micro-services approach and often results in a monolith or distributed monolith structure.

**What’s the best choice then?**

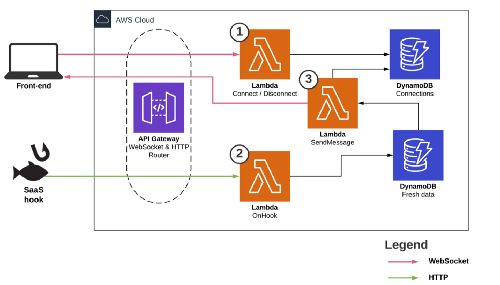
We recommend going for **WebSocket**, because:

1. It’s the only flexible and standard solution officially supported by AWS.
2. It has the smaller cost impact as the connection is kept alive by API Gateway and Lambdas are only triggered on useful events.
3. You’ll find lots of online content and libraries helping you implement WebSocket in a serverless context.

# How to set it up?

**Pre-requesite:** In the context of the Serverless Framework on AWS in TypeScript.**Development time:** This solution takes less than half a day to set-up, test and deploy (not including the front-end).





* Start and keep track of the live WebSocket connection
* An event, like a SaaS hook, triggers a DB update
* A DynamoDB event then triggering a Lambda to notify the front-end of the updated data

**Code example:**

Configure connect and disconnect Lambdas to keep track of the active connections. When you declare the websocket event type for the first time, Serverless Framework will create a new WebSocket API Gateway. $connect and $disconnect are official WebSocket route events.

<https://gist.github.com/xavierlefevre/9ae394f6551c9eb5ed0b7fbe72757d23#file-websockets-connect-disconnect-serverless-yml>

Configure your Lambda connection handlers in your serverless.yml

<https://gist.github.com/xavierlefevre/d733254fad596947cc42993ba8cfb4aa#file-websockets-connect-lambda-ts>

The Lambda called on connect is responsible to keep the connection ID, in your DynamoDB table for instance

<https://gist.github.com/xavierlefevre/d8dd6f8af9bee60ef94bba7dc98523d5#file-websockets-disconnect-lambda-ts>

The Lambda called on disconnect is responsible to remove the connection ID

Then add a Lambda responsible to push WebSocket messages to the front-ends.

<https://gist.github.com/xavierlefevre/07dfde2483b245bbdbc41e9acc7f88ed#file-websockets-send-message-serverless-yml>

Configure the lambda responsible to update your front-end, preferably triggered with an EventBridge event

The Lambda will pass the message to the interested live connections

<https://gist.github.com/xavierlefevre/aceccd6a473b8e0430ff4da5af9c39c4#file-websockets-send-message-lambda-ts>

And that’s it! With a few lines of code your serverless back-end is ready to handle live WebSocket connections.

# Security

### JSON Web Tokens (JWT)

<https://frontendmasters.com/courses/secure-auth-jwt/>

[https://auth0.com/learn/json-web-tokens/#](https://auth0.com/learn/json-web-tokens/)!

**SON Web Token (JWT)** is an open standard ([RFC 7519](https://tools.ietf.org/html/rfc7519)) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with **HMAC** algorithm) or a public/private key pair using **RSA**.

1. **Compact**: Because of its size, it can be sent through an URL, POST parameter, or inside an HTTP header. Additionally, due to its size its transmission is fast.
2. **Self-contained**: The payload contains all the required information about the user, to avoid querying the database more than once.

# Testing

Functional UI – testing the user interface as if a real user was interacting with our app through the browser.

Subcutaneous – just under the graphical user interface – testing HTTP requests to our app.

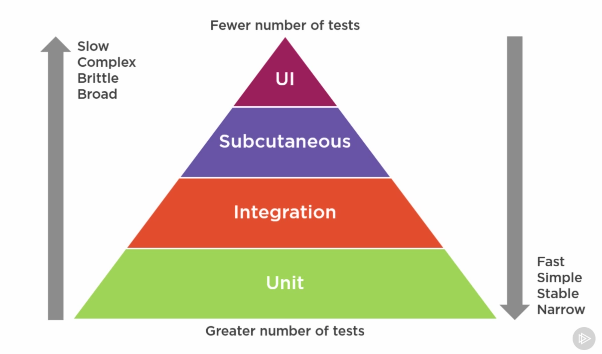
Integration – test different parts of the system together.

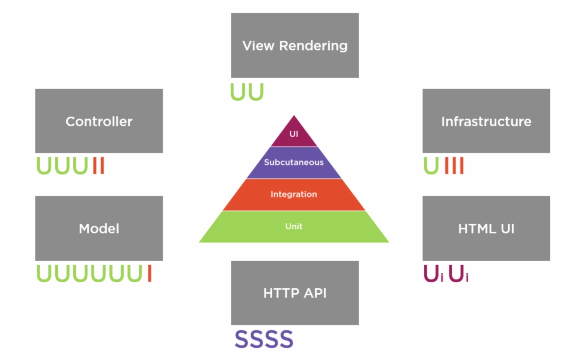
Unit – tests a single class

## Balanced testing model:

The combination of tests that will give us the most cost-to-value benefit.

The Testing Pyramid:





# Metrics and Logs

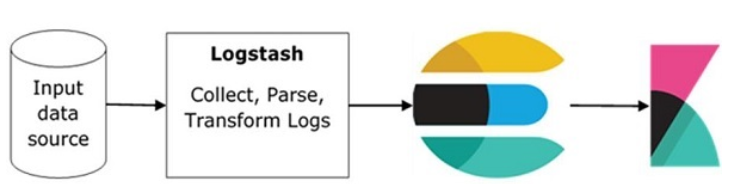
* P99 - The latency that 99% of request were served quicker then.

example: if 99% of requests were served within 50ms => P99==50ms

* n - number of requests
* errors - number of errors or number of errors as percentage of requests.
* In Amazon, we didn’t have absolute rules about any of the above, except to not make it worse.

## Elasticsearch-Logstash-Kibana – ELK

1. **Logstash** extracts the logging data or other events from different input sources. It processes the events and later stores them in **Elasticsearch**.
2. **Beats** is a light-weight component for moving metric-data into Elasticsearch.
3. **Kibana** is a visualization tool, which accesses the logs from Elasticsearch and is able to display to the user in the form of line graph, bar graph, pie charts etc.



### Installation

<https://www.tutorialspoint.com/kibana/kibana_environment_setup.htm>

### Running

In Kibana->devTools

Example commands for updating/querying ES in devTools: <https://assets.contentstack.io/v3/assets/bltefdd0b53724fa2ce/blt56ad3f4e2c755f29/5d37c1602a506857d64eff48/es_commands.txt>

POST <index-name>/<type>   
- write a new document into the index. ES will create and assign the document ID automatically.   
This means that if I run the same command over and over again, ES will create that many documents with the same data but with different document \_ids.

Index-name for example inspections

<type>: \_doc

PUT <index-name>/<type>/<id> - the same as post but the user need to specify the document ID. If I run the same command over and over again – ES will update the same document (since it’s the same id) so the “\_version” field will update but the document \_id will stay the same.

PUT /<index-name> - create a new index for you. You can specify:

number\_of\_shards:

number\_of\_replicas:

Bulk commands:

POST /inspections/\_doc/\_bulk  
{ "index": { "\_id": 1 }}  
{"business\_address":"315 California St”,…}  
{ "index": { "\_id": 2 }}  
{"business\_address":"3 California St”,…}  
…

Partial Document Update:  
This will perform a partial update of the document, only touching the included fields:

POST /<index-name>/\_update/<id>  
{  
 “doc”: {  
 … additional/updated fields  
 }  
}

DELETE /<index-name> - will delete the index and all the documents under it.

We can also delete individual document (using its id) in the index.

GET /<index-name>/<type>/\_search – return all the results for this index.  
GET /inspections/\_doc/\_search

To search for more specific parameters, we can add:

{  
 “query”: {  
 “match”: {  
 <field-name>: <low-case search value>  
 }  
 }  
}

<search value> is case-insensitive

The returned values will contain \_score field which shows how closely the result match my search (the higher, the better)

Additional queries:

“match\_phrase” - To search for a phrase (more than 1 word)

“bool” + “must” – condition1 AND condition 2

…

### Security

<https://www.elastic.co/guide/en/elasticsearch/reference/master/configuring-stack-security.html>

### ELK Stack Playground

Following this tutorial: <https://www.tutorialspoint.com/kibana/kibana_environment_setup.htm>

ELK-Playground installation folder: F:\MyFiles\Sarit\_Training\\_git\_projects\open-source-friday\ELK\_playground\   
under it:

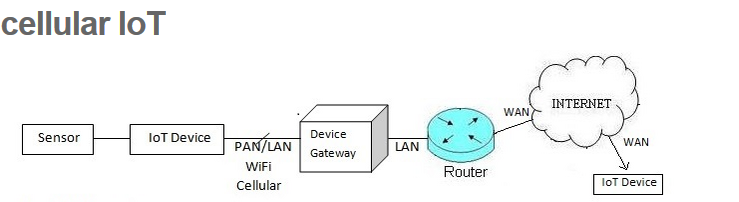
ES is under {ELK-Playground}\elasticsearch-7.13.4\bin

After running it it seemed like it’s stuck and requires me to install JDK 11 but after I pressed <enter> it continued ok and I was able to access it in the browser on <http://localhost:9200/>

# Internet of Things (IoT)

There are two major subsystems involved in the IoT network viz. front end part and back end part. Front end is mainly consists of IoT sensors which are MEMS based. It includes optical sensors, light sensors, gesture and proximity sensors, touch and fingerprint sensors, pressure sensors and more.

Back end consists of cellular, wireless and wired networks which are interfaced with IoT devices. The devices will report to the central servers and also interact with databases in the backbone network. Routers and gateways are part of the wireless backbone networks.



MQTT and CoAP are two of the most promising protocols for IoT; see <http://www.eclipse.org/community/eclipse_newsletter/2014/february/article2.php>

## The IoT Protocols

Can you build an IoT system with familiar Web technologies? Yes you can, although the result would not be as efficient as with the newer protocols.

HTTP(S) and Websockets are common existing standards, which can be used to deliver XML or JavaScript Object Notation (JSON) in the payload. JSON provides an abstraction layer for Web developers to create a stateful Web application with a persistent connection to a Web server.

**HTTP**

HTTP is the foundation of the client-server model used for the Web. The more secure method to implement HTTP is to include only a client in your IoT device, not a server. In other words, it is safer to build an IoT device that can only initiate connections, not receive. After all, you do not want to allow outside access to your local network.

**WebSocket**

WebSocket is a protocol that provides full-duplex communication over a single TCP connection between client and server. It is part of the HTML 5 specification. The WebSocket standard simplifies much of the complexity around bi-directional Web communication and connection management.

In the next sections, we’ll discuss some of the most promising new IoT protocols and when/how to use them.

## XMPP

[XMPP (Extensible Messaging and Presence Protocol)](http://wiki.xmpp.org/web/Tech_pages/IoT_systems) is a good example of an existing Web technology finding new use in the IoT space.

XMPP has its roots in instant messaging and presence information. It has expanded into signaling for VoIP, collaboration, lightweight middleware, content syndication, and generalized routing of XML data. It is a contender for mass scale management of consumer white goods such as washers, dryers, refrigerators, and so on.

## MQTT Protocol

[MQ Telemetry Transport (MQTT)](http://www.mqtt.org) is an open source protocol for constrained devices and low-bandwidth, high-latency networks. It is a publish/subscribe messaging transport that is extremely lightweight and ideal for connecting small devices to constrained networks.

MQTT is bandwidth efficient, data agnostic, and has continuous session awareness. It helps minimize the resource requirements for your IoT device, while also attempting to ensure reliability and some degree of assurance of delivery with grades of service.

MQTT targets large networks of small devices that need to be monitored or controlled from a back-end server on the Internet. It is not designed for device-to-device transfer. Nor is it designed to “multicast” data to many receivers. MQTT is extremely simple, offering few control options.

MQTT is designed for constrained devices and low-bandwidth, high-latency or unreliable networks. The design principles are to minimise network bandwidth and device resource requirements whilst also attempting to ensure reliability and some degree of assurance of delivery. These principles also turn out to make the protocol ideal of the emerging “machine-to-machine” (M2M) or “Internet of Things” world of connected devices, and for mobile applications where bandwidth and battery power are at a premium.

TCP/IP port 1883 is reserved with [IANA](http://www.iana.org/) for use with MQTT. TCP/IP port 8883 is also registered, for using MQTT over SSL.

**Does MQTT support security?**

You can pass a user name and password with an MQTT packet in V3.1 of the protocol. Encryption across the network can be handled with SSL, independently of the MQTT protocol itself (it is worth noting that SSL is not the lightest of protocols, and does add significant network overhead). Additional security can be added by an application encrypting data that it sends and receives, but this is not something built-in to the protocol, in order to keep it simple and lightweight.

**How Does MQTT Works?**

* + 1. MQTT has a client/server model, where every sensor is a client and connects to a server, known as a broker, over TCP.
    2. MQTT is message oriented. Every message is a discrete chunk of data, opaque to the broker.
    3. Every message is published to an address, known as a topic. Clients may subscribe to multiple topics. Every client subscribed to a topic receives every message published to the topic.
    4. In MQTT, topics are hierarchical, like a filing system (eg. kitchen/oven/temperature). Wildcards are allowed when registering a subscription (but not when publishing) allowing whole hierarchies to be observed by clients.
    5. Quality Of Service:   
       MQTT supports three quality of service levels, “Fire and forget”, “delivered at least once” and “delivered exactly once”.
    6. Last Will and Testament:  
       MQTT clients can register a custom “last will and testament” message to be sent by the broker if they disconnect. These messages can be used to signal to subscribers when a device disconnects.
    7. Persistence:  
       MQTT has support for persistent messages stored on the broker. When publishing messages, clients may request that the broker persists the message. Only the most recent persistent message is stored.
    8. Security:  
       MQTT brokers may require username and password authentication from clients to connect. To ensure privacy, the TCP connection may be encrypted with SSL/TLS.
    9. MQTT-SN:
       - Even though MQTT is designed to be lightweight, it has two drawbacks for very constrained devices.
       - Every MQTT client must support TCP and will typically hold a connection open to the broker at all times. For some environments where packet loss is high or computing resources are scarce, this is a problem.
       - MQTT topic names are often long strings which make them impractical for 802.15.4.
       - Both of these shortcomings are addressed by the MQTT-SN protocol, which defines a UDP mapping of MQTT and adds broker support for indexing topic names.

## CoAP (Constrained Application) Protocol

1. **Document Transfer Protocol:**  
   Like HTTP, CoAP is a document transfer protocol. It is a RESTful protocol and support the same commands as HTTP.  
   Unlike HTTP, CoAP is designed for the needs of constrained devices.
2. **Packet Size:**  
   CoAP packets are much smaller than HTTP TCP flows. Bitfields and mappings from strings to integers are used extensively to save space. Packets are simple to generate and can be parsed in place without consuming extra RAM in constrained devices.
3. **UDP:**  
   CoAP runs over UDP, not TCP. Clients and servers communicate through connectionless datagrams. Retries and reordering are implemented in the application stack. Removing the need for TCP may allow full IP networking in small microcontrollers. CoAP allows UDP broadcast and multicast to be used for addressing.
4. **Client/Server:**  
   CoAP follows a client/server model. Clients make requests to servers, servers send back responses. Clients may GET, PUT, POST and DELETE resources.
5. **Interoperability:**  
   CoAP is designed to interoperate with HTTP and the RESTful web at large through simple proxies.
6. **CoAP Over SMS:**  
   Because CoAP is datagram based, it may be used on top of SMS and other packet based communications protocols.
7. **Application Level Quality of Service:**Since CoAP runs over UDP, the application layer needs to implement the quality of service.
8. **Security:**  
   Because CoAP is built on top of UDP not TCP, SSL/TLS are not available to provide security. DTLS, Datagram Transport Layer Security provides the same assurances as TLS but for transfers of data over UDP. Typically, DTLS capable CoAP devices will support RSA and AES or ECC and AES.

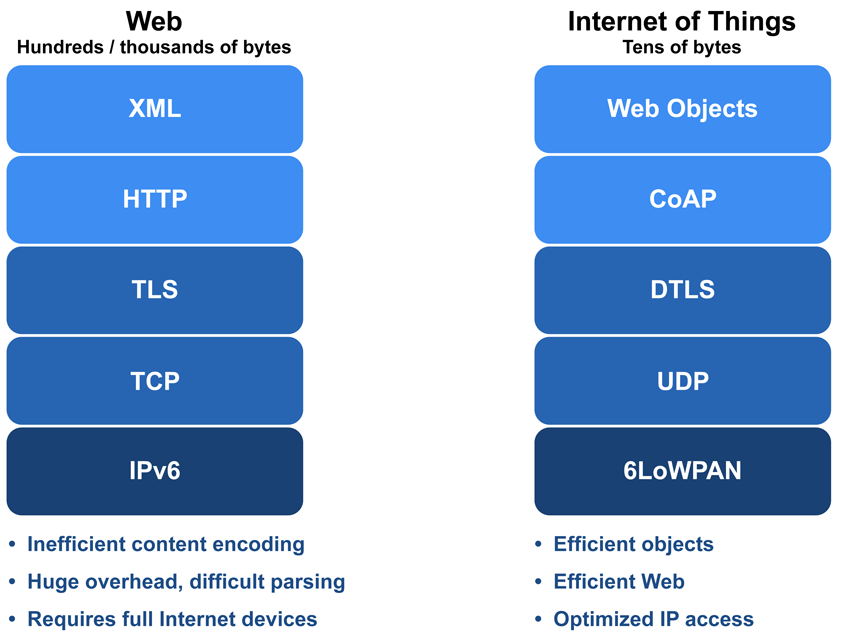
## Comparison

The table below contains a summary of the IoT protocol landscape.

| [Beyond MQTT: A Cisco View on IoT Protocols](http://blogs.cisco.com/ioe/beyond-mqtt-a-cisco-view-on-iot-protocols/), Paul Duffy, April 30 2013 | | | | |
| --- | --- | --- | --- | --- |
| **Protocol** | **CoAP** | **XMPP** | **RESTful HTTP** | **MQTT** |
| Transport | UDP | TCP | TCP | TCP |
| Messaging | Request/Response | Publish/Subscribe Request/Response | Request/Response | Publish/Subscribe Request/Response |
| 2G, 3G, 4G Suitability (1000s nodes) | Excellent | Excellent | Excellent | Excellent |
| LLN Suitability (1000s nodes) | Excellent | Fair | Fair | Fair |
| Compute Resources | 10Ks RAM/Flash | 10Ks RAM/Flash | 10Ks RAM/Flash | 10Ks RAM/Flash |
| Success Stories | Utility Field Area Networks | Remote management of consumer white goods | Smart Energy Profile 2 (premise energy management, home services) | Extending enterprise messaging into IoT applications |

### Comparing Web and IoT Protocols

The illustration below provides another good summary of the performance benefit that these protocols bring to IoT. From Zach Shelby's presentation “[Standards Drive the Internet of Things](http://www.slideshare.net/zdshelby/standards-drive-the-internet-of-things).”

[](http://micrium.com/wp-content/uploads/2014/03/Web-and-IoT-Stacks.png)

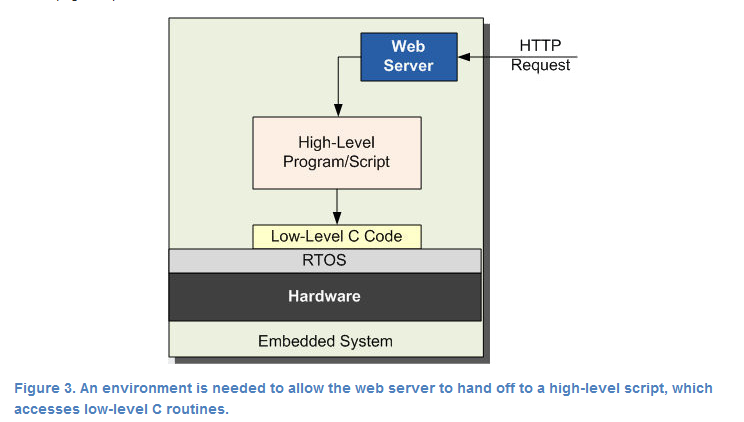
Comparison of Web and IoT protocol stacks

On the left, the protocol stack for Web applications can easily produce a **data overhead of hundreds or thousands of bytes**. By comparison, IoT protocols are optimized for constrained devices and networks, and produce a much smaller data overhead of tens of bytes.

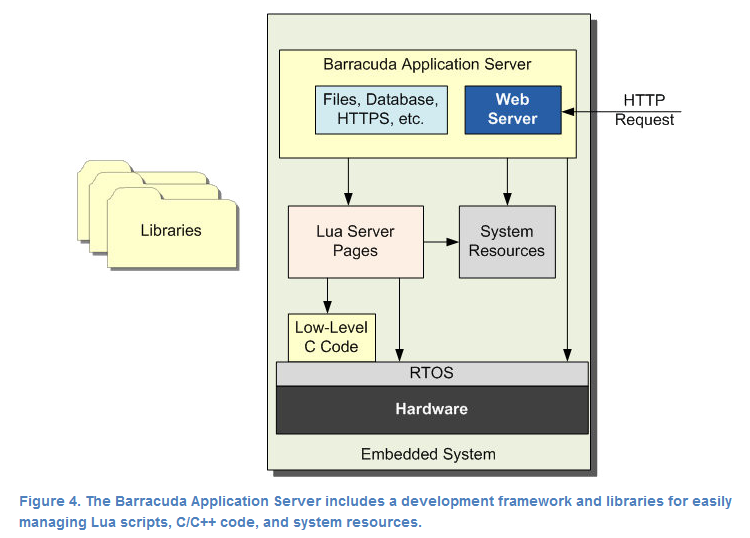
# Web Interface for Embedded Devices

**Web Servers vs Application Servers**

Web servers and application servers both offer a way to provide an embedded system with an Internet presence. The difference lies in the amount of programming required to adapt the server to the system:

**Classic web servers simply process HTTP requests and responses**; anything else must be developed from scratch. Much of that work amounts to spending time and money "reinventing the wheel" for common functions and infrastructure that aren't supported by a web server.  


By contrast, **application servers include frameworks that dramatically shorten the development cycle**. Choosing an application server means that you don't have to write commonly-used communication and user interface functions or their underlying infrastructure for each system. Programming resources can instead be focused on functionality that is specific to the system.



In short, application servers will ultimately provide the lowest total cost of ownership by removing months of development time, dramatically reducing the risks of overly-optimistic project scoping, and reducing the possibility of unanticipated debugging delays.

In order to implement a web interface on an embedded device, we need:

* + 1. **Network connection**: Ethernet/GPRS
    2. **Web server** e.g. Apache / lighttpd
    3. **Our Software** will include:
       - CGIs – scripts that communicate between the server and the application
       - Sockets to the main application
       - Main embedded application

## Embedded Web Servers

<https://www.linux.com/news/software/applications/807641-which-light-weight-open-source-web-server-is-right-for-you>

**Nginx – light weight, fast:**

light weight (10MB) and very fast. Used in Netflix, Hulu, Pinterest, Wordpress.com

**Lighttpd – very light weight, CPU load balancing:**

Very light weight. The perfect server for any machine suffering from load problems. Used in Raspberry Pi,

Who is lighttpd right for? If you’re looking to create an embedded system (with far less available resources) that includes a web component, Lighttpd is most likely what you want. Lighttpd is very simple to use and set up. Configuration of this particular server is handled in a single .conf file.

**Hiawatha – light weight and secure:**

Lightweight, open source web server with a focus geared toward security and ease of use.  
Who is Hiawatha right for? If you’re looking for a robust web server for either a standard setup or embedded system, and require a higher level of built-in security, Hiawatha is the server for you.

**Additional servers (not as recommended):**

Apache:

Boa – no longer supported.

## Scripts Languages Recommended for Embedded Systems

Python – with web.py (basic) or bottlepy.py (more complete)

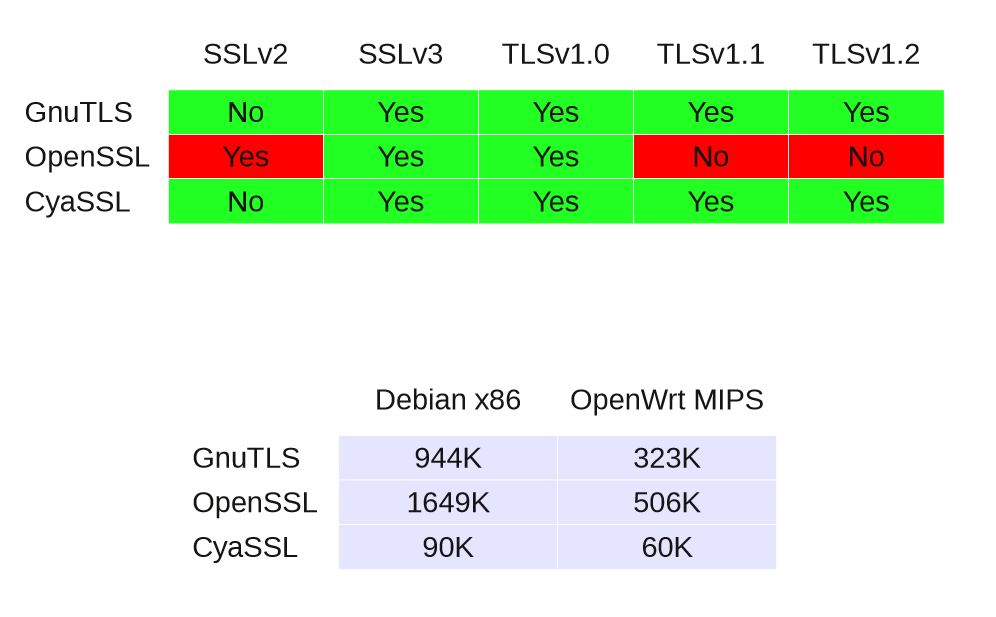
## Security

Web based embedded system is vulnerable

1. Port 80 is attacked
2. Anybody can try to connect
3. Text based communication → buffer overflows
4. Authentication → password sniffing
5. Request forgery and replay attacks

Tips to protect the web service:

1. Time out connections – otherwise, you run out of threads
2. HTTP Digest Authentication – otherwise, passwords can be sniffed
3. URL-encoding of session:
   1. Always use a different URL
   2. If bookmarked – redirect to login page first
4. SSL/TLS:
   1. GnuTLS:
      1. License: LGPL
      2. Pretty complete
   2. OpenSSL:
      1. License: BSD with advertising clause
      2. Most well known
      3. Large and clumsy
   3. CyaSSL:
      1. License: GPL/Commercial
      2. Specifically targeted at embedded:
         * Tiny compared to the others
         * focuses on most used features
      3. Optimised for speed (e.g. assembly for embedded uPs)
      4. OpenSSL API (simplified)



# Tools and Utils

## Mockup Tools

Presentation apps: <https://developer.apple.com/videos/play/wwdc2016/805/>

## Yahoo Query Language

Allow the user to get parts of a web page without needing to read the whole page and then parse it. It returns only the requested data in JSON or XML format.