Contents

[HTML5 4](#_Toc457826198)

[HTML5 Reference 4](#_Toc457826199)

[Document Type 4](#_Toc457826200)

[Quirks Mode 4](#_Toc457826201)

[Document Structure 4](#_Toc457826202)

[Encoding 4](#_Toc457826203)

[Form Encoding 5](#_Toc457826204)

[CSS3 6](#_Toc457826205)

[CSS3 References 6](#_Toc457826206)

[SASS 6](#_Toc457826207)

[LESS 6](#_Toc457826208)

[Bootstrap 6](#_Toc457826209)

[Bootstrap References 6](#_Toc457826210)

[Bootstrap Environment 6](#_Toc457826211)

[The Grid System 7](#_Toc457826212)

[Typography 7](#_Toc457826213)

[Elements 7](#_Toc457826214)

[Tables 8](#_Toc457826215)

[Forms 8](#_Toc457826216)

[Buttons 9](#_Toc457826217)

[Images 9](#_Toc457826218)

[Helper Classes 9](#_Toc457826219)

[Foundation 9](#_Toc457826220)

[Javascript 10](#_Toc457826221)

[Javascript References 10](#_Toc457826222)

[Javscript Core 10](#_Toc457826223)

[*this* & Object Prototypes 10](#_Toc457826224)

[*this* binding 10](#_Toc457826225)

[JS objects 12](#_Toc457826226)

[The Object Prototype Chain 13](#_Toc457826227)

[Behavior Delegation Oriented Design 14](#_Toc457826228)

[Design Patterns 15](#_Toc457826229)

[Types 15](#_Toc457826230)

[jQuery 16](#_Toc457826231)

[jQuery References 16](#_Toc457826232)

[jQuery Environment 16](#_Toc457826233)

[The jQuery Object 16](#_Toc457826234)

[Element Data 17](#_Toc457826235)

[Selecting Elements 17](#_Toc457826236)

[Manipulating Elements 18](#_Toc457826237)

[Traversing 18](#_Toc457826238)

[Utility Functions 18](#_Toc457826239)

[Iteration 19](#_Toc457826240)

[Effects 19](#_Toc457826241)

[Plugins 20](#_Toc457826242)

[Widgets 20](#_Toc457826243)

[Code Organization 21](#_Toc457826244)

[AngularJS 2.0 21](#_Toc457826245)

[React 21](#_Toc457826246)

[Ember.js 21](#_Toc457826247)

[Backbone.js 21](#_Toc457826248)

[RequireJS 21](#_Toc457826249)

[RequireJS References 21](#_Toc457826250)

[PHP 22](#_Toc457826251)

[Laravel 22](#_Toc457826252)

[Yii 22](#_Toc457826253)

[CodeIgniter 22](#_Toc457826254)

[Symfony 22](#_Toc457826255)

[Phalcon 22](#_Toc457826256)

[Wordpress 22](#_Toc457826257)

[Drupal 22](#_Toc457826258)

[Ruby 22](#_Toc457826259)

[Ruby on Rails 22](#_Toc457826260)

[Python 22](#_Toc457826261)

[Django 22](#_Toc457826262)

[Go 23](#_Toc457826263)

[Java 23](#_Toc457826264)

[Objective-C 23](#_Toc457826265)

[Swift 23](#_Toc457826266)

[Meteor 23](#_Toc457826267)

[Node.js 23](#_Toc457826268)

[Ionic 23](#_Toc457826269)

[Phonegap / Cordova 23](#_Toc457826270)

[MySQL 23](#_Toc457826271)

[PostgreSQL 23](#_Toc457826272)

[MongoDB 23](#_Toc457826273)

[Redis 23](#_Toc457826274)

[Oracle 23](#_Toc457826275)

[DDP 23](#_Toc457826276)

[Regular Expressions 23](#_Toc457826277)

[RegEx References 23](#_Toc457826278)

[Code Review 24](#_Toc457826279)

[UI/UX Design 24](#_Toc457826280)

[Marketing Concepts 24](#_Toc457826281)

[getmwf 25](#_Toc457826282)

[Styles 25](#_Toc457826283)

[Color 25](#_Toc457826284)

[Grid 25](#_Toc457826285)

[Motion 25](#_Toc457826286)

[Typography 26](#_Toc457826287)

[Imagery 26](#_Toc457826288)

[Picchu 26](#_Toc457826289)

[ARIA 26](#_Toc457826290)

getmwf Documentation

<http://getmwf.com/>

HCL webmail  
<https://ukmail.hcl.com/owa/>

[A Baseline for Front-End Developers](http://rmurphey.com/2012/04/12/a-baseline-for-front-end-developers)

<http://rmurphey.com/blog/2012/04/12/a-baseline-for-front-end-developers>

MDN Web Development Guides

<https://developer.mozilla.org/en-US/Apps>

# HTML5

## HTML5 Reference

* HTML5 Mastery Tutorials

<http://code.tutsplus.com/series/html5-mastery-class--cms-897>

## Document Type

### Quirks Mode

## Document Structure

* HTML4 used all divs and headings to exhibit structure, causing nested documents to add confusion to the structure and complicating the generation of outlines. HTML5 fixes this problem by introducing the <section> tag to add more precision to the structure. Each section can have its own set of headings and even h1s. With the section tag the true document structure is much more apparent.
* HTML5 added <aside>, <nav>, <header>, and <footer> tags which help distinguish content not truly part of the document structure.
* Self closing tags such as img, source, input are not supposed to have children and are closed immediately by the html parser. In html it is recommended to omit the trailing slash.
* The html parser resolves errors in improperly closed tags via the html scoping rules.

## Encoding

There are three important text encoding rules for HTML:

1. Load the content with the right encoding.
2. Transmit the content with the same encoding.
3. Ensure that the client reads the content with the specified encoding.

* UTF-8 has the major advantage of only requiring a single byte if we do not use a special character. At most 4 bytes per character are consumed. This is dynamic and makes UTF-8 an ideal format for text storage and transmission. The caveat is, however, that UTF-8 is not the best format for using strings from memory.
* The HTTP protocol transmits data as plain text. Even if we decide to encode the transmitted content as GZip or if we use HTTPS, which encrypts the content, the underlying content is still just plain text. We’ve already learned that there is no such thing as just plain text. We always need to associate the content with some encoding to get a text representation.
* An HTTP message is split in two parts. The upper part is called the headers. Separated by an empty line is the lower part: the body. There are always at least two HTTP messages: a request and its associated response. Both types of messages share this structure. The body of a response is the content we want to transmit. The body of a request is only of interest for form submission.  
    
  If we want to provide some information on the encoding of the content, we have to supply some information in the header. The following header tells the receiving side that the body contains a special text format called HTML, using the UTF-8 character set:  
    
  Content-Type: text/html; charset=utf-8
* There is also the Content-Encoding header. We can easily confuse the content encoding with the actual text encoding of the content. The former is used to specify encoding of the whole package, e.g. GZip, while the latter is used as an initial setting for, e.g., parsing the provided content.
* Once the DOM constructor hits a meta tag, it will look for a charset declaration. If one is found, the character set will be extracted. If we can extract it successfully and if the encoding is valid, we set the new encoding for scanning further characters. At this point the encoding will be frozen, and no further changes are possible.
* Place the <meta charset=utf-8> (or some other encoding) tag as soon as possible and only use ASCII characters before specifying the charset attribute in HTML.
* If no initial encoding is given from the HTTP headers, the browser will select the initial encoding based on the user’s locale. Windows-1252 is actually the default for most countries.
* In principle we do not have to worry about this initial encoding if we followed the best practices. ASCII is a subset of Unicode, and most of the listed encodings are actually just ASCII extensions to satisfy the specific needs of one or more countries. If we only use basic ASCII characters until the character set is specified, we should be fine.
* Declaring the encoding during transport is certainly useful, although not required, especially if we follow best practices for placing a <meta> element with the charset attribute. Most often its best to just use UTF-8. Consistency is a good thing.

### Form Encoding

* A form is submitted with a certain encoding type, which is not the same as the encoding type of a server’s response, e.g. GZip. The form’s encoding type determines how the form is serialized before sending it to the server. It is particularly useful in conjunction with the HTTP verb.
* Ordinary form submissions use POST as HTTP verb, but GET, PUT and DELETE are also common. Only POST and PUT are supposed to use the body for content transmission in the request. The browser will construct the content with respect to the choice of the enctype attribute of the <form> element, specifying the encoding type. The encoding type is transported by setting the Content-Type header in the HTTP request.

There are three well-established encoding types:

1. URL encoded (default value, explicitly application/x-www-form-urlencoded)
2. Plain text (text/plain)
3. Multipart (multipart/form-data)

The first and the second are quite similar, and multipart the most powerful method. It even allows the transporting of arbitrary files as attachments. URL encoded form transmission percent-encodes all names and values, which is not done by plain text. Multipart uses a boundary string to separate the entries which solves multi-line errors occurring in plain text submissions, but increases the content length and therefore the size of the request message.

* Form transmission primarily uses the accept-charset attribute of the corresponding <form> element. If no such attribute is given, the encoding of the page is used.
* In general we can always choose multipart/form-data as enctype, even though the default encoding type might be better (smaller) in most scenarios. In production we should never use text/plain.

# CSS3

## CSS3 References

* CSS3 Feature Demos

<http://css3clickchart.com>

* A Complete Guide to Flexbox  
  <https://css-tricks.com/snippets/css/a-guide-to-flexbox/>

# SASS

# LESS

# Bootstrap

## Bootstrap References

* Bootstrap CSS Documentation  
  <http://getbootstrap.com/css/>

## Bootstrap Environment

* Always use the HTML5 doctype with Bootstrap
* To ensure proper rendering and touch zooming, add the viewport meta tag to your <head>.  
  <meta name="viewport" content="width=device-width, initial-scale=1">
* Bootstrap uses Normalize.css and sets basic global display, typography, and link styles.
* Bootstrap containers are required to house the grid system. There are 2 classes to choose from: use “container” for a responsive fixed width container or “container-fluid” for a full width container spanning the entire viewport. These are not nestable.
* Bootstrap can be used in at least two ways: with the compiled CSS or with the source Less files. Many centralized variables can be altered to modify your site’s skeleton. Use LESS build tools to make these changes with overriding LESS files while keeping the core Bootstrap LESS files unaltered.
* While Bootstrap is built on Less, it also has an [official Sass port](https://github.com/twbs/bootstrap-sass).

## The Grid System

* Bootstrap includes a responsive, mobile first fluid grid system that appropriately scales up to 12 columns as the device or viewport size increases.
* The .row class is used to make groups of columns. Content should be placed in columns.
* Columns create gutters (gaps between column content) via padding. That padding is offset in rows for the first and last column via negative margin on .rows.
* Grid classes apply to devices with screen widths greater than or equal to the breakpoint sizes, and override grid classes targeted at smaller devices. Therefore, e.g. applying any .col-md-\* class to an element will not only affect its styling on medium devices but also on large devices if a .col-lg-\* class is not present.
* With the four tiers of grids available you're bound to run into issues where, at certain breakpoints, your columns don't clear quite right as one is taller than the other. To fix that, use a combination of a .clearfix and our [responsive utility classes](http://getbootstrap.com/css/#responsive-utilities):  
  <div class="clearfix visible-xs-block"></div>
* Move columns to the right using .col-(size)-offset-\* classes. These classes increase the left margin of a column by \* columns. For example, .col-(size)-offset-4 moves .col-(size)-4 over four columns. The same as with column clearing at responsive breakpoints above, you may need to reset offsets, pushes, or pulls. See this in action in [the grid example](http://getbootstrap.com/examples/grid/).
* To nest your content with the default grid, add a new .row inside of the column of which you want to nest and inside of the new row add a set of .col-(size)-\* columns. Nested rows should include a set of columns that add up to 12 or fewer.

## Typography

* All HTML headings, <h1> through <h6>, are available. .h1 through .h6 classes are also available, for when you want to match the font styling of a heading but still want your text to be displayed inline.
* Create lighter, secondary text in any heading with a generic <small> tag or the .small class:  
  <h1>h1. Bootstrap heading <small>Secondary text</small></h1>

## Elements

### Tables

* You must add the .table class to table elements to use the Boostrap table stylings. Available types are: .table, .table-striped, .table-bordered, .table-hover, .table-condensed
* **Responsive Tables**: create by wrapping any .table in .table-responsive to make them scroll horizontally on small devices (under 768px). When viewing on anything larger than 768px wide, you will not see any difference in these tables.

### Forms

* **Inline Forms**: add .form-inline to your form (which doesn't have to be a <form>) for left-aligned and inline-block controls. This only applies to forms within viewports that are at least 768px wide. Inputs and selects have width: 100%; applied by default in Bootstrap. Within inline forms, we reset that to width: auto; so multiple controls can reside on the same line. Depending on your layout, additional custom widths may be required.
* **Horizontal forms**: use Bootstrap's predefined grid classes to align labels and groups of form controls in a horizontal layout by adding .form-horizontal to the form (which doesn't have to be a <form>). Doing so changes .form-groups to behave as grid rows, so no need for .row.
* **Inputs**: most common form control, text-based input fields. Includes support for all HTML5 types: text, password, datetime, datetime-local, date, month, time, week, number, email, url, search, tel, and color.
* Do not mix form groups directly with [input groups](http://getbootstrap.com/components/#input-groups). Instead, nest the input group inside of the form group.
* Use the .checkbox-inline or .radio-inline classes on a series of checkboxes or radios for controls that appear on the same line.
* Always add labels. Screen readers will have trouble with your forms if you don't include a label for every input. For these inline forms, you can hide the labels using the .sr-only class. There are further alternative methods of providing a label for assistive technologies, such as the aria-label, aria-labelledby or title attribute.
* When you need to place plain text next to a form label within a form, use the .form-control-static class on a <p>. For instance when a field value is known and no user input is required.
* Add the disabled attribute to a <fieldset> to disable all the controls within the <fieldset> at once. Beware disabling fieldsets which contain buttons that are actually <a> tags with button styles. They will not be totally disabled via css and should be disabled properly via Javascript. Also, disabling fieldsets is not implemented in IE11 or below. Use Javascript to disable in these cases.
* **Validation states**: Bootstrap includes validation styles for error, warning, and success states on form controls. To use, add .has-warning, .has-error, or .has-success to the parent element. Any .control-label, .form-control, and .help-block within that element will receive the validation styles. You can also add optional feedback icons with the addition of .has-feedback and the right icon.
* **Control sizing**: set heights using classes like .input-lg, and set widths using grid column classes like .col-lg-\*. Horizontal form group sizes: quickly size labels and form controls within .form-horizontal by adding .form-group-lg or .form-group-sm.

### Buttons

* While button classes can be used on <a> and <button> elements, only <button> elements are supported within our nav and navbar components.
* If the <a> elements are used to act as buttons – triggering in-page functionality, rather than navigating to another document or section within the current page – they should also be given an appropriate role="button".
* As a best practice, we highly recommend using the <button> element whenever possible to ensure matching cross-browser rendering.
* Add .btn-lg, .btn-sm, or .btn-xs for additional sizes.
* Create block level buttons—those that span the full width of a parent— by adding .btn-block.

### Images

* **Responsive images**: images in Bootstrap 3 can be made responsive-friendly via the addition of the .img-responsive class. This applies max-width: 100%;, height: auto; and display: block; to the image so that it scales nicely to the parent element. To center images which use the .img-responsive class, use .center-block instead of .text-center. In Internet Explorer 8-10, SVG images with .img-responsive are disproportionately sized. To fix this, add width: 100% \9; where necessary. Bootstrap doesn't apply this automatically as it causes complications to other image formats.
* **Image shapes**: add classes to an <img> element to easily style images in any project. .img-rounded, .img-circle, .img-thumbnail

### Helper Classes

* **Quick Floats**: use .pull-left and .pull-right to float left/right. .pull-left/right(); mixins can be used in CSS definitions. Do not use quick floats in nav bars instead refer to the navbar docs.
* **Centering content blocks**: use .center-block class or .center-block() mixin.
* Easily clear floats by adding .**clearfix** to the parent element. Utilizes [the micro clearfix](http://nicolasgallagher.com/micro-clearfix-hack/) as popularized by Nicolas Gallagher. Can also be used as a mixin.
* Utilize the .text-hide class or mixin to help replace an element's text content with a background image.
* **Responsive utilities**: for faster mobile-friendly development, use these utility classes for showing and hiding content by device via media query:   
  .visible/hidden-(size)-(block/inline/inline-block)
* Similar to the regular responsive classes, use these for toggling content for print:  
  .visible-print-(block/inline/inline-block)

# Foundation

# Javascript

## Javascript References

* ES6 Feature Browser  
  <http://es6-features.org/>
* ES6 Compatibility Table

<http://kangax.github.io/compat-table/es6/>

* Google Javascript Style Guide

<https://google.github.io/styleguide/javascriptguide.xml>

* Learning JavaScript Design Patterns  
  <https://addyosmani.com/resources/essentialjsdesignpatterns/book/#jquerypluginpatterns>

## Javscript Core

* **Immediately Invoked Function Expressions (IIFEs)**: because an IIFE is just a function, and functions create variable scope, using an IIFE in this fashion is often used to declare variables that won't affect the surrounding code outside the IIFE.
* **Closures**: You can think of closure as a way to "remember" and continue to access a function's scope (its variables) even once the function has finished running.
* The most common non-JavaScript JavaScript you'll encounter is the **DOM API**. The document variable exists as a global variable when your code is running in a browser. It's not provided by the JS engine, nor is it particularly controlled by the JavaScript specification. It takes the form of something that looks an awful lot like a normal JS object, but it's not really exactly that. It's a special object, often called a "**host object**." It is most likely written in C/C++ depending on the browser. Another example is with input/output (I/O). Everyone's favorite alert(..) pops up a message box in the user's browser window. alert(..) is provided to your JS program by the browser, not by the JS engine itself. The call you make sends the message to the browser internals and it handles drawing and displaying the message box. The same goes with console.log(..); your browser provides such mechanisms and hooks them up to the developer tools.

## *this* & Object Prototypes

### *this* binding

* To understand this binding, we have to understand the call-site: the location in code where a function is called (not where it's declared). What's important is to think about the call-stack (the stack of functions that have been called to get us to the current moment in execution). The call-site we care about is in the invocation before the currently executing function.  
    
  4 rules for determining how the call-site sets what *this* points too:  
  **Default Binding**: the most common case and the default catch-all rule – *this* will point to the call-site.  
    
  func1 => call-site: func1  
    
  **Implicit Binding**: if the function is called in context of an object, *this* will point to the object. Only the top/last level of an object property reference chain matters to the call-site. This type of binding also occurs when passing a function callback to a function. When the callback function is executed inside, the call-site would be the function which calls callback not the callback itself or any object which the callback is called in context too. These same rules hold true for built-in Javascript functions.  
    
  obj.func => call-site: obj  
  obj2.obj1.func => call-site: obj2  
  func1(obj.func2) => call-site: func1  
    
  **Explicit Binding**: using call() and apply() you supply an object as the first parameter to use for *this*. The fundamental difference between call() and apply() is that call() accepts an argument list, while apply() accepts a single array of arguments.  
    
  obj1.call(obj2) => call-site: obj2  
    
  Due to the asynchronous nature of Javascript sometimes implicit bindings are lost. To combat this you can use **Hard Binding** using the bind function or manually by wrapping call()/apply() in a function. bind(..) returns a new function that is hard-coded to call the original function with the this context set as you specified. You can also bind arguments to the function. All arguments passed to .bind() after the first argument will be permanently bound to the returned function so that any arguments passed to that function will be placed after the bound ones. Many libraries' functions, and indeed many new built-in functions in the JavaScript language and host environment, provide an optional parameter, usually called "context", which is designed as a work-around for you not having to use bind(..) to ensure your callback function uses a particular this. Ex: [1, 2, 3].forEach( foo, obj );  
    
  **new Binding**: do not make assumptions about similarity between class-oriented languages’ constructors and so called constructors in Javascript. In JS, constructors are just functions that happen to be called with the new operator in front of them. They are not attached to classes, nor are they instantiating a class. They are not even special types of functions. They're just regular functions that are, in essence, hijacked by the use of new in their invocation. When a function is invoked with new in front of it, otherwise known as a constructor call, the following things are done automatically:  
    
  1. a brand new object is created (aka, constructed) out of thin air  
  2. the newly constructed object is [[Prototype]]-linked  
  3. the newly constructed object is set as the this binding for that function call  
  4. unless the function returns its own alternate object, the new-invoked function call will automatically return the newly constructed object.  
    
  Precedence: new > explicit > implicit > default

### JS objects

* Functions are a sub-type of object. They are technically callable objects and are therefore considered to be first class objects.
* obj.prop (property access) vs obj[“prop”] (key access): using key access prop can be any UTF-8 compatible string, while property access can only use valid identifier names. Technically both methods access the same location in memory.
* As of ES6 you can use computed property names: obj[“abc” + “def”]
* There is no true concept of object methods in Javascript. Properties containing functions are only references and are not “owned” by the object itself even if the function is declared as part of the object literal. Also note that this bindings are set purely according to call-site rules and do not blindly refer to the containing object.
* **Arrays** are objects, so even though each index is a positive integer, you can also add properties onto the array. Adding named properties (regardless of . or [ ] operator syntax) does not change the reported length of the array. You could use an array as a plain key/value object, and never add any numeric indices, but this is a bad idea because arrays have behavior and optimizations specific to their intended use, and likewise with plain objects. Use objects to store key/value pairs, and arrays to store values at numeric indices. Be careful: If you try to add a property to an array, but the property name looks like a number, it will end up instead as a numeric index (thus modifying the array contents):
* Automatically creating deep copies in Javascript is highly complex due to possible circular references and function copying therefore no built-in function exists to do so. As of ES6, Object.assign(..) is provided to make shallow copies. Object.assign(..) takes a target object as its first parameter, and one or more source objects as its subsequent parameters. It iterates over all the enumerable, owned keys (immediately present) on the source object(s) and copies them (via = assignment only) to target and returns the newly created object.
* As of ES5 all properties have **property descriptors**: value, writable, enumerable, and configurable. By default all of the latter 3 are true, but can be modified using Object.defineProperty(..) to alter it’s behavior. The configurable property descriptor must be set to true to use Object.defineProperty(..). configurable:false also prevents the ability to use the delete keyword on that property. The enumerable property descriptor controls whether or not your properties show up in certain enumerations for instance for .. in loops.
* The **delete** **keyword** is only used to remove object properties, not free up allocated memory like in other languages. If the delete keyword is used and the property deleted is the last reference to some object/function it will be garbage collected.
* **Immutability**:  
    
  **Constants**: by combining writable:false and configurable:false, you can essentially create a constant (cannot be changed, redefined or deleted) as an object property  
    
  **Prevent Extensions**: if you want to prevent an object from having new properties added to it, but otherwise leave the rest of the object's properties alone, call Object.preventExtensions(..).  
    
  **Seal**: Object.seal(..) creates a "sealed" object, which means it takes an existing object and essentially calls Object.preventExtensions(..) on it, but also marks all its existing properties as configurable:false. So, not only can you not add any more properties, but you also cannot reconfigure or delete any existing properties (though you can still modify their values).  
    
  **Freeze**: Object.freeze(..) creates a frozen object, which means it takes an existing object and essentially calls Object.seal(..) on it, but it also marks all "data accessor" properties as writable:false, so that their values cannot be changed. This approach is the highest level of immutability that you can attain for an object itself, as it prevents any changes to the object or to any of its direct properties (though, as mentioned above, the contents of any referenced other objects are unaffected).
* **[[Get]] and [[Put]]**: are the 2 internal functions which handle property accesses. Either through object-literal syntax with get a() { .. } or through explicit definition with defineProperty(..) we can define our own get and set functions for an object property. When you define a property to have either a getter or a setter or both, its definition becomes an "accessor descriptor" (as opposed to a "data descriptor"). For accessor-descriptors, the value and writable characteristics of the descriptor are moot and ignored, and instead JS considers the set and get characteristics of the property (as well as configurable and enumerable).
* **Existence**: (“prop” in obj) or obj.hasOwnProperty(“prop”) evaluate to a boolean describing the existence of the property in an object. The in operator will check to see if the property is in the object, or if it exists at any higher level of the [[Prototype]] chain object traversal. By contrast, hasOwnProperty(..) checks to see if only myObject has the property or not, and will not consult the [[Prototype]] chain.
* Object.keys(..) returns an array of all enumerable properties, whereas Object.getOwnPropertyNames(..) returns an array of all properties, enumerable or not. propertyIsEnumerable(..) tests whether the given property name exists directly on the object and is also enumerable:true.
* ES5 also added several **iteration helpers for arrays**, including **forEach**(..), **every**(..), and **some**(..). Each of these helpers accepts a function callback to apply to each element in the array, differing only in how they respectively respond to a return value from the callback.  
    
  forEach(..) will iterate over all values in the array, and ignores any callback return values. every(..) keeps going until the end or the callback returns a false (or "falsy") value, whereas some(..) keeps going until the end or the callback returns a true (or "truthy") value. These special return values inside every(..) and some(..) act somewhat like a break statement inside a normal for loop, in that they stop the iteration early before it reaches the end.  
    
  The order of iteration over an object's properties is not guaranteed and may vary between different JS engines. Do not rely on any observed ordering for anything that requires consistency among environments.
* As of ES6, Javascript has a for .. of loop for iterating over arrays and objects. Arrays have a built-in @@iterator (iterator object), but Objects need a custom @@iterator to be defined before using a for .. of loop on it. This can be either declared directly with [Symbol.iterator]: or with Object.defineProperty(..., Symbol.iterator, …).

### The Object Prototype Chain

* **[[Prototype]]** is a link/reference to a possible chain of other related objects. A [[Prototype]] linkage can be made with Object.create(..) like: var obj2 = Object.create( obj1 ); When trying to access a property that does not exist, Javascript checks the [[Prototype]] chain. If no matching property is ever found by the end of the chain, the return result from the [[Get]] operation is undefined.
* Similar to this [[Prototype]] chain look-up process, if you use a for..in loop to iterate over an object, any property that can be reached via its chain (and is also enumerable) will be enumerated. If you use the in operator to test for the existence of a property on an object, in will check the entire chain of the object (regardless of enumerability).
* All Javascript objects have Object.prototype at the top of their [[Prototype]] chain which contains functions like .toString(), .valueOf(), .hasOwnProperty(), and .isPrototypeOf().
* **Shadowing**: when an object (obj1) sets the value of a property belonging not to itself, but belonging to an object (obj2) in its [[Prototype]] link chain. When this occurs, a property of the same name is created inside of obj1 and that property is said to shadow obj2’s version of the property.   
    
  Shadowing is complicated and not recommended as there are cleaner ways to execute this same concept. Be very careful when dealing with delegated properties that you modify. If you wanted to increment obj2.a, the only proper way is obj2.a++.
* In JavaScript, classes can't (being that they don't exist) describe what an object can do. The object defines its own behavior directly. There's just the object. True class instantiated objects are all separate copies of each other with their inherited functionality copied from their ancestors. Javascript “so-called class” objects are nothing more than separate objects pointing to each other and there is no true inheritance. Rather than inheritance a better description is delegation.
* Javascript has no special constructor functions. When you put the new keyword in front of a normal function call (any function), that makes that function call a "constructor call". In fact, new sort of hijacks any normal function and calls it in a fashion that constructs an object, in addition to whatever else it was going to do. Functions aren't constructors, but function calls are "constructor calls" if and only if new is used.
* **Linking object prototypes**:  
  obj1.prototype = Object.create( obj2.prototype ); // pre ES6  
  Object.setPrototypeOf( obj1.prototype, obj2.prototype ); // ES6+
* **Introspection**: obj instanceof ClassName;  
    
  The instanceof operator takes a plain object as its left-hand operand and a function as its right-hand operand. The question instanceof answers is: in the entire [[Prototype]] chain of a, does the object arbitrarily pointed to by Foo.prototype ever appear?
* Object.create(null) creates an object that has an empty (aka, null) [[Prototype]] linkage, and thus the object can't delegate anywhere. Since such an object has no prototype chain, the instanceof operator has nothing to check, so it will always return false. These special empty-[[Prototype]] objects are often called "**dictionaries**" as they are typically used purely for storing data in properties, mostly because they have no possible surprise effects from any delegated properties/functions on the [[Prototype]] chain, and are thus purely flat data storage.

### Behavior Delegation Oriented Design

* Do not attempt to setup an object heirarchy (as with classes and OOP) with Javascript rather create utility objects to organize common, reusable functionality to which other objects can delegate those tasks to. Rather than organizing the objects in your mind vertically, with Parents flowing down to Children, think of objects side-by-side, as peers, with any direction of delegation links between the objects as necessary. Other objects can now also delegate to the utility function. Store data properties on the delegator objects not the delegate (utility function) to maintain state.  
    
  var utilObj = { setID: function(ID) { this.id = ID } };  
  var obj1 = Object.create( utilObj );  
  obj1.init = function () { this.setID(); };
* Avoid naming function members the same on the delegator and delegate objects. In OOP this would be equivalent to polymorphism (overriding), in Javascript it is shawdowing which produces unreliable results and is not recommended.
* When calling a function which is delegated and this refers to the delegator (most common), Javascript will look for the function on the delegator and won’t find it then it will follow the [[Prototype]] chain and will find it on the delegate then execute it. Even though this is true, this will still refer to the delegator due to the call-site rules. Having our data members as part of the delegator object, this will be what we want to happen.
* You cannot create a cycle where two or more objects are mutually delegated (bi-directionally) to each other. If you make B linked to A, and then try to link A to B, you will get an error.

## The Module Pattern

### Variations

* **Augmentation**: useful to split up module implementation code. Allows overrides.  
  var MODULE = (function (api) { return api; } (MODULE));
* **Loose Augmentation**: augmentation with the added benefit of not having to be in a specific order. No Overrides.  
  var MODULE = (function (api) { return api; } (MODULE || {}));
* **Submodules**:  
  var MODULE.sub = (function () { return api; } ());

## Asynchrony

* **Async** is about the gap between now and later, but **parallel** is about things being able to occur simultaneously.
* Up until ES6, JavaScript itself has actually never had any direct notion of asynchrony built into it.
* Javascript is single threaded, but race conditions are still created via asynchrony.
* Use the **setTimeout(..0) (hack)** for async scheduling, which basically just means "stick this function at the end of the current event loop queue."
* As of ES6, there's a new concept layered on top of the event loop queue, called the "**Job queue**." The most likely exposure you'll have to it is with the asynchronous behavior of Promises. The "Job queue" is like a queue hanging off the end of every tick in the event loop queue.
* For long running processes, break the processing up into chunks so that it does not hog the event loop.
* Ways to avoid **race conditions** in non-deterministic situations:  
  ensure ordering: avoid pushing/queueing async results rather use specific/known locations  
  use gates: conditionals to test expected async results before proceding
* **Callbacks** are by far the most common way that asynchrony in JS programs is expressed and managed. Indeed, the callback is the most fundamental async pattern in the language.

## Promises

* Once a **Promise** is resolved, it stays that way forever -- it becomes an immutable value at that point -- and can then be observed as many times as necessary.
* Do not use instanceof Promise to recognize a Promise. It was decided that the way to recognize a Promise (or something that behaves like a Promise) would be to define something called a "**thenable**" as any object or function which has a then(..) method on it. This method of identification is known as duck typing (looks like a duck, quacks like a duck). It is not a bulletproof way to determine if it’s actually a Promise, be wary.
* Even when a task may finish synchronously, Promises are automatically called asynchronously so no need to insert setTimeout(..0) hacks to prevent race conditions.
* When a Promise is resolved, all then(..) registered callbacks on it will be called, in order, immediately at the next asynchronous opportunity and nothing that happens inside of one of those callbacks can affect/delay the calling of the other callbacks.
* Promises are defined so that they can only be resolved once. If for some reason the Promise creation code tries to call **resolve**(..) or **reject**(..) multiple times, or tries to call both, the Promise will accept only the first resolution, and will silently ignore any subsequent attempts. Of course, if you register the same callback more than once, (e.g., p.then(f); p.then(f);), it'll be called as many times as it was registered.
* If a Javascript error or exception is encountered before resolving a Promise, it will be rejected.
* It is good practice to wrap potentially Promise returning function calls in Promise.resolve(..). It will ensure that a trustable Promise will always be returned.
* Every time you call then(..) on a Promise, it creates and returns a new Promise, which we can chain with. The return value of the then call is set as the fulfillment value of the returned Promise:  
    
  p.then(function(v){ return ‘a’; })  
   .then(function(v){ return v + ‘b’; })  
   .then(function(v){ return v + ‘c’; }); // ‘abc’  
    
  These steps will execute in order even if they contain asynchronous operations.
* If an error is thrown along the Promise chain, the error will continue propagating until an explicitly defined rejection handler is encountered.
* In JS try..catch statements are synchronous and will not catch errors in asynchronous situations.

## Design Patterns

### Types

1. **Creational** patterns focus on ways to create objects or classes. This may sound simple (and it is in some cases), but large applications need to control the object creation process.
2. **Structural** design patterns focus on ways to manage relationships between objects so that your application is architected in a scalable way. A key aspect of structural patterns is to ensure that a change in one part of your application does not affect all other parts.
3. **Behavioral** patterns focus on communication between objects.

# jQuery

## jQuery References

* jQuery Best Practices

<http://lab.abhinayrathore.com/jquery-standards/>

* jQuery Cheat Sheet

<http://lab.abhinayrathore.com/jquery-cheatsheet/>

## jQuery Environment

* Code included inside $( window ).load(function() { ... }) will run once the entire page (images or iframes), not just the DOM, is ready.
* $(function() { … }); is shorthand for $( document ).ready(function() { … });
* It is good practice to pass a named function to document ready rather than an anonymous one: $( document ).ready( readyFn );
* You can change the jQuery alias name to avoid conflicts on the $ variable with other js libraries: var $j = jQuery.noConflict();
* The best approach to avoiding conflicts is by providing document ready with $ as an argument. This is best so the references to the $ variable need not be replaced throughout the code: jQuery( document ).ready(function( $ ) { … }); or you can use an immediately invoked function expression: (function( $ ) { … })( jQuery );

## The jQuery Object

* The jQuery Object has many similarities to an array, but it is not an array.
* The jQuery Object extends functionality and takes care of compatibility issues for native DOM elements. Simply wrap DOM elements in jQuery like so to benefit:  
  var target = document.getElementById( "target" );  
  $( target ).html( "<td>Hello <b>World</b>!</td>" );
* To get the native DOM element itself from a jQuery object, use .get()  
  var firstHeadingElem = $( "h1" ).get( 0 ); or  
  var firstHeadingElem = $( "h1" )[ 0 ];
* All jQuery objects are unique, even when created with the exact same selector, therefore you must compare their native DOM elements: $("#myElem").get(0) == $("#myElem ").get(0) evaluates to true.
* Native DOM methods and properties are not present on the jQuery object, and vice versa.
* jQuery object methods are part of the $.fn namespace, aka the "jQuery prototype".
* Methods in the $ namespace (aka core jQuery methods) are generally utility-type methods, and do not work with selections; they are not automatically passed any arguments, and their return value will vary. Do not confuse or make assumptions about similarities between methods in the $ namespace and those in the jQuery prototype even if they have the same name. Ex: $.each() vs .each(). Make sure you read the documentation to know the particular differences.

## Element Data

* Avoid adding properties to DOM elements to store data/info, instead use the .data() method which will manage any memory issues.
* A common thing to store via the **.data()** method is a relationship between elements:  
  $li.data( "contentDiv", $( "#contentDiv" ) );
* You can also pass .data() an object containing key value pairs.
* HTML data attributes (data-) are automatically accessible in jQuery via .data(). You can use either snake or camel case as the key passed to .data():  
  data-my-attribute=”blah” can be accessed via: .data(‘my-attribute’); or .data(‘myAttribute’);

## Selecting Elements

* When using the :visible and :hidden pseudo-selectors, jQuery tests the actual **visibility** of the element, not its CSS visibility or display properties. jQuery looks to see if the element's physical height and width on the page are both greater than zero except in the case of <tr> elements in which it checks the display property. Also, elements that have not been added to the DOM will always be considered hidden.
* It is important to know jQuery best practices for using selectors to retrieve jQuery collections. Minor details in selector choice can have a great impact on the amount of DOM traversal performed.
* This is the proper way to check for an empty set when retrieving jQuery collections:   
  if ( $("selector").length ) { … }
* Always leverage jQuery variable caching when possible to avoid unnecessary jQuery operations: var $divs = $( "div" );
* Filtering selections is very useful. Use: .has(), .not(), .filter(), .first(), .eq().
* **Pseudo selectors** for form elements are very useful. Use: :checked, :disabled, :enabled, :input, :selected. Also there are pseudo selectors for forms according to type, but beware of specific performance concerns.
* Setters return a jQuery object, allowing you to continue calling jQuery methods on your selection. Getters return whatever they were asked to get, so you can't continue to call jQuery methods on the value returned by the getter (no chaining).
* Using **.end()** in the chain will reset the jQuery collection back to what it originally was should you change it.
* Use discretion when chaining jQuery methods as it can make code difficult to modify and debug.
* Use .index() to determine a particular element’s index position in respect to its siblings.
* $('<img />') creates a new <img /> element to be inserted to the DOM.  
  $('img') selects all existing <img /> elements.
* Performance recommendations:  
  - avoid jQuery extensions (ex: “:even”) when possible  
  - avoid excessive specificity  
  - use id based selectors  
  - be specific on the right-hand side of your selector, and less specific on the left

## Manipulating Elements

* When moving, copying, or removing elements around the DOM, the returned jQuery collection will consist of the element(s) selected in the $ function not the manipulating method:  
  var val = $("#li123").appendTo("#my-list"); // val contains a reference to the element with id == li123
* Rather than moving an element around, if you need a new copy of an element to insert into the DOM somewhere, use .clone(): $( "#myList li:first" ).clone().appendTo( "#myList" );
* .empty() removes all child elements of a selection, .remove() removes all elements from the DOM matching a selection including all associated data set with .data() and bound events. .detach() does the same as .remove() except it leaves all data and events in place. .remove() is faster than .detach(), but using .detach() can be a great performance boost if doing heavy manipulation to elements which are frequently reinserted into the DOM.
* You can create new elements using the $ function. Advanced syntax uses an attribute array:  
    
  $( "<a/>", {  
   html: "This is a <strong>new</strong> link",  
   "class": "new",  
   href: "foo.html"  
  });
* You can also add html elements directly in string form using the various insertion methods (.append(), .prepend(), etc.), but always concatenate the html strings before insertion to avoid DOM touches.
* **.attr()** can take a string and a primitive, a string and a function, or an object containing key value pairs to modify.
* Do not use **.css()** as a setter only as a getter to check css values.
* The DOM is slow; you want to avoid manipulating it as much as possible. jQuery introduced **.detach()** in version 1.4 to help address this issue, allowing you to remove an element from the DOM while you work with it.
* jQuery won't tell you if you're trying to run a whole lot of code on an **empty selection** – it will proceed as though nothing's wrong. It's up to you to verify that your selection contains some elements.

## Traversing

* Retrieving the desired node or collection is accomplished by using traversal and filter methods.
* jQuery traversal methods are most often overloaded to accept string selectors, jQuery objects, etc. to filter down or traverse the current set of jQuery objects.
* Its best practice to avoid traversing too far. 1 or 2 moves is ok, and don’t traverse across containers if possible.

## Utility Functions

* $.each() iterates over arrays and objects.
* $.inArray() returns a value's index in an array, or -1 if the value is not in the array.
* $.extend() changes the properties of the first object using the properties of subsequent objects.
* $.proxy() allows you to provide the scope of this
* Use $.isArray(), $.isFunction(), $.isNumeric() instead of the typeof operator

## Iteration

* Use $.each() to replace fors on arrays and for ins on objects
* Many jQuery methods implicitly iterate over the entire collection, applying their behavior to each matched element. Conversely, the following is a list of methods that require .each():  
  [.attr()](http://api.jquery.com/attr/#attr1) (getter), [.css()](http://api.jquery.com/css/#css1) (getter), [.data()](http://api.jquery.com/data/#data2) (getter), [.height()](http://api.jquery.com/height/#height1) (getter), [.html()](http://api.jquery.com/html/#html1) (getter), [.innerHeight()](http://api.jquery.com/innerHeight/), [.innerWidth()](http://api.jquery.com/innerWidth/), [.offset()](http://api.jquery.com/offset/#offset1) (getter), [.outerHeight()](http://api.jquery.com/outerHeight/), [.outerWidth()](http://api.jquery.com/outerWidth/), [.position()](http://api.jquery.com/position/), [.prop()](http://api.jquery.com/prop/#prop1) (getter), [.scrollLeft()](http://api.jquery.com/scrollLeft/#scrollLeft1) (getter), [.scrollTop()](http://api.jquery.com/scrollTop/#scrollTop1) (getter), [.val()](http://api.jquery.com/val/#val1) (getter), [.width()](http://api.jquery.com/width/#width1) (getter)

## Effects

* When jQuery hides an element, it sets its CSS display property to none. This means the content will have zero width and height; it does not mean that the content will simply become transparent and leave an empty area on the page.
* Inside of an **animation callback**, you can use the keyword this to refer to the DOM element being animated. Also note that the callback will be called for each element in the jQuery object. This means that if your selector returns no elements, your animation callback will never run. You can solve this problem by testing whether your selection returned any elements; if not, you can just run the callback immediately.
* **.stop()** will immediately terminate all animations running on the elements in your selection. You might give end-users control over page animations by rigging a button they can click to stop the animations.
* **.delay()** is used to introduce a delay between successive animations:  
  $( "elem" ).hide( 500 ).delay( 1500 ).show( 300 );
* **jQuery.fx** holds a number of properties which controls how effects are implemented such as jQuery.fx.speeds, jQuery.fx.interval, jQuery.fx.off
* jQuery makes it possible to animate arbitrary CSS properties via the **.animate()** method. The .animate() method lets you animate to a set value, or to a value relative to the current value. Color-related properties cannot be animated with .animate() using jQuery out of the box. Color animations can easily be accomplished by including the color plugin.
* **Easing** describes the manner in which an effect occurs — whether the rate of change is steady, or varies over the duration of the animation. jQuery includes only two methods of easing: swing and linear. If you want more natural transitions in your animations, various easing plugins are available. As of jQuery 1.4, it is possible to do per-property easing when using the .animate() method.
* **Queues** allow a series functions to be executed asynchronously on an element. Rather than passing a callback directly to the animate function, we daisy chain a .queue() function which defines another link in the chain of animations. The default queue if none is specified as the first parameter is “fx”. You can define as many custom named queues as you like to setup concurrent animation chains. All custom queues need to be kicked off with .dequeue(“queueName”). We can also clear a queue or replace a queue along the daisy chain. You can also call .queue() without passing it functions, which will return the queue of that element as an array.

## Plugins

* A jQuery object contains all of the jQuery prototype methods and all of the elements that matched the selector when the $ function executed. It gets these methods from the $.fn object.
* To create a plugin add a function to $.fn
* To avoid conflicts with the $ variable with other js libs, use an [Immediately Invoked Function Expression](http://benalman.com/news/2010/11/immediately-invoked-function-expression/) to encapsulate the plugin code and pass in the jQuery object: (function ( $ ) { $.fn.pluginName = function() { … }; }( jQuery )); this will also enable us to have private members.
* Only use one slot in $.fn per plugin and pass parameters in.
* Pass in options which extend defaults and expose variables and event callbacks which are likely to be modified.
* Give your HTML elements classes or ids to allow users access to them.

## Widgets

* Widgets are stateful plugins. They are different from a standard jQuery plugin in two important ways. First, the context is an object, not a DOM element. Second, the context is always a single object, never a collection.
* The plugin name must contain a namespace exactly 1 level deep: $.widget(“ns.name”. { … })
* Private methods are preceded with an underscore: \_methodName: function() { }
* To call a method on a plugin instance, you pass the name of the method to the jQuery plugin. If you are calling a method that accepts parameters, you simply pass those parameters after the method name: ns.pluginName( "method", param );
* Add callbacks so users can react when the state of your plugin changes using the \_trigger method. It takes 3 params: the name of the callback, a native event object that initiated the callback, and a hash of data relevant to the event.
* Plugin instances are stored in jquery.data and are therefore directly linked to the DOM element and can be accessed directly.
* Use the \_destroy method to unapply the plugin to the element.

## Code Organization

* Use RequireJS for dependency management and build system.
* Separate code into Services and Modules. Use inheritance. Don’t put DOM references where they don’t belong. Units of functionality should be loosely coupled and should stand alone. Avoid direct communication between units, instead use custom events or pub/sub.
* An object literal is the simplest way to encapsulate code.
* Modules provide privacy for variables and functions while exposing a public API if desired.
* Anonymous functions are difficult to debug, maintain, test, or reuse. Instead, use an object literal to organize and name your handlers and callbacks.
* Use Promise.all() to

# AngularJS 2.0

# React

# Ember.js

# Backbone.js

* The single most important thing that Backbone can help you with is keeping your business logic separate from your user interface. When the two are entangled, change is hard; when logic doesn't depend on UI, your interface becomes easier to work with.
* **Model**:  
  Orchestrates data and business logic.  
  Loads and saves from the server.  
  Emits events when data changes.
* **View**:  
  Listens for changes and renders UI.  
  Handles user input and interactivity.  
  Sends captured input to the model.
* A **Collection** helps you deal with a group of related models, handling the loading and saving of new models to the server and providing helper functions for performing aggregations or computations against a list of models. Aside from their own events, collections also proxy through all of the events that occur to models within them, allowing you to listen in one place for any change that might happen to any model in the collection.
* **API Integration**: Backbone is pre-configured to sync with a RESTful API. Simply create a new Collection with the url property set to your resource endpoint url. The Collection and Model components together form a **direct mapping** of REST resources using the following methods:  
    
  GET /books/ ... collection.fetch();  
  POST /books/ ... collection.create();  
  GET /books/1 ... model.fetch();  
  PUT /books/1 ... model.save();  
  DEL /books/1 ... model.destroy();
* Set the **parse** property given to .extend( .. ) to a function to properly handle/signify where the data actually is in the response object when the data is buried in another structure and not the actual root object structure.
* **Events** is a module that can be mixed in to any object, giving the object the ability to bind and trigger custom named events to callback functions which accept parameters. The convention is to namespace events using a “:” like “event-class:event-name”.
* All Backbone event methods also support an **event map** syntax when registering multiple events.
* To supply a context value for **this** when the callback is invoked, pass the optional last argument.

Defining models:  
Backbone.Model.extend({ … });

Validation:  
Use the validate function inside of a model for validation functionality

# RequireJS

## RequireJS References

* [http://www.requirejs.org/docs/jquery.html#noconflictmap](http://www.requirejs.org/docs/jquery.html%23noconflictmap)  
  Using RequireJS with jQuery
* Only one script tag is needed to include RequireJS and declare where the initialization point of the app is:  
  <script data-main="scripts/main" src="scripts/require.js"></script>
* All code is wrapped in require() and define() functions. RequireJS uses dependency injection to load modules. If a.js is structured as below, then it is indicated that module “a” is dependent on b and c which would be defined in b.js and c.js:  
  define(["b","c"], function(b,c) { … });
* General rule of thumb is you use define when you want to define a module that will be reused by your application and you use require to simply load a dependency.
* RequireJS waits until all the dependent modules are loaded before executing the functionality. If any modules are missing, it doesn’t execute any code. This helps us maintain the consistency of our data.
* RequireJS uses Asynchronous Module Loading (AMD) for loading files. Each dependent module will start loading through asynchronous requests in the given order. Even though the file order is considered, we cannot guarantee that the first file is loaded before the second file due to the asynchronous nature. So, RequireJS allows us to use the shim config to define the sequence of files which need to be loaded in correct order:  
    
  requirejs.config({  
   shim: {  
   'source1': ['dependency1','dependency2'],  
   'source2': ['source1']  
   }  
  });

# Linux

## Shell

* **Ctrl-A** – go to the beginning of prompt
* **Ctrl-E** – go to the end of prompt
* **Ctrl-R** – reverse history search
* **Symbolic links** are pointers to files. Useful when users don’t have access to certain files.
* Shutting down – **shutdown**, **poweroff**, **init 0**
* User information – **w**, **who**
* Monitoring processes – **top** – kill processes using the PID  
  More robust process monitor - **htop**
* Network activity – **netstat**
* Use **man** (manual) to pull up any documentation on a command
* Every process that is being run by the Linux kernel has 3 channels **STDIN**(0), **STDOUT**(1), **STDERR**(2)
* Redirect output to files using **>** or **>>** to append.
* The | hooks up the STDOUT of one command to the STDIN of another
* **grep** is a powerful tool to search for text
* Only users who have **sudo** powers can use sudo. The first user created on the system in Ubuntu automatically gets sudo powers
* **apt-get** is the package manager for Ubuntu – use **update**, **upgrade**, **search**, **install**, **remove**
* Ubuntu is a Linux distribution of Debian
* File permissions: 10 bits  
  1st bit is filetype, next 3 are owner perms, next 3 are group perms, next 3 are world perms
* **chmod** codes:  
  0 = no permissions whatsoever; this person cannot read, write, or execute the file  
  1 = execute only  
  2 = write only  
  3 = write and execute (1+2)  
  4 = read only  
  5 = read and execute (4+1)  
  6 = read and write (4+2)  
  7 = read and write and execute (4+2+1)
* When using chmod on directories, use -R to recursively set the same perms to everything inside.
* /etc/login.defs  
  UMASK – permissions which you take away (are not given) when creating a file/directory
* Use chown to change owner of a file/directory
* /etc/passwd - users  
  /etc/group - groups  
  /etc/shadow - passwords  
  /etc/skel – skeleton directory which is copied as the home directory when a new user is created.
* useradd – create a new user  
  userdel – delete a user  
  passwd – set a user’s password  
  usermod – perform tasks on user such as locking and unlocking account  
  newusers – batch mode for creating new users  
  whoami – prints out the current user  
  id – shows current user’s ids: uid, gid, group ids

# PHP

## Laravel

## Yii

## CodeIgniter

## Symfony

## Phalcon

## Wordpress

## Drupal

# Ruby

## Ruby on Rails

# Python

## Django

# Go

# Java

# Objective-C

# Swift

# Meteor

# Node.js

# Ionic

# Phonegap / Cordova

# MySQL

# PostgreSQL

# MongoDB

# Firebase

# Redis

# Oracle

# DDP

# Regular Expressions

## RegEx References

* Awesome testing and learning site

<http://www.regexr.com/>

* Easy to read tutorial  
  <https://ponyfoo.com/articles/learn-regular-expressions>
* **Modifiers**: i - case insensitive, g - global
* **Anchors**: ^ - beginning of string, $ - end of string
* **Quantifiers**: ? – optionally matches preceding string, + - must occur at least once, \* - “” 0, 1, or more times, {n} – “” exactly n times, {n,} – “” at least n times, {n,m} – “” from n to m times.
* **Built-in Patterns** - . – any char except new line, \ - will escape any char, \s – whitespace, \d – digits same as [0-9], \D opposite of \d, \w – words same as [A-z0-9], \W opposite of \w
* There are two kinds of groups. Capturing, and non-capturing. Capturing groups will match strings and can be referenced like $1, $2, … Adding ?: at the beginning of the group signifies a non-capturing string.
* Javascript String.prototype.replace function: this function takes a regex as the first parameter and the second param could be either a string, a string with replacements ($1, $2, etc.), or a callback function to calculate replacement values. The supplied parameters to the callback are as follows: (match (the matched substring), p1, p2, … (the group values), offset (the offset of the matched substring), string (the entire string being examined))

# Software Development Life Cycles

## Waterfall Model

* Waterfall is the oldest and most straightforward of the structured SDLC methodologies — finish one phase, then move on to the next. No going back. Each stage relies on information from the previous stage and has its own project plan. Waterfall is easy to understand and simple to manage. But early delays can throw off the entire project timeline. And since there is little room for revisions once a stage is completed, problems can’t be fixed until you get to the maintenance stage. This model doesn’t work well if flexibility is needed or if the project is long term and ongoing.

## V-Shaped Model

* Also known as the Verification and Validation model, the V-shaped model grew out of Waterfall and is characterized by a corresponding testing phase for each development stage. Like Waterfall, each stage begins only after the previous one has ended. This model is useful when there are no unknown requirements, as it’s still difficult to go back and make changes.

## Iterative Model

* The Iterative model is repetition incarnate. Instead of starting with fully known requirements, you implement a set of software requirements, then test, evaluate and pinpoint further requirements. A new version of the software is produced with each phase, or iteration. Rinse and repeat until the complete system is ready.
* One advantage over other SDLC methodologies: This model gives you a working version early in the process and makes it less expensive to implement changes. One disadvantage: Resources can quickly be eaten up by repeating the process again and again.

## Spiral Model

* One of the most flexible SDLC methodologies, the Spiral model takes a cue from the Iterative model and its repetition; the project passes through four phases over and over in a “spiral” until completed, allowing for multiple rounds of refinement. This model allows for the building of a highly customized product, and user feedback can be incorporated from early on in the project. But the risk you run is creating a never-ending spiral for a project that goes on and on.

## Big Bang Model

* A bit of an anomaly among SDLC methodologies, the Big Bang model follows no specific process, and very little time is spent on planning. The majority of resources are thrown toward development, and even the client may not have a solid grasp of the requirements. This is one of the SDLC methodologies typically used for small projects with only one or two software engineers.
* Big Bang is not recommended for large or complex projects, as it’s a high-risk model; if the requirements are misunderstood in the beginning, you could get to the end and realize the project may have to be started all over again.

## Agile Model

* By breaking the product into cycles, the Agile model quickly delivers a working product and is considered a very realistic development approach. The model produces ongoing releases, each with small, incremental changes from the previous release. At each iteration, the product is tested.
* This model emphasizes interaction, as the customers, developers and testers work together throughout the project. But since this model depends heavily on customer interaction, the project can head the wrong way if the customer is not clear on the direction he or she wants to go.

# Agile

Agile methods support a broad range of the software development life cycle. Some focus on the practices (e.g., XP, pragmatic programming, agile modeling), while some focus on managing the flow of work (e.g., Scrum, Kanban). Some support activities for requirements specification and development (e.g., FDD), while some seek to cover the full development life cycle (e.g., DSDM, RUP).  
  
Popular agile software development frameworks include (but are not limited to):

### [Adaptive software development](https://en.wikipedia.org/wiki/Adaptive_software_development) (ASD)

### [Agile modeling](https://en.wikipedia.org/wiki/Agile_modeling)

### [Agile Unified Process](https://en.wikipedia.org/wiki/Agile_Unified_Process) (AUP)

### [Crystal Clear methods](https://en.wikipedia.org/wiki/Crystal_Clear_(software_development))

### [Disciplined agile delivery](https://en.wikipedia.org/wiki/Disciplined_agile_delivery)

### [Dynamic systems development method](https://en.wikipedia.org/wiki/Dynamic_systems_development_method) (DSDM)

### [Extreme programming](https://en.wikipedia.org/wiki/Extreme_programming) (XP)

### [Feature-driven development](https://en.wikipedia.org/wiki/Feature-driven_development) (FDD)

### [Lean software development](https://en.wikipedia.org/wiki/Lean_software_development)

### [Kanban](https://en.wikipedia.org/wiki/Kanban_(development))

### [Scrum](https://en.wikipedia.org/wiki/Scrum_(software_development))

### [Scrumban](https://en.wikipedia.org/wiki/Scrumban)

## Agile practices

Agile development is supported by a number of concrete practices, covering areas like requirements, design, modelling, coding, testing, planning, risk management, process, quality, etc. Some notable agile practices include:

### [Acceptance test-driven development](https://en.wikipedia.org/wiki/Acceptance_test-driven_development) (ATDD)

### [Agile modeling](https://en.wikipedia.org/wiki/Agile_modeling)

### [Backlogs](https://en.wikipedia.org/wiki/Scrum_(development)#Product_Backlog) (Product and Sprint)

### [Behavior-driven development](https://en.wikipedia.org/wiki/Behavior-driven_development) (BDD)

### Business analyst designer method (BADM)

### [Cross-functional team](https://en.wikipedia.org/wiki/Cross-functional_team)

### [Continuous integration](https://en.wikipedia.org/wiki/Continuous_integration) (CI)

### [Domain-driven design](https://en.wikipedia.org/wiki/Domain-driven_design) (DDD)

### Information radiators (scrum board, task board, visual management board, [burndown chart](https://en.wikipedia.org/wiki/Burndown_chart))

### [Iterative and incremental development](https://en.wikipedia.org/wiki/Iterative_and_incremental_development) (IID)

### [Pair programming](https://en.wikipedia.org/wiki/Pair_programming)

### [Planning poker](https://en.wikipedia.org/wiki/Planning_poker)

### [Refactoring](https://en.wikipedia.org/wiki/Refactoring)

### [Scrum events](https://en.wikipedia.org/wiki/Scrum_(development)) (sprint planning, daily scrum, sprint review and retrospective)

### [Test-driven development](https://en.wikipedia.org/wiki/Test-driven_development) (TDD)

### [Agile testing](https://en.wikipedia.org/wiki/Agile_testing)

### [Timeboxing](https://en.wikipedia.org/wiki/Timeboxing)

### [User story](https://en.wikipedia.org/wiki/User_story)

### [Story-driven modeling](https://en.wikipedia.org/wiki/Story-driven_modeling)

### [Retrospective](https://en.wikipedia.org/wiki/Retrospective)

### [Velocity tracking](https://en.wikipedia.org/wiki/Velocity_(software_development))

### User Story Mapping

# Software Testing Methodologies

## Functional Testing

## Unit Testing

## Integration Testing

## System Testing

## Acceptance Testing

## Non-Functional Testing

## Performance, Load, Stress Testing

## Security, Vulnerability Testing

## Usability Testing

## Compatibility Testing

## A/B Testing

# Unit Testing Frameworks

## Mocha

## Chai

## Jasmine

## PhantomJS

## QUnit

## Sinon

# UI/UX Design

# Marketing Concepts

# getmwf

## Styles

### Color

* Only use brand color palette colors for primary brand elements and calls to action.
* When coupling neutrals with accent and brand colors, make sure there is suitable contrast between them.
* Position darker colors next to lighter neutrals, and brighter colors next to darker neutrals.

### Grid

* Base Unit: try to use multiples of 4 in sizing your elements.
* The grids content column reaches a maximum width of 1600 pixels. Its recommended to cap images at 1920px.

### Motion

* For on-screen transitions, use a short ease-in (exit animations) and long ease-out (entrance animations). Avoid linear transitions.
* Avoid having elements pass through each other.
* Items should not simply appear, nor should they shift about as other elements load.
* Use short fades to introduce new items, and cascade item entrances from the top down.
* Translate objects in from off of the canvas.
* Only use instant feedback for user interaction.
* Where possible, avoid animating elements in multiple directions.
* Use motion to reinforce content hierarchy and reveal user opportunities.
* Use directional transitions to suggest off-screen opportunities for navigation.
* Direct user to items that can be found by animating FROM the source of interaction.
* Use motion direction to inform the user of connections between separate content on screen.
* Use Feedback animations to inform the user of system intent, or reveal additional information

### Typography

* All-caps should only be used in calls-to-action, kickers, and badges. For badges, strings consist of one or two words inside a solid container. For calls to action, the all caps strings are coupled with a glyph. Always use 13px Segoe UI Regular with a tracking value of 75.
* Use bold to add visual interest, for legibility on top of imagery, and for content hierarchy. Use bold in scenarios with paired primary and secondary strings (ie: product name and price). The only sizes able to be bolded are 20/24px (mosaic headlines) and 34/40px (hero headlines).

### Imagery

#### Don’ts:

* No clear focal point
* Subject matter seems ambiguous
* Features non-Microsoft products
* Awkward cropping of faces or limbs
* Different light sources
* Busy graphics
* Distorted depth of field
* Awkward composition

# Picchu

# ARIA