[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**

**HTML** - HyperText Markup Language - the standard markup language used to create web pages. The documents of the Web - content and structure of web pages.

**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).

**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.

**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.

# 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:

* **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.
* **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.
* **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).
* **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.
* **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.
* **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:

* assets/html
* assets/images
* assets/styles
* assets/styles/fonts
* assets/scripts
* assets/data ; some xml files and other stuff
* assets/media ; for music etc
* assets/lib (or 3rd-party); this one is intended for code you don't make or modify, the libraries as you get them
* 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
* 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
  + assets-server**/fonts**
  + assets-server**/lib**
  + assets-server/lib-modded

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

* ./
* images
* styles
* styles/fonts
* 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

* scripts/source
* 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:

* **Layout vs. design**

layout.css design.css

* **Page Sections**

main.css nav.css

* **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

* 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:

* Hold down Ctrl and press F5.

In Apple Safari:

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

In Chrome and Firefox for Mac:

* 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:**

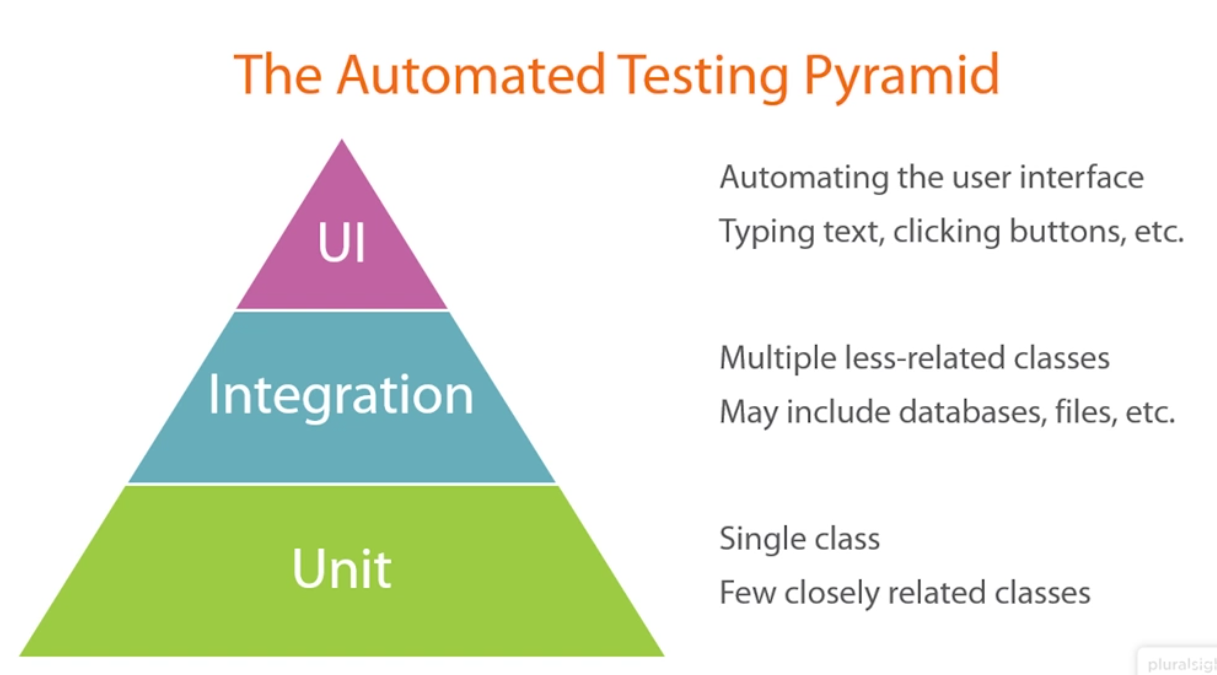
Bootstrap: <http://getbootstrap.com/>   
Foundation: <http://foundation.zurb.com/>   
Yaml: <http://www.yaml.de/>

**Lightweight Frameworks:**

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

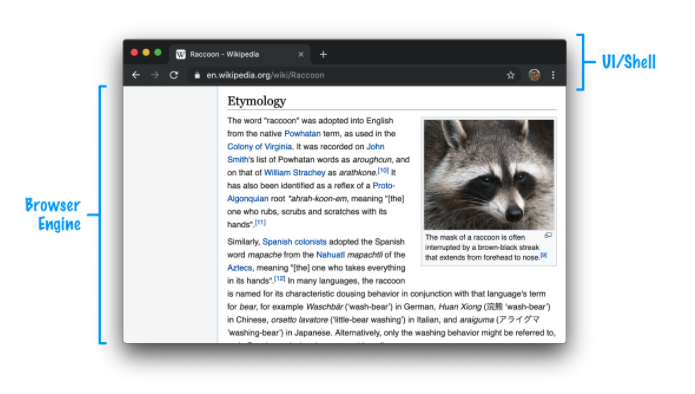
## Bootstrap –

# Testing



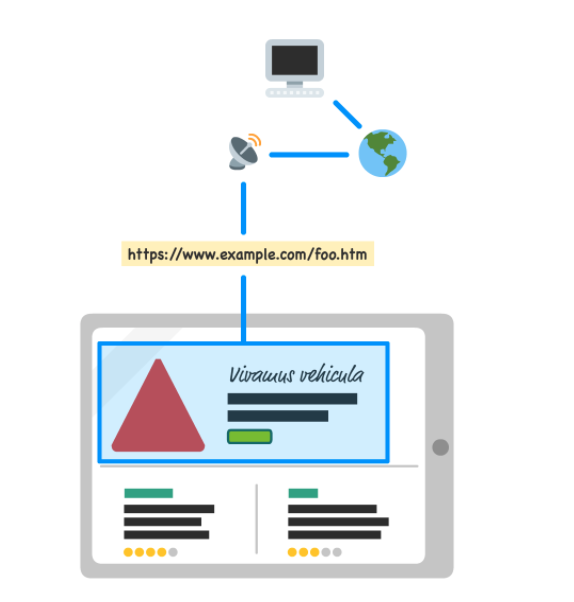
# Webview

**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.



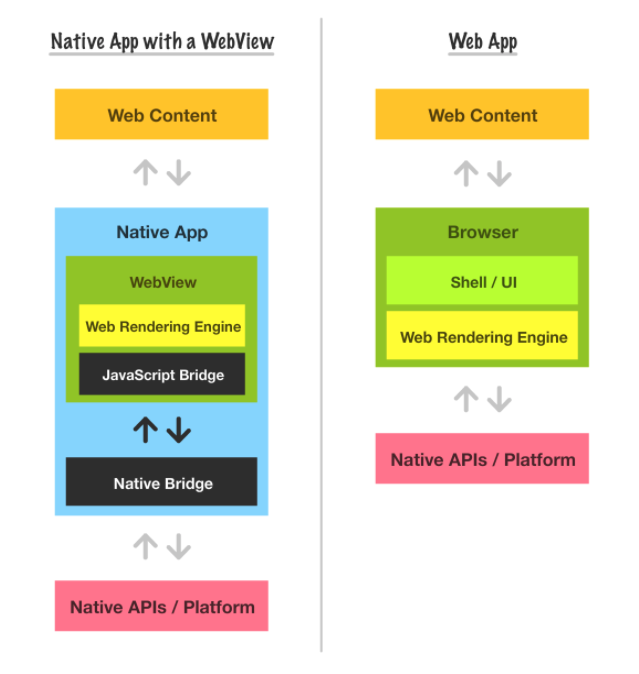
A **WebView** is just the **browser engine** part that you can insert sort of like an iframe into your native app and programmatically tell it what web content to load.

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:

* 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.
* On Android, the rendering engine under the covers is ***usually*** Blink, the same one that powers Chrome.
* 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

## Security in HTML

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

## Security in Python Script

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

Import cgi

def escape\_html(s):

return cgi.escape(s,quote=True)

* + 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:

* Manually:

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

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

# Frameworks

Web frameworks:

* Django,
* Ruby on Rails,
* 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.

## 

# 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.

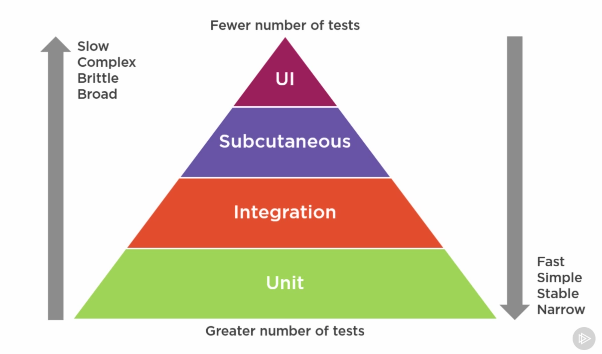
Integration – test different parts of the system together.

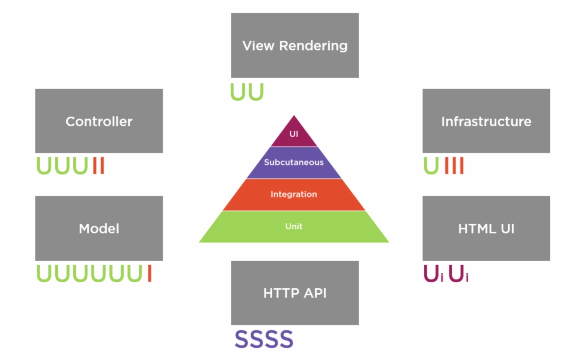
Unit – tests a single class

## Balanced testing model:

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

The Testing Pyramid:

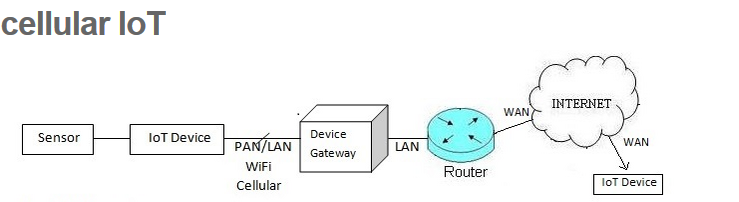




# Internet of Things (IoT)

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?**

* + - MQTT has a client/server model, where every sensor is a client and connects to a server, known as a broker, over TCP.
    - MQTT is message oriented. Every message is a discrete chunk of data, opaque to the broker.
    - 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.
    - 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.
    - Quality Of Service:   
      MQTT supports three quality of service levels, “Fire and forget”, “delivered at least once” and “delivered exactly once”.
    - 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.
    - 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.
    - 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.
    - 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

* **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.
* **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.
* **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.
* **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.
* **Interoperability:**  
  CoAP is designed to interoperate with HTTP and the RESTful web at large through simple proxies.
* **CoAP Over SMS:**  
  Because CoAP is datagram based, it may be used on top of SMS and other packet based communications protocols.
* **Application Level Quality of Service:**Since CoAP runs over UDP, the application layer needs to implement the quality of service.
* **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.

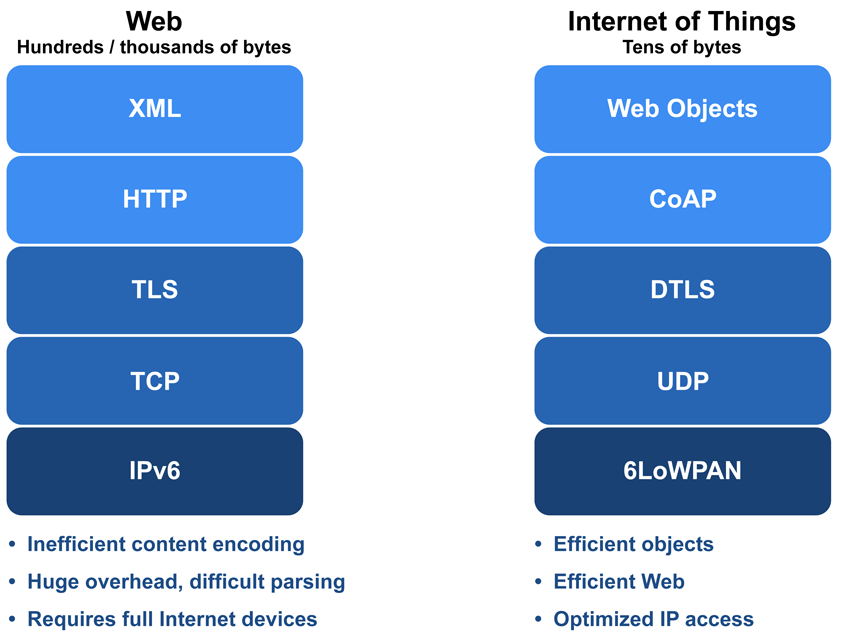
## Comparison

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)

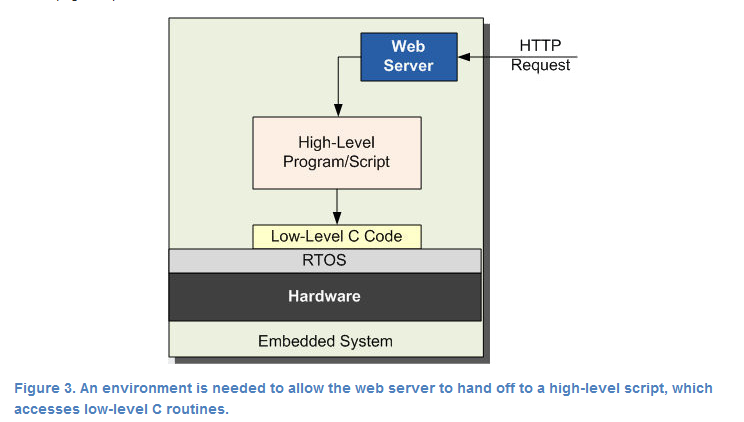
Comparison of Web and IoT protocol stacks

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.

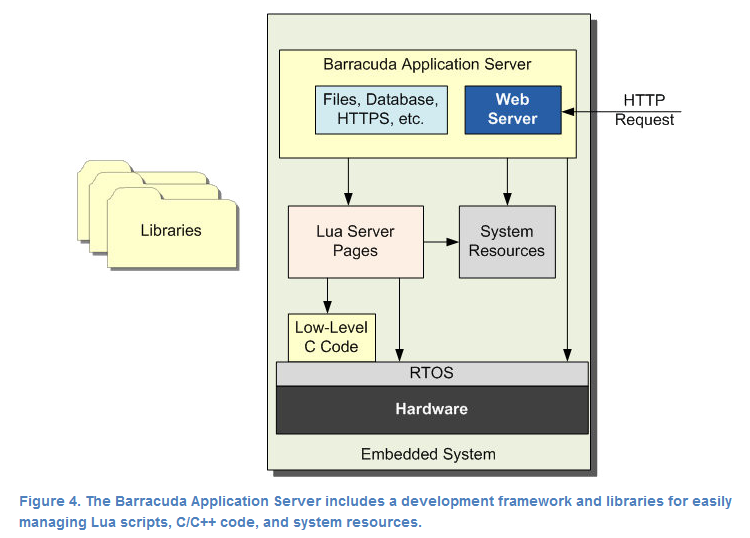
# Web Interface for Embedded Devices

**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:

* + - **Network connection**: Ethernet/GPRS
    - **Web server** e.g. Apache / lighttpd
    - **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)

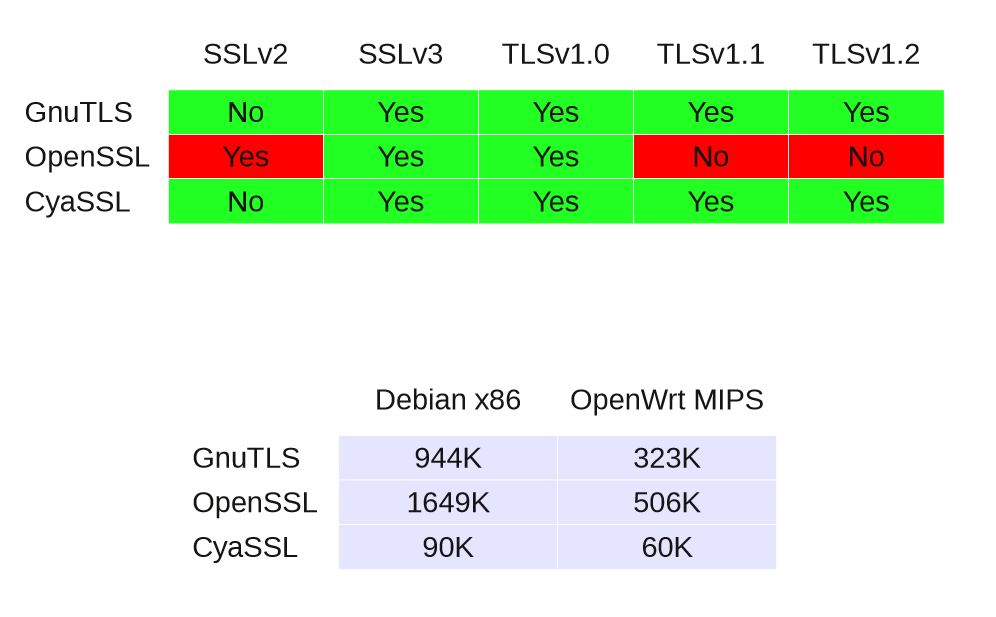
## Security

Web based embedded system is vulnerable

* Port 80 is attacked
* Anybody can try to connect
* Text based communication → buffer overflows
* Authentication → password sniffing
* Request forgery and replay attacks

Tips to protect the web service:

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



# Tools and Utils

## 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.