

Different types of directive scopes

Scope : False (Directive uses its parent scope)

Scope : True (Directive gets a new scope)

When scope is set to `scope: true`, AngularJS will create a new scope by inheriting parent scope (usually controller scope, otherwise the application's rootScope). Any changes made to this new scope will not reflect back to the parent scope. However, since the new scope is inherited from the parent scope, any changes made in the Ctrl1 (the parent scope) will be reflected in the directive scope.

When scope is set to `scope: false`, the controller Ctrl1 and directive are using the same scope object. This means any changes to the controller or directive will be in sync.

Scope : { } (Directive gets a new isolated scope)

This is the most interesting section. Till now, we saw two situations for directive scope creation. In the third type, we are going to set scope property in DDO to an Object literal. When an object literal is passed to the scope property, things are bit different. This time, there will be a new scope created for the directive, but it will not be inherited from the parent scope. This new scope also known as Isolated scope because it is completely detached from its parent scope.

So far, this is the recommended way of setting the scope on DDO while creating custom directives. Why? Because:

It'll make sure that our directive is generic, and placed anywhere inside the application. Parent scope is not going to interfere with the directive scope

There are 3 types of prefixes AngularJS provides used to communicate with parent directive.

- 1. "@" (Text binding / one-way binding)
- 2. "=" (Direct model binding / two-way binding)]
- 3. "&" (Behaviour binding / Method binding)

Summary of Compile, Controller, Pre-link & Post-link

Reference: <http://www.undefinednull.com/2014/07/07/practical-guide-to-prelink-postlink-and-controller-methods-of-angular-directives/>

Post : This is the most commonly used for data binding

Safe to attach event handlers to the DOM element

All children directives are linked, so it's safe to access them

Never set any data required by the child directive here. Because child directive's will be linked already.

Pre :

Used rarely. One of the use case is when a child directive requires data from its parent, the parent directive should set it through its pre-link function.

Set data required for its child directives

Safe to attach event handlers to the DOM element

Not safe to access DOM elements belong to child directives. Theyâ€™re not linked yet.

Controller :

Used for defining a proper API to the directive. Using controller, directives can communicate and share data each other.

Set the data required to other directives.

Never access DOM element inside the controller; itâ€™s against Angularâ€™s philosophy and make testing hard.

Angular directives - How to declare the various functions?

Compile, Controller, Pre-link & Post-link

If one is to use all four function, the directive will follow this form:

```

1  myApp.directive( 'myDirective', function () {
2      return {
3          restrict: 'EA',
4          controller: function( $scope, $element, $attrs, $transclude ) {
5              // Controller code goes here.
6          },
7          compile: function compile( tElement, tAttributes, transcludeFn ) {
8              // Compile code goes here.
9              return {
10                 pre: function preLink( scope, element, attributes, controller, transcludeFn ) {
11                     // Pre-link code goes here
12                 },
13                 post: function postLink( scope, element, attributes, controller, transcludeFn ) {
14                     // Post-link code goes here
15                 }
16             };
17         }
18     };
19 });
20
```

Notice that compile returns an object containing both the pre-link and post-link functions; in Angular lingo we say the compile function returns a template function.

Compile, Controller & Post-link

If **pre-link** isn't necessary, the compile function can simply return the post-link function instead of a definition object, like so:

```

1  myApp.directive( 'myDirective', function () {
2      return {
3          restrict: 'EA',
4          controller: function( $scope, $element, $attrs, $transclude ) {
5              // Controller code goes here.
6          },
7          compile: function compile( tElement, tAttributes, transcludeFn ) {

```

```

8         // Compile code goes here.
9         return function postLink( scope, element, attributes, controller, tra
10             // Post-link code goes here
11     };
12     }
13 };
14 });
15

```

Sometimes, one wishes to add a **compile** method, after the (post) **link** method was defined. For this, one can use:

```

1  myApp.directive( 'myDirective', function () {
2      return {
3          restrict: 'EA',
4          controller: function( $scope, $element, $attrs, $transclude ) {
5              // Controller code goes here.
6          },
7          compile: function compile( tElement, tAttributes, transcludeFn ) {
8              // Compile code goes here.
9          }
10         return this.link;
11     },
12     link: function( scope, element, attributes, controller, transcludeFn ) {
13         // Post-link code goes here
14     }
15 };
16 });
17
18

```

Controller & Post-link

If no compile function is needed, one can skip its declaration altogether and provide the post-link function under the **link** property of the directive's configuration object:

```

1  myApp.directive( 'myDirective', function () {
2      return {
3          restrict: 'EA',
4          controller: function( $scope, $element, $attrs, $transclude ) {
5              // Controller code goes here.
6          },
7          link: function postLink( scope, element, attributes, controller, transclu
8              // Post-link code goes here
9          },
10     };
11 });
12

```

No controller

In any of the examples above, one can simply remove the **controller** function if not needed. So for instance, if only **post-link** function is needed, one can use:

In which order the directive functions are executed?

For a single directive

Based on the following plunk, consider the following HTML markup:

```
1 <div log="some-div"></div>
2
3
4
```

?

With the following directive declaration:

```
1 myApp.directive('log', function() {
2     return {
3         controller: function( $scope, $element, $attrs, $transclude ) {
4             console.log( $attrs.log + ' (controller)' );
5         },
6         compile: function compile( tElement, tAttributes ) {
7             console.log( tAttributes.log + ' (compile)' );
8             return {
9                 pre: function prelink( scope, element, attributes ) {
10                     console.log( attributes.log + ' (pre-link)' );
11                 },
12                 post: function postLink( scope, element, attributes ) {
13                     console.log( attributes.log + ' (post-link)' );
14                 }
15             };
16         }
17     };
18 });
19
20
```

?

The console output will be:

```
1 some-div (compile)
2 some-div (controller)
3 some-div (pre-link)
4 some-div (post-link)
5
```

?

We can see that compile is executed first, then controller, then pre-link and finally followed by post-link.

For nested directives

```
1 Note: The following does not apply to directives that render their children in t
2
```

The original HTML markup is often made of nested elements, each with its own directive. Like in the following markup (see plunk):

```
1 <div log="parent">
2     <div log="..first-child"></div>
3     <div log="..second-child"></div>
4 </div>
5
6
7
```

?

The console output will look like this (without the comments):

```
1 // The compile phase
2 parent (compile)
3 ..first-child (compile)
4 ..second-child (compile)
5
```

?

```

6 // The link phase
7 parent (controller)
8 parent (pre-link)
9 ..first-child (controller)
10 ..first-child (pre-link)
11 ..first-child (post-link)
12 ..second-child (controller)
13 ..second-child (pre-link)
14 ..second-child (post-link)
15 parent (post-link)
16

```

We can distinguish two phases here - the compile phase and the link phase.

The compile phase

When the DOM is loaded Angular starts the compile phase, where it traverses the markup top-down, and calls compile on all directives. Graphically, we could express it like so:



It is perhaps important to mention that at this stage, the templates the compile function gets are the source templates (not instance template).

The link phase

DOM instances are often simply the result of a source template being rendered to the DOM, but they may be created by ng-repeat, or introduced on the fly.

Whenever a new instance of an element with a directive is rendered to the DOM, the link phase starts.

In this phase, Angular calls controller, pre-link, iterates children, and call post-link on all directives, like so:



What is the difference between a source template and an instance template?

The fact that Angular allows DOM manipulation means that the input markup into the compilation process sometimes differ from the output. Particularly, some input markup may be cloned a few times (like with ng-repeat) before being rendered to the DOM.

Angular terminology is a bit inconsistent, but it still distinguishes between two types of markups:

- **Source template** - the markup to be cloned, if needed. If cloned, this markup will not be rendered to the DOM.
- **Instance template** - the actual markup to be rendered to the DOM. If cloning is involved, each instance will be a clone.

The following markup demonstrates this:

```

1 <div ng-repeat="i in [0,1,2]">
2   <my-directive>{{i}}</my-directive>
3 </div>
4

```



The source html defines

```
1 <my-directive>{{i}}></my-directive>  
2
```



which serves as the source template

But as it is wrapped within an ng-repeat directive, this source template will be cloned (3 times in our case). These clones are instance template, each will appear in the DOM and be bound to the relevant scope.

Function nature, do's and dont's

Compile

Each directive's compile function is only called once, when Angular bootstraps.

Officially, this is the place to perform (source) template manipulations that do not involve scope or data binding.

Primarily, this is done for optimisation purposes; consider the following markup:

```
1 <my-raw></my-raw>  
2  
3  
4
```



The directive will render a particular DOM Markup. So we can either:

- Allow ng-repeat to duplicate the source template (), only then modify the markup of each instance template (outside the compile function).
- First modify the source template to involve the desired markup (in the compile function), only then allow ng-repeat duplicating it.

If there are 1000 raws, the latter option may be faster than the former one.

Do's

- Manipulate markup so it serves as a template to instances (clones).

Dont's

- Attach event handlers..
- Inspect child elements.
- Setup observations on attributes.
- Setup watches on the scope.

;

ngroute vs ui router?

ui-router is a 3rd-party module and is very powerful. It supports everything the normal ngRoute can do as well as many extra functions.

Here are some common reason ui-router is chosen over ngRoute:

- ui-router allows for nested views and multiple named views. This is very useful with larger app where you may have pages that inherit from other sections.
- ui-router allows for you to have strong-type linking between states based on state names. Change the url in one place will update every link to that state when you build your links with ui-sref. Very useful for larger projects where URLs might change.
- There is also the concept of the decorator which could be used to allow your routes to be dynamically created based on the URL that is trying to be accessed. This could mean that you will not need to specify all of your routes before hand.
- states allow you to map and access different information about different states and you can easily pass information between states via \$stateParams.
- You can easily determine if you are in a state or parent of a state to adjust UI element (highlighting the navigation of the current state) within your templates via \$state provided by ui-router which you can expose via setting it in \$rootScope on run.

In essence, ui-router is ngRouter with more features, under the sheets it is quite different. These additional features are very useful for larger applications.

State Manager.

The simplest form of state A state in its simplest form can be added like this (typically within module.config):

```

1  <!-- in index.html -->
2
3      <section ui-view=""></section>
4
5  // in app-states.js (or whatever you want to name it)
6  $stateProvider.state('contacts', {
7      template: '<h1>My Contacts</h1>'
8  })
9

```

;

How to empty an array in JavaScript?

Method 1

If you need to keep the original array because you have other references to it that should be updated too, you can clear it without creating a new array by setting its length to zero:

```

1  A.length = 0;
2

```

Method 2

This solution is not very succinct but it is by far the fastest solution (apart from setting the array to a new array). If you care about performance, consider using this method to clear an array. Benchmarks show that this is at least 10 times faster than setting the length to 0 or using splice().

```

1  while(A.length > 0) {
2      A.pop();
3  }

```

What is the difference between using the delete operator on the array element as opposed to using the

Array.splice method?

delete does remove the element, but does not reindex the array or update its length. The deleted element will shown as undefined.

```
1  myArray = ['a', 'b', 'c', 'd']
2  // ["a", "b", "c", "d"]
3  delete myArray[0]
4  // true
5  myArray
6  // [undefined, "b", "c", "d"]
7  myArray.splice(0, 2)
8  // [undefined, "b"]
9  myArray
10 // ["c", "d"]
11
```



Loop through array in JavaScript?

Use a sequential for loop:

```
1  var myStringArray = ["Hello","World"];
2  var arrayLength = myStringArray.length;
3  for (var i = 0; i < arrayLength; i++) {
4      alert(myStringArray[i]);
5  }
6
```



the for...in statement should be avoided, this is meant to enumerate object properties.

It shouldn't be used for array-like objects because:

- The order of iteration is not guaranteed, the array indexes may not visited in the numeric order.
- Inherited properties are also enumerated.

The for-in statement as I said before is there to enumerate object properties, for example:

```
1  var obj = {
2      "a": 1,
3      "b": 2,
4      "c": 3
5  };
6
7  for (var prop in obj) {
8      if (obj.hasOwnProperty(prop)) {
9          // or if (Object.prototype.hasOwnProperty.call(obj,prop)) for safety...
10         alert("prop: " + prop + " value: " + obj[prop])
11     }
12 }
```



Remove specific element from an array?

The splice() method changes the content of an array, removing/adding new elements while removing old elements.

index - Index at which to start changing the array. If greater than the length of the array, actual starting index will be set to the length of the array. If negative, will begin that many elements from the end.

howMany - An integer indicating the number of old array elements to remove. If howMany is 0, no elements are removed. In this case, you should specify at least one new element. If howMany is greater than the number of elements left in the array starting at index, then all of the elements through the end of the array will be deleted. If no howMany parameter is specified (second syntax above, which is a SpiderMonkey extension), all elements after index are removed.

element1, ..., elementN - The elements to add to the array. If you don't specify any elements, splice simply removes elements from the array.

Returns - An array containing the removed elements. If only one element is removed, an array of one element is returned. If no elements are removed, an empty array is returned.

```
1  var myFish = ["angel", "clown", "mandarin", "surgeon"];
2  //removes 0 elements from index 2, and inserts "drum"
3  var removed = myFish.splice(2, 0, "drum");
4  //myFish is ["angel", "clown", "drum", "mandarin", "surgeon"]
5  //removed is [], no elements removed
6
7  //removes 1 element from index 3
8  removed = myFish.splice(3, 1);
9  //myFish is ["angel", "clown", "drum", "surgeon"]
10 //removed is ["mandarin"]
11
12 //removes 1 element from index 2, and inserts "trumpet"
13 removed = myFish.splice(2, 1, "trumpet");
14 //myFish is ["angel", "clown", "trumpet", "surgeon"]
15 //removed is ["drum"]
16
17 //removes 2 elements from index 0, and inserts "parrot", "anemone" and "blue"
18 removed = myFish.splice(0, 2, "parrot", "anemone", "blue");
19 //myFish is ["parrot", "anemone", "blue", "trumpet", "surgeon"]
20 //removed is ["angel", "clown"]
21
22 //removes 2 elements from index 3
23 removed = myFish.splice(3, Number.MAX_VALUE);
24 //myFish is ["parrot", "anemone", "blue"]
25 //removed is ["trumpet", "surgeon"]
26
```

Appending to array?

Use push method

```
1  var arr = [  
2    "Hi",  
3    "Hello",  
4    "Bonjour"  
5  ];  
6  
7  // append new value to the array  
8  arr.push("Hola");  
9  
10 // To push specified number of arguments  
11 arr.push('second', 'third');  
12  
13 // To push entire array  
14 Array.prototype.push.apply( arr, ["Hola", "Hask"]);  
15
```

?

Use concat method

```
1  var ar1 = [1, 2, 3];  
2  var ar2 = [4, 5, 6];  
3  var ar3 = ar1.concat(ar2);  
4
```

?

Why does [1,2] + [3,4] = "1,23,4" in JavaScript?

JavaScript's + operator has two purposes: adding two numbers, or joining two strings. It doesn't have a specific behaviour for arrays, so it's converting them to strings and then joining them.

If you want to join two arrays to produce a new one, use the .concat method instead:

```
1  [1, 2].concat([3, 4]) // [1, 2, 3, 4]  
2
```

?

fastest way to add all the elements of the array?

```
1  var arr = [11,5,7,9];  
2  eval( arr.join("+") );  
3
```

?

different REST methods used in backbone?

The HTTP Protocol

I like examples, so here is an HTTP request to get the HTML for this page:

```
1 GET /questions/18504235/understand-backbone-js-rest-calls HTTP/1.1
2 Host: stackoverflow.com
3
```

[Optional] If you have ever played with command line or terminal, try running the command `telnet stackoverflow.com 80` and pasting in the above, followed by pressing enter a couple of times. Voila! HTML in all of it's glory.

In this example...

- GET is the method.
- /questions/18504235/understand-backbone-js-rest-calls is the path.
- HTTP/1.1 is the protocol.
- Host: stackoverflow.com is an example of a header.

Your browser does approximately the same, just with more headers, in order to get the HTML for this page. Cool, huh?

Since you work in front end, you've probably seen the form tag many times. Here's an example of one:

```
1 <form action="/login" method="post">
2   <input type="text" name="username">
3   <input type="password" name="password">
4   <input type="submit" name="submit" value="Log In">
5 </form>
6
```

When you submit this form along with appropriate data, your browser sends a request that looks something like this:

```
1 POST /login HTTP/1.1
2 Host: stackoverflow.com
3
4 username=testndtv&password=zachrabbitisawesome123&submit=Log%20In
5
```

There are three differences between the previous example and this one.

- The method is now POST.
- The path is now /login.
- There is an extra line, called the body.

While there are a bunch of other methods, the ones used in RESTful applications are POST, GET, PUT, and DELETE. This tells the server what type of action it's supposed to take with the data, without having to have different paths for everything.

Back to Backbone

So hopefully now you understand a bit more about how HTTP works. But how does this relate to Backbone? Let's find out!

Here's a small chunk of code you might find in a Backbone application.

```
1 var BookModel = Backbone.Model.extend({
```

```
2     urlRoot: '/books'
3   });
4   var BookCollection = Backbone.Collection.extend({
5     model: BookModel
6     , url: '/books'
7   });
8
```

Create (POST)

Since we're using a RESTful API, that's all the information Backbone needs to be able to create, read, update, and delete all of our book information! Let's start by making a new book. The following code should suffice:

```
1   var brandNewBook = new BookModel({ title: '1984', author: 'George Orwell' });
2   brandNewBook.save();
3
```

Backbone realizes you're trying to create a new book, and knows from the information it's been given to make the following request:

```
1   POST /books HTTP/1.1
2   Host: example.com
3
4   {"title":"1984","author":"George Orwell"}
5
```

Read (GET)

See how easy that was? But we want to get that information back at some point. Let's say we ran `new BookCollection().fetch()`. Backbone would understand that you're trying to read a collection of books, and it would make the following request:

```
1   GET /books HTTP/1.1
2   Host: example.com
3
```

BAM. That easy. But say we only wanted the information for one book. Let's say book #42. Say we ran `new BookModel({ id: 42 }).fetch()`. Backbone sees you're trying to read a single book:

```
1   GET /books/42 HTTP/1.1
2   Host: example.com
3
```

Update (PUT)

Oh darn, I just realized I spelled Mr. Orwell's name wrong. Easy to fix!

```
1   brandNewBook.set('author', 'George Orwell');
2   brandNewBook.save();
3
```

Backbone is smart enough to know that despite being called `brandNewBook`, it's already been saved. So it **updates** the book:

```
1   PUT /books/84 HTTP/1.1
2   Host: example.com
3
4   {"title":"1984","author":"George Orwell"}
5
```

Delete (DELETE)

Finally, you realize that the government is tracking your every move, and you need to bury the fact that you have read 1984. It's probably too late, but it never hurts to try. So you run `brandNewBook.destroy()`, and Backbone becomes sentient and realizes your danger deletes the book with the following request:

```
1 DELETE /books/84 HTTP/1.1
2 Host: example.com
3
```

And it's gone.

Other Useful Tidbits

While we've talked a lot about what we're sending TO the server, we should probably also take a look at what we're getting back. Let's return to our collection of books. If you remember, we made a GET request to `/books`. In theory, we should get back something like this:

```
1 [
2   { "id": 42, "author": "Douglas Adams", "title": "The Hitchhiker's Guide to the Galaxy" },
3   { "id": 3, "author": "J. R. R. Tolkien", "title": "The Lord of the Rings: The Fellowship of the Ring" }
4 ]
5
```

Nothing too scary. And even better, Backbone knows how to handle this out of the box. But what if we changed it a bit? Instead of `id` being the identifying field, it was `bookId`?

```
1 [
2   { "bookId": 42, "author": "Douglas Adams", "title": "The Hitchhiker's Guide to the Galaxy" },
3   { "bookId": 3, "author": "J. R. R. Tolkien", "title": "The Lord of the Rings: The Fellowship of the Ring" }
4 ]
5
```

Backbone gets that every API is a bit different, and it's okay with that. All you have to do is let it know the `idAttribute`, like so:

```
1 var BookModel = Backbone.Model.extend({
2   urlRoot: '/books'
3   , idAttribute: 'bookId'
4 });
5
```

You only have to add that information to the model, since the collection checks the model anyway. So just like that, Backbone understands your API! Even if I don't...

The downside of this is that you have to remember to use `bookId` in certain cases. For example, where we previously used `new BookModel({ id: 42 }).fetch()` to load the data about a single book, we would now have to use `new BookModel({ bookId: 42 }).fetch()`.

CSS rule to disable text selection highlighting?

```
1 -webkit-touch-callout: none;
2 -webkit-user-select: none;
3 -khtml-user-select: none;
4 -moz-user-select: none;
5 -ms-user-select: none;
6 user-select: none;
```

7

article vs section vs div

The difference between <article> and <section>

There's been a lot of confusion over the difference (or perceived lack of a difference) between the **<article>** and **<section>** elements in HTML5. The **<article>** element is a specialised kind of **<section>**; it has a more specific semantic meaning than **<section>** in that it is an independent, self-contained block of related content. We could use **<section>**, but using **<article>** gives more semantic meaning to the content.

By contrast **<section>** is only a block of related content, and **<div>** is only a block of content. Also as mentioned above the **pubdate** attribute doesn't apply to **<section>**. To decide which of these three elements is appropriate, choose the first suitable option:

- Would the content would make sense on its own in a feed reader? If so use **<article>**
- Is the content related? If so use **<section>**
- Finally if there's no semantic relationship use **<div>**

You can still use div

Sorry, can you say that again?, I hear you ask. Certainly: you can still use **<div>**! Despite HTML5 bringing us new elements like **<article>**, **<section>**, and **<aside>**, the **<div>** element still has its place. Let the HTML5 Doctor tell you why.

Status: Unchanged

In HTML 4, the

element was defined to be a generic element for structuring a page. Although you can allude to the nature of its content by assigning id and class attributes with meaningful names, a has almost no semantic meaning. The HTML5 definition is basically the same as in HTML 4:

- 1 The div element has no special meaning at all. It represents its children. It ca?
- 2 W3C Specification
- 3

typeof !== 'undefined' vs. !== null

typeof allows the identifier to never have been declared before. So it's safer in that regard:

- 1 **if(typeof neverDeclared == "undefined")**

?

```
2 //no errors
3
4 if(neverDeclared == null)
5 //throws ReferenceError: neverDeclared is not defined
6
```

What is the difference between a function expression vs declaration in JavaScript?

```
1 //Function declaration
2 function foo() { return 5; }
3
4 //Anonymous function expression
5 var foo = function() { return 5; }
6
7 //Named function expression
8 var foo = function foo() { return 5; }
9
```



They're actually really similar. How you call them is exactly the same, but the difference lies in how the browser loads them into the execution context.

function declarations loads before any code is executed.

While function expressions loads only when the interpreter reaches that line of code.

So if you try to call a function expression before it's loaded, you'll get an error

But if you call a function declaration, it'll always work. Because no code can be called until all declarations are loaded.

ex. Function Declaration

```
1 alert(foo()); // Alerts 5. Declarations are loaded before any code can run.
2 function foo() { return 5; }
3
```



ex. Anonymous Function Expression

```
1 alert(foo()); // ERROR! foo wasn't loaded yet
2 var foo = function() { return 5; }
3
```



ex. Named Function Expression

It can access the function name, usually used for recursion

Below example implements fibonacci series using named function expression and recursion

```
1 var fib = function fibonacci(n) {
2     if (n < 2) {
3         return 1;
4     } else {
5         return fibonacci(n - 2) + fibonacci(n - 1);
6     }
7 };
8
```



To print the first 10 fibonacci numbers

```
1  for (var i=1; i<10; i++) {  
2      fib(i);  
3  }  
4
```



JavaScript Closures?

Whenever you see the function keyword within another function, the inner function has access to variables in the outer function.

```
1  function foo(x) {  
2      var tmp = 3;  
3      function bar(y) {  
4          alert(x + y + (++tmp));  
5      }  
6      bar(10);  
7  }  
8  foo(2);  
9
```



This will always alert 16, because bar can access the x which was defined as an argument to foo, and it can also access tmp from foo.

That is a closure. A function doesn't have to return in order to be called a closure. Simply accessing variables outside of your immediate lexical scope creates a closure.

Common uses of closures

- The most common use is when someone wants to "delay" use of a variable that is increased upon each loop, but because the variable is scoped then each reference to the variable would be after the loop has ended, resulting in the end state of the variable

```
1  for (var i = 0; i < someVar.length; i++) {  
2      window.setTimeout(function () {  
3          alert("Value of i was "+i+" when this timer was set" )  
4      }, 10000);  
5  }  
6
```



This would result in every alert showing the same value of i, the value it was increased to when the loop ended. The solution is to create a new closure, a separate scope for the variable. This can be done using an instantly executed anonymous function, which receives the variable and stores its state as an argument:

```
1  for (var i = 0; i < someVar.length; i++)  
2      (function (i) {  
3          window.setTimeout(function () {  
4              alert("Value of i was "+i+" when this timer was set" )  
5          }, 10000);  
6      })(i);
```




```
6 | })(i);
7 |
```

- **Emulating private methods with closures:**

```
1 | a = (function () {
2 |     var privatefunction = function () {
3 |         alert('hello');
4 |     }
5 |
6 |     return {
7 |         publicfunction : function () {
8 |             privatefunction();
9 |         }
10 |     }
11 | })();
12 |
13 |
```

?

Performance considerations

It is unwise to unnecessarily create functions within other functions if closures are not needed for a particular task, as it will negatively affect script performance both in terms of processing speed and memory consumption.

For instance, when creating a new object/class, methods should normally be associated to the object's prototype rather than defined into the object constructor. The reason is that whenever the constructor is called, the methods would get reassigned (that is, for every object creation).

Consider the following impractical but demonstrative case:

```
1 | function MyObject(name, message) {
2 |     this.name = name.toString();
3 |     this.message = message.toString();
4 |     this.getName = function() {
5 |         return this.name;
6 |     };
7 |
8 |     this.getMessage = function() {
9 |         return this.message;
10 |     };
11 | }
12 |
```

?

The previous code does not take advantage of the benefits of closures and thus could instead be formulated:

```
1 | function MyObject(name, message) {
2 |     this.name = name.toString();
3 |     this.message = message.toString();
4 | }
5 | MyObject.prototype = {
6 |     getName: function() {
7 |         return this.name;
8 |     },
9 |     getMessage: function() {
10 |         return this.message;
11 |     }
12 | };
13 |
```

?

However, redefining the prototype is not recommended, so the following example is even better because it appends to the existing prototype:

```
1 function MyObject(name, message) {  
2   this.name = name.toString();  
3   this.message = message.toString();  
4 }  
5 MyObject.prototype.getName = function() {  
6   return this.name;  
7 };  
8 MyObject.prototype.getMessage = function() {  
9   return this.message;  
10 };  
11
```

What are the differences between Deferred, Promise and Future in Javascript?

Rather than directly passing callbacks to functions, something which can lead to tightly coupled interfaces, using promises allows one to separate concerns for code that is synchronous or asynchronous.

Personally, I've found deferred especially useful when dealing with e.g. templates that're populated by asynchronous requests, loading scripts that have networks of dependencies, and providing user feedback to form data in a non-blocking manner.

Indeed, compare the pure callback form of doing something after loading CodeMirror in JS mode asynchronously (apologies, I've not used jQuery in a while):

```
1 /* assume getScript has signature like: function (path, callback, context) and ?  
2 $(function () {  
3   getScript('path/to/CodeMirror', getJSMODE);  
4  
5   // onreadystatechange is not reliable for callback args.  
6   function getJSMODE() {  
7     getScript('path/to/CodeMirror/mode/javascript/javascript.js',  
8       ourAwesomeScript);  
9   };  
10  
11   function ourAwesomeScript() {  
12     console.log("CodeMirror is awesome, but I'm too impatient.");  
13   };  
14 });  
15
```

To the promises formulated version (again, apologies, I'm not up to date on jQuery):

```
1 /* Assume getScript returns a promise object */  
2 $(function () {  
3   $.when(  
4     getScript('path/to/CodeMirror'),  
5     getScript('path/to/CodeMirror/mode/javascript/javascript.js')  
6   ).then(function () {  
7     console.log("CodeMirror is awesome, but I'm too impatient.");  
8   });  
9 });  
10
```

The point of promises is to give us back functional composition and error bubbling in the async world.

In other word, promises are a way that lets us write asynchronous code that is almost as easy to write as if it was synchronous.

Consider this example, with promises:

```

1  getTweetsFor("domenic") // promise-returning async function
2    .then(function (tweets) {
3      var shortUrls = parseTweetsForUrls(tweets);
4      var mostRecentShortUrl = shortUrls[0];
5      return expandUrlUsingTwitterApi(mostRecentShortUrl); // promise-returning
6    })
7    .then(doHttpRequest) // promise-returning async function
8    .then(
9      function (responseBody) {
10       console.log("Most recent link text:", responseBody);
11     },
12     function (error) {
13       console.error("Error with the twitterverse:", error);
14     }
15   );
16
```

It works as if you were writing this synchronous code:

```

1  try {
2    var tweets = getTweetsFor("domenic"); // blocking
3    var shortUrls = parseTweetsForUrls(tweets);
4    var mostRecentShortUrl = shortUrls[0];
5    var responseBody = doHttpRequest(expandUrlUsingTwitterApi(mostRecentShortUrl));
6    console.log("Most recent link text:", responseBody);
7  } catch (error) {
8    console.error("Error with the twitterverse: ", error);
9  }
10
```

Regarding Deferred, it's a way to .resolve() or .reject() promises. In the Promises/B spec, it is called .defer(). In jQuery, it's \$.Deferred().

local storage vs cookie?

To delete a localStorage item

```

1  localStorage.removeItem(key);
2
```

What is a cookie? Quite simply, a cookie is a small text file that is stored by a browser on the user's machine. Cookies are plain text; they contain no code. A web page or server instructs a browser to store this information and then send it back with each subsequent request based on a set of rules. Web servers can then use this information to identify individual users. Most sites requiring a login will typically set a cookie once your credentials have been verified, and you are then free to navigate to all parts of the site so

long as that cookie is present and validated. Once again, the cookie just contains data and isn't harmful in and of itself.

Cookies and local storage really serve different purposes. Cookies are primarily for reading server-side, local storage can only be read client-side. So the question is, in your app, who needs this data – the client or the server?

If it's your client (your JavaScript), then by all means switch. You're wasting bandwidth by sending all the data in each HTTP header.

If it's your server, local storage isn't so useful because you'd have to forward the data along somehow (with Ajax or hidden form fields or something). This might be okay if the server only needs a small subset of the total data for each request.

You'll want to leave your session cookie as a cookie either way though.

Apart from being an old way of saving data, **Cookies give you a limit of 4096 bytes** (4095, actually) - its per cookie. **Local Storage is as big as 5MB per domain**

localStorage is implementation of Storage Interface - stores data with no expiration date, and gets cleared only through JavaScript, or clearing the Browser Cache / Locally Stored Data - unlike cookie expiry

Publish/Subscribe pattern (in JS/jQuery)?

It's all about loose coupling and single responsibility, which goes hand to hand with MV* (MVC/MVP/MVVM) patterns in JavaScript which are very modern in the last few years.

So that you don't have to hardcode method / function calls, you just publish the event without caring who listens. This makes the publisher independent from subscriber, reducing dependency (or coupling, whatever term you prefer) between 2 different parts of the application.

Here are some disadvantages of coupling as mentioned by wikipedia

Tightly coupled systems tend to exhibit the following developmental characteristics, which are often seen as disadvantages:

- A change in one module usually forces a ripple effect of changes in other modules.
- Assembly of modules might require more effort and/or time due to the increased inter-module dependency.
- A particular module might be harder to reuse and/or test because dependent modules must be included.

NOTE: Loose coupling is an *Object-oriented principle* in which each component of the system knows its responsibility and doesn't care about the other components (or at least tries to not care about them as much as possible). Loosening coupling is a good thing because you can easily reuse the different modules. You're not coupled with the interfaces of other modules. Using publish/subscribe you're only coupled with the publish/subscribe interface which is not a big deal – just two methods. So if you decide to reuse a module in different project you can just copy and paste it and it'll probably work or at least you won't need much effort to make it work.

method vs function?

A function is a piece of code that is called by name. It can be passed data to operate on (ie. the parameters) and can optionally return data (the return value).

All data that is passed to a function is explicitly passed.

A method is a piece of code that is called by name that is associated with an object. In most respects it is identical to a function except for two key differences.

- It is implicitly passed the object for which it was called
- It is able to operate on data that is contained within the class (remembering that an object is an instance of a class - the class is the definition, the object is an instance of that data)

What is namespacing?

JavaScript has a big design flaw, where it is very easy to create global variables that have the potential to interact in negative ways. The practice of namespacing is usually to create an object literal encapsulating your own functions and variables, so as not to collide with those created by other libraries:

```
1  var MyApp = {
2      isMobile: true,
3      foo: function() {
4      },
5      bar: function() {
6      }
7  };
8
9  // Then instead of calling foo() globally, it would always be called as:
10 MyApp.foo();
11
12 // Likewise, var1 always accessed as:
13 MyApp.isMobile
14
```

Dis advantages of Spa?

The biggest disadvantage is that the client must have JavaScript enabled and be powerful enough to run a fair amount of it. It's also harder to satisfy accessibility concerns or anything else that relies on parsing static HTML (though something knowing your specific API can probably do better than HTML scraping). Finally, it's easier to have significant memory leaks.

As far as duplicating code or putting business logic on the client - I'm not sure how much of that you have to do. If the model on the client is a View-Model (a model that matches up with the world as the UI sees it, not a business model) then the logic that matches the ViewModel up to the business model can reside on the client, the server, or a bit of both. It depends on how you feel about having your API contain a client-specific facade vs. having the client translate UI inputs into API calls.

OOP In JavaScript?

The two important principles with OOP in JavaScript are Object Creation patterns (Encapsulation) and Code Reuse patterns (Inheritance). When building applications, you create many objects, and there are many ways of creating these objects: you can use the very basic object literal pattern, for example:

```
1 | var myObj = {name: "Richard", profession: "Developer"};
2 |
```



Websockets and their uses?

WebSockets enables instant data exchange and is supported by all modern browsers. Put it to use immediately in your Web apps

Determining whether to use WebSockets for the job at hand is simple:

- Does your app involve multiple users communicating with each other?
- Is your app a window into server-side data that's constantly changing?

If you answered yes to either of these questions, consider using WebSockets. If you're still unsure and want some inspiration, here are a few killer use cases.

1. Social feeds

One of the benefits of social apps is knowing what all your friends are doing when they do it. Sure, it's a little creepy, but we all love it. You don't want to wait minutes to find out a family member won a pie-baking contest or a friend has become engaged. You're online, so your feed should update in real time.

2. Multiplayer games

The Web is quickly coming into its own as a gaming platform. Without having to rely on plug-ins (I'm looking at you, Flash) Web developers are now able to implement and experiment with high-performance gaming in the browser. Whether you're dealing with DOM elements, CSS animations, HTML5 canvas or you're experimenting with WebGL, efficient interaction between players is crucial. I don't want to discover my opponent has moved after I pull the trigger.

3. Collaborative editing/coding

We live in the age of distributed dev teams. Working on a copy of a document used to suffice, but then you had to figure out a way to merge all the edited copies together. Version control systems like Git can help with certain files, but you'll still have to track people down when Git finds a conflict it can't deal with. With a collaborative solution like WebSockets, we can work on the same document and skip all the merges. It's easy to see who is editing what and if you're working on the same portion of a document as someone else.

4. Clickstream data

Being able to analyze how users interact with your website is critical to improving it. The cost of HTTP has forced us to prioritize and collect only the most important data. Then, six months down the line, we realize we should've been collecting a different metric -- one that looked unimportant but would now shed light on a critical decision. With the overhead of HTTP requests out of the way, you can be less restrictive about the kind of data you're sending from the client. Want to track mouse movement in addition to page loads? Just send the data through a WebSocket connection to the back end and persist it in your favorite NoSQL store. (MongoDB is good for logging events like this.) Now you can play back customer interactions to see what was really going on.

5. Financial tickers

The finance world moves fast -- microsecond fast. Our human brains can't keep up with processing data at that speed, so we write algorithms to do it for us. Even if you're not dealing with high-frequency trading, however, stale information can only hurt. When you have a dashboard tracking companies you're interested in, you want to know what they're worth now, not 10 seconds ago. Use WebSockets to stream that data and no one needs to wait.

6. Sports updates

While we're on the topic of silly things people are passionate about, let's talk sports. I'm not a sports guy, but I know what sports guys want. When the Patriots are playing, my brother-in-law surrenders himself to the pace of the game. It's a trancelike state prone to violent, visceral reactions. I don't understand it, but I respect the intensity of the connection, so the last thing I want to do is add latency to his experience. If you're including sports information in your Web app, WebSockets can keep your users up to speed.

7. Multimedia chat

While there's no substitute for holding a meeting in person, videoconferences are about as good as it gets when we can't get everybody in the same room. The videoconference route is plug-in heavy, though, and full of proprietary "goodness." I'm a fan of the open Web, so using WebSockets with getUserMedia API's and the HTML5 audio and video elements is an obvious win. WebRTC, the logical predecessor to the combination I just outlined, looks promising, but the lack of current browser support eliminates it as a candidate.

8. Location-based apps

More and more developers are leveraging the GPS capabilities of mobile devices to make their Web apps location-aware. If you're tracking user locations over time (such as running an app that tracks your progress along a route), you'll be collecting fine-grained data. If you want to update a Web dashboard in real time (say, a track coach monitoring the progress of runners) HTTP is going to be unnecessarily bulky. Leverage the TCP connection a WebSocket uses and let that data fly.

9. Online education

School keeps getting more expensive, while the Internet keeps getting faster and cheaper. Online education can be a great way to learn, especially if you can interact with teachers and other students. WebSockets is the natural choice, allowing for multimedia chat, text chat, and other perks like collaborative drawing on a digital communal chalkboard.

Why Spa?

Single page applications are more capable of decreasing load time of pages by storing the functionality once it is loaded the first time, allowing easier data transfer between pages and a more complex user interface instead of trying to control so much from the server. This allows for more interference from an end user, but proper checks on the server eliminate pretty much all of that risk. Most end users won't attempt this, so that does not warrant much if any worry. Also, the processing of data can be moved into generic service handlers which could result in an architecture utilizing the following layers:

- Database
- BL
 - Transport Service
 - Processing Service

- User Interface The ajax functionality used allows for a smoother and more controlled user experience (my favorite perk)

And as for handling large components, the benefit is that you have the option to defer the load time to an ajax process and do so whenever you like, making the initial load faster.

The process for coding it is a bit different in that using services is pretty much a requirement, but it is very similar to using a master page and controlling what content is loaded from the client.

As for best practices, there are quite a few optimizations that could and should be made to a design implementation, such as storing information as it comes, loading in script, html and js files through ajax only when needed, and using data loaded on one page in another if it can be instead of reloading for each new "page".

There is all sorts of information on this, and all sorts of libraries to use, but I would suggest using your own code as much as possible for the basic functionality(including the libraries you would typically use) and enhance from there.

As a side note, I would suggest looking up concepts for designing for one web, such as media queries for adjusting content..

I believe that this is the direction most websites should be moving in considering the number of devices that users utilize today, and the abilities and limitations of each.

Why REST?

REST is using HTTP verbs GET, POST, PUT, DELETE to respectively get, insert, update, delete resources on a server.

E.g. GET with url: controller/customer/{id} The alternative is adding methods to a controller such as GetCustomerById(id), InsertCustomer(customer), UpdateCustomer(customer), DeleteCustomer(customer). This is what's known as a RPC approach. E.g. GET with url: controller/GetCustomerById?id={id} One of the differences is, is that a REST api is kind of predictable. If you know REST and you know the name of the resource you want (e.g. Customer), then you can immediately jump in and use the REST controller with the standard verbs.

- **With an RPC approach, you need to know what methods are on your controller**, what kind of arguments they take, etc. These signatures can vary from controller to controller, from app to app.
- **AJAX on the other hand is just an asynchronous way of placing the aforementioned requests, whether as a REST call or a RPC call.**

REST-JSON has several advantages over SOAP-XML

- **Size** REST-JSON is a lot smaller and less bloated than SOAP-XML therefore you will be passing much less data over the network, which is particularly important for mobile devices.
- **Efficiency** REST-JSON is also easier to parse the data so therefore easier to extract and convert the data, so therefore requires much less from the CPU from the client.
- **Caching** REST-JSON provides improved response times and server loading due to support from caching
- **Implementation** REST-JSON interfaces are much easier to design and implement.

For these reasons REST-JSON is generally preferred to SOAP-XML for mobile application which require a web service to retrieve data from a web service, where there is no need to the heavy-weight XML structure.

SOAP-XML is generally preferred when:

passing around a lot of text since XML excels and wrapping and marking up text. you require secure, transactional services such as banking services. Due to the strict nature of SOAP-XML any changes in the server code needs to be implemented on any and all clients.

Custom Events

Without custom events, the code will be more meaningful

Publish / Subscribe demonstrating custom events With jquery

```
1 $(document).on('testEvent', function(e, eventInfo) {  
2     subscribers = $('.subscribers-testEvent');  
3     subscribers.trigger('testEventHandler', [eventInfo]);  
4 });  
5  
6 $('#myButton').on('click', function() {  
7     $(document).trigger('testEvent', [1011]);  
8 });  
9  
10 $('#notifier1').on('testEventHandler', function(e, eventInfo) {  
11     alert('(notifier1)The value of eventInfo is: ' + eventInfo);  
12 });  
13  
14 $('#notifier2').on('testEventHandler', function(e, eventInfo) {  
15     alert('(notifier2)The value of eventInfo is: ' + eventInfo);  
16 });  
17
```

jQuery Miscellaneous?

• \$(document).ready equivalent without jQuery?

We have three options:

- If script is the last tag of the body, DOM would be ready before script tag executes
- When DOM is ready, "readyState" will change to "complete"
- Put everything under 'DOMContentLoaded' event listener

onreadystatechange

```
1 document.onreadystatechange = function () {  
2     if (document.readyState == "complete") {  
3         //document is ready. Do your stuff here  
4     }
```

```
5 | }
6 |
```

DOMContentLoaded

```
1 | document.addEventListener('DOMContentLoaded', function(){
2 |     console.log('document is ready. I can sleep now');
3 | });
4 |
```

• Preloading images with jQuery?

```
1 | $.fn.preload = function() {
2 |     this.each(function(){
3 |         $('<img>')[0].src = this;
4 |     });
5 | }
6 | // Usage:
7 | $(['img1.jpg', 'img2.jpg', 'img3.jpg']).preload();
8 |
```

• How to change the href for a hyperlink using jQuery?

```
1 | $("a").attr("href", "http://www.google.com/")
2 |
```

• Scroll to bottom of div?

```
1 | $("#mydiv").scrollTop($("#mydiv")[0].scrollHeight);
2 |
```

• Using jQuery to center a DIV on the screen

```
1 | jQuery.fn.center = function () {
2 |     this.css("position", "absolute");
3 |     this.css("top", Math.max(0, (($ (window).height() - $ (this).outerHeight()
4 |                                     $ (window).scrollTop()) + "px");
5 |     this.css("left", Math.max(0, (($ (window).width() - $ (this).outerWidth()
6 |                                     $ (window).scrollLeft()) + "px");
7 |     return this;
8 | }
9 | //Now call
10 | $(element).center();
11 |
12 | //With CSS only
13 | .center {
14 |     position: absolute;
15 |     left: 50%;
16 |     top: 50%;
17 |     transform: translate(-50%, -50%); /* Yep! */
18 |     width: 48%;
19 |     height: 59%;
20 | }
21 |
```

• jQuery scrollTop (Smoothly scroll to an element with jquery)

Get the current vertical position of the scroll bar for the first element in the set of matched elements or set the vertical position of the scroll bar for every matched element.

```
1 | var container = $('div'),
2 |     scrollTo = $('#row_8');
3 |
4 | container.scrollTop(
```

```

5 |         scrollTo.offset().top - container.offset().top + container.scrollTop()
6 |     );
7 |
8 |     // Or you can animate the scrolling:
9 |     container.animate({
10 |         scrollTop: scrollTo.offset().top - container.offset().top + container.sc
11 |     });
12 |

```

- **How to get the current URL in javascript?**

```

1 | window.location.pathname;
2 |

```

- **Get selected text from drop down list (select box) using jQuery**

```

1 | // Using plain javascript
2 | dropdownid.options[dropdownid.selectedIndex].innerHTML
3 |
4 | $("#dropdownid option:selected").text();
5 | // the fastest using jquery is
6 | $("#dropdownid").children("option").filter(":selected").text()
7 |

```

- **jQuery multiple class selector?**

To select a <div class="a b"></div>

```

1 | // Use
2 | $('a.b')
3 |

```

and we don't need that ".length > 0" part.

- **check the existence of an element in jQuery?**

In JavaScript, everything is truthy or falsy and for numbers, 0 means false, everything else true. So you could write:

```

1 | if ($(selector).length)
2 |

```

and we don't need that ".length > 0" part.

- **which radio is selected via jQuery?**

```

1 | $("#myform input[type='radio']:checked").val();
2 | ( or )
3 | $('input[name=radioName]:checked', '#myForm').val()
4 |

```

- **event.preventDefault() vs. return false**

return false from within a jQuery event handler is effectively the same as calling both *e.preventDefault* and *e.stopPropagation* on the passed jQuery.Event object.

e.preventDefault() will prevent the default event from occurring, *e.stopPropagation()* will prevent the event from bubbling up and return false will do both. Note that this behaviour differs from normal (non-jQuery) event handlers, in which, notably, return false does not stop the event from bubbling up.

- **How do I check a checkbox with jQuery?**

In jQuery 1.6+, Use the new .prop() function:

```

1 | $('.myCheckbox').prop('checked', true);

```

```
2 | $('myCheckbox').prop('checked', false);
3 |
```

in jQuery 1.5 and below, the .prop() function is not available, so you need to use .attr().

To check the checkbox (by setting the value of the checked attribute) do

```
1 | $('myCheckbox').attr('checked', 'checked');
```

and for un-checking (by removing the attribute entirely) do

```
1 | $('myCheckbox').removeAttr('checked');
2 |
```

If you want to check if a checkbox is checked or not:

```
1 | $('myCheckbox').is(':checked');
2 |
```

NOTE: Using only javascript

```
1 | DOMElement.checked = true; // to set in javascript
2 |
```

jQuery Utilities?

Type testing functions -

These are very helpful in optional parameters and also when validating the parameters for functions.

- **isArray(array) -**

\$.isArray() returns a Boolean indicating whether the object is a JavaScript array (not an array-like object, such as a jQuery object).

Array.isArray works fast, but it isn't supported by all versions of browsers. So you could make an exception for others and use universal method:

```
1 | var a = ["A", "AA", "AAA"];
2 |
3 | // javascript version
4 | if(Array.isArray){
5 |     Array.isArray(a) ? alert("a is an array") : alert("a is not an array");
6 | } else {
7 |     (Object.prototype.toString.call(a) === '[object Array]')
8 |     ? alert("a is an array") : alert("a is not an array");
9 | }
10 |
11 | // jquery version
12 | if($.isArray(a)) {
13 |     alert("a is an array!");
14 | } else {
15 |     alert("a is not an array!");
16 | }
17 |
```

- **isFunction(fun) -**

- Determine if the argument passed is a Javascript function object.

- **isEmptyObject(obj)** -

Checks to see if an object is empty (contains no enumerable properties).

- **jQuery.isPlainObject(obj)** -

Note: Host objects (or objects used by browser host environments to complete the execution environment of ECMAScript) have a number of inconsistencies which are difficult to robustly feature detect cross-platform. As a result of this, \$.isPlainObject() may evaluate inconsistently across browsers in certain instances.

An example of this is a test against document.location using \$.isPlainObject() as follows:

```
1 console.log( $.isPlainObject( document.location ) );  
2
```

which throws an invalid pointer exception in IE8. With this in mind, it's important to be aware of any of the gotchas involved in using \$.isPlainObject() against older browsers. A couple basic examples that do function correctly cross-browser can be found below.

Check to see if an object is a plain object (created using "{}" or "new Object").

```
1 jQuery.isPlainObject({}) // true  
2 jQuery.isPlainObject( "test" ) // false  
3
```

- **isXmlDoc(doc)** -

- **isNumeric(num)** -

available in jQuery 1.7+

The \$.isNumeric() method checks whether its argument represents a numeric value. If so, it returns true. Otherwise it returns false. The argument can be of any type.

- **isWindow(obj)** -

Determine whether the argument is a window.

Collection functions -

These are very helpful in optional parameters and also when validating the parameters

- **makeArray** -

Convert an array-like object into a true JavaScript array.

```
1 var obj = $( "li" );  
2 var arr = $.makeArray( obj );  
3
```

- **inArray** -

- returns index of the element (Search for a specified value within an array and return its index (or -1 if not found))

The \$.inArray() method is similar to JavaScript's native .indexOf() method in that it returns -1 when it doesn't find a match. If the first element within the array matches value, \$.inArray() returns 0.

- **unique** -

filters duplicates

- **merge** -

merge two arrays

- **map -**

Translate all items in an array or object to new array of items.

`$(selector).map`

`$.map`

Example: To simply flatten the array [1, 2, [3, 4], [5, 6], 7] to [1, 2, 3, 4, 5, 6, 7]

```
1 | $.map( [1, 2, [3, 4], [5, 6], 7], function(n){
2 |     return n;
3 | });
4 | // returns [1, 2, 3, 4, 5, 6, 7]
5 |
```

?

- **grep -**

Finds the elements of an array which satisfy a filter function. The original array is not affected.

only receives the dom object invert parameter - which performs inverse

```
1 | var arr = [ 1, 9, 3, 8, 6, 1, 5, 9, 4, 7, 3, 8, 6, 9, 1 ];
2 | arr = jQuery.grep(arr, function( item, index ) {
3 |     return ( item !== 5 && index > 4 );
4 | });
5 | // 1, 9, 4, 7, 3, 8, 6, 9, 1
6 |
```

?

- **extend Function**

copy members from source object to target object - Conflicts will cause members to be overwritten

```
1 |
2 |
```

?

- **parseJSON Function**

Turns a JSON String into javascript objects, no need to do as jquery ajax functions will convert the result into JSON automatically, but in special cases this is needed.

```
1 | var json = '{"fname": "john", "lname": "smith", "age":24}';
2 | var jsonObj = $.parseJSON(json);
3 | console.log( jsonObj.fname, jsonObj.lname );
4 | // output
5 | // john smith
6 |
```

?

- **each Function (\$.each() and \$(selector).each())**

Iterate over a jQuery object, executing a function for each matched.

Uses:

- Can be used to iterate over json object.
- Can be used to iterate over array.

The \$.each() function is not the same as \$(selector).each(), which is used to iterate, exclusively, over a jQuery object.

```
1 | $.each(".indent", function(index){
```

?

doesn't iterate over the elements of `$('.indent')` but over the `".indent"` string whose length is 7 chars.

See reference, A more detailed explanation based on jQuery source code : jQuery first checks if the first parameter, obj (here your string), has a length :

```
1  var ...
2      length = obj.length,
3      isObj = length === undefined || jQuery.isFunction( obj );
4  Your string having a length (and not being a function), isObj is false.
5
```

In this case, the following code is executed :

```
1  for ( ; i < length; ) {
2      if ( callback.call( obj[ i ], i, obj[ i++ ] ) === false ) {
3          break;
4      }
5  }
6
```

So, given the function f, the following code

```
1  $.each(".indent", f);
2
```

is equivalent to

```
1  for (var i=0; i<".indent".length; i++) {
2      var letter = ".indent"[i];
3      f.call(letter, i, letter);
4  }
5
```

(you can log the letters using var f = function(i,v){console.log(v)}; or be reminded one of the subtleties of call using var f = function(){console.log(this)};)

- **pushStack**

Adds new array of DOM elements onto the jQuery stack. (or) pushStack() creates a new jQuery object that inherits state from a previous jQuery object.

This inherited state allows methods like .end() and .self() to work properly.

When working with jQuery objects, it's important to know that most operations that change what DOM objects are in a jQuery object return a new jQuery object with the change in it rather than modifying the existing jQuery object. It is this design characteristic that allows them to maintain this stack of previously modified jQuery objects.

The main use of pushStack is, in jQuery methods that return a new jQuery object. In that case, you create the new list of elements you want in the new jQuery object and then rather than just turning that into a regular jQuery object and returning it, you call **return this.pushStack(elems)**. This ends up creating a new jQuery object and returning it, but it also links that new object to the previous object so that special commands like .end() can work when used in chaining.

The jQuery .add() method is a classic example. In pseudo-code, what it does is this:

```
1  add: function(selector, context) {
2      // get the DOM elements that correspond to the new selector (the ones to |
3      // get the DOM elements from the current jQuery object with this.get()
4      // merge the two arrays of elements together into one array
5      return this.pushStack(combinedElems);
6  }
7
```

- **holdReady**

holds the ready function until we tell it can fire it (Holds or releases the execution of jQuery's ready event.)

Uses: waiting for a specific asset to load

The `$.holdReady()` method allows the caller to delay jQuery's ready event. This advanced feature would typically be used by dynamic script loaders that want to load additional JavaScript such as jQuery plugins before allowing the ready event to occur, even though the DOM may be ready. This method must be called early in the document, such as in the immediately after the jQuery script tag. Calling this method after the ready event has already fired will have no effect.

To delay the ready event, first call `$.holdReady(true)`. When the ready event should be released to execute, call `$.holdReady(false)`. Note that multiple holds can be put on the ready event, one for each `$.holdReady(true)` call. The ready event will not actually fire until all holds have been released with a corresponding number of `$.holdReady(false)` calls and the normal document ready conditions are met. (See ready for more information.)

Example: Delay the ready event until a custom plugin has loaded.

```
1  $.holdReady( true );
2  $.getScript( "myplugin.js", function() {
3      $.holdReady( false );
4  });
5
```

- **getScript**

wrapper for `$.ajax` so non blocking (Load a JavaScript file from the server using a GET HTTP request, then execute it.)

Executes retrieved javascript immediately

```
1  $.getScript("script.js", function(data, textStatus) {
2      console.log(textStatus); // success
3  })
4      ;
```

- **jQuery.noConflict**

Releases (Relinquish) jQuery's control of the `$` variable.

Many JavaScript libraries use `$` as a function or variable name, just as jQuery does. In jQuery's case, `$` is just an alias for jQuery, so all functionality is available without using `$`. If you need to use another JavaScript library alongside jQuery, return control of `$` back to the other library with a call to `$.noConflict()`. Old references of `$` are saved during jQuery initialization; `noConflict()` simply restores them.

If for some reason two versions of jQuery are loaded (which is not recommended), calling `$.noConflict(true)` from the second version will return the globally scoped jQuery variables to those of the first version.

Example: Load two versions of jQuery (not recommended). Then, restore jQuery's globally scoped variables to the first loaded jQuery.

```
1  <!doctype html>
2  <html lang="en">
3  <head>
4      <meta charset="utf-8">
5      <title>jQuery.noConflict demo</title >
6      <script src="//code.jquery.com/jquery-1.10.2.js"> </script>
```



```

7
8 <body>
9
10 <div id="log">
11   <h3>Before $.noConflict(true)</h3>
12 </div>
13 <script src="//code.jquery.com/jquery-1.6.2.js"> </script>
14
15 <script>
16 var $log = $( "#log" );
17
18 $log.append( "2nd loaded jQuery version ($): " + $.fn.jquery + "<br>" );
19
20 // Restore globally scoped jQuery variables to the first version loaded
21 // (the newer version)
22
23 jq162 = jQuery.noConflict( true );
24
25 $log.append( "<h3>After $.noConflict(true)</h3>" );
26 $log.append( "1st loaded jQuery version ($): " + $.fn.jquery + "<br>" );
27 $log.append( "2nd loaded jQuery version (jq162): " + jq162.fn.jquery + "<br>" );
28 </script>
29 </body>
30 </html>
31

```

Why Backbone.Marionette JS?

Marionette has a few very distinct goals:

- Composite application architecture
- Enterprise messaging pattern influence
- Modularization options
- Incremental use (no all-or-nothing requirement)
- No server dependency
- Make it easy to change those defaults
- Code as configuration / over configuration

Composite Application Architecture

The basic idea is to **"compose"** your application's runtime experience and process out of many smaller, individual pieces that don't necessarily know about each other. They register themselves with the overall composite application system and then they communicate through various means of decoupled messages and calls.

Message Queues / Patterns

The same large scale, distributed systems also took advantage of message queuing, enterprise integration patterns (messaging patterns), and service buses to handle the messages. This, more than anything else, had a tremendous influence on my approach to decoupled software development. I began to see single-process, in-memory WinForms applications from this perspective, and soon my server side and web application development took influence from this.

This has directly translated itself in to how I look at Backbone application design. I provide an event aggregator in Marionette, for both the high level Application object, and for each module that you create within the application.

I think about messages that I can send between my modules: command messages, event messages, and more. I also think about the server side communication as messages with these same patterns. Some of the patterns have made their way in to Marionette already, but some haven't yet.

Modularization

Modularization of code is tremendously important. Creating small, well encapsulated packages that have a singular focus with well defined entry and exit points is a must for any system of any significant size and complexity. Marionette provides modularization directly through it's 'module' definitions. But I also recognize that some people like RequireJS and want to use that. So I provide both a standard build and a RequireJS compatible build.

```
1 MyApp = new Backbone.Marionette.Application();
2 MyApp.module("MyModule", function(MyModule, MyApp, Backbone, Marionette, $, _){
3   // your module code goes here
4 });
5
```

Incremental Use

This is one of the core philosophies that I bake in to every part of Marionette that I can: no "all-or-nothing" requirement for use of Marionette. Backbone itself takes a very incremental and modular approach with all of it's building block objects. You are free to choose which ones you want to use, when. I strongly believe in this principle and strive to make sure Marionette works the same way. To that end, the majority of the pieces that I have built in to Marionette are built to stand alone, to work with the core pieces of Backbone, and to work together even better.

For example, nearly every Backbone application needs to dynamically show a Backbone view in a particular place on the screen. The apps also need to handle closing old views and cleaning up memory when a new one is put in place. This is where Marionette's 'Region' comes in to play. A region handles the boilerplate code of taking a view, calling render on it, and stuffing the result in to the DOM for you. Then will close that view and clean it up for you, provided your view has a "close" method on it.

```
1 MyApp.addRegions({
2   someRegion: "#some-div"
3 });
4 MyApp.someRegion.show(new MyView());
5
```

But you're not required to use Marionette's views in order to use a region. The only requirement is that you are extending from Backbone.View at some point in the object's prototype chain. If you choose to provide a 'close' method, a 'onShow' method, or others, Marionette's Region will call it for you at the right time.

No Server dependency

Backbone / Marionette apps can be build on top of a wide variety of server technologies, for example:

- ASP.NET MVC
- Ruby on Rails
- Ruby / Sinatra
- NodeJS / ExpressJS
- PHP / Slim
- Java

JavaScript is JavaScript, when it comes to running in a browser. Server side JavaScript is awesome, too, but it has zero affect or influence on how I write my browser based JavaScript.

Because of the diversity in projects that I built and back-end technologies that my clients use, I cannot and will not lock Marionette in to a single server side technology stack for any reason. I won't provide a boilerplate project. I won't provide a ruby gem or an npm package. I want people to understand that Marionette doesn't require a specific back-end server. It's browser based JavaScript, and the back-end doesn't matter.

Of course, I fully support other people providing packages for their language and framework. I list those packages in the Wiki and hope that people continue to build more packages as they see a need. But that is community support, not direct support from Marionette.

Easily Change The Defaults

In my effort to reduce boilerplate code and provide sensible defaults (which is an idea that I directly â€œborrowedâ€ from Tim Branyen's LayoutManager), I recognize the need for other developers to use slightly different implementations than I do.

I provide rendering based on inline `<script>` tags for templates, using Underscore.js templating by default. But you can replace this by changing the **'Renderer'** and/or **'TemplateCache'** objects in Marionette. These two objects provide the core of the rendering capabilities, and there are wiki pages that show how to change this out for specific templating engines and different ways of loading templates.

With v0.9 of Marionette, it gets even easier. For example, if you want to replace the use of in-line template script blocks with pre-compiled templates, you only have to replace one method on the Renderer:

```
1 // use pre-compiled template functions
2 Backbone.Marionette.Renderer.render = function(template, data){
3   return template(data);
4 };
5
```

and now the entire application will use pre-compiled templates that you attach to your view's 'template' attribute. There is a Marionette.Async add-on with v0.9 that allows you to support asynchronously rendering views. I continuously strive to make it as easy as possible to replace the default behaviours in Marionette.

;;;

Differences Between jQuery .bind() vs .live() vs .delegate() vs .on()??

Introduction

I've seen quite a bit of confusion from developers about what the real differences are between the jQuery .bind(), .live(), .delegate(), and .on() methods and when they should be used.

If you want, you can jump to the [TL;DR](#) section and get a high-level overview what this article is about.

Before we dive into the ins and outs of these methods, let's start with some common HTML markup that

we'll be using as we write sample jQuery code.

```

1  <ul id="members" data-role="listview" data-filter="true">
2      <!-- ... more list items ... -->
3      <li>
4          <a href="detail.html?id=10">
5              <h3>John Resig</h3>
6              <p><strong>jQuery Core Lead</strong></p>
7              <p>Boston, United States</p>
8          </a>
9      </li>
10     <!-- ... more list items ... -->
11 </ul>

```

[_snippet.html](#) hosted with ❤ by GitHub

[view raw](#)

```

1 2 3 4 5 6 7 8 9 10 11
    <ul id="members" data-role="listview" data-filter="true">
        <!-- ... more list items ... -->
        <li>
            <a href="detail.html?id=10">
                <h3>John Resig</h3>
                <p><strong>jQuery Core Lead</strong></p>
                <p>Boston, United States</p>
            </a>
        </li>
        <!-- ... more list items ... -->
    </ul>

```

[_snippet.html](#) hosted with ❤ by GitHub

[view raw](#)

Using the Bind Method

The `.bind()` method registers the type of event and an event handler directly to the DOM element in question. This method has been around the longest and in its day it was a nice abstraction around the various cross-browser issues that existed. This method is still very handy when wiring-up event handlers, but there are various performance concerns as are listed below.

```

1  /* The .bind() method attaches the event handler directly to the DOM
2     element in question ( "#members li a" ). The .click() method is
3     just a shorthand way to write the .bind() method. */
4
5  $( "#members li a" ).bind( "click", function( e ) {} );
6  $( "#members li a" ).click( function( e ) {} );

```

[bind.js](#) hosted with ❤ by GitHub


[view raw](#)

```

1 2 3 4 5 6
    /* The .bind() method attaches the event handler directly to the DOM
        element in question ( "#members li a" ). The .click() method is
        just a shorthand way to write the .bind() method. */

    $( "#members li a" ).bind( "click", function( e ) {} );
    $( "#members li a" ).click( function( e ) {} );

```

bind.js hosted with  by **GitHub**[view raw](#)

The `.bind()` method will attach the event handler to all of the anchors that are matched! That is not good. Not only is that expensive to implicitly iterate over all of those items to attach an event handler, but it is also wasteful since it is the same event handler over and over again.

Pros

- This methods works across various browser implementations.
- It is pretty easy and quick to wire-up event handlers.
- The shorthand methods (`.click()`, `.hover()`, etc...) make it even easier to wire-up event handlers.
- For a simple ID selector, using `.bind()` not only wires-up quickly, but also when the event fires the event handler is invoked almost immediately.


Cons

- The method attaches the same event handler to every matched element in the selection.
- It doesn't work for elements added dynamically that matches the same selector.
- There are performance concerns when dealing with a large selection.
- The attachment is done upfront which can have performance issues on page load.

Using the Live Method


The `.live()` method uses the concept of event delegation to perform its so called "magic". The way you call `.live()` looks just like how you might call `.bind()`, which is very convenient. However, under the covers this method works much different. The `.live` method attaches the event handler to the root level document along with the associated selector and event information. By registering this information on the document it allows one event handler to be used for all events that have bubbled (a.k.a. delegated, propagated) up to it. Once an event has bubbled up to the document jQuery looks at the selector/event metadata to determine which handler it should invoke, if any. This extra work has some impact on performance at the point of user interaction, but the initial register process is fairly speedy.

```
1  /* The .live() method attaches the event handler to the root level
2     document along with the associated selector and event information
3     ( "#members li a" & "click" ) */
4
5  $( "#members li a" ).live( "click", function( e ) {} );
```

live.js hosted with  by **GitHub**[view raw](#)

```
/* The .live() method attaches the event handler to the root level
   document along with the associated selector and event information
1 2 3 4 5  ( "#members li a" & "click" ) */

$( "#members li a" ).live( "click", function( e ) {} );
```

live.js hosted with  by **GitHub**[view raw](#)

The good thing about this code as compared to the `.bind()` example above is that it is only attaching the event handler once to the document instead of multiple times. This not only is faster, but less

wasteful, however, there are many problems with using this method and they are outlined below.

Pros

- There is only one event handler registered instead of the numerous event handlers that could have been registered with the `.bind()` method.
- The upgrade path from `.bind()` to `.live()` is very small. All you have to do is replace "bind" to "live".
- Elements dynamically added to the DOM that match the selector magically work because the real information was registered on the document.
- You can wire-up event handlers before the document ready event helping you utilize possibly unused time.

Cons

- This method is deprecated as of jQuery 1.7 and you should start phasing out its use in your code.
- Chaining is not properly supported using this method.
- The selection that is made is basically thrown away since it is only used to register the event handler on the document.
- Using `event.stopPropagation()` is no longer helpful because the event has already delegated all the way up to the document.
- Since all selector/event information is attached to the document once an event does occur jQuery has match through its large metadata store using the `matchesSelector` method to determine which event handler to invoke, if any.
- Your events always delegate all the way up to the document. This can affect performance if your DOM is deep.

Using the Delegate Method

The `.delegate()` method behaves in a similar fashion to the `.live()` method, but instead of attaching the selector/event information to the document, you can choose where it is anchored. Just like the `.live()` method, this technique uses event delegation to work correctly.

If you skipped over the explanation of the `.Live()` method you might want to go back up and read it as I described some of the internal logic that happen.

```
1  /* The .delegate() method behaves in a similar fashion to the .live()
2     method, but instead of attaching the event handler to the document,
3     you can choose where it is anchored ( "#members" ). The selector
4     and event information ( "li a" & "click" ) will be attached to the
5     "#members" element. */
6
7  $( "#members" ).delegate( "li a", "click", function( e ) {} );
```

delegate.js hosted with ❤ by GitHub

[view raw](#)

```
/* The .delegate() method behaves in a similar fashion to the .live()
method, but instead of attaching the event handler to the document,
you can choose where it is anchored ( "#members" ). The selector
```

```
1 2 3 4 5 6 7 and event information ( "li a" & "click" ) will be attached to the
                "#members" element. */
```

```
$( "#members" ).delegate( "li a", "click", function( e ) { } );
```

delegate.js hosted with  by **GitHub**

[view raw](#)

The `.delegate()` method is very powerful. The above code will attach the event handler to the unordered list (`"#members"`) along with the selector/event information. This is much more efficient than the `.live()` method that always attaches the information to the document. In addition a lot of other problematic issues were resolved by introducing the `.delegate()` method. See the following outline for a detailed list.

Pros

- You have the option of choosing where to attach the selector/event information.
- The selection isn't actually performed up front, but is only used to register onto the root element.
- Chaining is supported correctly.
- jQuery still needs to iterate over the selector/event data to determine a match, but since you can choose where the root is the amount of data to sort through can be much smaller.
- Since this technique uses event delegation, it can work with dynamically added elements to the DOM where the selectors match.
- As long as you delegate against the document you can also wire-up event handlers before the document ready event.

Cons

- Changing from a `.bind()` to a `.delegate()` method isn't as straight forward.
- There is still the concern of jQuery having to figure out, using the `matchesSelector` method, which event handler to invoke based on the selector/event information stored at the root element. However, the metadata stored at the root element should be considerably smaller compared to using the `.live()` method.

Using the On Method

Did you know that the jQuery `.bind()`, `.live()`, and `.delegate()` methods are just one line pass throughs to the new jQuery 1.7 `.on()` method? The same is true of the `.unbind()`, `.die()`, and `.undelegate()` methods. The following code snippet is taken from the [jQuery 1.7.1 codebase in GitHub](#)...

```
1 // ... more code ...
2
3 bind: function( types, data, fn ) {
4     return this.on( types, null, data, fn );
5 },
6 unbind: function( types, fn ) {
7     return this.off( types, null, fn );
8 },
9
10 live: function( types, data, fn ) {
11     jQuery( this.context ).on( types, this.selector, data, fn );
```

```

12     return this;
13 },
14 die: function( types, fn ) {
15     jQuery( this.context ).off( types, this.selector || "**", fn );
16     return this;
17 },
18
19 delegate: function( selector, types, data, fn ) {
20     return this.on( types, selector, data, fn );
21 },
22 undelegate: function( selector, types, fn ) {
23     return arguments.length == 1 ?
24         this.off( selector, "**" ) :
25         this.off( types, selector, fn );
26 },
27
28 // ... more code ...

```

jquery-1.7.1.js hosted with ❤ by GitHub

[view raw](#)

```

// ... more code ...

bind: function( types, data, fn ) {
    return this.on( types, null, data, fn );
},
unbind: function( types, fn ) {
    return this.off( types, null, fn );
},

live: function( types, data, fn ) {
    jQuery( this.context ).on( types, this.selector, data, fn );
    return this;
1 2 3 4 5 6 7 8 9 10 11 },
12 13 14 15 16 17 18 19 die: function( types, fn ) {
20 21 22 23 24 25 26 27     jQuery( this.context ).off( types, this.selector || "**", fn );
28     return this;
    },

    delegate: function( selector, types, data, fn ) {
        return this.on( types, selector, data, fn );
    },
    undelegate: function( selector, types, fn ) {
        return arguments.length == 1 ?
            this.off( selector, "**" ) :
            this.off( types, selector, fn );
    },

    // ... more code ...

```

jquery-1.7.1.js hosted with â by GitHub

[view raw](#)

With that in mind, the usage of the new `.on()` method looks something like the following...


```

1  /* The jQuery .bind(), .live(), and .delegate() methods are just one
2     line pass throughs to the new jQuery 1.7 .on() method */
3
4  // Bind
5  $( "#members li a" ).on( "click", function( e ) {} );
6  $( "#members li a" ).bind( "click", function( e ) {} );
7
8  // Live
9  $( document ).on( "click", "#members li a", function( e ) {} );
10 $( "#members li a" ).live( "click", function( e ) {} );
11
12 // Delegate
13 $( "#members" ).on( "click", "li a", function( e ) {} );
14 $( "#members" ).delegate( "li a", "click", function( e ) {} );

```

on.js hosted with ❤ by GitHub

[view raw](#)

```

/* The jQuery .bind(), .live(), and .delegate() methods are just one
   line pass throughs to the new jQuery 1.7 .on() method */

// Bind
$( "#members li a" ).on( "click", function( e ) {} );
$( "#members li a" ).bind( "click", function( e ) {} );
1 2 3 4 5 6 7 8 9 10
11 12 13 14 // Live
$( document ).on( "click", "#members li a", function( e ) {} );
$( "#members li a" ).live( "click", function( e ) {} );

// Delegate
$( "#members" ).on( "click", "li a", function( e ) {} );
$( "#members" ).delegate( "li a", "click", function( e ) {} );

```

on.js hosted with â by GitHub

[view raw](#)

You'll notice that depending how I call the `.on()` method changes how it performs. You can consider the `.on()` method as being "overloaded" with different signatures, which in turn changes how the event binding is wired-up. The `.on` method bring a lot of consistency to the API and hopefully makes things slightly less confusing.

Pros

- Brings uniformity to the various event binding methods.
- Simplifies the jQuery code base and removes one level of redirection since the `.bind()`, `.live()`, and `.delegate()` call this method under the covers.
- Still provides all the goodness of the `.delegate()` method, while still providing support for the `.bind()` method if you need it.

Cons

- Brings confusion because the behavior changes based on how you call the method.

Conclusion (tl;dr)

If you have been confused about the various different types of event binding methods then don't worry, there has been a lot of history and evolution in the API over time. There are many people that view these methods as magic, but once you uncover some of how they work it will help you understand how to better code inside of your projects.

The biggest take aways from this article are that...

- Using the `.bind()` method is very costly as it attaches the same event handler to every item matched in your selector.
- You should stop using the `.live()` method as it is deprecated and has a lot of problems with it.
- The `.delegate()` method gives a lot of "bang for your buck" when dealing with performance and reacting to dynamically added elements.
- That the new `.on()` method is mostly syntax sugar that can mimic `.bind()`, `.live()`, or `.delegate()` depending on how you call it.
- The new direction is to use the new `.on` method. Get familiar with the syntax and start using it on all your jQuery 1.7+ projects.

Were there any pros or cons that you would have added to the above lists? Have you found yourself using the `.delegate` method more recently? What are your thoughts on the new `.on` method? Leave your thoughts in the comments. Thanks!

Webapps Security?

Generally, as a developer you can improve code security by:

- validating untrusted user input. If you must allow HTML in user responses, make sure to allow a small white list of HTML tags. Any tag that contains attributes should fail validation.
- generally rejecting input if it's not in an expected format. Sanitising the input has its place, but you need to cover any escape codes that an attacker could use
- always using server-side validation in addition to client side validation. The latter can be bypassed, and thus cannot be relied upon.
- properly encode/escape all output, paying particular attention to HTML and attributes, XML and attributes, URLs, and Javascript (or other script)

The [OWASP Top Ten](#) provides a powerful awareness document for web application security. The OWASP Top Ten represents a broad consensus about what the most critical web application security flaws are. Contributors include security experts from around the world who have shared their expertise to produce this list.

Below is a summary of the 2010 report - the previous having being in 2007.

1. *Injection including SQL injection*

This occurs when untrusted data is passed in a query to a data process e.g. the famous `param=' OR '1'='1` example put in a SQL query. The OWASP project also has some suggestions for protecting yourself against this attack - [SQL Injection prevention cheat sheet](#).

2. *Cross-site scripting (XSS)*

XSS is the most common web application security flaw. This occurs when an application includes user supplied data in a page sent to the browser, without properly validating or escaping that data. There are 3 variants:

- Stored XSS - where the data is already in the application's data store e.g. in forums, CMS
- Reflected XSS - where data is obtained from a user and displayed to another user
- DOM-based/local XSS - attacks client side Javascript code and variables

In the first two, the server serves a contaminated page to the client, while for the last the code is injected at the client end using Javascript.

Note that using HTTPS is not a defence against XSS.

3. *Broken Authentication and Session Management*

This could be from anonymous attackers wanting to gain access to other people's accounts. Insiders may also try this to mimick another user and thereby cover their tracks. Authentication can be tricky, and leaks or flaws in the authentication can be exploited. This risk can be minimised by having a single set of authentication and session management controls for developers on a project. An example interface can be found at [ESAPI Authenticator and User APIs](#).

4. *Insecure Direct Object References*

This is where an authorised person gains access to a prohibited resource simply by changing a reference in a web request e.g. changing `some.site?reportid=1` (which is the user is authorised to access) to `some.site?reportid=2` (which should be prohibited). This can be mitigated by using indirect references to restricted resources. The user can then only select from an allowed subset.

5. *Cross-site Request Forgery (CSRF)*

A request is made to your site from another site, using forged HTTP requests. Such requests can be submitted via img tags. If the user is authenticated on your site, the attack could make changes to your site in the user's name. The changes should be restricted to only what the user is authorised to do. This risk can be mitigated by using hidden token for each HTTP request, which is validated when a user makes a request. A request is only allowed if the token validates. Such a token should at least be unique per user session - see [CSRF cheat sheet](#).

6. *Security Misconfiguration*

An attacker exploits default accounts, unused pages, flaws, unprotected resources to gain unauthorised access to or knowledge of the system. This attack could lead to compromising of the entire system. You can reduce this risk by making sure that default accounts are properly configured/disabled, error messages do not unnecessarily give away system configuration information, and system is kept up to date.

7. *Insecure Cryptographic Storage*

This risk is mainly from authorised users of your system, including system administrators. The most common flaw in this area is simply not encrypting data that deserves encryption. When encryption is employed, unsafe key generation and storage, not rotating keys, and weak algorithm usage is common. Use of weak or unsalted hashes to protect passwords is also common. For hashes, SHA-256 is recommended over MD5 or SHA-1 - see [Guide to Cryptography](#).

8. *Failure to Restrict URL Access*

This is when an attacker can be granted access to a privileged page, simply by specifying the url, e.g. an anonymous user getting access to an administrator-only page. Mitigate by making

authentication and authorisation role-based. Also allow for a flexible association between roles and permissions. By default, users should be denied access to restricted pages.

9. *Insufficient Transport Layer Protection*

Requests not using SSL/TLS are open to network sniffing by a malicious user, which can lead to exposure of session IDs. This can be mitigated by using SSL/TLS for all requests. However, for performance reasons, it's more common to use SSL/TLS for the authentication stages only. A good practice is to use SSL/TLS for all sensitive pages, including administrative access. Also this [TLS cheat sheet](#) recommends not redirecting users from a non-TLS login page to a TLS login page.

10. *Unvalidated Redirects and Forwards*

The risk is when a site redirects users based on some parameter in the request. Always verify the target of the redirect is a valid URL or page, and the user is authorised to access that page. If possible, avoid redirects altogether.