Web Development Fundamentals

AUTHOR: Isagani Mendoza (ITJUMPSTART.NET)

PREFACE:

There are only two types of application development. Alex Payne calls it "scaling in the small" versus "scaling in the large". The top 1% of all developers are the likes of Google, Facebook, Amazon and others who are scaling in the large. The rest is a continuum of solo developers and small-and-medium sized businesses who simply want to get the job done.

Linode founder Chris Aker said it best:

> I think, largely the difference between a VPS and a cloud server is all marketing... When people think cloud, they think ephemeral, scale-out, highly flexible infrastructure. But really at the essence of it, what you're getting out the end of either one of those is a Linux virtual machine... I think plenty of people get sold on the promise of Amazon. Cloud isn't magic, you can't just put your application in the cloud and expect it to scale to a billion users. It just doesn't work like that... it's not a magic button.

The same is true of Web application development. There are solutions for trivial tasks all the way up to the most complex. For most developers, all you need is jQuery, not client-side frameworks.

This is why you need to know the fundamentals. Once you know the basics, you can tell if somebody is selling you snake oil or the real deal. Once you know the fundamentals, you can tell if somebody is selling you bandaids and chicken wire.

As Joel Spolsky has said:

> So the abstractions save us time working, but they don't save us time learning.

I hope this book about Web fundamentals will save you some time learning.

SHARE THIS:

References:

- Chad Hower on three-tier architecture

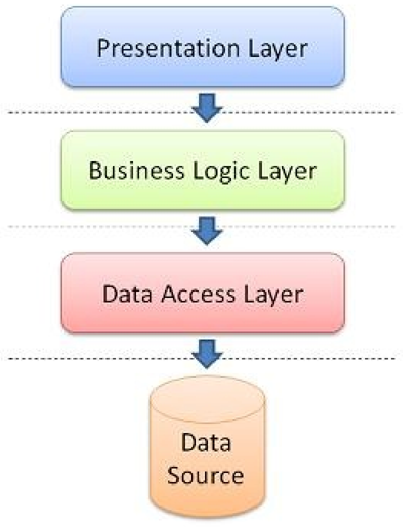
- Jake Spurlock, author of "Bootstrap"
- jQuery Foundation for chapters on front-end development
- Michael Fogus, author of "Functional JavaScript"
- Peter Smith, author of "Professional Website Performance"
- Auth0 for chapter on JSON Web Token
- Rob Pike for concurrency vs parallelism
- Baron Schwartz et al, authors of "High Performance MySQL"
- Adam Wiggins for The Twelve-Factor App
- Leonard Richardson and Sam Ruby, authors of "RESTful Web Services"
- Martin Abbott and Michael Fisher, authors of "Scalability Rules"
- Jeff Atwood, SQL joins at "Coding Horror" blog
- C.L.Mofatt, SQL joins at CodeProject
- MathIsFun.com for Venn diagram examples
- Thomas Frank for MySQL to JSON script

Acknowledgement:

- Thanks to all developers of open-source projects without whom this guidebook would have been non-existent

Part I. Three-Tier Architecture

Presentation Layer



Build static pages

The presentation tier is about the user interface and should be static as much as possible unless you are building a media site. The debate between static page applications and dynamic pages is as divisive as the debate between Web apps and native mobile apps. Technology debates are akin to religious wars. The merits of each side depend on each situation and thus, there is no one-size-fits-all solution. The only thing that matters is context.

In this book, I want to state the **context** once and for all: **that of creating CRUD Web apps using a relational database, not mobile apps**.

In this context, static page applications (SPA) are composed of three parts:

- HTML - presentation
- CSS - style
- JavaScript - behavior

When the application needs data, it simply invokes AJAX calls in the server and render it at the client-side with or without using a template engine. The data is being transported in JSON format. Read how LinkedIn left JSP in the dust.

A static app is also known as single-page app (SPA) since it embodies one aspect of an application in a single Web page. In reality, SPA rarely stands alone as it needs to invoke other SPA to round out all the business aspects of an application.

More info at staticapps.org.

Use JSON as data transport

JSON (JavaScript Object Notation) is the de facto data transport for Web pages since it speaks the same JavaScript language on the client-side, and is easily consumed on the server-side regardless if it is JavaScript or not. However, do not abuse JSON. For simple response data like a one-line status message, you may use plain text.

Do not use client-side frameworks

Client-side frameworks like Angular, Ember, Knockout, Backbone and others were built by advanced JavaScript developers as an abstraction for their own use.

> Instead of thinking in terms of MVC, you should think in terms of Three Tier Architecture. MVC is a leaky abstraction because Web development straddles at least three sides.

Model - leaks to SQL or NoSQL query language (storage tier)

View - HTML and CSS (presentation tier)

Controller - RESTful API (logic tier)

You should use an interface instead of a leaky abstraction (see Rule 4 at Part III).

Today, there is one universal interface that can do everything what all those client-side frameworks can do: jQuery.

Why jQuery?

- It is modular, extensible and easy to learn. It can be used in small projects and can scale to large ones by using building blocks like custom events and promise-based programming. jQuery can interface with functional programming libraries as well like Underscore/LoDash and BaconJS.

- Instead of thinking in terms of models, you can simply use JSON and it will work using jQuery AJAX API regardless of server-side languages.

- The issue with client-side frameworks is that they impose a certain methodology in doing the same thing. But there lies the problem. Software development just like any human endeavor is, in the words of Austrian philosopher Paul Feyerabend, "anything goes". There is no universal methodology that will hammer out all use cases but you can have a universal interface. Like Lego blocks, you can mix and match components however you like as long as it conforms to the interface. The loose coupling of interface and implementation is the hallmark of great software design.

What about two-way data binding?

I will quote Michael S. Mikowski and Josh C. Powell, authors of "Single Page Web Applications":

> Beware SPA “framework” libraries bearing gifts

Some SPA “framework” libraries promise “automatic two-way data binding” which certainly sounds good. But we’ve learned a few points of caution about such promises in spite of the impressive canned demos:

- We’ll need to learn the language of the library—its API and terminology to make it do the things a well-groomed presenter can do. This can be a significant investment.
- The library author often has a vision of how an SPA is supposed to be structured. If our SPA doesn’t meet that vision, retrofitting can get expensive.
- Libraries can be big, buggy, and offer another layer of complexity where things can go wrong.

- The library’s data binding may often not meet our SPA requirements.

Let’s focus on that last point. Perhaps we’d like a user to be able to edit a row in a table, and when finished, either accept the entire row or cancel (in which case the row should revert back to old values). And, when the user is done editing rows, we’d like to have the user accept or cancel the entire edited table. And only then would we even consider saving the table to the backend.

The probability of a framework library supporting this kind of reasonable interaction “out-of-the-box” is low. So if we go with a library, we’ll need to create a custom override method to circumvent the default behavior. If we have to do that just a few times, we can easily end up with more code, more layers, more files, and more complexity than if we’d written the damn thing ourselves in the first place.

After a few well-intended attempts, we’ve learned to approach framework libraries with caution. We’ve found they can add complexity to an SPA rather than making development better, faster, or easier to understand. That doesn’t mean we should never use framework libraries—they have their place. But our example SPAs (and quite a few in production) work fine with just jQuery, some plugins, and a few specialized tools like TaffyDb. Often, simpler is better.

See my other rules "Do not use ORM" and "Do not use MVC framework" and I will rest my case against client-side frameworks.

Do not reinvent the JavaScript module system

There is only one JavaScript module system that works now and into the future: the Node module system (not AMD or RequireJS).

Breaking JavaScript code into modules is the key to developing large-scale JavaScript applications. The Node module system works in the server only but it can also work in the Web browser using Browserify.

The simplicity of Node Package Manager trumps all the other alternatives because

- it enables you to use one model both for server-side and client-side development
- it does nested dependency which frees the developer from doing it on their own.

Use token for authentication, not cookies

User authentication in the context of Web application refers to a mechanism where a user has been authenticated (for example, with a username/email address and password) and gets a corresponding cookie or token which is being sent to the Web server for every protected API request. For more info about why cookies are bad, read Jesse Hallett's post. Hint: A cookie is bad when used as authentication mechanism but works just fine when used as storage mechanism. But why rely on cookies when desktop and mobile browsers already support Web storage (localStorage and sessionStorage)?

Overview of User Authentication (courtesy: Mozilla)

Static websites are easy to scale. You can cache the heck out of them and you don't have state to propagate between the various servers that deliver this content to end-users.

Unfortunately, most web applications need to carry some state in order to offer a personalized experience to users. If users can log into your site, then you need to keep sessions for them. The typical way that this is done is by setting a cookie with a random session identifier and storing session details on the server under this identifier.

Scaling a stateful service

Now, if you want to scale that service, you essentially have three options:

- replicate that session data across all of the web servers,
- use a central store that each web server connects to, or
- ensure that a given user always hits the same web server

These all have downsides:

- Replication has a performance cost and increases complexity.
- A central store will limit scaling and increase latency.
- Confining users to a specific server leads to problems when that
- server needs to come down.

However, if you flip the problem around, you can find a fourth option: storing the session data on the client.

Client-side sessions

Pushing the session data to the browser has some obvious advantages:

- the data is always available, regardless of which machine is serving a user
- there is no state to manage on servers
- nothing needs to be replicated between the web servers
- new web servers can be added instantly

There is one key problem though: you cannot trust the client not to tamper with the session data.

For example, if you store the user ID for the user's account in a cookie, it would be easy for that user to change that ID and then gain access to someone else’s account.

While this sounds like a deal breaker, there is a clever solution to work around this trust problem: **store the session data in a tamper-proof package**. That way, there is no need to trust that the user hasn’t modified the session data. It can be verified by the server.

What that means in practice is that you encrypt and sign the cookie using a server key to keep users from reading or modifying the session data. This is what node-client-sessions does.

Benefits of Token-based Authentication

Now, if you replace cookies with tokens, here are the benefits: (courtesy: Auth0)

- Cross-domain / CORS: cookies + CORS don't play well across different domains. A token-based approach allows you to make AJAX calls to any server, on any domain because you use an HTTP header to transmit the user information.
- Stateless (a.k.a. Server side scalability): there is no need to keep a session store, the token is a self-contained entity that conveys all the user information. The rest of the state lives in cookies or local storage on the client side.
- CDN: you can serve all the assets of your app from a CDN (e.g. javascript, HTML, images, etc.), and your server side is just the API.
- Decoupling: you are not tied to a particular authentication scheme. The token might be generated anywhere, hence your API can be called from anywhere with a single way of authenticating those calls.
- Mobile ready: when you start working on a native platform (iOS, Android, Windows 8, etc.) cookies are not ideal when consuming a secure API (you have to deal with cookie containers). Adopting a token-based approach simplifies this a lot.
- CSRF: since you are not relying on cookies, you don't need to protect against cross site requests (e.g. it would not be possible to your site, generate a POST request and re-use the existing authentication cookie because there will be none).
- Performance: we are not presenting any hard perf benchmarks here, but a network roundtrip (e.g. finding a session on database) is likely to take more time than calculating an HMACSHA256 to validate a token and parsing its contents.
- Login page is not a special case: If you are using Protractor to write your functional tests, you don't need to handle any special case for login.

- Standard-based: your API could accepts a standard JSON Web Token (JWT). This is a standard and there are multiple backend libraries (.NET, Ruby, Java, Python, PHP) and companies backing their infrastructure (e.g. Firebase, Google, Microsoft). As an example, Firebase allows their customers to use any authentication mechanism, as long as you generate a JWT with certain pre-defined properties, and signed with the shared secret to call their API.

What's JSON Web Token?

JSON Web Token (JWT, pronounced jot) is a relatively new token format used in space-constrained environments such as HTTP Authorization headers. JWT is architected as a method for transferring security claims-based between parties.

For Web applications that have user login requirements, it is better to use sessionStorage rather than localStorage to store the JSON Web token but not sensitive data such as password! Data in sessionStorage persists until the Web browser is closed.

More about Tokens (courtesy: https://auth0.com/blog/2014/01/27/ten-things-you-should-know-about-tokens-and-cookies/)

Tokens need to be stored somewhere (local/session storage or cookies)

In the context of tokens being used on single page applications, some people have brought up the issue about refreshing the browser, and what happens with the token. The answer is simple: you have to store the token somewhere: in session storage, local storage or a client side cookie. Most session storage polyfills fallback to cookies when the browser doesn't support it.

If you are wondering

> "but if I store the token in the cookie I'm back to square one".

Not really, in this case you are using cookies as a storage mechanism, not as an authentication mechanism (i.e. the cookie won't be used by the web framework to authenticate a user, hence no XSRF attack)

Tokens can expire like cookies, but you have more control

Tokens have an expiration (in JSON Web Tokens is represented by exp property), otherwise someone could authenticate forever to the API once they logged in at least once. Cookies also have expiration for the same reasons.

In the world of cookies, there are different options to control the lifetime of the cookie:

- Cookies can be destroyed after the browser is closed (session cookies).
- In addition to this you can implement a server side check (typically done for you by the web framework in use), and you could implement expiration or sliding window expiration.
- Cookies can be persistent (not destroyed after the browser is closed) with an expiration.

In the tokens world, once the token expires, you simply want to get a new one. You could implement an endpoint to refresh a token that will:

- Validate the old token
- Check if the user still exists or access hasn't been revoked or whatever makes sense for your application
- Issue a new token with a renewed expiration

Local/session storage won't work across domains, use a marker cookie

If you set a cookie's domain to .yourdomain.com it can be accessed from yourdomain.com and app.yourdomain.com, making it easier to detect from the main domain (where you probably have, let's say, your marketing site) that the user is already logged in and redirect her to app.yourdomain.com.

Tokens stored in local/session storage, on the other hand, can't be accessed from different domains (even if these are subdomains). So what can you do?

One possible option is, when the user authenticates on app.yourdomain.com and you generate a token you can also set a cookie set to .yourdomain.com

Then, in yourdomain.com you can check the existence of that cookie and redirect to app.yourdomain.com if the cookie exists. The token will be available on app subdomain, and from there on, the usual flow applies (if the token is still valid use it, if not get a new one unless last login was more than the threshold you set up).

It could happen that the cookie exists but the token was deleted or something else happened. In that case, the user would have to login again. But what's important to highlight here is, as we said before, we are not using the cookie as an authentication mechanism, just as a storage mechanism that happens to support storing information across different domains.

Preflight requests will be sent on each CORS request

Someone pointed out that the Authorization header is not a simple header, hence a pre-flight request would be required for all the requests to a particular URLs.

But this happens if you are sending Content-Type: application/json for instance. So this is already happening for most applications.

One small caveat, the OPTIONS request won't have the Authorization header itself, so your web framework should support treating OPTIONS and the subsequent requests differently (Note: Microsoft IIS for some reason seems to have issues with this).

When you need to stream something, use the token to get a signed request

When using cookies, you can trigger a file download and stream the contents easily. However, in the tokens world, where the request is done via XHR, you can't rely on that. The way you solve this is by generating a signed request like AWS does, for example. Hawk Bewits is a nice framework to enable this:

Request:

```
POST /download-file/123
Authorization: Bearer...
```

Response:

```
ticket=lahdoiasdhoiwdowijaksjdoaisdjoasidja
```

This ticket is stateless and it is built based on the URL: host + path + query + headers + timestamp + HMAC, and has an expiration. So it can be used in the next, say 5 minutes, to download the file.

You would then redirect to

```
/download-file/123?ticket=lahdoiasdhoiwdowijaksjdoaisdjoasidja
```

The server will check that the ticket is valid and continue with business as usual.

It's easier to deal with XSS than XSRF

Cookies have this feature that allows setting an HttpOnly flag from server side so they can only be accessed on the server and not from JavaScript. This is useful because it protects the content of that cookie to be accessed by injected client-side code (XSS).

Since tokens are stored in local/session storage or a client side cookie, they are open to an XSS attack getting the attacker access to the token. This is a valid concern, and for that reason **you should keep your token expiration low.**

But if you think about the attack surface on cookies, one of the main ones is XSRF. The reality is that XSRF is one of the most misunderstood attacks, and the average developer, might not even understand the risk, so lots of applications lack anti-XSRF mechanism. However, everybody understands what injection is. Put simply, if you allow input on your website and then render that without escaping it, you are open to XSS. So based on our experience, it is easier to protect against XSS than protecting against XSRF. Adding to that, anti-XSRF is not built-in on every web framework. XSS on the other hand is easy to prevent by using the escape syntax available by default on most template engines.

The token gets sent on every request, watch out its size

Every time you make an API request you have to send the token in the Authorization header.

Depending on how much information you store in that token, it could get big. On the other hand, session cookies usually are just an identifier (connect.sid, PHPSESSID, etc.) and the rest of the content lives on the server (in memory if you just have one server or a database if you run on a server farm).

Now, nothing prevents you from implementing a similar mechanism with tokens. The token would have the basic information needed and on the server side you would enrich it with more data on every API call. This is exactly the same thing cookies will do, with the difference that you have the additional benefit that this is now a conscious decision, you have full control, and is part of your code.

It is worth mentioning that you could also have the session stored completely on the cookie (instead of being just an identifier). Some web platforms support that, others not. For instance, in node.js you can use mozilla/node-client-sessions.

If you store confidential info, encrypt the token

The signature on the token prevents tampering with it. TLS/SSL prevents man in the middle attacks. But if the payload contains sensitive information about the user (e.g. SSN, whatever), you can also encrypt them. The JWT spec points to the JWE spec but most of the libraries don't implement JWE yet, so the simplest thing is to just encrypt with AES-CBC.

Of course you can use the approach on #7 and keep confidential info in a database.

JSON Web Tokens can be used in OAuth

Tokens are usually associated with OAuth. OAuth 2 is an authorization protocol that solves identity delegation. The user is prompted for consent to access his/her data and the authorization server gives back an access_token that can be used to call the APIs acting as that user.

Typically these tokens are opaque. They are called bearer tokens and are random strings that will be stored in some kind of hash-table storage on the server (db, cache, etc.) together with an expiration, the scope requested (e.g. access to friend list) and the user who gave consent. Later, when the API is called, this token is sent and the server lookup on the hash-table, rehydrating the context to make the authorization decision (did it expire? does this token has the right scope associated for the API that wants to be accessed?).

The main difference between these tokens and the ones we've been discussing is that signed tokens (e.g.: JWT) are "stateless". They don't need to be stored on a hash-table, hence it's a more lightweight approach. OAuth2 does not dictate the format of the access_token so you could return a JWT from the authorization server containing the scope/permissions and the expiration.

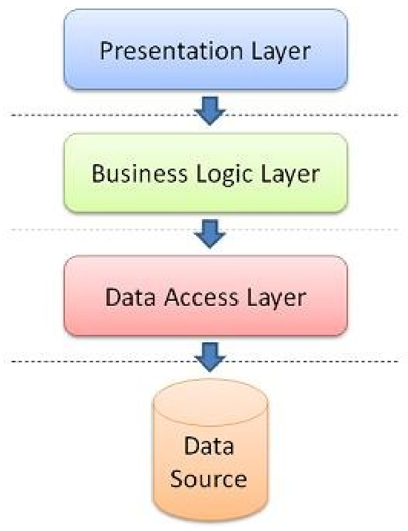
Tokens are not silver bullets, think about your authorization use cases carefully

Couple of years ago Auth0 helped a big company implement a token-based architecture. This was a 100.000+ employees company with tons of information to protect. They wanted to have a centralized organization-wide store for "authentication & authorization". Think about "user X can read field id and name of clincial trial Y for hospital Z on country W" use cases. This fine-grained authorization, as you can imagine, can get unmanageable pretty quickly, not only technically but also administratively.

- Tokens can get really big
- Your apps/APIs gets more complicated
- Whoever grant these permissions will have a hard time managing all this.

We ended up working more on the information architecture side of things to make sure the right scopes and permissions were created. Conclusion: resist the temptation of putting everything into tokens and do some analysis and sizing before going all the way with this approach.

Application Layer



Do not use ORM

ORM (Object/Relational Mapper) is a leaky abstraction borne out of a need to abstract specific use cases when dealing with an RDBMS. Since SQL is the query language of relational databases, you better start learning it at the soonest time possible because ORM won't cover all use cases that is natively available with SQL. Granted, SQL has different syntax among RDBMS products but that is no excuse compared to differing and more complex syntax among ORMs in every programming language! SQL is not hard to learn. And if you will venture to the world of NoSQL, you will find that there are even more query languages to learn with respect to a particular NoSQL data store. In the worlds of relational and non-relational data stores, SQL and NoSQL's respective query language are the gatekeepers to the source of data.

Do not use MVC framework

MVC is a leaky abstraction since it obscures the mechanics of the Web. MVC whether you use it on the client-side or server-side reminds me of anybody who lumps everything as an object (read Steve Yegge's article: Execution in the Kingdom of Nouns).

Here, I quote Jason Roelofs' article "Where's Your Business Logic?",

> If I sat down with your code base and asked you how such-and-such a feature is implemented, what would you say? Would you lead me through controller filters and actions? What about model methods, includes, callbacks or observers? How about objects in lib/ (oh, and there's also this other call to a mailer to make sure emails get sent out…), or would you even dive down into the database itself? Or would you send me all of these paths at once? Can you even remember or properly follow the code flow for a given use case? If any of this sounds familiar, and you're nodding in a “yeah I know but…” fashion, but feel stuck, I've got an answer. The fundamental problem with almost every Rails project (and I'm sure in other frameworks as well), is that there is no direct codifying of the business rules and use cases of the application. There is no single location you can point to and say “here, these objects implement our use cases”. I put some of the blame on Rails itself, which has guided developers to use Controllers, Models, or Libraries, and nothing else.

Here is a code snippet from the jQuery version of TodoMVC app:

```
    var App = {
        init: function () {
            this.todos = util.store('todos-jquery');
            this.cacheElements();
            this.bindEvents();

            Router({
                '/:filter': function (filter) {
                    this.filter = filter;
                    this.render();
                }.bind(this)
            }).init('/all');
        },
        cacheElements: function () {
            this.todoTemplate = Handlebars.compile($('#todo-template').html());
            this.footerTemplate = Handlebars.compile($('#footer-template').html());
            this.$todoApp = $('#todoapp');
            this.$header = this.$todoApp.find('#header');
            this.$main = this.$todoApp.find('#main');
            this.$footer = this.$todoApp.find('#footer');
            this.$newTodo = this.$header.find('#new-todo');
            this.$toggleAll = this.$main.find('#toggle-all');
            this.$todoList = this.$main.find('#todo-list');
            this.$count = this.$footer.find('#todo-count');
            this.$clearBtn = this.$footer.find('#clear-completed');
        },
        bindEvents: function () {
            var list = this.$todoList;
            this.$newTodo.on('keyup', this.create.bind(this));
            this.$toggleAll.on('change', this.toggleAll.bind(this));
            this.$footer.on('click', '#clear-completed', this.destroyCompleted.bind(this));
            list.on('change', '.toggle', this.toggle.bind(this));
            list.on('dblclick', 'label', this.edit.bind(this));
            list.on('keyup', '.edit', this.editKeyup.bind(this));
            list.on('focusout', '.edit', this.update.bind(this));
            list.on('click', '.destroy', this.destroy.bind(this));
        },
        render: function () {
            var todos = this.getFilteredTodos();
            this.$todoList.html(this.todoTemplate(todos));
            this.$main.toggle(todos.length > 0);
            this.$toggleAll.prop('checked', this.getActiveTodos().length === 0);
            this.renderFooter();
            this.$newTodo.focus();
            util.store('todos-jquery', this.todos);
        },
        renderFooter: function () {
            var todoCount = this.todos.length;
            var activeTodoCount = this.getActiveTodos().length;
            var template = this.footerTemplate({
                activeTodoCount: activeTodoCount,
                activeTodoWord: util.pluralize(activeTodoCount, 'item'),
                completedTodos: todoCount - activeTodoCount,
                filter: this.filter
            });

            this.$footer.toggle(todoCount > 0).html(template);
        },
```

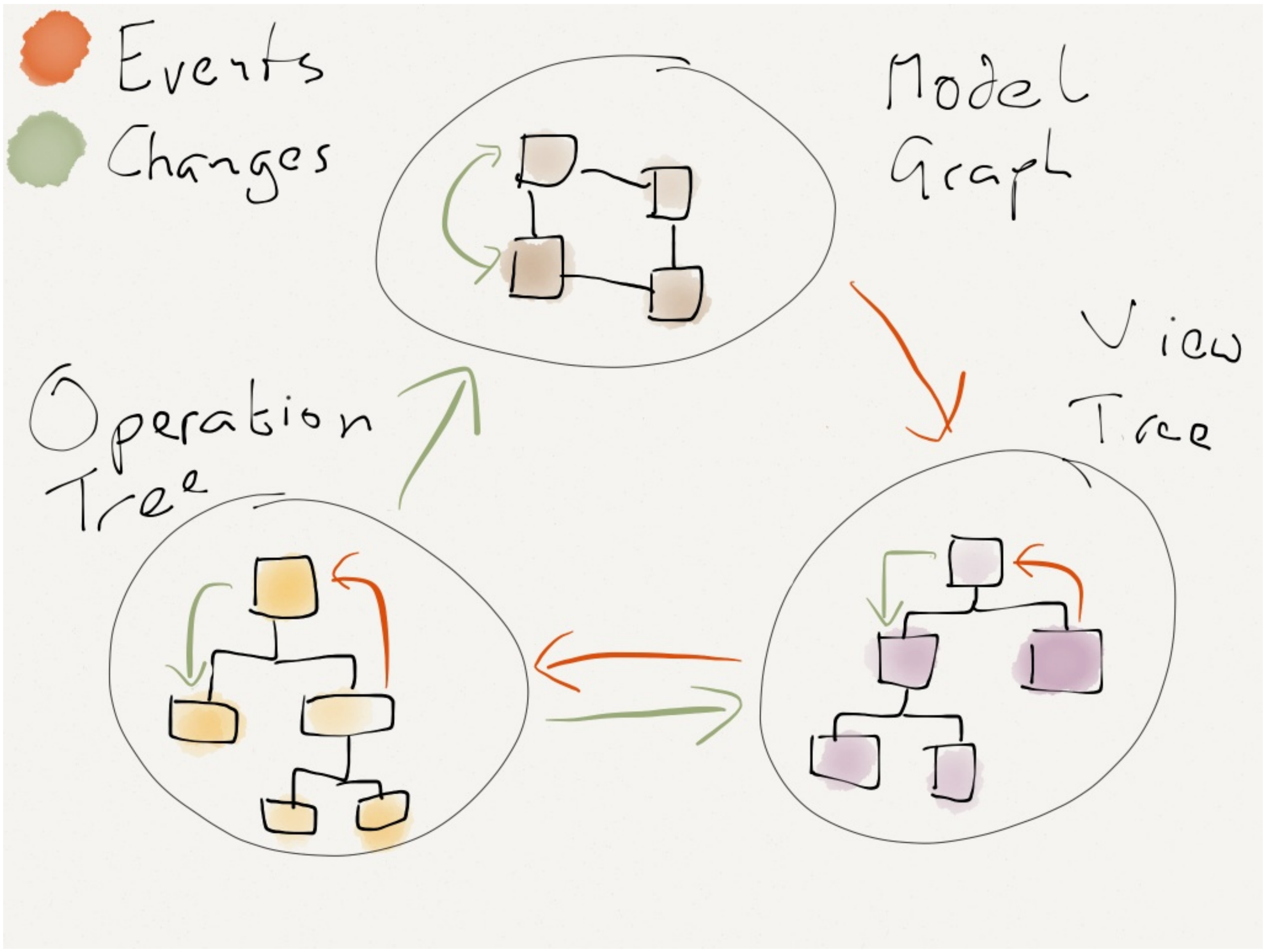
I mean no disrespect to the author but this code is more abstract than using jQuery's handling of built-in or custom events. The problem lies with the abstraction of an object as a noun instead of the more intuitive object as an event (verb).

> Wouldn't it be simpler if we code in jQuery like below and capture state using closures, deferred/promises or by emitting custom events?

```
$( "#element" ).on( "click", function() {
    ...
});
```

In my view, MVC is more abstract since it refactors code from a noun-based object perspective. Is there a more intuitive way to do this?

> I quote Conrad Irwin in his article "MVC is dead, it's time to MOVE on."



MVC is a phenomenal idea. You have models, which are nice self-contained bits of state, views which are nice self-contained bits of UI, and controllers which are nice self-contained bits of …

What?

I'm certainly not the first person to notice this, but the problem with MVC as given is that you end up stuffing too much code into your controllers, because you don't know where else to put it.

To fix this I've been using a new pattern: MOVE. Models, Operations, Views, and Events.

I'll define the details in a minute, but this diagram shows the basic structure of a MOVE application.

- Models encapsulate everything that your application knows.
- Operations encapsulate everything that your application does.
- Views mediate between your application and the user.
- Events are used to join all these components together safely.

In order to avoid spaghetti code, it's also worth noting that there are recommendations for what objects of each type are allowed to do. I've represented these as arrows on the diagram. For example, views are allowed to listen to events emitted by models, and operations are allowed to change models, but models should not refer to either views or operations.

Models

The archetypal model is a "user" object. It has at the very least an email address, and probably also a name and a phone number.

In a MOVE application models only wrap knowledge. That means that, in addition to getters and setters, they might contain functions that let you check "is this the user's password?", but they don't contain functions that let you save them to a database or upload them to an external API. That would be the job of an operation.

Operations

A common operation for applications is logging a user in. It's actually two sub-operations composed together: first get the email address and password from the user, second load the "user" model from the database and check whether the password matches.

Operations are the doers of the MOVE world. They are responsible for making changes to your models, for showing the right views at the right time, and for responding to events triggered by user interactions. In a well factored application, each sub-operation can be run independently of its parent; which is why in the diagram events flow upwards, and changes are pushed downwards.

What's exciting about using operations in this way is that your entire application can itself be treated as an operation that starts when the program boots. It spawns as many sub-operations as it needs, where each concurrently existing sub-operation is run in parallel, and exits the program when they are all complete.

Views

The login screen is a view which is responsible for showing a few text boxes to the user. When the user clicks the "login" button the view will yield a "loginAttempt" event which contains the username and password that the user typed.

Everything the user can see or interact with should be powered by a view. They not only display the state of your application in an understandable way, but also simplify the stream of incoming user interactions into meaningful events. Importantly views don't change models directly, they simply emit events to operations, and wait for changes by listening to events emitted by the models.
Events

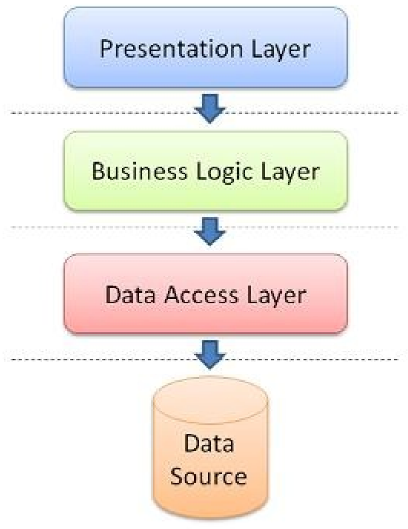
The "loginAttempt" event is emitted by the view when the user clicks login. Additionally, when the login operation completes, the "currentUser" model will emit an event to notify your application that it has changed.

Listening on events is what gives MOVE (and MVC) the inversion of control that you need to allow models to update views without the models being directly aware of which views they are updating. This is a powerful abstraction technique, allowing components to be coupled together without interfering with each other.

Why now?

I don't wish to be misunderstood as implying that MVC is bad; it truly has been an incredibly successful way to structure large applications for the last few decades. Since it was invented however, new programming techniques have become popular. Without closures (or anonymous blocks) event binding can be very tedious; and without deferrables (also known as deferreds or promises) the idea of treating individual operations as objects in their own right doesn't make much sense.

Database Layer



Do not put application logic at the database level

Why?

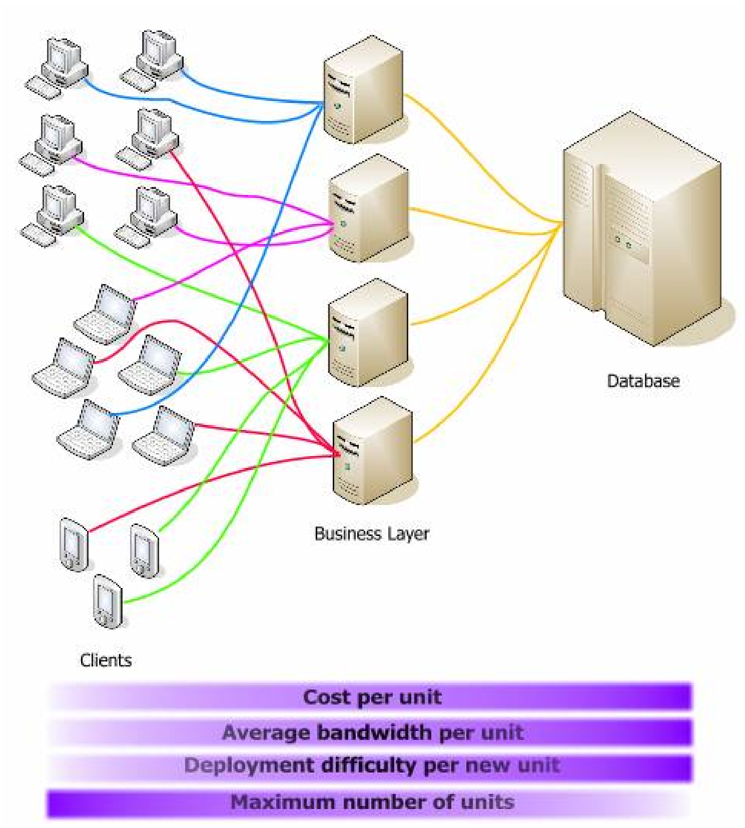
a) Because applications have its own life cycle (aborted, shelved, rewrite, refactor, etc)

b) Baron Schwartz have said in "Web Operations" book:

> The database should store the data, not the application itself.

c) Stored procedures are not enough to account for business logic complexity

d) Application deployment can be complex (see Chad Hower)



Enforce foreign key constraints at the application level

Foreign keys are part of a sound relational database design. However, do not enforce foreign key constraints at the database level. Every developer worth his salt ought to know the relational structure of the database because the database is the lifeblood of every business. You have no business developing code if you don't know the structure of your relational database. Of course, foreign keys are not applicable to NoSQL data stores.

Furthermore, it's a matter of performance. To quote the authors of High Performance MySQL, Third Edition:

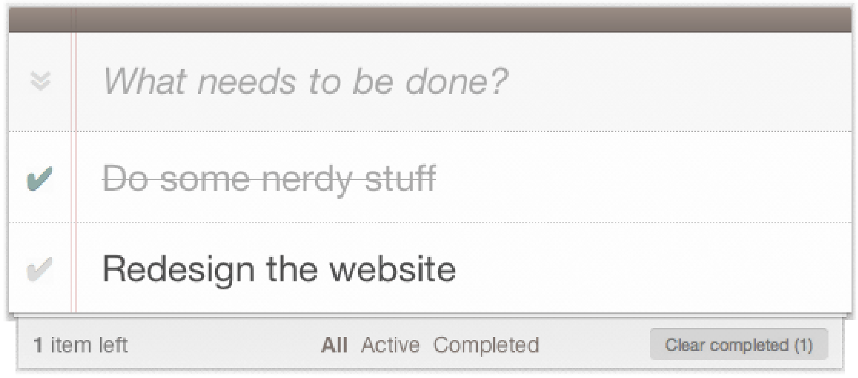
> Instead of using foreign keys as constraints, it’s often a good idea to constrain the values in the application. Foreign keys can add significant overhead. We don’t have any benchmarks to share, but we have seen many cases where server profiling revealed that foreign key constraint checks were the performance problem, and removing the foreign keys improved performance greatly.

Do not abuse RDBMS

Relational databases are designed for a specific use case: relational data model. If you don't need the data constraints inherent in a relational model, you may use NoSQL where it is more appropriate (key/value, columnar, document-oriented, graph databases). In the same vein, don't abuse the NoSQL proposition. Don't fall into the trap of IT dogma wars.

Part II. Front-End Development

Build your own Todo App



Before you begin learning front-end development, I have adopted the ToDo template from TodoMVC.com and this will serve as your main exercise throughout the book.

The ToDo application created at TodoMVC.com spawns a lot of examples using different frameworks. That application leans heavily on the MVC pattern and is geared towards intermediate JavaScript developers. In this book, I want you to deviate from the TodoMVC specification and tinker on your own. After all, the aim is not to submit your own ToDo app but to learn organically.

Here are some deviations:

- For consistency, use jQuery
- You need not use Bower for now
- You could have your own coding style but at least, use double-quotes in HTML and single-quotes in JS and CSS, and use a constant instead of the keyCode directly: var ENTER_KEY = 13;
- Instead of using MVC, use event-driven and promise-based programming of jQuery
- Use {{ }} delimiter in Underscore template

Client-side templating is beyond the scope of this book. Read Ben Nadel's article which I quote below:

> All client-side template engines attempt to answer the same question: How do I merge this data into this block of HTML?

You can read the following references regarding client-side templates:

- Brad Dunbar Underscore Templates

- LinkedIn's move to DustJS client-side template

So leaving aside the client-side template issue, here are the functionality specifications of your ToDo app copied verbatim from the site (a zip template is included in this book excluding jQuery):

Specs

No todos

When there are no todos, #main and #footer should be hidden.

New todo

New todos are entered in the input at the top of the app. Pressing Enter creates the todo, appends it to the todo list and clears the input. Make sure to .trim() the input and then check that it's not empty before creating a new todo.
Mark all as complete

This checkbox toggles all the todos to the same state as itself. Make sure to clear the checked state after the the "Clear completed" button is clicked. The "Mark all as complete" checkbox should also be updated when single todo items are checked/unchecked. Eg. When all the todos are checked it should also get checked.

Item

A todo item has three possible interactions:

- Clicking the checkbox marks the todo as complete by updating its completed value and toggling the class completed on its parent "li"
- Double-clicking the activates editing mode, by toggling the .editing class on its "li"
- Hovering over the todo shows the remove button (.destroy)

Editing

When editing mode is activated it will hide the other controls and bring forward an input that contains the todo title, which should be focused (.focus()). The edit should be saved on both blur and enter, and the editing class should be removed. Make sure to .trim() the input and then check that it's not empty. If it's empty the todo should instead be destroyed. If escape is pressed during the edit, the edit state should be left and any changes be discarded.

Counter

Displays the number of active todos in a pluralized form. Make sure the number is wrapped by a **tag. Also make sure to pluralize the item word correctly: 0 items, 1 item, 2 items. Example: 2 items left**

Clear completed button

Displays the number of completed todos, and when clicked, removes them. Should be hidden when there are no completed todos.

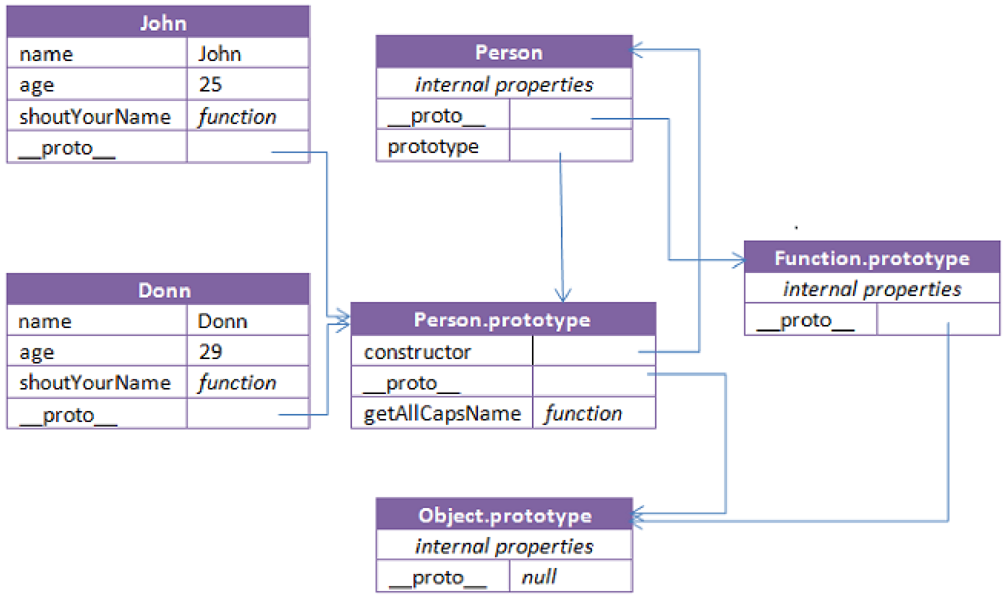
Persistence

Your app should dynamically persist the todos to localStorage. If the framework has capabilities for persisting data (i.e. Backbone.sync), use that, otherwise vanilla localStorage. If possible, use the keys id, title, completed for each item. Make sure to use this format for the localStorage name: todos-[framework]. Editing mode should not be persisted.

Routing

Routing is required for all frameworks. Use the built-in capabilities if supported, otherwise use the Flatiron Director routing library located in the /js folder. The following routes should be implemented: #/ (all - default), #/active and #/completed (#!/ is also allowed). When the route changes the todo list should be filtered on a model level and the selected class on the filter links should be toggled. When an item is updated while in a filtered state, it should be updated accordingly. E.g. if the filter is Active and the item is checked, it should be hidden. Make sure the active filter is persisted on reload.

JavaScript Basics



(Courtesy: http://viralpatel.net/blogs/javascript-objects-functions)

This chapter is courtesy of jQuery Learning Center:

Getting Started

Anatomy of a Web Page

Before diving into JavaScript, it helps to understand how it aligns with the other web technologies.

HTML is for Content

HTML is a markup language used to define and describe content. Whether it be a blog post, a search engine result, or an e-commerce site, the core content of a web page is written in HTML. A semantic markup, HTML is used to describe content in universal terms (headers, paragraphs, images, etc.)

CSS is for Presentation

CSS is a supplemental language that applies style to HTML documents. CSS is all about making content look better by defining fonts, colors, and other visual aesthetics. The power of CSS comes from the fact that styling is not intermingled with content. This means you can apply different styles to the same piece of content, which is critical when building responsive websites that look good across a range of devices.

JavaScript is for Interactivity

In the browser, JavaScript adds interactivity and behavior to HTML content. Without JavaScript, web pages would be static and boring. JavaScript helps bring a web page to life.

Look at this simple HTML page that includes CSS and JavaScript to see how it all fits together:

```
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8" />
<title>Hello World</title>
<!-- CSS for presentation. -->
<style>
h1 { font-size: 14px; color: hotpink; }
button { color: red; }
</style>
</head>
<body>
<h1>Hello World</h1>
<button>Click Me!</button>
<!-- JavaScript for interactivity. -->
<script>
// Get a handle on the first button element in the document.
var button = document.querySelector("button");
// If a user clicks on it, say hello!
button.addEventListener("click", function (ev) {
    alert("Hello");
}, false);
</script>
```

```
</body>
</html>
```

In the example above, HTML is used to describe the content. The "Hello World" text is described as a heading with the h1 tag and "Click Me!" is described as a button with the button tag. The style block contains CSS that changes the font size and color of the header text. The script block contains JavaScript that adds interactivity to the button. When a user clicks on the button, an alert message will appear that says "Hello!"

A Scripting Language for the Web

JavaScript was originally designed to add interactivity to web pages, not to be a general programming language, which makes it a scripting language. Scripting languages are regarded to be more productive than general languages because they are optimized for their specific domain (in this case, the web browser). However, recent advancements have brought JavaScript to the server-side (via Node.js) so it can now be used in place of languages like PHP, Ruby, or ASP. This guide will focus exclusively on JavaScript running in the browser with jQuery.

The name "JavaScript" is a bit misleading. Despite the similarity in name, JavaScript has no relationship with Java, a general purpose language. JavaScript is based on an open web standard called ECMAScript. Standards-based languages are not controlled by any one entity or corporation – instead, developers work together to define the language, which is why JavaScript will run in every web browser regardless of the operating system or device.

What You Need to Get Started with JavaScript and jQuery

```
Web Browser
Text Editor
Developer Tools (optional)
```

One of JavaScript's greatest strengths is its simplicity. It can be written and run on any operating system, and the only requirements are a web browser and a text editor. There are also numerous tools that can make JavaScript development more productive, but they are completely optional.

Developer Tools

Commonly referred to as "developer tools," many browsers ship with built-in features that provide better insight into JavaScript and jQuery while they run in the browser. Although they aren't required, you may find developer tools helpful when it comes to debugging errors in your code.

Running Code

External

The first and recommended option is to write code in an external file (with a .js extension), which can then be included on our web page using an HTML script tag and pointing the src attribute to the file's location. Having JavaScript in a separate file will reduce code duplication if you want to reuse it on other pages. It will also allow the browser to cache the file on the remote client's computer, decreasing page load time.

```
<!-- Code is written in a .js file, included via the script tag src attribute. -->
<script src="/path/to/example.js"></script>
```

Inline

The second option is to inline the code directly on the web page. This is also achieved using HTML script tags, but instead of pointing the src attribute to a file, the code is placed between the tags. While there are use cases for this option, the majority of the time it is best to keep our code in an external file as described above.

```
<!-- Embed code directly on a web page using script tags. -->
<script>
alert( "Hello World!" );
</script>
```

Attributes

The last option is to use the event handler attributes of HTML elements. This method is strongly discouraged:

```
<!-- Inline code directly on HTML elements being clicked. -->
<a href="javascript:alert( 'Hello World' );">Click Me!</a>
<button onClick="alert( 'Good Bye World' );">Click Me Too!</button>
```

Placement

Placement of the previous two options is important and can vary depending on the situation. If you are including JavaScript that doesn't access the elements on the page, you can safely place the script before the closing HTML head tag. However, if the code will interact with the elements on the page, you have to make sure those elements exist at the time the script is executed. This common pitfall can be seen in the example below. The script for finding the element with the ID hello-world will be executed before the element is defined in the document.

```
<!doctype html>
<html>
<head>
<script>
// Attempting to access an element too early will have unexpected results.
var title = document.getElementById( "hello-world" );
console.log( title );
</script>
</head>
<body>
<h1 id="hello-world">Hello World</h1>
</body>
</html>
```

It is a common pattern to move scripts to the bottom of the page, prior to the closing HTML body tag. This will guarantee that elements are defined when the script is executed:

```
<!doctype html>
<html>
<head></head>
<body>
<h1 id="hello-world">Hello World</h1>
<script>
// Moving the script to the bottom of the page will make sure the element exists.
var title = document.getElementById( "hello-world" );
console.log( title );
</script>
</body>
</html>
```

Syntax Basics

Comments

JavaScript has support for single- and multi-line comments. Comments are ignored by the JavaScript engine and therefore have no side-effects on the outcome of the program. Use comments to document the code for other developers. Libraries like JSDoc are available to help generate project documentation pages based on commenting conventions.

```
// Single- and multi-line comments.
// This is an example of a single-line comment.
/*
* this is an example
* of a
* multi-line
* comment.
*/
```

Whitespace

Whitespace is also ignored in JavaScript. There are many tools that will strip out all the whitespace in a program, reducing the overall file size and improving network latency. Given the availability of tools like these, whitespace should be leveraged to make the code as readable as possible.

```
// Whitespace is insignificant.
var hello = "Hello";
var world = "World!";

// Semantic whitespace promotes readability.
// Readable code is good!
var foo = function () {
    for (var i = 0; i < 10; i++) {
        alert(i);
    }
};
foo();
// This is much harder to read!
```

```
var foo = function () {
    for (var i = 0; i < 10; i++) {
        alert(i);
    }
};
foo();
```

Reserved Words

There are a handful of reserved words that can't be used when declaring user-defined variables and functions. Some of these reserved words are currently implemented, some are saved for future use, and others are reserved for historical reasons. A list of reserved words can be found here, and in-depth explanations for each can be found on the MDN JavaScript Reference site.

Identifiers

Identifiers are used to give variables and functions a unique name so they can subsequently be referred to by that name. The name of an identifier must follow a few rules:

```
Cannot be a reserved word.
Can only be composed of letters, numbers, dollar signs, and underscores.
The first character cannot be a number.
```

It's a best practice to name identifiers in a way that will make sense to you and other developers later on.

```
// Valid identifier names.
var myAwesomeVariable = "a";
var myAwesomeVariable2 = "b";
var my_awesome_variable = "c";
var $my_AwesomeVariable = "d";
var _my_awesome_variable_$ = "e";
```

Variable Definition

Variables can be defined using multiple var statements, or in a single combined var statement.

```
// This works:
var test = 1;
var test2 = function() { ... };
var test3 = test2( test );
// And so does this:
var test4 = 1,
test5 = function() { ... },
test6 = test2( test );
```

Variables can be declared without assigning them a value. The value of a variable declared without a value is undefined.

```
var x;
x === undefined; // true
```

Types

Types in JavaScript fall into two categories: primitives and objects. Primitive types include:

- String
- Number
- Boolean
- null
- undefined

String

Strings are text wrapped in single or double quotation marks. It is best practice to consistently use one or the other. There may be times when the string contains quotation marks that collide with the ones used to create the string. In this case, either escape the characters using a \ backslash or use different quotes around the string.

```
// Strings can be created with double or single quotes.
var a = "I am a string";
var b = 'So am I!';
alert( a );
alert( b );

// Sometimes a string may contain quotation marks.
var statement1 = 'He said "JavaScript is awesome!"';
var statement2 = "He said \"JavaScript is awesome!\"";
```

Number

Number types are any positive or negative numeric value. There is no distinction between integer and floating point values.

```
// Numbers are any whole or floating point integer.
var num1 = 100;
var num2 = 100.10;
var num3 = 0.10;
var statement2 = "He said \"JavaScript is awesome!\"";
```

Boolean

Boolean types are either true or false.

```
// Boolean values.
var okay = true;
var fail = false;
```

null and undefined

null and undefined are special types in JavaScript. Null types are values that represent the absence of a value, similar to many other programming languages. Undefined types represent a state in which no value has been assigned at all. This type is created in two ways: by using the undefined keyword or by not defining a value at all.

```
// Define a null value.
var foo = null;
// Two ways to achieve an undefined value.
var bar1 = undefined;
var bar2;
```

Objects

Everything else in JavaScript is considered an object. While there are numerous built-in objects, this chapter will cover:

- Object
- Array
- Function

The simplest way to create an object is either through the Object constructor or the shorthand syntax known as object literal. These simple objects are unordered key/value pairs. The key is formally known as a property and the value can be any valid JavaScript type, even another object. To create or access a property on an object, we use what is known as "dot notation" or "bracket notation."

```
// Creating an object with the constructor:
var person1 = new Object;
person1.firstName = "John";
person1.lastName = "Doe";
alert(person1.firstName + " " + person1.lastName);
// Creating an object with the object literal syntax:
var person2 = {
    firstName : "Jane",
    lastName : "Doe"
};
alert(person2.firstName + " " + person2.lastName);

// As mentioned, objects can also have objects as a property.
var people = {};
people["person1"] = person1;
people["person2"] = person2;
alert(people["person1"].firstName);
alert(people["person2"].firstName);
```

If a property is accessed that has not been defined, it will return a type of undefined.

```
// Properties that have not been created are undefined.
var person = { name: "John Doe" };
alert( person.email ); // undefined
```

Objects contain one or more key-value pairs. The key portion can be any string. The value portion can be any type of value: a number, a string, an array, a function, or even another object. When one of these values is a function, it’s called a method of the object. Otherwise, they are called properties.

As it turns out, nearly everything in JavaScript is an object – arrays, functions, numbers, even strings – and they all have properties and methods.

```
// Creating an object literal.
var myObject = {
    sayHello : function () {
        console.log("hello");
    },
    myName : "Rebecca"
};
myObject.sayHello(); // "hello"
console.log(myObject.myName); // "Rebecca"
```

When creating object literals, note that the key portion of each key-value pair can be written as any valid JavaScript identifier, a string (wrapped in quotes), or a number:

```
var myObject = {
    validIdentifier : 123,
    "some string" : 456,
    99999 : 789
};
```

Array

An array is a type of object that is ordered by the index of each item it contains. The index starts at zero and extends to however many items have been added, which is a property of the array known as the .length. Similar to a basic object, an array can be created with the Array constructor or the shorthand syntax known as array literal.

```
// Creating an array with the constructor:
var foo = new Array;
// Creating an array with the array literal syntax:
var bar = [];
```

There is an important distinction to be made between the two. Both an array constructor and an array literal can contain items to be added to the array upon creating it. However, if just a single numeric item is passed in, the array constructor will assume its length to be that value.

```
// The array literal returns a foo.length value of 1:
var foo = [ 100 ];
alert( foo[ 0 ] ); // 100
alert( foo.length ); // 1
// The array constructor returns a bar.length value of 100:
var bar = new Array( 100 );
alert( bar[ 0 ] ); // undefined
alert( bar.length ); // 100
```

An array can be manipulated through methods that are available on the instance of the array. Items in the array can be accessed using bracket notation with a given index. If the index does not exist or contains no value, the return type will be undefined.

A few common array methods are shown below:

```
// Using the push(), pop(), unshift() and shift() methods on an array.
var foo = [];
foo.push( "a" );
foo.push( "b" );
alert( foo[ 0 ] ); // a
alert( foo[ 1 ] ); // b
alert( foo.length ); // 2
foo.pop();
alert( foo[ 0 ] ); // a
alert( foo[ 1 ] ); // undefined
alert( foo.length ); // 1
foo.unshift( "z" );
alert( foo[ 0 ] ); // z
alert( foo[ 1 ] ); // a
alert( foo.length ); // 2
foo.shift();
alert( foo[ 0 ] ); // a
alert( foo[ 1 ] ); // undefined
alert( foo.length ); // 1
```

Operators

Basic operators allow you to manipulate values.

```
// Concatenation
var foo = "hello";
var bar = "world";
console.log( foo + " " + bar ); // "hello world"

// Multiplication and division
2 * 3;
2 / 3;

// Incrementing and decrementing
// The pre-increment operator increments the operand before any further processing.
var i = 1;
console.log( ++i ); // 2 - because i was incremented before evaluation
console.log( i ); // 2
// The post-increment operator increments the operand after processing it.
var i = 1;
console.log( i++ ); // 1 - because i was evaluated to 1 and _then_ incremented
console.log( i ); // 2 - incremented after using it
```

**

Operations on Numbers & Strings**

In JavaScript, numbers and strings will occasionally behave in unexpected ways.

```
// Addition vs. Concatenation
var foo = 1;
var bar = "2";
console.log( foo + bar ); // 12

// Coercing a string to act as a number.
var foo = 1;
var bar = "2";
console.log( foo + Number(bar) ); // 3
```

The Number constructor, when called as a function (as in the above example), will have the effect of casting its argument into a number. The unary plus operator also does the same thing:

```
// Forcing a string to act as a number (using the unary plus operator).
console.log( foo + +bar ); // 3
```

Logical Operators

Logical operators allow evaluation of a series of operands using AND (&&) and OR (||) operations.

```
// Logical AND and OR operators
var foo = 1;
var bar = 0;
var baz = 2;
// returns 1, which is true
foo || bar;
// returns 1, which is true
bar || foo;
// returns 0, which is false
foo && bar;
// returns 2, which is true
foo && baz;
// returns 1, which is true
baz && foo;
```

In the above example, the || operator returns the value of the first truthy operand, or in cases where neither operand is truthy, it returns the last operand. The && operator returns the value of the first false operand, or the value of the last operand if both operands are truthy.

You'll sometimes see developers use these logical operators for flow control instead of using if statements. For example:

```
// Do something with foo if foo is truthy.
foo && doSomething( foo );
// Set bar to baz if baz is truthy;
// otherwise, set it to the return value of createBar()
var bar = baz || createBar();
```

This style is quite elegant and pleasantly terse; that said, it can be really hard to read or use, especially for beginners. See the section on truthy and falsy things in the Conditional Code article for more about evaluating truthiness.

Comparison Operators

Comparison operators allow you to test whether values are equivalent or whether values are identical.

```
// Comparison operators
var foo = 1;
var bar = 0;
var baz = "1";
var bim = 2;
foo == bar; // false
foo != bar; // true
foo == baz; // true; but note that the types are different
foo === baz; // false
foo !== baz; // true
foo === parseInt( baz ); // true
foo > bim; // false
bim > baz; // true
foo <= baz; // true
```

Conditional Code

Sometimes a block of code should only be run under certain conditions. Flow control – via if and else blocks – lets you run code if certain conditions have been met.

```
// Flow control
var foo = true;
var bar = false;
if (bar) {
    // This code will never run.
    console.log("hello!");
}
if (bar) {
    // This code won't run.
} else {
    if (foo) {
        // This code will run.
    } else {
        // This code would run if foo and bar were both false.
    }
}
```

While curly braces aren't strictly required around single-line if statements, using them consistently, even when they aren't strictly required, makes for vastly more readable code.

Be mindful not to define functions with the same name multiple times within separate if/else blocks, as doing so may not have the expected result.

Conditional Code

Sometimes a block of code should only be run under certain conditions. Flow control – via if and else blocks – lets you run code if certain conditions have been met.

```
// Flow control
var foo = true;
var bar = false;
if (bar) {
    // This code will never run.
    console.log("hello!");
}
if (bar) {
    // This code won't run.
} else {
    if (foo) {
        // This code will run.
    } else {
        // This code would run if foo and bar were both false.
    }
}
```

While curly braces aren't strictly required around single-line if statements, using them consistently, even when they aren't strictly required, makes for vastly more readable code.

Be mindful not to define functions with the same name multiple times within separate if/else blocks, as doing so may not have the expected result.

Truthy and Falsy Things

In order to use flow control successfully, it's important to understand which kinds of values are "truthy" and which kinds of values are "falsy." Sometimes, values that seem like they should evaluate one way actually evaluate another.

```
// Values that evaluate to false:
false
"" // An empty string.
NaN // JavaScript's "not-a-number" variable.
null
undefined // Be careful -- undefined can be redefined!
0 // The number zero.
```

```
// Everything else evaluates to true, some examples:
"0"
"any string"
[] // An empty array.
{} // An empty object.
1 // Any non-zero number.
```

Conditional Variable Assignment with the Ternary Operator

Sometimes a variable should be set depending on some condition. An if/else statement works, but in many cases the ternary operator is more convenient. The ternary operator tests a condition; if the condition is true, it returns a certain value, otherwise it returns a different value.

The ternary operator:

```
// Set foo to 1 if bar is true; otherwise, set foo to 0:
var foo = bar ? 1 : 0;
```

While the ternary operator can be used without assigning the return value to a variable, this is generally discouraged.

Switch Statements

Rather than using a series of if/else blocks, sometimes it can be useful to use a switch statement instead. switch statements look at the value of a variable or expression, and run different blocks of code depending on the value.

```
// A switch statement
switch (foo) {
case "bar":
    alert("the value was bar -- yay!");
    break;
case "baz":
    alert("boo baz :(");
    break;
default:
    alert("everything else is just ok");
}
```

Switch statements have somewhat fallen out of favor in JavaScript, because often the same behavior can be accomplished by creating an object that has more potential for reuse or testing. For example:

```
var stuffToDo = {
    "bar" : function () {
        alert("the value was bar -- yay!");
    },
    "baz" : function () {
        alert("boo baz :(");
    },
    "default" : function () {
        alert("everything else is just ok");
    }
};
// Check if the property exists in the object.
if (stuffToDo[foo]) {
    // This code won't run.
    stuffToDo[foo]();
} else {
    // This code will run.
    stuffToDo["default"]();
}
```

Loops

Loops let a block of code run a certain number of times:

```
for (var i = 0; i < 5; i++) {
    // Logs "try 0", "try 1", ..., "try 4".
    console.log("try " + i);
}
```

> Note that in loops, the variable i is not "scoped" to the loop block even though the keyword var is used before the variable name. Scope is covered in more depth in the Scope section.

The for Loop

A for loop is made up of four statements and has the following structure:

```
for ([initialization]; [conditional]; [iteration]) {
    [loopBody]
}
```

The initialization statement is executed only once, before the loop starts. It gives you an opportunity to prepare or declare any variables.

The conditional statement is executed before each iteration, and its return value decides whether the loop is to continue. If the conditional statement evaluates to a falsy value, then the loop stops.

The iteration statement is executed at the end of each iteration and gives you an opportunity to change the state of important variables. Typically, this will involve incrementing or decrementing a counter and thus bringing the loop closer to its end.

The loopBody statement is what runs on every iteration. It can contain anything. Typically, there will be multiple statements that need to be executed, and should be wrapped in a block ({...}).

Here's a typical for loop:

```
for (var i = 0, limit = 100; i < limit; i++) {
    // This block will be executed 100 times.
    console.log("Currently at " + i);
    // Note: The last log will be "Currently at 99".
}
```

The while loop

A while loop is similar to an if statement, except that its body will keep executing until the condition evaluates to a falsy value.

```
while ([conditional]) {
    [loopBody]
}
```

Here's a typical while loop:

```
var i = 0;
while (i < 100) {
    // This block will be executed 100 times.
    console.log("Currently at " + i);
    i++; // Increment i
}
```

Notice that the counter is incrementing within the loop's body. It's possible to combine the conditional and incrementer, like so:

```
var i = -1;
while (++i < 100) {
    // This block will be executed 100 times.
    console.log("Currently at " + i);
}
```

Notice that the counter starts at -1 and uses the prefix incrementer (++i).

The do-while Loop

This is almost exactly the same as the while loop, except for the fact that the loop's body is executed at least once before the condition is tested.

```
do {
    [loopBody]
} while ([conditional])
```

Here's a do-while loop:

```
do {
    // Even though the condition evaluates to false
    // this loop's body will still execute once.
    alert("Hi there!");
} while (false);
```

These types of loops are quite rare since only few situations require a loop that blindly executes at least once. Regardless, it's good to be aware of it.

Breaking and Continuing

Usually, a loop's termination will result from the conditional statement not evaluating to a truthy value, but it is possible to stop a loop in its tracks from within the loop's body with the break statement:

```
// Stopping a loop
for (var i = 0; i < 10; i++) {
    if (something) {
        break;
    }
}
```

You may also want to continue the loop without executing more of the loop's body. This is done using the continue statement:

```
// Skipping to the next iteration of a loop
for (var i = 0; i < 10; i++) {
    if (something) {
        continue;
    }
    // The following statement will only be executed
    // if the conditional "something" has not been met
    console.log("I have been reached");
}
```

Reserved Words

- break
- case
- catch
- class
- const

- continue
- debugger
- default
- delete
- do
- else
- enum
- export
- extends
- false
- finally
- for
- function
- if
- implements
- import
- in
- instanceof
- interface
- let
- new
- null
- package
- private
- protected
- public
- return
- static
- super
- switch
- this
- throw
- true

- try
- typeof
- var
- void
- while
- with
- yield

Arrays

Arrays are zero-indexed, ordered lists of values. They are a handy way to store a set of related items of the same type (such as strings), though in reality, an array can include multiple types of items, including other arrays.

To create an array, either use the object constructor or the literal declaration, by assigning the variable a list of values after the declaration.

```
// A simple array with constructor.
var myArray1 = new Array( "hello", "world" );
// Literal declaration, the preferred way.
var myArray2 = [ "hello", "world" ];
```

The literal declaration is generally preferred. See the Google Coding Guidelines for more information.

If the values are unknown, it is also possible to declare an empty array, and add elements either through functions or through accessing by index:

```
// Creating empty arrays and adding values
var myArray = [];
// Adds "hello" on index 0
myArray.push( "hello" );
// Adds "world" on index 1
myArray.push( "world" );
// Adds "!" on index 2
myArray[ 2 ] = "!";
```

.push() is a function that adds an element on the end of the array and expands the array respectively. You also can directly add items by index. Missing indices will be filled with undefined.

```
// Leaving indices
var myArray = [];
myArray[ 0 ] = "hello";
myArray[ 1 ] = "world";
myArray[ 3 ] = "!";
console.log( myArray ); // [ "hello", "world", undefined, "!" ];
```

If the size of the array is unknown, .push() is far more safe. You can both access and assign values to array items with the index.

```
// Accessing array items by index
var myArray = [ "hello", "world", "!" ];
console.log( myArray[ 2 ] ); // "!"
```

Array Methods and Properties

.length

The .length property is used to determine the amount of items in an array.

```
// Length of an array
var myArray = [ "hello", "world", "!" ];
console.log( myArray.length ); // 3
```

You will need the .length property for looping through an array:

```
// For loops and arrays - a classic
var myArray = ["hello", "world", "!"];
for (var i = 0; i < myArray.length; i = i + 1) {
    console.log(myArray[i]);
}
```

.concat()

Concatenate two or more arrays with .concat():

```
var myArray = [ 2, 3, 4 ];
var myOtherArray = [ 5, 6, 7 ];
var wholeArray = myArray.concat( myOtherArray ); // [ 2, 3, 4, 5, 6, 7 ]
```

.join()

.join() creates a string representation of an array by joining all of its elements using a separator string. If no separator is supplied (i.e., .join() is called without arguments) the array will be joined using a comma.

```
// Joining elements
var myArray = [ "hello", "world", "!" ];
// The default separator is a comma.
console.log( myArray.join() ); // "hello,world,!"
// Any string can be used as separator...
console.log( myArray.join( " " ) ); // "hello world !";
console.log( myArray.join( "!!" ) ); // "hello!!world!!!";
// ...including an empty one.
console.log( myArray.join( "" ) ); // "helloworld!"
```

.pop()

.pop() removes the last element of an array. It is the opposite method of .push():

```
// Pushing and popping
var myArray = [];
myArray.push( 0 ); // [ 0 ]
myArray.push( 2 ); // [ 0 , 2 ]
myArray.push( 7 ); // [ 0 , 2 , 7 ]
myArray.pop(); // [ 0 , 2 ]
```

.reverse()

As the name suggests, the elements of the array are in reverse order after calling this method:

```
var myArray = [ "world" , "hello" ];
myArray.reverse(); // [ "hello", "world" ]
```

.shift()

Removes the first element of an array. With .push() and .shift(), you can recreate the method of a queue:

```
// Queue with shift() and push()
var myArray = [];
myArray.push( 0 ); // [ 0 ]
myArray.push( 2 ); // [ 0 , 2 ]
myArray.push( 7 ); // [ 0 , 2 , 7 ]
myArray.shift(); // [ 2 , 7 ]
```

.slice()

Extracts a part of the array and returns that part in a new array. This method takes one parameter, which is the starting index:

```
// Slicing
var myArray = [ 1, 2, 3, 4, 5, 6, 7, 8 ];
var newArray = myArray.slice( 3 );
console.log( myArray ); // [ 1, 2, 3, 4, 5, 6, 7, 8 ]
console.log( newArray ); // [ 4, 5, 6, 7, 8 ]
```

.splice()

Removes a certain amount of elements and adds new ones at the given index. It takes at least three parameters:

```
myArray.splice( index, length, values, ... );
```

- Index – The starting index.
- Length – The number of elements to remove.
- Values – The values to be inserted at the index position.

For example:

```
var myArray = [ 0, 7, 8, 5 ];
myArray.splice( 1, 2, 1, 2, 3, 4 );
console.log( myArray ); // [ 0, 1, 2, 3, 4, 5 ]
```

.sort()

Sorts an array. It takes one parameter, which is a comparing function. If this function is not given, the array is sorted ascending:

```
// Sorting without comparing function.
var myArray = [ 3, 4, 6, 1 ];
myArray.sort(); // 1, 3, 4, 6
```

```
// Sorting with comparing function.
function descending(a, b) {
    return b - a;
}
var myArray = [3, 4, 6, 1];
myArray.sort(descending); // [ 6, 4, 3, 1 ]
```

The return value of descending (for this example) is important. If the return value is less than zero, the index of a is before b, and if it is greater than zero it's vice-versa. If the return value is zero, the elements' index is equal.

.unshift()

Inserts an element at the first position of the array:

```
var myArray = [];
myArray.unshift( 0 ); // [ 0 ]
myArray.unshift( 2 ); // [ 2 , 0 ]
myArray.unshift( 7 ); // [ 7 , 2 , 0 ]
```

.forEach()

In modern browsers it is possible to traverse through arrays with a .forEach() method, where you pass a function that is called for each element in the array.

The function takes up to three arguments:

- Element – The element itself.
- Index – The index of this element in the array.
- Array – The array itself.

All of these are optional, but you will need at least the Element parameter in most cases.

```
// Native .forEach()
function printElement( elem ) {
console.log( elem );
}
function printElementAndIndex( elem, index ) {
console.log( "Index " + index + ": " + elem );
}
function negateElement( elem, index, array ) {
array[ index ] = -elem;
}
myArray = [ 1, 2, 3, 4, 5 ];
```

```
// Prints all elements to the console
myArray.forEach( printElement );
// Prints "Index 0: 1", "Index 1: 2", "Index 2: 3", ...
myArray.forEach( printElementAndIndex );
// myArray is now [ -1, -2, -3, -4, -5 ]
myArray.forEach( negateElement );
```

Functions

Functions contain blocks of code that need to be executed repeatedly. Functions can take zero or more arguments, and can optionally return a value.

Functions can be created in a variety of ways, two of which are shown below:

```
// Function declaration.
function foo() {
// Do something.
}
```

```
// Named function expression.
var foo = function() {
// Do something.
};
```

Using Functions

```
// A simple function.
var greet = function (person, greeting) {
    var text = greeting + ", " + person;
    console.log(text);
};
greet("Rebecca", "Hello"); // "Hello, Rebecca"
```

```
// A function that returns a value.
var greet = function (person, greeting) {
    var text = greeting + ", " + person;
    return text;
};
console.log(greet("Rebecca", "Hello")); // "Hello, Rebecca"
```

```
// A function that returns another function.
var greet = function (person, greeting) {
    var text = greeting + ", " + person;
    return function () {
        console.log(text);
    };
};
var greeting = greet("Rebecca", "Hello");
greeting(); // "Hello, Rebecca"
```

Immediately-Invoked Function Expression (IIFE)

A common pattern in JavaScript is the immediately-invoked function expression. This pattern creates a function expression and then immediately executes the function. This pattern is extremely useful for cases where you want to avoid polluting the global namespace with code – no variables declared inside of the function are visible outside of it.

```
// An immediately-invoked function expression.
(function () {
    var foo = "Hello world";
})();
console.log(foo); // undefined!
```

Functions as Arguments

In JavaScript, functions are "first-class citizens" – they can be assigned to variables or passed to other functions as arguments. Passing functions as arguments is an extremely common idiom in jQuery.

```
// Passing an anonymous function as an argument.
var myFn = function (fn) {
    var result = fn();
    console.log(result);
};
// Logs "hello world"
myFn(function () {
    return "hello world";
});
```

```
// Passing a named function as an argument
var myFn = function (fn) {
    var result = fn();
    console.log(result);
};
var myOtherFn = function () {
    return "hello world";
};
myFn(myOtherFn); // "hello world"
```

Testing Type

JavaScript offers a way to test the type of a variable. However, the result can be confusing – for example, the type of an array is "Object."

It's common practice to use the typeof operator when trying to determining the type of a specific value.

```
// Testing the type of various variables.
var myFunction = function() {
console.log( "hello" );
};
var myObject = {
foo: "bar"
};
var myArray = [ "a", "b", "c" ];
var myString = "hello";
var myNumber = 3;
var myRegExp = /(\w+)\s(\w+)/;
typeof myFunction; // "function"
typeof myObject; // "object"
typeof myArray; // "object" -- Careful!
typeof myString; // "string"
typeof myNumber; // "number"
typeof null; // "object" -- Careful!
typeof undefined; // "undefined"
typeof meh; // "undefined" -- undefined variable.
typeof myRegExp; // "function" or "object" depending on environment.
if ( myArray.push && myArray.slice && myArray.join ) {
// probably an array (this is called "duck typing")
}
if ( Object.prototype.toString.call( myArray ) === "[object Array]" ) {
// Definitely an array!
// This is widely considered as the most robust way
// to determine if a specific value is an Array.
}
```

The "this" keyword

In JavaScript, as in most object-oriented programming languages, this is a special keyword that is used in methods to refer to the object on which a method is being invoked. The value of this is determined using a simple series of steps:

1. If the function is invoked using Function.call() or Function.apply(), this will be set to the first argument passed to .call()/.apply(). If the first argument passed to .call()/.apply() is null or undefined, this will refer to the global object (which is the window object in web browsers).
2. If the function being invoked was created using Function.bind(), this will be the first argument that was passed to .bind() at the time the function was created.
3. If the function is being invoked as a method of an object, this will refer to that object.
4. Otherwise, the function is being invoked as a standalone function not attached to any object, and this will refer to the global object.

```
// A function invoked using Function.call()
var myObject = {
    sayHello : function () {
        console.log("Hi! My name is " + this.myName);
    },
    myName : "Rebecca"
};
var secondObject = {
    myName : "Colin"
};
myObject.sayHello(); // "Hi! My name is Rebecca"
myObject.sayHello.call(secondObject); // "Hi! My name is Colin"
```

```
// A function created using Function.bind()
var myName = "the global object";
var sayHello = function () {
    console.log("Hi! My name is " + this.myName);
};
var myObject = {
    myName : "Rebecca"
};
var myObjectHello = sayHello.bind(myObject);
sayHello(); // "Hi! My name is the global object"
myObjectHello(); // "Hi! My name is Rebecca"
```

```
// A function being attached to an object at runtime.
var myName = "the global object";
var sayHello = function () {
    console.log("Hi! My name is " + this.myName);
};
var myObject = {
    myName : "Rebecca"
};
var secondObject = {
    myName : "Colin"
};
myObject.sayHello = sayHello;
secondObject.sayHello = sayHello;
sayHello(); // "Hi! My name is the global object"
myObject.sayHello(); // "Hi! My name is Rebecca"
secondObject.sayHello(); // "Hi! My name is Colin"
```

When invoking a function deep within a long namespace, it is often tempting to reduce the amount of code you need to type by storing a reference to the actual function as a single, shorter variable. It is important not to do this with instance methods as this will cause the value of this within the function to change, leading to incorrect code operation. For instance:

```
var myNamespace = {
    myObject : {
        sayHello : function () {
            console.log("Hi! My name is " + this.myName);
        },
        myName : "Rebecca"
    }
};
var hello = myNamespace.myObject.sayHello;
hello(); // "Hi! My name is undefined"
```

You can, however, safely reduce everything up to the object on which the method is invoked:

```
var myNamespace = {
    myObject : {
        sayHello : function () {
            console.log("Hi! My name is " + this.myName);
        },
        myName : "Rebecca"
    }
};
var obj = myNamespace.myObject;
obj.sayHello(); // "Hi! My name is Rebecca"
```

Scope

"Scope" refers to the variables that are available to a piece of code at a given time. A lack of understanding of scope can lead to frustrating debugging experiences. The idea of scope is that it's where certain functions or variables are accessible from in our code, and the context in which they exist and are executed in.

There are two types of scopes in JavaScript: global and local. Let's talk about each of them in turn.

Global Scope

The first scope is Global Scope. This is very easy to define. If a variable or function is global, it can be accessed from anywhere within a program. In a browser, the global scope is the window object. If a variable declaration occurs outside of a function, then that variable exists on the global object. For example:

```
var x = 9;
```

Once that variable had been defined, it could be referenced as window.x, but because it exists on the global object we can simply refer to it as x.

Local Scope

JavaScript also creates a Local Scope inside each function body. For example:

```
function myFunc() {
    var x = 5;
}
console.log(x); // ReferenceError: x is not defined
```

Since x was initialized within .myFunc(), it is only accessible within .myFunc(), and we get a reference error if we try to access it outside of .myFunc().

A Word of Caution

If you declare a variable and forget to use the var keyword, that variable is automatically made global. So this code would work:

```
function myFunc() {
    x = 5;
}
console.log(x); // 5
```

This is a bad idea. Any variable that is global can have its value changed by any other parts of a program or any other script. This is undesirable, as it could lead to unforseen side effects.

Secondly, Immediately-Invoked Function Expressions provide a way to avoid global variables. You'll see many libraries such as jQuery often use these:

```
(function () {
    var jQuery = {
        /* All my methods go here. */
    };
    window.jQuery = jQuery;
})();
```

Wrapping everything in a function which is then immediately invoked means all the variables within that function are bound to the local scope. At the very end you can then expose all your methods by binding the jQuery object to the window, the global object. To read more about Immediately-Invoked Functions, check out Ben

Alman's Immediately-Invoked Function Expression article.

Because local scope works through functions, any functions defined within another have access to variables defined in the outer function:

```
function outer() {
    var x = 5;
    function inner() {
        console.log(x);
    }
    inner(); // 5
}
```

But the .outer() function doesn't have access to any variables declared within .inner():

```
function outer() {
    var x = 5;
    function inner() {
        console.log(x);
        var y = 10;
    }
    inner(); // 5
    console.log(y); // ReferenceError: y is not defined
}
```

Furthermore, variables that are declared inside a function without the var keyword are not local to the function – JavaScript will traverse the scope chain all the way up to the window scope to find where the variable was previously defined. If the variable wasn't previously defined, it will be defined in the global scope, which can have unexpected consequences.

```
// Functions have access to variables defined in the same scope.
var foo = "hello";
var sayHello = function () {
    console.log(foo);
};
sayHello(); // "hello"
console.log(foo); // "hello"
```

Variables with the same name can exist in different scopes with different values:

```
var foo = "world";
var sayHello = function () {
    var foo = "hello";
    console.log(foo);
};
sayHello(); // "hello"
console.log(foo); // "world"
```

When, within a function, you reference a variable defined in an outer scope, that function can see changes to the variable's value after the function is defined.

```
var myFunction = function () {
    var foo = "hello";
    var myFn = function () {
        console.log(foo);
    };
    foo = "world";
    return myFn;
};
var f = myFunction();
f(); // "world"
```

Here's a more complex example of scopes at play:

```
(function () {
    var baz = 1;
    var bim = function () {
        console.log(baz);
    };
    bar = function () {
        console.log(baz);
    };
})();
```

In this instance, running:

```
console.log( baz ); // baz is not defined outside of the function
```

Gives us a ReferenceError. baz was only defined within the function, and was never exposed to the global scope.

```
bar(); //  1
```

.bar() may have been defined within the anonymous function, but it was defined without the var keyword, which means it wasn't bound to the local scope and was instead created globally. Furthermore, it has access to the baz variable because .bar() was defined within the same scope as baz. This means it has access to it, even though other code outside of the function does not.

```
bim(); // ReferenceError: bim is not defined
```

.bim() was only defined within the function, so it does not exist on the global object as it was defined locally.

Closures

Closures are an extension of the concept of scope. With closures, functions have access to variables that were available in the scope where the function was created. If that seems confusing, don’t worry: closures are generally best understood by example.

As shown in the Scope section, functions have access to changing variable values. The same sort of behavior exists with functions defined within loops – the function "sees" the change in the variable's value even after the function is defined, resulting in each function referencing the last value stored in the variable.

```
// Each function executed within the loop will reference
// the last value stored in i (5).
// This won't behave as we want it to - every 100 milliseconds, 5 will alert
```

```
for (var i = 0; i < 5; i++) {
    setTimeout(function () {
        alert(i);
    }, i * 100);
}
```

Closures can be used to prevent this by creating a unique scope for each iteration – storing each unique value of the variable within its scope.

```
// Using a closure to create a new private scope
// fix: “close” the value of i inside createFunction, so it won't change
var createFunction = function (i) {
    return function () {
        alert(i);
    };
};
for (var i = 0; i < 5; i++) {
    setTimeout(createFunction(i), i * 100);
}
```

Closures can also be used to resolve issues with the this keyword, which is unique to each scope:

```
// Using a closure to access inner and outer object instances simultaneously.
var outerObj = {
    myName : "outer",
    outerFunction : function () {
        // Provide a reference to outerObj through innerFunction's closure
        var self = this;
        var innerObj = {
            myName : "inner",
            innerFunction : function () {
                console.log(self.myName, this.myName); // "outer inner"
            }
        };
        innerObj.innerFunction();
        console.log(this.myName); // "outer"
    }
};
outerObj.outerFunction();
```

Function.bind

Closures can be particularly useful when dealing with callbacks. However, it is often better to use Function.bind, which will avoid any overhead associated with scope traversal.

Function.bind is used to create a new function. When called, the new function then calls itself in the context of the supplied this value, using a given set of arguments that will precede any arguments provided when the new function was initially called.

As .bind() is a recent addition to ECMAScript 5, it may not be present in all browsers, which is something to be wary of when deciding whether to use it. However, it's possible to work around support by using this shim from MDN:

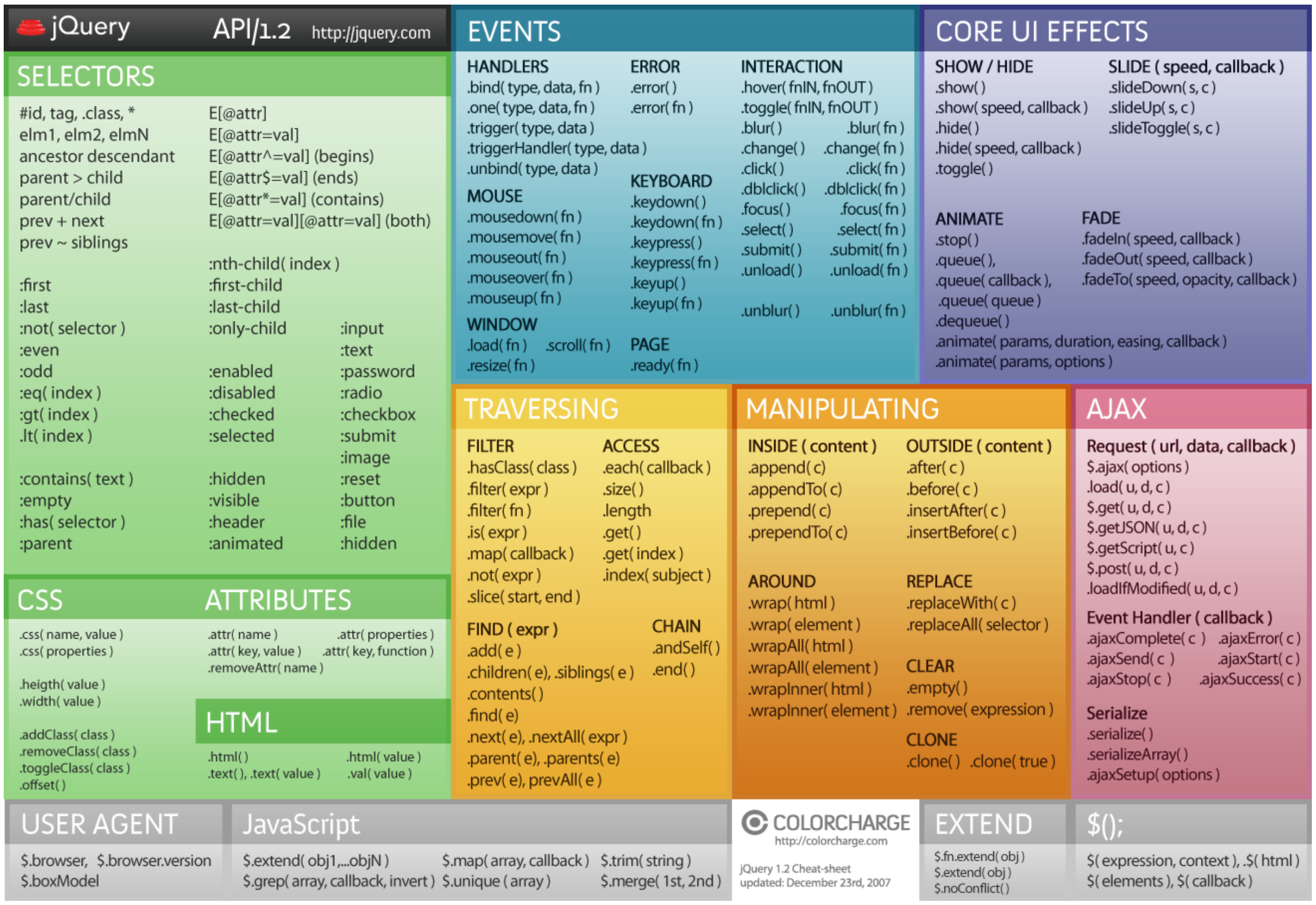
```
// Shim from MDN
```

```
if (!Function.prototype.bind) {
    Function.prototype.bind = function (oThis) {
        if (typeof this !== "function") {
            // closest thing possible to the ECMAScript 5 internal
            // IsCallable function
            throw new TypeError("Function.prototype.bind - what is trying to be bound is not callable
        }
        var fSlice = Array.prototype.slice,
        aArgs = fSlice.call(arguments, 1),
        fToBind = this,
        fNOP = function () {},
        fBound = function () {
            return fToBind.apply(this instanceof fNOP
                 ? this
                 : oThis || window,
                aArgs.concat(fSlice.call(arguments)));
        };
        fNOP.prototype = this.prototype;
        fBound.prototype = new fNOP();
        return fBound;
    };
}
```

One of the simplest uses of .bind() is making a function that is called with a particular value for this, regardless of how it's called. A common mistake developers make is attempting to extract a method from an object, then later calling that method and expecting it to the use the origin object as its this. However, this can be solved by creating a bound function using the original object as demonstrated below:

```
// Let's manipulate "this" with a basic example.
var user = "johnsmith";
var module = {
    getUser : function () {
        return this.user;
    },
    user : "janedoe"
};
// module.getUser() is called where "module" is "this"
// and "module.user" is returned.
// janedoe
module.getUser();
// let's now store a reference in the global version of "this"
var getUser = module.getUser;
// getUser() called, "this" is global, "user" is returned
// johnsmith
getUser();
// store a ref with "module" bound as "this"
var boundGetUser = getUser.bind(module);
// boundGetUser() called, "module" is "this" again, "module.user" returned.
// janedoe
boundGetUser();
```

Learn jQuery


jQuery API/1.2 http://jquery.com

SELECTORS

#id, tag, .class, *
elm1, elm2, elmN
ancestor descendant
parent > child
parent/child
prev + next
prev ~ siblings

E[@attr]
E[@attr=val]
E[@attr^=val] (begins)
E[@attr$=val] (ends)
E[@attr*=val] (contains)
E[@attr=val][@attr=val] (both)

:nth-child(index)
:first-child
:last-child
:only-child

:first
:last
:not(selector)
:even
:odd
:eq(index)
:gt(index)
:lt(index)

:enabled
:disabled
:checked
:selected

:input
:text
:password
:radio
:checkbox
:submit
:image
:reset
:button
:file
:hidden

:contains(text)
:empty
:has(selector)
:parent

:hidden
:visible
:header
:animated

CSS

.css(name, value)
.css(properties)

.heigth(value)
.width(value)

.addClass(class)
.removeClass(class)
.toggleClass(class)
.offset()

ATTRIBUTES

.attr(name)
.attr(key, value)
.removeAttr(name)
.attr(properties)
.attr(key, function)

HTML

.html()
.text(), .text(value)
.html(value)
.val(value)

EVENTS

HANDLERS
.bind(type, data, fn)
.one(type, data, fn)
.trigger(type, data)
.triggerHandler(type, data)
.unbind(type, data)

ERROR
.error()
.error(fn)

INTERACTION
.hover(fnIN, fnOUT)
.toggle(fnIN, fnOUT)
.blur() .blur(fn)
.change() .change(fn)
.click() .click(fn)
.dblclick() .dblclick(fn)
.focus() .focus(fn)
.select() .select(fn)
.submit() .submit(fn)
.unload() .unload(fn)
.unblur() .unblur(fn)

MOUSE
.mousedown(fn)
.mousemove(fn)
.mouseout(fn)
.mouseover(fn)
.mouseup(fn)

KEYBOARD
.keydown()
.keydown(fn)
.keypress()
.keypress(fn)
.keyup()
.keyup(fn)

WINDOW
.load(fn) .scroll(fn)
.resize(fn)

PAGE
.ready(fn)

CORE UI EFFECTS

SHOW / HIDE
.show()
.show(speed, callback)
.hide()
.hide(speed, callback)
.toggle()

SLIDE (speed, callback)
.slideDown(s, c)
.slideUp(s, c)
.slideToggle(s, c)

ANIMATE
.stop()
.queue(),
.queue(callback),
.queue(queue)
.dequeue()
.animate(params, duration, easing, callback)
.animate(params, options)

FADE
.fadeIn(speed, callback)
.fadeOut(speed, callback)
.fadeTo(speed, opacity, callback)

TRAVERSING

FILTER
.hasClass(class)
.filter(expr)
.filter(fn)
.is(expr)
.map(callback)
.not(expr)
.slice(start, end)

ACCESS
.each(callback)
.size()
.length
.get()
.get(index)
.index(subject)

FIND (expr)
.add(e)
.children(e), .siblings(e)
.contents()
.find(e)
.next(e), .nextAll(expr)
.parent(e), .parents(e)
.prev(e), prevAll(e)

CHAIN
.andSelf()
.end()

MANIPULATING

INSIDE (content)
.append(c)
.appendTo(c)
.prepend(c)
.prependTo(c)

OUTSIDE (content)
.after(c)
.before(c)
.insertAfter(c)
.insertBefore(c)

AROUND
.wrap(html)
.wrap(element)
.wrapAll(html)
.wrapAll(element)
.wrapInner(html)
.wrapInner(element)

REPLACE
.replaceWith(c)
.replaceAll(selector)

CLEAR
.empty()
.remove(expression)

CLONE
.clone() .clone(true)

AJAX

Request (url, data, callback)
$.ajax(options)
.load(u, d, c)
$.get(u, d, c)
$.getJSON(u, d, c)
$.getScript(u, c)
$.post(u, d, c)
.loadIfModified(u, d, c)

Event Handler (callback)
.ajaxComplete(c) .ajaxError(c)
.ajaxSend(c) .ajaxStart(c)
.ajaxStop(c) .ajaxSuccess(c)

Serialize
.serialize()
.serializeArray()
.ajaxSetup(options)

USER AGENT
$.browser, $.browser.version
$.boxModel

JavaScript
$.extend(obj1,...objN) $.map(array, callback) $.trim(string)
$.grep(array, callback, invert) $.unique (array) $.merge(1st, 2nd)

COLORCHARGE
http://colorcharge.com
jQuery 1.2 Cheat-sheet
updated: December 23rd, 2007

EXTEND
$.fn.extend(obj)
$.extend(obj)
$.noConflict()

$();
$(expression, context), .$(html)
$(elements), $(callback)

This chapter is courtesy of The jQuery Foundation:

How jQuery Works

This is a basic tutorial, designed to help you get started using jQuery. If you don't have a test page setup yet, start by creating the following HTML page:

```
<!doctype html>
<html>
<head>
<meta charset="utf-8" />
<title>Demo</title>
</head>
<body>
<a href="http://jquery.com/">jQuery</a>
<script src="jquery.js"></script>
<script>
// Your code goes here.
</script>
</body>
</html>
```

The src attribute in the script element must point to a copy of jQuery. Download a copy of jQuery from http://jquery.com/download/ or from Google CDN (https://developers.google.com/speed/libraries/) and store the jquery.js file in the same directory as your HTML file.

Launching Code on Document Ready

To ensure that their code runs after the browser finishes loading the document, many JavaScript programmers wrap their code in an onload function:

```
window.onload = function () {
    alert("welcome");
}
```

Unfortunately, the code doesn't run until all images are finished downloading, including banner ads. To run code as soon as the document is ready to be manipulated, jQuery has a statement known as the ready event:

```
$( document ).ready(function() {
// Your code here.
});
```

For example, inside the ready event, you can add a click handler to the link:

```
$(document).ready(function () {
    $("a").click(function (event) {
        alert("Thanks for visiting!");
    });
});
```

Save your HTML file and reload the test page in your browser. Clicking the link should now first display an alert pop-up, then continue with the default behavior of navigating to http://jquery.com.

For click and most other events, you can prevent the default behavior by calling event.preventDefault() in the event handler:

```
$(document).ready(function () {
    $("a").click(function (event) {
        alert("As you can see, the link no longer took you to jquery.com");
        event.preventDefault();
    });
});
```

Complete Example

The following example illustrates the click handling code discussed above, embedded directly in the HTML body.

> Note that in practice, it is usually better to place your code in a separate JS file and load it on the page with a script element's src attribute.

```
<!doctype html>
<html>
<head>
<meta charset="utf-8" />
<title>Demo</title>
</head>
```

```
<body>
<a href="http://jquery.com/">jQuery</a>
<script src="jquery.js"></script>
<script>
$(document).ready(function () {
    $("a").click(function (event) {
        alert("The link will no longer take you to jquery.com");
        event.preventDefault();
    });
});
</script>
</body>
</html>
```

Adding and Removing an HTML Class

Important: You must place the remaining jQuery examples inside the ready event so that your code executes when the document is ready to be worked on.

Another common task is adding or removing a class.

First, add some style information into the head of the document, like this:

```
<style>
a.test {
font-weight: bold;
}
</style>
```

Next, add the .addClass() call to the script:

```
$( "a" ).addClass( "test" );
```

All "a" elements are now bold.

To remove an existing class, use .removeClass():

```
$( "a" ).removeClass( "test" );
```

Special Effects

jQuery also provides some handy effects to help you make your web sites stand out. For example, if you create a click handler of:

```
$("a").click(function (event) {
    event.preventDefault();
    $(this).hide("slow");
});
```

Then the link slowly disappears when clicked.

Callbacks and Functions

Unlike many other programming languages, JavaScript enables you to freely pass functions around to be executed at a later time. A callback is a function that is passed as an argument to another function and is executed after its parent function has completed. Callbacks are special because they patiently wait to execute until their parent finishes. Meanwhile, the browser can be executing other functions or doing all sorts of other work.

To use callbacks, it is important to know how to pass them into their parent function.

Callback without Arguments

If a callback has no arguments, you can pass it in like this:

```
$.get( "myhtmlpage.html", myCallBack );
```

When $.get() finishes getting the page myhtmlpage.html, it executes the myCallBack() function.

> Note: The second parameter here is simply the function name (but not as a string, and without parentheses).

Callback with Arguments

Executing callbacks with arguments can be tricky.

Wrong

This code example will not work:

```
$.get( "myhtmlpage.html", myCallBack( param1, param2 ) );
```

The reason this fails is that the code executes myCallBack(param1, param2) immediately and then passes myCallBack()'s return value as the second parameter to $.get(). We actually want to pass the function myCallBack(), not myCallBack(param1, param2)'s return value (which might or might not be a function). So, how to pass in myCallBack() and include its arguments?

Right

To defer executing myCallBack() with its parameters, you can use an anonymous function as a wrapper.

> Note the use of function() {. The anonymous function does exactly one thing: calls myCallBack(), with the values of param1 and param2.

```
$.get("myhtmlpage.html", function () {
    myCallBack(param1, param2);
});
```

When $.get() finishes getting the page myhtmlpage.html, it executes the anonymous function, which executes myCallBack(param1, param2).

$ vs $()

Until now, we've been dealing entirely with methods that are called on a jQuery object. For example:

```
$( "h1" ).remove();
```

Most jQuery methods are called on jQuery objects as shown above; these methods are said to be part of the $.fn namespace, or the "jQuery prototype," and are best thought of as jQuery object methods.

However, there are several methods that do not act on a selection; these methods are said to be part of the jQuery namespace, and are best thought of as core jQuery methods.

This distinction can be incredibly confusing to new jQuery users. Here's what you need to remember:

- Methods called on jQuery selections are in the $.fn namespace, and automatically receive and return the selection as this.
- Methods in the $ namespace are generally utility-type methods, and do not work with selections; they are not automatically passed any arguments, and their return value will vary.

There are a few cases where object methods and core methods have the same names, such as $.each() and .each(). In these cases, be extremely careful when reading the documentation that you are exploring the correct method.

In this guide, if a method can be called on a jQuery selection, we'll refer to it just by its name: .each(). If it is a utility method -- that is, a method that isn't called on a selection -- we'll refer to it explicitly as a method in the jQuery namespace: $.each().

$(document).ready()

A page can't be manipulated safely until the document is "ready." jQuery detects this state of readiness for you. Code included inside $(document).ready() will only run once the page Document Object Model (DOM) is ready for JavaScript code to execute. Code included inside $(window).load(function() { ... }) will run once the entire page (images or iframes), not just the DOM, is ready.

```
// A $( document ).ready() block.
$(document).ready(function () {
    console.log("ready!");
});
```

Experienced developers sometimes use shorthand for $(document).ready(). If you are writing code that people who aren't experienced with jQuery may see, it's best to use the long form.

```
// Shorthand for $( document ).ready()
$(function () {
    console.log("ready!");
});
```

You can also pass a named function to $(document).ready() instead of passing an anonymous function.

```
// Passing a named function instead of an anonymous function.
function readyFn(jQuery) {
    // Code to run when the document is ready.
}
$(document).ready(readyFn);
// or:
$(window).load(readyFn);
```

The below example shows $(document).ready() and $(window).load() in action. The code tries to load a website URL in an iframe and checks for both events:

```
<html>
<head>
<script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
<script>
$(document).ready(function () {
    console.log("document loaded");
});
$(window).load(function () {
    console.log("window loaded");
});
</script>
</head>
<body>
<iframe src="http://techcrunch.com"></iframe>
</body>
</html>
```

Avoiding Conflicts with Other Libraries

The jQuery library and virtually all of its plugins are contained within the jQuery namespace. As a general rule, global objects are stored inside the jQuery namespace as well, so you shouldn't get a clash between jQuery and any other library (like prototype.js, MooTools, or YUI).

That said, there is one caveat: by default, jQuery uses $ as a shortcut for jQuery. Thus, if you are using another JavaScript library that uses the $ variable, you can run into conflicts with jQuery. In order to avoid these conflicts, you need to put jQuery in no-conflict mode immediately after it is loaded onto the page and before you attempt to use jQuery in your page.

Putting jQuery Into No-Conflict Mode

When you put jQuery into no-conflict mode, you have the option of assigning a new variable name to replace the $ alias.

```
<!-- Putting jQuery into no-conflict mode. -->
<script src="prototype.js"></script>
<script src="jquery.js"></script>
<script>
var $j = jQuery.noConflict();
// $j is now an alias to the jQuery function; creating the new alias is optional.
$j(document).ready(function () {
    $j("div").hide();
});
// The $ variable now has the prototype meaning, which is a shortcut for
// document.getElementById(). mainDiv below is a DOM element, not a jQuery object.
window.onload = function () {
    var mainDiv = $("main");
}
</script>
```

In the code above, the $ will revert back to its meaning in original library. You'll still be able to use the full function name jQuery as well as the new alias $j in the rest of your application. The new alias can be named anything you'd like: jq, $J, awesomeQuery, etc.

Finally, if you don't want to define another alternative to the full jQuery function name (you really like to use $ and don't care about using the other library's $ method), then there's still another approach you might try: simply add the $ as an argument passed to your jQuery(document).ready() function. This is most frequently used in the case where you still want the benefits of really concise jQuery code, but don't want to cause conflicts with other libraries.

```
<!-- Another way to put jQuery into no-conflict mode. -->
<script src="prototype.js"></script>
<script src="jquery.js"></script>
<script>
jQuery.noConflict();
jQuery(document).ready(function ($) {
    // You can use the locally-scoped $ in here as an alias to jQuery.
    $("div").hide();
});
// The $ variable in the global scope has the prototype.js meaning.
window.onload = function () {
    var mainDiv = $("main");
}
</script>
```

This is probably the ideal solution for most of your code, considering that there'll be less code that you'll have to change in order to achieve complete compatibility.

Including jQuery Before Other Libraries

The code snippets above rely on jQuery being loaded after prototype.js is loaded. If you include jQuery before other libraries, you may use jQuery when you do some work with jQuery, but the $ will have the meaning defined in the other library. There is no need to relinquish the $ alias by calling jQuery.noConflict().

```
<!-- Loading jQuery before other libraries. -->
```

```
<script src="jquery.js"></script>
<script src="prototype.js"></script>
<script>
// Use full jQuery function name to reference jQuery.
jQuery(document).ready(function () {
    jQuery("div").hide();
});
// Use the $ variable as defined in prototype.js
window.onload = function () {
    var mainDiv = $("main");
};
</script>
```

Summary of Ways to Reference the jQuery Function

Here's a recap of ways you can reference the jQuery function when the presence of another library creates a conflict over the use of the $ variable:

Create a New Alias

The jQuery.noConflict() method returns a reference to the jQuery function, so you can capture it in whatever variable you'd like:

```
<script src="prototype.js"></script>
<script src="jquery.js"></script>
<script>
// Give $ back to prototype.js; create new alias to jQuery.
var $jq = jQuery.noConflict();
</script>
```

Use an Immediately Invoked Function Expression

You can continue to use the standard $ by wrapping your code in an immediately invoked function expression; this is also a standard pattern for jQuery plugin authoring, where the author cannot know whether another library will have taken over the $.

```
<!-- Using the $ inside an immediately-invoked function expression. -->
<script src="prototype.js"></script>
<script src="jquery.js"></script>
<script>
jQuery.noConflict();
(function ($) {
    // Your jQuery code here, using the $
})(jQuery);
</script>
```

> Note that if you use this technique, you will not be able to use prototype.js methods inside the immediately invoked function that expect $ to be prototype.js's $.

Use the Argument That's Passed to the jQuery(document).ready() Function

```
<script src="jquery.js"></script>
<script src="prototype.js"></script>
<script>
```

```
jQuery(document).ready(function ($) {
    // Your jQuery code here, using $ to refer to jQuery.
});
</script>
```

Or using the more concise syntax for the DOM ready function:

```
<script src="jquery.js"></script>
<script src="prototype.js"></script>
<script>
jQuery(function ($) {
    // Your jQuery code here, using the $
});
</script>
```

Attributes

An element's attributes can contain useful information for your application, so it's important to be able to get and set them.

The .attr() method

The .attr() method acts as both a getter and a setter. As a setter, .attr() can accept either a key and a value, or an object containing one or more key/value pairs.

.attr() as a setter:

```
$( "a" ).attr( "href", "allMyHrefsAreTheSameNow.html" );
$( "a" ).attr({
title: "all titles are the same too!",
href: "somethingNew.html"
});
```

.attr() as a getter:

```
$( "a" ).attr( "href" ); // Returns the href for the first a element in the document
```

Selecting Elements

The most basic concept of jQuery is to "select some elements and do something with them." jQuery supports most CSS3 selectors, as well as some non-standard selectors.

****Selecting Elements by ID****

```
$( "#myId" ); // Note IDs must be unique per page.
```

Selecting Elements by Class Name

```
$( ".myClass" );
```

Selecting Elements by Attribute

```
$( "input[name='first_name']" ); // Beware, this can be very slow in older browsers
```

Selecting Elements by Compound CSS Selector

```
$( "#contents ul.people li" );
```

Pseudo-Selectors

```
$( "a.external:first" );
$( "tr:odd" );
// Select all input-like elements in a form (more on this below).
$( "#myForm :input" );
$( "div:visible" );
// All except the first three divs.
$( "div:gt(2)" );
// All currently animated divs.
$( "div:animated" );
```

> Note: When using the :visible and :hidden pseudo-selectors, jQuery tests the actual visibility of the element, not its CSS visibility or display properties. jQuery looks to see if the element's physical height and width on the page are both greater than zero.

However, this test doesn't work with

elements. In the case of jQuery does check the CSS display property, and considers an element hidden if its display property is set to none.

Elements that have not been added to the DOM will always be considered hidden, even if the CSS that would affect them would render them visible.

Choosing Selectors

Choosing good selectors is one way to improve JavaScript's performance. A little specificity – for example, including an element type when selecting elements by class name – can go a long way. On the other hand, too much specificity can be a bad thing. A selector such as #myTable thead tr th.special is overkill if a selector such as #myTable th.special will get the job done.

jQuery offers many attribute-based selectors, allowing selections based on the content of arbitrary attributes using simplified regular expressions.

```
// Find all <a> elements whose rel attribute ends with "thinger".
$( "a[rel$='thinger']" );
```

Does My Selection Contain Any Elements?

Once you've made a selection, you'll often want to know whether you have anything to work with. A common mistake is to use:

```
// Doesn't work!
if ($("div.foo")) {
    ...
}
```

This won't work. When a selection is made using $(), an object is always returned, and objects always evaluate to true. Even if the selection doesn't contain any elements, the code inside the if statement will still run.

The best way to determine if there are any elements is to test the selection's .length property, which tells you how many elements were selected. If the answer is 0, the .length property will evaluate to false when used as a boolean value:

```
// Testing whether a selection contains elements.
if ($("div.foo").length) {
    ...
}
```

Saving Selections

jQuery doesn't cache elements for you. If you've made a selection that you might need to make again, you should save the selection in a variable rather than making the selection repeatedly.

```
var divs = $( "div" );
```

Once the selection is stored in a variable, you can call jQuery methods on the variable just like you would have called them on the original selection.

A selection only fetches the elements that are on the page at the time the selection is made. If elements are added to the page later, you'll have to repeat the selection or otherwise add them to the selection stored in the variable. Stored selections don't magically update when the DOM changes.

Refining & Filtering Selections

Sometimes the selection contains more than what you're after. jQuery offers several methods for refining and filtering selections.

```
// Refining selections.
$( "div.foo" ).has( "p" ); // div.foo elements that contain <p> tags
$( "h1" ).not( ".bar" ); // h1 elements that don't have a class of bar
$( "ul li" ).filter( ".current" ); // unordered list items with class of current
$( "ul li" ).first(); // just the first unordered list item
$( "ul li" ).eq( 5 ); // the sixth
```

Working with Selections

Getters & Setters

jQuery "overloads" its methods, so the method used to set a value generally has the same name as the method used to get a value. When a method is used to set a value, it's called a setter. When a method is used to get (or read) a value, it's called a getter. Setters affect all elements in a selection. Getters get the requested value only for the first element in the selection.

```
// The .html() method used as a setter:
$( "h1" ).html( "hello world" );

// The .html() method used as a getter:
$( "h1" ).html();
```

Setters return a jQuery object, allowing you to continue calling jQuery methods on your selection. Getters return whatever they were asked to get, so you can't continue to call jQuery methods on the value returned by the getter.

```
// Attempting to call a jQuery method after calling a getter.
// This will NOT work:
$( "h1" ).html().addClass( "test" );
```

Chaining

If you call a method on a selection and that method returns a jQuery object, you can continue to call jQuery methods on the object without pausing for a semicolon. This practice is referred to as "chaining":

```
$( "#content" ).find( "h3" ).eq( 2 ).html( "new text for the third h3!" );
```

It may help code readability to break the chain over several lines:

```
$( "#content" )
.find( "h3" )
.eq( 2 )
.html( "new text for the third h3!" );
```

jQuery also provides the .end() method to get back to the original selection should you change the selection in the middle of a chain:

```
$( "#content" )
.find( "h3" )
.eq( 2 )
.html( "new text for the third h3!" )
.end() // Restores the selection to all h3s in #content
.eq( 0 )
.html( "new text for the first h3!" );
```

Chaining is extraordinarily powerful, and it's a feature that many libraries have adapted since it was made popular by jQuery. However, it must be used with care – extensive chaining can make code extremely difficult to modify or debug. There is no hard-and-fast rule to how long a chain should be – just know that it's easy to get carried away.

Manipulating Elements

Getting and Setting Information About Elements

There are many ways to change an existing element. Among the most common tasks is changing the inner HTML or attribute of an element. jQuery offers simple, cross-browser methods for these sorts of manipulations. You can also get information about elements using many of the same methods in their getter incarnations. For more information on getters and setters, see the Working with Selections section. Here are a few methods you can use to get and set information about elements:

- .html() – Get or set the HTML contents.
- .text() – Get or set the text contents; HTML will be stripped.
- .attr() – Get or set the value of the provided attribute.
- .width() – Get or set the width in pixels of the first element in the selection as an integer.
- .height() – Get or set the height in pixels of the first element in the selection as an integer.
- .position() – Get an object with position information for the first element in the selection, relative to its first positioned ancestor. This is a getter only.
- .val() – Get or set the value of form elements.

Changing things about elements is trivial, but remember that the change will affect all elements in the selection. If you just want to change one element, be sure to specify that in the selection before calling a setter method.

```
// Changing the HTML of an element.
$( "#myDiv p:first" ).html( "New <strong>first</strong> paragraph!" );
```

Cloning Elements

Methods such as .appendTo() move the element, but sometimes a copy of the element is needed instead. In this case, use .clone() first:

```
// Making a copy of an element.
// Copy the first list item to the end of the list:
$( "#myList li:first" ).clone().appendTo( "#myList" );
```

If you need to copy related data and events, be sure to pass true as an argument to .clone().

Removing Elements

There are two ways to remove elements from the page: .remove() and .detach(). Use .remove() when you want to permanently remove the selection from the page. While .remove() does return the removed element(s), those elements will not have their associated data and events attached to them if you return them to the page.

Use .detach() if you need the data and events to persist. Like .remove(), it returns the selection, but it also maintains the data and events associated with the selection, so you can restore the selection to the page at a later time.

The .detach() method is extremely valuable if you are doing heavy manipulation on an element. In that case, it's beneficial to .detach() the element from the page, work on it in your code, then restore it to the page when you're done. This limits expensive "DOM touches" while maintaining the element's data and events.

If you want to leave the element on the page but remove its contents, you can use .empty() to dispose of the element's inner HTML.

Creating New Elements

jQuery offers a trivial and elegant way to create new elements using the same $() method used to make selections:

```
// Creating new elements from an HTML string.
$("<p>This is a new paragraph</p>");
$("<li class=\"new\">new list item</li>");

// Creating a new element with an attribute object.
$("<a/>", {
    html : "This is a <strong>new</strong> link",
    "class" : "new",
    href : "foo.html"
});
```

Note that the attributes object in the second argument above, the property name class is quoted, although the property names html and href are not. Property names generally do not need to be quoted unless they are reserved words (as class is in this case).

When you create a new element, it is not immediately added to the page. There are several ways to add an element to the page once it's been created.

```
// Getting a new element on to the page.
var myNewElement = $( "<p>New element</p>" );
myNewElement.appendTo( "#content" );
myNewElement.insertAfter( "ul:last" ); // This will remove the p from #content!
$( "ul" ).last().after( myNewElement.clone() ); // Clone the p so now we have
```

The created element doesn't need to be stored in a variable – you can call the method to add the element to the page directly after the $(). However, most of the time you'll want a reference to the element you added so you won't have to select it later.

You can also create an element as you're adding it to the page, but note that in this case you don't get a reference to the newly created element:

```
// Creating and adding an element to the page at the same time.
$( "ul" ).append( "<li>list item</li>" );
```

The syntax for adding new elements to the page is easy, so it's tempting to forget that there's a huge performance cost for adding to the DOM repeatedly. If you're adding many elements to the same container, you'll want to concatenate all the HTML into a single string, and then append that string to the container instead of appending the elements one at a time. Use an array to gather all the pieces together, then join them into a single string for appending:

```
var myItems = [];
var myList = $("#myList");
for (var i = 0; i < 100; i++) {
    myItems.push("<li>item " + i + "</li>");
}
myList.append(myItems.join(""));
```

Manipulating Attributes

jQuery's attribute manipulation capabilities are extensive. Basic changes are simple, but the .attr() method also allows for more complex manipulations. It can either set an explicit value, or set a value using the return value of a function. When the function syntax is used, the function receives two arguments: the zero-based index of the element whose attribute is being changed, and the current value of the attribute being changed.

```
// Manipulating a single attribute.
$("#myDiv a:first").attr("href", "newDestination.html");

// Manipulating multiple attributes.
$("#myDiv a:first").attr({
    href : "newDestination.html",
    rel : "nofollow"
});

// Using a function to determine an attribute's new value.
$("#myDiv a:first").attr({
    rel : "nofollow",
    href : function (idx, href) {
        return "/new/" + href;
    }
});
$("#myDiv a:first").attr("href", function (idx, href) {
    return "/new/" + href;
});
```

The jQuery Object

When creating new elements (or selecting existing ones), jQuery returns the elements in a collection. Many developers new to jQuery assume that this collection is an array. It has a zero-indexed sequence of DOM elements, some familiar array functions, and a .length property, after all. Actually, the jQuery object is more complicated than that.

DOM and DOM Elements

The Document Object Model (DOM for short) is a representation of an HTML document. It may contain any number of DOM elements. At a high level, a DOM element can be thought of as a "piece" of a web page. It may contain text and/or other DOM elements. DOM elements are described by a type, such as

, , or

, and any number of attributes such as src, href, class and so on. For a more thorough description, refer to the official DOM specification from the W3C.

Elements have properties like any JavaScript object. Among these properties are attributes like .tagName and methods like .appendChild(). These properties are the only way to interact with the web page via JavaScript.

The jQuery Object

It turns out that working directly with DOM elements can be awkward. The jQuery object defines many methods to smooth out the experience for developers. Some benefits of the jQuery Object include:

Compatibility – The implementation of element methods varies across browser vendors and versions. The following snippet attempts to set the inner HTML of a

element stored in target:

```
var target = document.getElementById( "target" );
target.innerHTML = "<td>Hello <b>World</b>!</td>";
```

This works in many cases, but it will fail in most versions of Internet Explorer. In that case, the recommended approach is to use pure DOM methods instead. By wrapping the target element in a jQuery object, these edge cases are taken care of, and the expected result is achieved in all supported browsers:

```
// Setting the inner HTML with jQuery.
var target = document.getElementById( "target" );
$( target ).html( "<td>Hello <b>World</b>!</td>" );
```

Convenience – There are also a lot of common DOM manipulation use cases that are awkward to accomplish with pure DOM methods. For instance, inserting an element stored in newElement after the target element requires a rather verbose DOM method:

```
// Inserting a new element after another with the native DOM API.
var target = document.getElementById( "target" );
var newElement = document.createElement( "div" );
target.parentNode.insertBefore( newElement, target.nextSibling );
```

By wrapping the target element in a jQuery object, the same task becomes much simpler:

```
// Inserting a new element after another with jQuery.
```

```
var target = document.getElementById( "target" );
var newElement = document.createElement( "div" );
$( target ).after( newElement );
```

For the most part, these details are simply "gotchas" standing between you and your goals.

Data Methods

There's often data about an element you want to store with the element. In plain JavaScript, you might do this by adding a property to the DOM element, but you'd have to deal with memory leaks in some browsers. jQuery offers a straightforward way to store data related to an element, and it manages the memory issues for you.

```
// Storing and retrieving data related to an element.
$( "#myDiv" ).data( "keyName", { foo: "bar" } );
$( "#myDiv" ).data( "keyName" ); // Returns { foo: "bar" }
```

Any kind of data can be stored on an element. For the purposes of this article, .data() will be used to store references to other elements.

For example, you may want to establish a relationship between a list item and a

that's inside of it. This relationship could be established every single time the list item is touched, but a better solution would be to establish the relationship once, then store a pointer to the

on the list item using .data():

```
// Storing a relationship between elements using .data()
$("#myList li").each(function () {
    var li = $(this);
    var div = li.find("div.content");
    li.data("contentDiv", div);
});
// Later, we don't have to find the div again;
// we can just read it from the list item's data
var firstLi = $("#myList li:first");
firstLi.data("contentDiv").html("new content");
```

In addition to passing .data() a single key-value pair to store data, you can also pass an object containing one or more pairs.

Iterating over jQuery and non-jQuery Objects

jQuery provides an object iterator utility called $.each() as well as a jQuery collection iterator: .each(). These are not interchangeable. In addition, there are a couple of helpful methods called $.map() and .map() that can shortcut one of our common iteration use cases.

$.each()

$.each() is a generic iterator function for looping over object, arrays, and array-like objects. Plain objects are iterated via their named properties while arrays and array-like objects are iterated via their indices.

$.each() is essentially a drop-in replacement of a traditional for or for-in loop. Given:

```
var sum = 0;
var arr = [ 1, 2, 3, 4, 5 ];
```

Then this:

```
for (var i = 0, l = arr.length; i < l; i++) {
    sum += arr[i];
}
console.log(sum); // 15
```

Can be replaced with this:

```
$.each(arr, function (index, value) {
    sum += value;
});
console.log(sum); // 15
```

Notice that we don't have to access arr[index] as the value is conveniently passed to the callback in $.each().

In addition, given:

```
var sum = 0;
var obj = {
    foo : 1,
    bar : 2
}
```

Then this:

```
for (var item in obj) {
    sum += obj[item];
}
console.log(sum); // 3
```

Can be replaced with this:

```
$.each(obj, function (key, value) {
    sum += value;
});
console.log(sum); // 3
```

Again, we don't have to directly access obj[key] as the value is passed directly to the callback.

> Note that $.each() is for plain objects, arrays, array-like objects that are not jQuery collections.

This would be considered incorrect:

```
// Incorrect:
$.each($("p"), function () {
    // Do something
});
```

For jQuery collections, use .each().

.each()

.each() is used directly on a jQuery collection. It iterates over each matched element in the collection and performs a callback on that object. The index of the current element within the collection is passed as an argument to the callback. The value (the DOM element in this case) is also passed, but the callback is fired within the context of the current matched element so the this keyword points to the current element as expected in other jQuery callbacks.

For example, given the following markup:

```
<ul>
<li><a href="#">Link 1</a></li>
<li><a href="#">Link 2</a></li>
<li><a href="#">Link 3</a></li>
</ul>
```

.each() may be used like so:

```
$( "li" ).each( function( index, element ){
console.log( $( this ).text() );
});
// Logs the following:
// Link 1
// Link 2
// Link 3
```

The Second Argument

The question is often raised, "If this is the element, why is there a second DOM element argument passed to the callback?"

Whether intentional or inadvert, the execution context may change. When consistently using the keyword this, it's easy to end up confusing ourselves or other developers reading the code. Even if the execution context remains the same, it may be more readable to use the second parameter as a named parameter. For example:

```
$("li").each(function (index, listItem) {
    this === listItem; // true
    // For example only. You probably shouldn't call $.ajax() in a loop.
    $.ajax({
        success : function (data) {
            // The context has changed.
            // The "this" keyword no longer refers to listItem.
            this !== listItem; // true
        }
    });
});
```

Sometimes .each() Isn't Necessary

Many jQuery methods implicitly iterate over the entire collection, applying their behavior to each matched element. For example, this is unnecessary:

```
$("li").each(function (index, el) {
    $(el).addClass("newClass");
});
```

And this is fine:

```
$( "li" ).addClass( "newClass" );
```

Each li in the document will have the class "newClass" added.

On the other hand, some methods do not iterate over the collection. .each() is required when we need to get information from the element before setting a new value.

This will not work:

```
// Doesn't work:
$( "input" ).val( $( this ).val() + "%" );
// .val() does not change the execution context, so this === window
```

Rather, this is how it should be written:

```
$("input").each(function (i, el) {
    var elem = $(el);
    elem.val(elem.val() + "%");
});
```

The following is a list of methods that require .each():

- .attr() (getter)
- .css() (getter)
- .data() (getter)
- .height() (getter)
- .html() (getter)
- .innerHeight()
- .innerWidth()
- .offset() (getter)
- .outerHeight()
- .outerWidth()
- .position()
- .prop() (getter)

- .scrollLeft() (getter)
- .scrollTop() (getter)
- .val() (getter)
- .width() (getter)

> Note that in most cases, the "getter" signature returns the result from the first element in a jQuery collection while the setter acts over the entire collection of matched elements. The exception to this is .text() where the getter signature will return a concatenated string of text from all matched elements.

In addition to a setter value, the attribute, property, CSS setters, and DOM insertion "setter" methods (i.e. .text() and .html()) accept anonymous callback functions that are applied to each element in the matching set. The arguments passed to the callback are the index of the matched element within the set and the result of the 'getter' signature of the method.

For example, these are equivalent:

```
$("input").each(function (i, el) {
    var elem = $(el);
    elem.val(elem.val() + "%");
});
$("input").val(function (index, value) {
    return value + "%";
});
```

One other thing to keep in mind with this implicit iteration is that traversal methods such as .children() or .parent() will act on each matched element in a collection, returning a combined collection of all children or parent nodes.

.map()

There is a common iteration use case that can be better handled by using the .map() method. Anytime we want to create an array or concatenated string based on all matched elements in our jQuery selector, we're better served using .map().

For example instead of doing this:

```
var newArr = [];
$("li").each(function () {
    newArr.push(this.id);
});
```

We can do this:

```
$("li").map(function (index, element) {
    return this.id;
}).get();
```

Notice the .get() chained at the end. .map() actually returns a jQuery-wrapped collection, even if we return strings out of the callback. We need to use the argument-less version of .get() in order to return a basic JavaScript array that we can work with. To concatenate into a string, we can chain the plain JS .join() array method after .get().

$.map

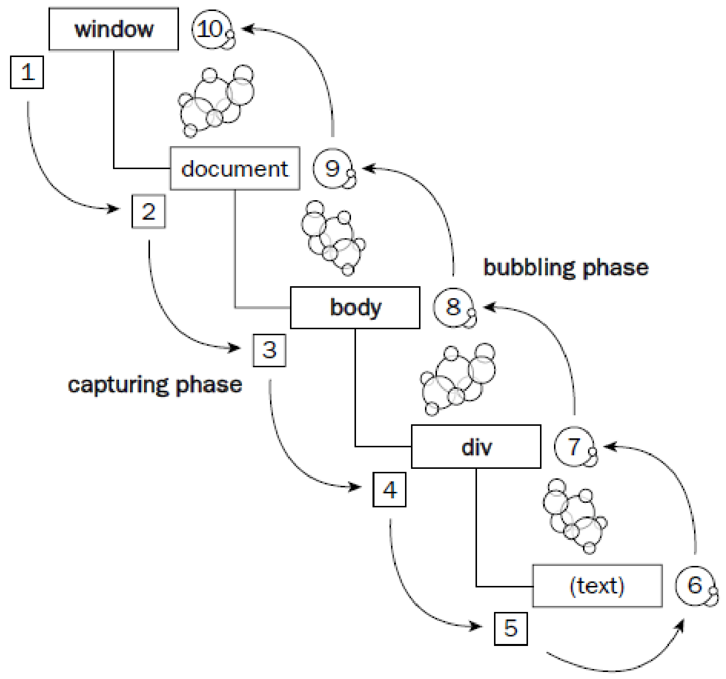
Like $.each() and .each(), there is a $.map() as well as .map(). The difference is also very similar to both .each() methods. $.map() works on plain JavaScript arrays while .map() works on jQuery element collections. Because it's working on a plain array, $.map() returns a plain array and .get() does not need to be called – in fact, it will throw an error as it's not a native JavaScript method.

A word of warning: $.map() switches the order of callback arguments. This was done in order to match the native JavaScript .map() method made available in ECMAScript 5.

For example:

```
<li id="a"></li>
<li id="b"></li>
<li id="c"></li>
<script>
var arr = [{
        id : "a",
        tagName : "li"
    }, {
        id : "b",
        tagName : "li"
    }, {
        id : "c",
        tagName : "li"
    }
];
// Returns [ "a", "b", "c" ]
$("li").map(function (index, element) {
    return element.id;
}).get();
// Also returns ["a", "b", "c"]
// Note that the value comes first with $.map
$.map(arr, function (value, index) {
    return value.id;
});
</script>
```

Events



(Courtesy: washington.edu)

jQuery provides simple methods for attaching event handlers to selections. When an event occurs, the provided function is executed. Inside the function, this refers to the element that was clicked.

For details on jQuery events, visit the Events documentation on api.jquery.com.

The event handling function can receive an event object. This object can be used to determine the nature of the event, and to prevent the event's default behavior.

For details on the event object, visit the Event object documentation on api.jquery.com.

Introduction to Events

Web pages are all about interaction. Users perform a countless number of actions such as moving their mice over the page, clicking on elements, and typing in textboxes — all of these are examples of events. In addition to these user events, there are a slew of others that occur, like when the page is loaded, when video begins playing or is paused, etc. Whenever something interesting occurs on the page, an event is fired, meaning that the browser basically announces that something has happened. It's this announcement that allows developers to "listen" for events and react to them appropriately.

What's a DOM event?

As mentioned, there are a myriad of event types, but perhaps the ones that are easiest to understand are user events, like when someone clicks on an element or types into a form. These types of events occur on an element, meaning that when a user clicks on a button for example, the button has had an event occur on it.

While user interactions aren't the only types of DOM events, they're certainly the easiest to understand when starting out. MDN has a good reference of available DOM events.

Ways to listen for events

There are many ways to listen for events. Actions are constantly occurring on a webpage, but the developer is only notified about them if they're listening for them. Listening for an event basically means you're waiting for the browser to tell you that a specific event has occurred and then you'll specify how the page should react.

To specify to the browser what to do when an event occurs, you provide a function, also known as an event handler. This function is executed whenever the event occurs (or until the event is unbound).

For instance, to alert a message whenever a user clicks on a button, you might write something like this:

```
<button onclick="alert('Hello')">Say hello</button>
```

The event we want to listen to is specified by the button's onclick attribute, and the event handler is the alert function which alerts "Hello" to the user. While this works, it's an abysmal way to achieve this functionality for a couple of reasons:

- First, we're coupling our view code (HTML) with our interaction code (JS). That means that whenever we need to update functionality, we'd have to edit our HTML which is just a bad practice and a maintenance nightmare.
- Second, it's not scalable. If you had to attach this functionality onto numerous buttons, you'd not only bloat the page with a bunch of repetitious code, but you would again destroy maintainability.

Utilizing inline event handlers like this can be considered obtrusive JavaScript, but its opposite, unobtrusive JavaScript is a much more common way of discussing the topic. The notion of *unobtrusive JavaScript* is that your HTML and JS are kept separate and are therefore more maintainable. Separation of concerns is important because it keeps like pieces of code together (i.e. HTML, JS, CSS) and unlike pieces of code apart, facilitating changes, enhancements, etc. Furthermore, unobtrusive JavaScript stresses the importance of adding the least amount of cruft to a page as possible. If a user's browser doesn't support JavaScript, then it shouldn't be intertwined into the markup of the page. Also, to prevent naming collisions, JS code should utilize a single namespace for different pieces of functionality or libraries. jQuery is a good example of this, in that the jQuery object/constructor (and also the $ alias to jQuery) only utilizes a single global variable, and all of jQuery's functionality is packaged into that one object.

To accomplish the desired task unobtrusively, let's change our HTML a little bit by removing the onclick attribute and replacing it with an id, which we'll utilize to "hook onto" the button from within a script file.

```
<button id="helloBtn">Say hello</button>
```

If we wanted to be informed when a user clicks on that button unobtrusively, we might do something like the following in a separate script file:

```
// Event binding using addEventListener
var helloBtn = document.getElementById("helloBtn");
helloBtn.addEventListener("click", function (event) {
    alert("Hello.");
}, false);
```

Here we're saving a reference to the button element by calling getElementById and assigning its return value to a variable. We then call addEventListener and provide an event handler function that will be called whenever that event occurs. While there's nothing wrong with this code as it will work fine in modern browsers, it won't fare well in versions of IE prior to IE9. This is because Microsoft chose to implement a different method, attachEvent, as opposed to the W3C standard addEventListener, and didn't get around to changing it until IE9 was released. For this reason, it's beneficial to utilize jQuery because it abstracts away browser inconsistencies, allowing developers to use a single API for these types of tasks, as seen below.

```
// Event binding using a convenience method
$("#helloBtn").click(function (event) {
    alert("Hello.");
});
```

The $("#helloBtn") code selects the button element using the $ (a.k.a. jQuery) function and returns a jQuery object. The jQuery object has a bunch of methods (functions) available to it, one of them named click, which resides in the jQuery object's prototype. We call the click method on the jQuery object and pass along an anonymous function event handler that's going to be executed when a user clicks the button, alerting "Hello." to the user.

There are a number of ways that events can be listened for using jQuery:

```
// The many ways to bind events with jQuery
// Attach an event handler directly to the button using jQuery's
// shorthand `click` method.
$("#helloBtn").click(function (event) {
    alert("Hello.");
});
// Attach an event handler directly to the button using jQuery's
// `bind` method, passing it an event string of `click`
$("#helloBtn").bind("click", function (event) {
    alert("Hello.");
});
// As of jQuery 1.7, attach an event handler directly to the button
// using jQuery's `on` method.
$("#helloBtn").on("click", function (event) {
    alert("Hello.");
});
// As of jQuery 1.7, attach an event handler to the `body` element that
// is listening for clicks, and will respond whenever *any* button is
// clicked on the page.
$("body").on({
    click : function (event) {
        alert("Hello.");
    }
}, "button");
// An alternative to the previous example, using slightly different syntax.
$("body").on("click", "button", function (event) {
    alert("Hello.");
});
```

As of jQuery 1.7, all events are bound via the on method, whether you call it directly or whether you use an alias/shortcut method such as bind or click, which are mapped to the on method internally. With this in mind, it's beneficial to use the on method because the others are all just syntactic sugar, and utilizing the on method is going to result in faster and more consistent code.

Let's look at the on examples from above and discuss their differences. In the first example, a string of click is passed as the first argument to the on method, and an anonymous function is passed as the second. This looks a lot like the bind method before it. Here, we're attaching an event handler directly to #helloBtn. If there were any other buttons on the page, they wouldn't alert "Hello" when clicked because the event is only attached to #helloBtn.

In the second on example, we're passing an object (denoted by the curly braces {}), which has a property of click whose value is an anonymous function. The second argument to the on method is a jQuery selector string of button. While examples 1–3 are functionally equivalent, example 4 is different in that the body element is listening for click events that occur on any button element, not just #helloBtn. The final example above is exactly the same as the one preceding it, but instead of passing an object, we pass an event string, a selector string, and the callback. Both of these are examples of event delegation, a process by which an element higher in the DOM tree listens for events occurring on its children.

Event delegation works because of the notion of event bubbling. For most events, whenever something occurs on a page (like an element is clicked), the event travels from the element it occurred on, up to its parent, then up to the parent's parent, and so on, until it reaches the root element, a.k.a. the window. So in our table example, whenever a td is clicked, its parent tr would also be notified of the click, the parent table would be notified, the body would be notified, and ultimately the window would be notified as well. While event bubbling and delegation work well, the delegating element (in our example, the table) should always be as close to the delegatees as possible so the event doesn't have to travel way up the DOM tree before its handler function is called.

The two main pros of event delegation over binding directly to an element (or set of elements) are performance and the aforementioned event bubbling. Imagine having a large table of 1,000 cells and binding to an event for each cell. That's 1,000 separate event handlers that the browser has to attach, even if they're all mapped to the same function. Instead of binding to each individual cell though, we could instead use delegation to listen for events that occur on the parent table and react accordingly. One event would be bound instead of 1,000, resulting in way better performance and memory management.

The event bubbling that occurs affords us the ability to add cells via AJAX for example, without having to bind events directly to those cells since the parent table is listening for clicks and is therefore notified of clicks on its children. If we weren't using delegation, we'd have to constantly bind events for every cell that's added which is not only a performance issue, but could also become a maintenance nightmare.

The event object

In all of the previous examples, we've been using anonymous functions and specifying an event argument within that function. Let's change it up a little bit.

```
// Binding a named function
function sayHello(event) {
    alert("Hello.");
}
$("#helloBtn").on("click", sayHello);
```

In this slightly different example, we're defining a function called sayHello and then passing that function into the on method instead of an anonymous function. So many online examples show anonymous functions used as event handlers, but it's important to realize that you can also pass defined functions as event handlers as well. This is important if different elements or different events should perform the same functionality. This helps to keep your code DRY.

But what about that event argument in the sayHello function — what is it and why does it matter? In all DOM event callbacks, jQuery passes an event object argument which contains information about the event, such as precisely when and where it occurred, what type of event it was, which element the event occurred on, and a plethora of other information. Of course you don't have to call it event; you could call it e or whatever you want to, but event is a pretty common convention.

If the element has default functionality for a specific event (like a link opens a new page, a button in a form submits the form, etc.), that default functionality can be cancelled. This is often useful for AJAX requests. When a user clicks on a button to submit a form via AJAX, we'd want to cancel the button/form's default action (to submit it to the form's action attribute), and we would instead do an AJAX request to accomplish the same task for a more seamless experience. To do this, we would utilize the event object and call its .preventDefault() method. We can also prevent the event from bubbling up the DOM tree using .stopPropagation() so that parent elements aren't notified of its occurrence (in the case that event delegation is being used).

```
// Preventing a default action from occurring and stopping the event bubbling
$("form").on("submit", function (event) {
    // Prevent the form's default submission.
    event.preventDefault();
    // Prevent event from bubbling up DOM tree, prohibiting delegation
    event.stopPropagation();
    // Make an AJAX request to submit the form data
});
```

When utilizing both .preventDefault() and .stopPropagation() simultaneously, you can instead return false to achieve both in a more concise manner, but it's advisable to only return false when both are actually necessary and not just for the sake of terseness.

> A final note on .stopPropagation() is that when using it in delegated events, the soonest that event bubbling can be stopped is when the event reaches the element that is delegating it.

It's also important to note that the event object contains a property called originalEvent, which is the event object that the browser itself created. jQuery wraps this native event object with some useful methods and properties, but in some instances, you'll need to access the original event via event.originalEvent for instance. This is especially useful for touch events on mobile devices and tablets.

Finally, to inspect the event itself and see all of the data it contains, you should log the event in the browser's console using console.log. This will allow you to see all of an event's properties (including the originalEvent) which can be really helpful for debugging.

```
// Logging an event's information
$("form").on("submit", function (event) {
    // Prevent the form's default submission.
    event.preventDefault();
    // Log the event object for inspectin'
    console.log(event);
    // Make an AJAX request to submit the form data
});
```

Handling Events

jQuery provides a method .on() to respond to any event on the selected elements. This is called an event binding. Although .on() isn't the only method provided for event binding, it is a best practice to use this for jQuery 1.7+. To learn more, read more about the evolution of event binding in jQuery.

The .on() method provides several useful features:

- Bind any event triggered on the selected elements to an event handler
- Bind multiple events to one event handler
- Bind multiple events and multiple handlers to the selected elements

- Use details about the event in the event handler
- Pass data to the event handler for custom events
- Bind events to elements that will be rendered in the future

Examples

Simple event binding

```
// When any <p> tag is clicked, we expect to see '<p> was clicked' in the console.
$("p").on("click", function () {
    console.log("<p> was clicked");
});
```

Many events, but only one event handler

Suppose you want to trigger the same event whenever the mouse hovers over or leaves the selected elements. The best practice for this is to use "mouseenter mouseleave".

> Note the difference between this and the next example.

```
// When a user focuses on or changes any input element, we expect a console message
// bind to multiple events
$("div").on("mouseenter mouseleave", function () {
    console.log("mouse hovered over or left a div");
});
```

Many events and handlers

Suppose that instead you want different event handlers for when the mouse enters and leaves an element. This is more common than the previous example. For example, if you want to show and hide a tooltip on hover, you would use this.

.on() accepts an object containing multiple events and handlers.

```
$("div").on({
    mouseenter : function () {
        console.log("hovered over a div");
    },
    mouseleave : function () {
        console.log("mouse left a div");
    },
    click : function () {
        console.log("clicked on a div");
    }
});
```

The event object

Handling events can be tricky. It's often helpful to use the extra information contained in the event object passed to the event handler for more control. To become familiar with the event object, use this code to inspect it in your browser console after you click on a div in the page.

```
$("div").on("click", function (event) {
    console.log("event object:");
    console.dir(event);
});
```

Passing data to the event handler

You can pass your own data to the event object.

```
$( "p" ).on( "click", { foo: "bar"  }, function( event ) {
    console.log( "event data: " + event.data.foo + " (should be 'bar')" );
});
```

Binding events to elements that don't exist yet

This is called event delegation. Here's an example just for completeness, but see the page on Event Delegation for a full explanation.

```
$("ul").on("click", "li", function () {
    console.log("Something in a <ul> was clicked, and we detected that it was an <li> element.");
});
```

Connecting Events to Run Only Once

Sometimes you need a particular handler to run only once — after that, you may want no handler to run, or you may want a different handler to run. jQuery provides the .one() method for this purpose.

```
// Switching handlers using the `.one()` method
$("p").one("click", function () {
    console.log("You just clicked this for the first time!");
    $(this).click(function () {
        console.log("You have clicked this before!");
    });
});
```

The .one() method is especially useful if you need to do some complicated setup the first time an element is clicked, but not subsequent times.

.one() accepts the same arguments as .on() which means it supports multiple events to one or multiple handlers, passing custom data and event delegation.

Disconnecting Events

Although all the fun of jQuery occurs in the .on() method, it's counterpart is just as important if you want to be a responsible developer. .off() cleans up that event binding when you don't need it anymore. Complex user interfaces with lots of event bindings can bog down browser performance, so using the .off() method diligently is a best practice to ensure that you only have the event bindings that you need, when you need them.

```
// Unbinding all click handlers on a selection
$( "p" ).off( "click" );
```

```
// Unbinding a particular click handler, using a reference to the function
var foo = function () {
    console.log("foo");
};
var bar = function () {
    console.log("bar");
};
$("p").on("click", foo).on("click", bar);
// foo will stay bound to the click event
$("p").off("click", bar);
```

Namespacing Events

For complex applications and for plugins you share with others, it can be useful to namespace your events so you don't unintentionally disconnect events that you didn't or couldn't know about. For details, see Event Namespacing.

Inside the Event Handling Function

Every event handling function receives an event object, which contains many properties and methods. The event object is most commonly used to prevent the default action of the event via the .preventDefault() method. However, the event object contains a number of other useful properties and methods, including:

pageX, pageY

The mouse position at the time the event occurred, relative to the top left of the page.

type

The type of the event (e.g. "click").

which

The button or key that was pressed.

data

Any data that was passed in when the event was bound.

target

The DOM element that initiated the event.

preventDefault()

Prevent the default action of the event (e.g. following a link).

stopPropagation()

Stop the event from bubbling up to other elements.

In addition to the event object, the event handling function also has access to the DOM element that the handler was bound to via the keyword this. To turn the DOM element into a jQuery object that we can use jQuery methods on, we simply do $(this), often following this idiom:

```
var elem = $( this );
```

```
// Preventing a link from being followed
$("a").click(function (event) {
    var elem = $(this);
    if (elem.attr("href").match("evil")) {
        event.preventDefault();
        elem.addClass("evil");
    }
});
```

Understanding Event Delegation

Event delegation allows us to attach a single event listener, to a parent element, that will fire for all descendants matching a selector, whether those descendants exist now or are added in the future.

Example

For the remainder of the lesson, we will reference the following HTML structure:

```
<html>
<body>
<div id="container">
<ul id="list">
<li><a href="http://domain1.com">Item #1</a></li>
<li><a href="/local/path/1">Item #2</a></li>
<li><a href="/local/path/2">Item #3</a></li>
<li><a href="http://domain4.com">Item #4</a></li>
</ul>
</div>
</body>
</html>
```

When an anchor in our #list group is clicked, we want to log its text to the console. Normally we could directly bind to the click event of each anchor using the .on() method:

```
// attach a directly bound event
$("#list a").on("click", function (event) {
    event.preventDefault();
    console.log($(this).text());
});
```

While this works perfectly fine, there are drawbacks. Consider what happens when we add a new anchor after having already bound the above listener:

```
// add a new element on to our existing list
$( "#list" ).append( "<li><a href='http://newdomain.com'>Item #5</a></li>" );
```

If we were to click our newly added item, nothing would happen. This is because of the directly bound event handler that we attached previously. Direct events are only attached to elements at the time the .on() method is called. In this case, since our new anchor did not exist when .on() was called, it does not get the event handler.

Event Propagation

Understanding how events propagate is an important factor in being able to leverage Event Delegation. Any time one of our anchor tags is clicked, a click event is fired for that anchor, and then bubbles up the DOM tree, triggering each of its parent click event handlers:

```
    <a>
    <li>
    <ul #list>
    <div #container>
    <body>
    <html>
    document root
```

This means that anytime you click one of our bound anchor tags, you are effectively clicking the entire document body! This is called event bubbling or event propagation.

Since we know how events bubble, we can create a delegated event:

```
// attach a delegated event
$("#list").on("click", "a", function (event) {
    event.preventDefault();
    console.log($(this).text());
});
```

Notice how we have moved the a part from the selector to the second parameter position of the .on() method. This second, selector parameter tells the handler to listen for the specified event, and when it hears it, check to see if the triggering element for that event matches the second parameter. In this case, the triggering event is our anchor tag, which matches that parameter. Since it matches, our anonymous function will execute. We have now attached a single click event listener to our ul that will listen for clicks on its descendant anchors, instead of attaching an unknown number of directly bound events to the existing anchor tags only.

Using the Triggering Element

What if we wanted to open the link in a new window if that link is an external one (as denoted here by beginning with "http")?

```
// attach a delegated event
$("#list").on("click", "a", function (event) {
    var elem = $(this);
    if (elem.is("[href^='http']")) {
        elem.attr("target", "_blank");
    }
});
```

This simply passes the .is() method a selector to see if the href attribute of the element starts with "http". We have also removed the event.preventDefault(); statement as we want the default action to happen (which is to follow the href).

We can actually simplify our code by allowing the selector parameter of .on() do our logic for us:

```
// attach a delegated event with a more refined selector
$("#list").on("click", "a[href^='http']", function (event) {
    $(this).attr("target", "_blank");
});
```

Summary

Event delegation refers to the process of using event propagation (bubbling) to handle events at a higher level in the DOM than the element on which the event originated. It allows us to attach a single event listener for elements that exist now or in the future.

Triggering Event Handlers

jQuery provides a way to trigger the event handlers bound to an element without any user interaction via the .trigger() method.

What handlers can be .trigger()'d?

jQuery's event handling system is a layer on top of native browser events. When an event handler is added using .on("click", function() {...}), it can be triggered using jQuery's .trigger("click") because jQuery stores a reference to that handler when it is originally added. Additionally, it will trigger the JavaScript inside the onclick attribute. The .trigger() function cannot be used to mimic native browser events, such as clicking on a file input box or an anchor tag. This is because, there is no event handler attached using jQuery's event system that corresponds to these events.

```
<a href="http://learn.jquery.com">Learn jQuery</a>
```

```
// This will not change the current page
$( "a" ).trigger( "click" );
```

How can I mimic a native browser event, if not .trigger()?

In order to trigger a native browser event, you have to use document.createEventObject for Internet Explorer less than version 9 and document.createEvent for all other browsers. Using these two APIs, you can programmatically create an event that behaves exactly as if someone has actually clicked on a file input box. The default action will happen, and the browse file dialog will display.

The jQuery UI Team created jquery.simulate.js in order to simplify triggering a native browser event for use in their automated testing. Its usage is modeled after jQuery's trigger.

```
// Triggering a native browser event using the simulate plugin
$( "a" ).simulate( "click" );
```

This will not only trigger the jQuery event handlers, but also follow the link and change the current page.

.trigger() vs .triggerHandler()

There are four differences between .trigger() and .triggerHandler()

1. .triggerHandler() only triggers the event on the first element of a jQuery object.
2. .triggerHandler() cannot be chained. It returns the value that is returned by the last handler, not a jQuery object.
3. .triggerHandler() will not cause the default behavior of the event (such as a form submission).
4. Events triggered by .triggerHandler(), will not bubble up the DOM hierarchy. Only the handlers on the single element will fire.

Don't use .trigger() simply to execute specific functions

While this method has its uses, it should not be used simply to call a function that was bound as a click handler. Instead, you should store the function you want to call in a variable, and pass the variable name when you do your binding. Then, you can call the function itself whenever you want, without the need for .trigger().

```
// Triggering an event handler the right way
var foo = function (event) {
    if (event) {
        console.log(event);
    } else {
        console.log("this didn't come from an event!");
    }
};
$("p").on("click", foo);
foo(); // instead of $( "p" ).trigger( "click" )
```

A more complex architecture can built on top of trigger using the publish-subscribe pattern using jQuery plugins. With this technique, .trigger() can be used to notify other sections of code that an application specific event has happened.

Custom Events

We're all familiar with the basic events — click, mouseover, focus, blur, submit, etc. — that we can latch on to as a user interacts with the browser. Custom events open up a whole new world of event-driven programming.

It can be difficult at first to understand why you'd want to use custom events, when the built-in events seem to suit your needs just fine. It turns out that custom events offer a whole new way of thinking about event-driven JavaScript. Instead of focusing on the element that triggers an action, custom events put the spotlight on the element being acted upon. This brings a bevy of benefits, including:

- Behaviors of the target element can easily be triggered by different elements using the same code.
- Behaviors can be triggered across multiple, similar, target elements at once.
- Behaviors are more clearly associated with the target element in code, making code easier to read and maintain.

Recap: .on() and .trigger()

In the world of custom events, there are two important jQuery methods: .on() and .trigger(). In the Events chapter, we saw how to use these methods for working with user events; for this chapter, it's important to remember two things:

- .on() method takes an event type and an event handling function as arguments. Optionally, it can also receive event-related data as its second argument, pushing the event handling function to the third argument. Any data that is passed will be available to the event handling function in the data property of the event object. The event handling function always receives the event object as its first argument.

- .trigger() method takes an event type as its argument. Optionally, it can also take an array of values. These values will be passed to the event handling function as arguments after the event object.

Here is an example of the usage of .on() and .trigger() that uses custom data in both cases:

```
$(document).on("myCustomEvent", {
    foo : "bar"
}, function (event, arg1, arg2) {
    console.log(event.data.foo); // "bar"
    console.log(arg1); // "bim"
    console.log(arg2); // "baz"
});
$(document).trigger("myCustomEvent", ["bim", "baz"]);
```

Ajax

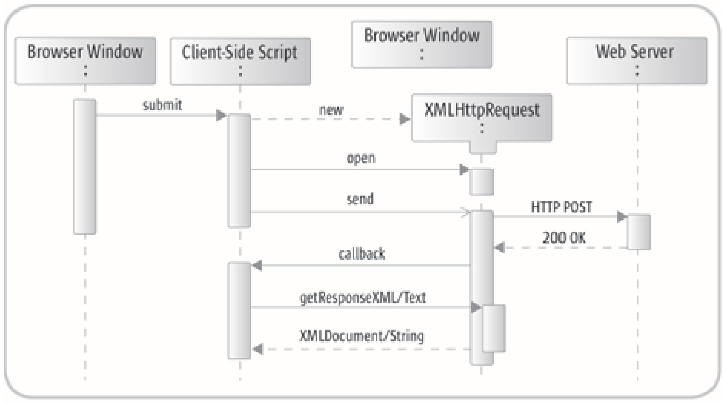


Figure 1 Sequence diagram of an XMLHttpRequest using the HTTP GET method

(Courtesy: Jonas Jacobi)

Traditionally webpages required reloading to update their content. For web-based email this meant that users had to manually reload their inbox to check and see if they had new mail. This had huge drawbacks: it was slow and it required user input. When the user reloaded their inbox, the server had to reconstruct the entire web page and resend all of the HTML, CSS, JavaScript, as well as the user's email. This was hugely inefficient. Ideally, the server should only have to send the user's new messages, not the entire page. By 2003, all the major browsers solved this issue by adopting the XMLHttpRequest (XHR) object, allowing browsers to communicate with the server without requiring a page reload.

The XMLHttpRequest object is part of a technology called Ajax (Asynchronous JavaScript and XML). Using Ajax, data could then be passed between the browser and the server, using the XMLHttpRequest API, without having to reload the web page. With the widespread adoption of the XMLHttpRequest object it quickly became possible to build web applications like Google Maps, and Gmail that used XMLHttpRequest to get new map tiles, or new email without having to reload the entire page.

Ajax requests are triggered by JavaScript code; your code sends a request to a URL, and when it receives a response, a callback function can be triggered to handle the response. Because the request is asynchronous, the rest of your code continues to execute while the request is being processed, so it's imperative that a callback be used to handle the response.

Unfortunately, different browsers implement the Ajax API differently. Typically this meant that developers would have to account for all the different browsers to ensure that Ajax would work universally. Fortunately, jQuery provides Ajax support that abstracts away painful browser differences. It offers both a full-featured $.ajax() method, and simple convenience methods such as $.get(), $.getScript(), $.getJSON(), $.post(), and $().load().

Most jQuery applications don't in fact use XML, despite the name "Ajax"; instead, they transport data as plain

HTML or JSON (JavaScript Object Notation).

In general, Ajax does not work across domains. For instance, a webpage loaded from example1.com is unable to make an Ajax request to example2.com as it would violate the same origin policy. As a work around, JSONP (JSON with Padding) uses script tags to load files containing arbitrary JavaScript content and JSON, from another domain. More recently browsers have implemented a technology called Cross-Origin Resource Sharing (CORS), that allows Ajax requests to different domains.

Key Concepts

Proper use of Ajax-related jQuery methods requires understanding some key concepts first.

GET vs. POST

The two most common "methods" for sending a request to a server are GET and POST. It's important to understand the proper application of each.

The GET method should be used for non-destructive operations — that is, operations where you are only "getting" data from the server, not changing data on the server. For example, a query to a search service might be a GET request. GET requests may be cached by the browser, which can lead to unpredictable behavior if you are not expecting it. GET requests generally send all of their data in a query string.

The POST method should be used for destructive operations — that is, operations where you are changing data on the server. For example, a user saving a blog post should be a POST request. POST requests are generally not cached by the browser; a query string can be part of the URL, but the data tends to be sent separately as post data.

Data Types

jQuery generally requires some instruction as to the type of data you expect to get back from an Ajax request; in some cases the data type is specified by the method name, and in other cases it is provided as part of a configuration object.

There are several options:

text

For transporting simple strings.

html

For transporting blocks of HTML to be placed on the page.

script

For adding a new script to the page.

json

For transporting JSON-formatted data, which can include strings, arrays, and objects.

Note: As of jQuery 1.4, if the JSON data sent by your server isn't properly formatted, the request may fail silently. See http://json.org for details on properly formatting JSON, but as a general rule, use built-in language methods for generating JSON on the server to avoid syntax issues.

jsonp

For transporting JSON data from another domain.

xml

For transporting data in a custom XML schema.

I am a strong proponent of using the JSON format in most cases, as it provides the most flexibility. It is especially useful for sending both HTML and data at the same time.

A is for Asynchronous

The asynchronicity of Ajax catches many new jQuery users off guard. Because Ajax calls are asynchronous by default, the response is not immediately available. Responses can only be handled using a callback. So, for example, the following code will not work:

```
var response;
$.get("foo.php", function (r) {
    response = r;
});
console.log(response); // undefined
```

Instead, we need to pass a callback function to our request; this callback will run when the request succeeds, at which point we can access the data that it returned, if any.

```
$.get("foo.php", function (response) {
    console.log(response); // server response
});
```

Same-Origin Policy and JSONP

In general, Ajax requests are limited to the same protocol (http or https), the same port, and the same domain as the page making the request. This limitation does not apply to scripts that are loaded via jQuery's Ajax methods.

The other exception is requests targeted at a JSONP service on another domain. In the case of JSONP, the provider of the service has agreed to respond to your request with a script that can be loaded into the page using a script tag, thus avoiding the same-origin limitation; that script will include the data you requested, wrapped in a callback function you provide.

Ajax and Firebug

Firebug (or the Webkit Inspector in Chrome or Safari) is an invaluable tool for working with Ajax requests. You can see Ajax requests as they happen in the Console tab of Firebug (and in the Resources > XHR panel of Webkit Inspector), and you can click on a request to expand it and see details such as the request headers, response headers, response content, and more. If something isn't going as expected with an Ajax request, this is the first place to look to track down what's wrong.

jQuery’s Ajax-Related Methods

While jQuery does offer many Ajax-related convenience methods, the core $.ajax() method is at the heart of all of them, and understanding it is imperative. We'll review it first, and then touch briefly on the convenience methods.

I generally use the $.ajax() method and do not use convenience methods. As you'll see, it offers features that the convenience methods do not, and its syntax is more easily understandable, in my opinion.

$.ajax()

jQuery’s core $.ajax() method is a powerful and straightforward way of creating Ajax requests. It takes a configuration object that contains all the instructions jQuery requires to complete the request. The $.ajax() method is particularly valuable because it offers the ability to specify both success and failure callbacks. Also, its ability to take a configuration object that can be defined separately makes it easier to write reusable code. For complete documentation of the configuration options, visit http://api.jquery.com/jQuery.ajax/.

```
// Using the core $.ajax() method
$.ajax({
    // the URL for the request
    url : "post.php",
    // the data to send (will be converted to a query string)
    data : {
        id : 123
    },
    // whether this is a POST or GET request
    type : "GET",
    // the type of data we expect back
    dataType : "json",
    // code to run if the request succeeds;
    // the response is passed to the function
    success : function (json) {
        $("<h1/>").text(json.title).appendTo("body");
        $("<div class=\"content\"/>").html(json.html).appendTo("body");
    },
    // code to run if the request fails; the raw request and
    // status codes are passed to the function
    error : function (xhr, status, errorThrown) {
        alert("Sorry, there was a problem!");
        console.log("Error: " + errorThrown);
        console.log("Status: " + status);
        console.dir(xhr);
    },
    // code to run regardless of success or failure
    complete : function (xhr, status) {
        alert("The request is complete!");
    }
});
```

> A note about the dataType setting: if the server sends back data that is in a different format than you specify, your code may fail, and the reason will not always be clear, because the HTTP response code will not show an error. When working with Ajax requests, make sure your server is sending back the data type you're asking for, and verify that the Content-type header is accurate for the data type. For example, for JSON data, the Content-type header should be application/json.

$.ajax() Options

There are many, many options for the $.ajax() method, which is part of its power. For a complete list of options, visit http://api.jquery.com/jQuery.ajax/; here are several that you will use frequently:

async

Set to false if the request should be sent synchronously. Defaults to true.

> Note that if you set this option to false, your request will block execution of other code until the response is received.

cache

Whether to use a cached response if available. Defaults to true for all dataTypes except "script" and "jsonp". When set to false, the URL will simply have a cachebusting parameter appended to it.

complete

A callback function to run when the request is complete, regardless of success or failure. The function receives the raw request object and the text status of the request.

context

The scope in which the callback function(s) should run (i.e. what this will mean inside the callback function(s)).

By default, this inside the callback function(s) refers to the object originally passed to $.ajax().

data

The data to be sent to the server. This can either be an object or a query string, such as foo=bar&baz=bim.

dataType

The type of data you expect back from the server. By default, jQuery will look at the MIME type of the response if no dataType is specified.

error

A callback function to run if the request results in an error. The function receives the raw request object and the text status of the request.

jsonp

The callback name to send in a query string when making a JSONP request. Defaults to "callback".

success

A callback function to run if the request succeeds. The function receives the response data (converted to a JavaScript object if the dataType was JSON), as well as the text status of the request and the raw request object.

timeout

The time in milliseconds to wait before considering the request a failure.

traditional

Set to true to use the param serialization style in use prior to jQuery 1.4. For details, see http://api.jquery.com/jQuery.param/.

type

The type of the request, "POST" or "GET". Defaults to "GET". Other request types, such as "PUT" and "DELETE" can be used, but they may not be supported by all browsers.

url

The URL for the request.

The url option is the only required property of the $.ajax() configuration object; all other properties are optional.

This can also be passed as the first argument to $.ajax(), and the options object as the second argument.

Convenience Methods

If you don't need the extensive configurability of $.ajax(), and you don't care about handling errors, the Ajax convenience functions provided by jQuery can be useful, terse ways to accomplish Ajax requests. These methods are just "wrappers" around the core $.ajax() method, and simply pre-set some of the options on the $.ajax() method.

The convenience methods provided by jQuery are:

$.get

Perform a GET request to the provided URL.

$.post

Perform a POST request to the provided URL.

$.getScript

Add a script to the page.

$.getJSON

Perform a GET request, and expect JSON to be returned.

In each case, the methods take the following arguments, in order:

url

The URL for the request. Required.

data

The data to be sent to the server. Optional. This can either be an object or a query string, such as foo=bar&baz=bim.

> Note: This option is not valid for $.getScript.

success callback

A callback function to run if the request succeeds. Optional. The function receives the response data (converted to a JavaScript object if the data type was JSON), as well as the text status of the request and the raw request object.

data type

The type of data you expect back from the server. Optional.

> Note: This option is only applicable for methods that don't already specify the data type in their name.

```
// Using jQuery's Ajax convenience methods
// get plain text or HTML
$.get("/users.php", {
    userId : 1234
}, function (resp) {
    console.log(resp); // server response
});
// add a script to the page, then run a function defined in it
$.getScript("/static/js/myScript.js", function () {
    functionFromMyScript();
});
// get JSON-formatted data from the server
$.getJSON("/details.php", function (resp) {
    // log each key in the response data
    $.each(resp, function (key, value) {
        console.log(key + " : " + value);
    });
});
```

$.fn.load

The .load() method is unique among jQuery's Ajax methods in that it is called on a selection. The .load() method fetches HTML from a URL, and uses the returned HTML to populate the selected element(s). In addition to providing a URL to the method, you can optionally provide a selector; jQuery will fetch only the matching content from the returned HTML.

```
// Using $.fn.load to populate an element
$( "#newContent" ).load( "/foo.html" );
```

```
// Using $.fn.load to populate an element based on a selector
$("#newContent").load("/foo.html #myDiv h1:first", function (html) {
    alert("Content updated!");
});
```

Ajax and Forms

jQuery's ajax capabilities can be especially useful when dealing with forms. There are several advantages, which can range from serialization, to simple client-side validation (e.g. "Sorry, that username is taken"), to prefilters (explained below), and even more!

Serialization

Serializing form inputs in jQuery is extremely easy. Two methods come supported natively: .serialize() and .serializeArray(). While the names are fairly self-explanatory, there are many advantages to using them.

The .serialize() method serializes a form's data into a query string. For the element's value to be serialized, it must have a name attribute. Please note that values from inputs with a type of checkbox or radio are included only if they are checked.

```
// Turning form data into a query string
$( "#myForm" ).serialize();
// creates a query string like this:
// field_1=something&field2=somethingElse
```

While plain old serialization is great, sometimes your application would work better if you sent over an array of objects, instead of just the query string. For that, jQuery has the .serializeArray() method. It's very similar to the .serialize() method listed above, except it produces an array of objects, instead of a string.

```
// Creating an array of objects containing form data
$( "#myForm" ).serializeArray();
// creates a structure like this:
// [
// {
// name : "field_1",
// value : "something"
// },
// {
// name : "field_2",
// value : "somethingElse"
// }
// ]
```

Client-side validation

Client-side validation is, much like many other things, extremely easy using jQuery. While there are several cases developers can test for, some of the most common ones are: presence of a required input, valid usernames/emails/phone numbers/etc…, or checking an "I agree…" box.

> Please note that it is advisable that you also perform server-side validation for your inputs. However, it typically makes for a better user experience to be able to validate some things without submitting the form.

With that being said, let's jump on in to some examples! First, we'll see how easy it is to check if a required field doesn't have anything in it. If it doesn't, then we'll return false, and prevent the form from processing.

```
// Using validation to check for the presence of an input
$("#form").submit(function (event) {
    // if .required's value's length is zero
    if ($(".required").val().length === 0) {
        // usually show some kind of error message here
        // this prevents the form from submitting
        return false;
    } else {
        // run $.ajax here
    }
});
```

Let's see how easy it is to check for invalid characters in a phone number:

```
// Validate a phone number field
$("#form").submit(function (event) {
    var inputtedPhoneNumber = $("#phone").val();
    // match only numbers
    var phoneNumberRegex = /^\d*$/;
    // if the phone number doesn't match the regex
    if (!phoneNumberRegex.test(inputtedPhoneNumber)) {
        // usually show some kind of error message here
        // prevent the form from submitting
        return false;
    } else {
        // run $.ajax here
    }
});
```

Prefiltering

A prefilter is a way to modify the ajax options before each request is sent (hence, the name prefilter).

For example, say we would like to modify all cross-domain requests through a proxy. To do so with a prefilter is quite simple:

```
// Using a proxy with a prefilter
$.ajaxPrefilter(function (options, originalOptions, jqXHR) {
    if (options.crossDomain) {
        options.url = "http://mydomain.net/proxy/" + encodeURIComponent(options.url);
        options.crossDomain = false;
    }
});
```

You can pass in an optional argument before the callback function that specifies which dataTypes you'd like the prefilter to be applied to. For example, if we want our prefilter to only apply to JSON and script requests, we'd do:

```
// Using the optional dataTypes argument
$.ajaxPrefilter("json script", function (options, originalOptions, jqXHR) {
    // do all of the prefiltering here, but only for
    // requests that indicate a dataType of "JSON" or "script"
});
```

Working with JSONP

The advent of JSONP — essentially a consensual cross-site scripting hack — has opened the door to powerful mashups of content. Many prominent sites provide JSONP services, allowing you access to their content via a predefined API. A particularly great source of JSONP-formatted data is the Yahoo! Query Language, which we'll use in the following example to fetch news about cats.

```
// Using YQL and JSONP
$.ajax({
    url : "http://query.yahooapis.com/v1/public/yql",
    // the name of the callback parameter, as specified by the YQL service
    jsonp : "callback",
    // tell jQuery we're expecting JSONP
    dataType : "jsonp",
    // tell YQL what we want and that we want JSON
    data : {
        q : "select title,abstract,url from search.news where query=\"cat\"",
        format : "json"
    },
    // work with the response
    success : function (response) {
        console.log(response); // server response
    }
});
```

jQuery handles all the complex aspects of JSONP behind-the-scenes — all we have to do is tell jQuery the name of the JSONP callback parameter specified by YQL ("callback" in this case), and otherwise the whole process looks and feels like a normal Ajax request.

Ajax Events

Often, you'll want to perform an operation whenever an Ajax requests starts or stops, such as showing or hiding a loading indicator. Rather than defining this behavior inside every Ajax request, you can bind Ajax events to elements just like you'd bind other events. For a complete list of Ajax events, visit Ajax Events documentation on docs.jquery.com.

```
// Setting up a loading indicator using Ajax Events
$("#loading_indicator").ajaxStart(function () {
    $(this).show();
}).ajaxStop(function () {
    $(this).hide();
});
```

Functional Programming

> Note: Functional programming in this book mainly refers to JavaScript context but the principles are the same.

Getting Started with Functional Programming

This chapter is courtesy of Michael Fogus, author of "Functional JavaScript" (O'Reilly).

You may have heard of functional programming on your favorite news aggregation site, or maybe you've worked in a language supporting functional techniques. If you've written JavaScript (and in this book I assume that you have) then you indeed have used a language supporting functional programming. However, that being the case, you might not have used JavaScript in a functional way. This book outlines a functional style of programming that aims to simplify your own libraries and applications, and helps tame the wild beast of JavaScript complexity.

As a bare-bones introduction, functional programming can be described in a single sentence:

> Functional programming is the use of functions that transform values into units of abstraction, subsequently used to build software systems.

This is a simplification bordering on libel, but it’s functional (ha!) for this early stage in the book. The library that I use as my medium of functional expression in JavaScript is Underscore, and for the most part, it adheres to this basic definition. However, this definition fails to explain the “why” of functional programming.

Why Functional Programming Matters

> The major evolution that is still going on for me is towards a more functional programming style, which involves unlearning a lot of old habits, and backing away from some OOP directions.
> —John Carmack

If you’re familiar with object-oriented programming, then you may agree that its primary goal is to break a problem into parts. Likewise, these parts/objects can be aggregated and composed to form larger parts. Based on these parts and their aggregates, a system is then described in terms of the interactions and values of the parts. This is a gross simplification of how object-oriented systems are formed, but I think that as a high-level description it works just fine. By comparison, a strict functional programming approach to solving problems also breaks a problem into parts (namely, functions).

> Whereas the object-oriented approach tends to break problems into groupings of “nouns,” or objects, a functional approach breaks the same problem into groupings of “verbs,” or functions.

As with object-oriented programming, larger functions are formed by “gluing” or “composing” other functions together to build high-level behaviors. Finally, one way that the functional parts are formed into a system is by taking a value and gradually “transforming” it — via one primitive or composed function — into another.

In a system observing a strict object-oriented style, the interactions between objects cause internal change to each object, leading to an overall system state that is the amalgamation of many smaller, potentially subtle state changes. These interrelated state changes form a conceptual “web of change” that, at times, can be confusing to keep in your head. This confusion becomes a problem when the act of adding new objects and system features requires a working knowledge of the subtleties of potentially far-reaching state changes.

> A functional system, on the other hand, strives to minimize observable state modification.

Therefore, adding new features to a system built using functional principles is a matter of understanding how new functions can operate within the context of localized, nondestructive (i.e., original data is never changed) data transformations.

However, I hesitate to create a false dichotomy and say that functional and object-oriented styles should stand in opposition. That JavaScript supports both models means that systems can and should be composed of both models. Finding the balance between functional and object-oriented styles is a tricky task that will be tackled much later in the book, when discussing mixins in Chapter 9. However, since this is a book about functional programming in JavaScript, the bulk of the discussion is focused on functional styles rather than object-oriented ones.

Having said that, a nice image of a system built along functional principles is an assembly-line device that takes raw materials in one end, and gradually builds a product that comes out the other end.

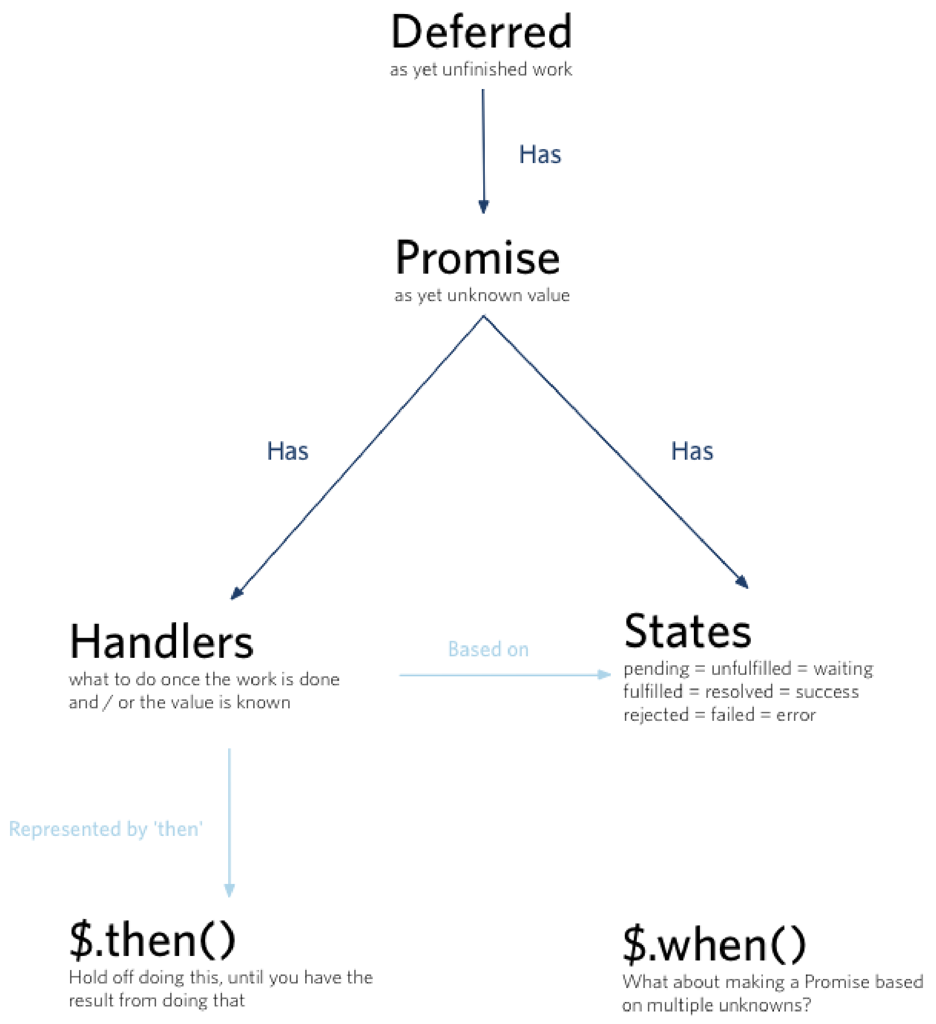
The assembly line analogy is, of course, not entirely perfect, because every machine I know consumes its raw materials to produce a product. By contrast, functional programming is what happens when you take a system built in an imperative way and shrink explicit state changes to the smallest possible footprint to make it more modular (Hughes 1984).

> Practical functional programming is not about eliminating state change, but instead about reducing the occurrences of mutation to the smallest area possible for any given system.

To summarize the essence of functional programming, here are some key ideas:

- functions as units of abstraction
- functions as units of behavior
- data as abstraction - UnderscoreJS/LoDash
- data flow versus control flow (app logic versus control logic)
- function chaining
- functional reactive programming - see BaconJS

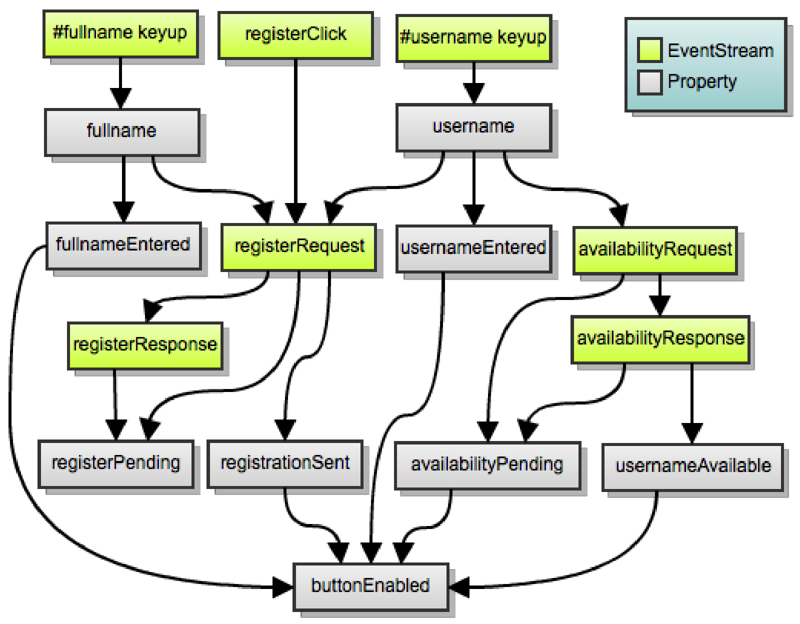
Promise-based Programming


Deferred
as yet unfinished work
Has
Promise
as yet unknown value
Has
Has
Handlers
what to do once the work is done and / or the value is known
Based on
States
pending = unfulfilled = waiting
fulfilled = resolved = success
rejected = failed = error
Represented by 'then'
$.then()
Hold off doing this, until you have the result from doing that
$.when()
What about making a Promise based on multiple unknowns?

(Courtesy: Chris Webb)

I will port BaconJS tutorial to promise-based programming using WhenJS via Browserify (well, it is not a full-blown port but it's functional).

> Note: The BaconJS tutorial includes Mockjax 1.5.1 which works for jQuery 1.8.0 only.



Here are the specifications using BaconJS library:

```
Disable button if username is missing
    define usernameEntered property
    assign side-effect: setDisabled for registerButton

Disable also if full name is missing
    define fullname and fullnameEntered properties
    use .and() to change the condition for enabling the button

Disable also if username unavailable
    include usernameAvailable to the condition for enabling the button

Show AJAX indicator when AJAX pending
    define usernameRequestPending property as usernameResponse.pending(usernameRequest)
    assign side effect to show usernameAjaxIndicator

Disable button when AJAX is pending

Implement registerClick stream
    tip: do(".preventDefault")

Implement registrationRequest
    combine username and password into a new property that would be the data given to JQuery.ajax
    can use username.combine(..) or Bacon.combineTemplate
    type: "POST"

Make this a stream of registration requests, send when the button is clicked
    .sampledBy(registerClick)

Create registrationResponse stream
    as in usernameResponse stream

Show feedback

Disable button after registration sent

Show ajax indicator for registration POST
```

My objectives here are twofold:

1. To show you a promise-based version of that tutorial without porting all the functionality found in BaconJS
2. To show coding around the event paradigm of jQuery (such as onClick event)

Here is the HTML and BaconJS code of that tutorial.

```
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="css/register.css">
    <script type="text/javascript" src="lib/jquery.js"></script>
    <script type="text/javascript" src="lib/jquery.mockjax.js"></script>
    <script type="text/javascript" src="lib/Bacon.js"></script>
    <script type="text/javascript" src="lib/Bacon.UI.js"></script>
    <script type="text/javascript" src="mocks.js"></script>
  </head>
  <body>
    <form id="login-container">
      <h1>Bacon Registration Form</h1>
      <div id="username">
        <input type="text" placeholder="username">
        <em class="ajax"></em>
        <em id="username-unavailable" class="tooltip">Username is unavailable</em>
      </div>
      <div id="fullname">
        <input type="text" placeholder="Full Name">
      </div>
      <div id="register">
        <button>Get some!</button>
        <em class="ajax"></em>
        <span id="result"></span>
      </div>
    </form>
  </body>
</html>
```

```
    <script type="text/javascript">
      function show(x) { console.log(x) }
      function nonEmpty(x) { return x.length > 0 }
      function setVisibility(element, visible) {
        element.toggle(visible)
      }
      function setEnabled(element, enabled) {
        element.attr("disabled", !enabled)
      }
      $(function() {
          // fields
          usernameField = $("#username input")
          fullnameField = $("#fullname input")
          registerButton = $("#register button")
          unavailabilityLabel = $("#username-unavailable")
          usernameAjaxIndicator = $("#username .ajax")
          registerAjaxIndicator = $("#register .ajax")

          // username
          username = Bacon.UI.textFieldValue(usernameField)
          usernameEntered = username.map(nonEmpty)
          availabilityRequest = username.changes().filter(nonEmpty).skipDuplicates()
            .throttle(300)
            .map(function(user) { return { url : "/usernameavailable/" + user } })
          availabilityResponse = availabilityRequest.ajax()
          usernameAvailable = availabilityResponse.toProperty(true)
          availabilityPending = availabilityResponse.pending(availabilityRequest)

          // fullname
          fullname = Bacon.UI.textFieldValue(fullnameField)
          fullnameEntered = fullname.map(nonEmpty)

          // registration
          registerClick = registerButton.asEventStream("click").do(".preventDefault")
          registrationRequest = username.combine(fullname, function(u, f) {
              return {
                type: "post",
                url : "/register",
                contentType: "application/json",
                data: JSON.stringify({ username: u, fullname: f })
              }
          }).sampledBy(registerClick)
          registrationResponse = registrationRequest.ajax()
          registrationPending = registrationResponse.pending(registrationRequest)
          registrationSent = registrationRequest.map(true).toProperty(false)

          // button enabled state
          buttonEnabled = usernameEntered.and(fullnameEntered).and(usernameAvailable)
            .and(availabilityPending.not()).and(registrationSent.not())

          // side-effects
          buttonEnabled.onValue(setEnabled, registerButton)
          usernameAvailable.not().and(availabilityPending.not())
            .onValue(setVisibility, unavailabilityLabel)
          availabilityPending.onValue(setVisibility, usernameAjaxIndicator)
          registrationPending.onValue(setVisibility, registerAjaxIndicator)
          registrationResponse.map("Thanks dude!").onValue($("#result"), "text")
      })
    </script>
```

Here is the Promise-based version using WhenJS using the same HTML.

Why WhenJS?

I recommend you download the source code of Ghost blog platform which offers a ton of examples using WhenJS.

```
function jqPromise() {

    var when = require('when');

    function show(x) {
        console.log(x)
    }
    function nonEmpty(x) {
        return x.length > 0
    }
    function setVisibility(element, visible) {
        element.toggle(visible)
    }
    function setEnabled(element, enabled) {
        element.attr("disabled", !enabled)
    }

    var registerButton = $("#register button");
    var unavailabilityLabel = $("#username-unavailable");
    var usernameAjaxIndicator = $("#username .ajax");
    var registerAjaxIndicator = $("#register .ajax");

    //initialize
    setVisibility(usernameAjaxIndicator, false);
    setVisibility(registerAjaxIndicator, false);
```

```
setEnabled(registerButton, false);

//a deferred's resolve/reject and notify are mutually exclusive

var fullname = when('');
var username = when('');
var usernameEntered = when(false);
var fullnameEntered = when(false);
var buttonEnabled = when(false);
var availabilityRequest = when.defer();
var jqAvailabilityRequest; //= $.Deferred();
var availabilityPending = when.defer();
var usernameAvailable = when(false);

//no availabilityResponse (less variables, the better)

function setButtonEnabled() {
    buttonEnabled = when.join(usernameEntered, usernameAvailable, fullnameEntered);

    buttonEnabled.then(function (arrayBool) {
        //alert(arrayBool);

        len = arrayBool.length;
        var result = true;
        for (var i = 0; i < len; i++) {
            result = result && arrayBool[i];
        }
        setEnabled(registerButton, result);

    });
}

//promises enable you to get out of the callback shell
$("#username input").on('keyup', function () {

    var user = $(this).val(); //string

    username = when(user);

    if (nonEmpty(user)) {

        usernameEntered = when(true);

        //simulate availabilityPending
        setVisibility(usernameAjaxIndicator, true);

        //jQuery AJAX calls implement a promise interface
        jqAvailabilityRequest = $.get("/usernameavailable/" + user);

        usernameAvailable = when(jqAvailabilityRequest);

        usernameAvailable.then(function (bool) {

            setVisibility(usernameAjaxIndicator, false);

            //when usernameavailable is false, show unavailabilityLabel
            setVisibility(unavailabilityLabel, !bool);
        });



    } else {
        usernameEntered = when(false);
    }
    setButtonEnabled();
});
```

```
    $("#fullname input").on('keyup', function () {
        strFullname = $(this).val(); //string
        fullname = when(strFullname);

        if (nonEmpty(strFullname)) {
            fullnameEntered = when(true);
        } else {
            fullnameEntered = when(false);
        }

        setButtonEnabled();
    });

    registerButton.on('click', function (event) {
        $(this).prop('disabled', true);

        event.preventDefault();

        setVisibility(registerAjaxIndicator, true);

        var input = when.join(username, fullname);


        input.then(function (values) {
            $.ajax({
                type : "post",
                url : "/register",
                contentType : "application/json",
                data : JSON.stringify({
                    username : values[0],
                    fullname : values[1]
                }),
                success : function (data) {
                    alert("Username: " + values[0] + " Fullname: " + values[1]);

                    $("#username input").val('');
                    $("#fullname input").val('');

                    setVisibility(registerAjaxIndicator, false);

                    window.open('demo3.html', "_self");  //refresh
                }
            });
        });


    });
}
```

Lazy Evaluation

The concept of promise-based programming is based on functional programming (FP). The key to understand FP is the concept of lazy evaluation. In this book, I will discuss about FP in the context of JavaScript language. Also, I'm going to discuss about promise-based programming in the context of WhenJS library (other libraries may have different syntax but the principles are the same).

> In imperative programming, you assign a value to a variable (`var str = 'string';`)
>
> In functional programming, you assign a value to an object which you can evaluate later (hence lazy

> evaluation)

For example,

```
var buttonEnabled = when(false);
```

When you want to evaluate the Boolean value being hold by `buttonEnabled` object, you can use its "then" method like so.

```
buttonEnabled.then(function(value) {
    console.log(value);  //false
});
```

In our example, see how we are able to capture state of three variables using promises without using a callback in two keyup event handlers .

```
    function setButtonEnabled() {
        buttonEnabled = when.join(usernameEntered, usernameAvailable, fullnameEntered);

        buttonEnabled.then(function (arrayBool) {
            //alert(arrayBool);

            len = arrayBool.length;
            var result = true;
            for (var i = 0; i < len; i++) {
                result = result && arrayBool[i];
            }
            setEnabled(registerButton, result);

        });
    }
```

From WhenJS documentation,

when()

```
var promise = when(x);
```

Get a trusted promise for x . If x is:

- a value, returns a promise fulfilled with x
- a promise, returns x .
- a foreign thenable, returns a promise that follows x

```
var transformedPromise = when(x, f);
```

Get a trusted promise by transforming x with f . If x is

- a value, returns a promise fulfilled with f(x)
- a promise or thenable, returns a promise that

a) if x fulfills, will fulfill with the result of calling f with x 's fulfillment value.

b) if x rejects, will reject with the same reason as x

Alternative to Callback Hell

Since JavaScript is built around the concept of callbacks, consider the following jQuery GET AJAX example.

```
$.get( "test.php", function( data ) {
    alert( "Data Loaded: " + data );
});
```

If you need to pass that data outside of that anonymous function, you need to call another callback, ad infinitum...aka callback hell.

```
function callback(data) {
    ...
}

$.get( "test.php", function( data ) {
    alert( "Data Loaded: " + data );
    callback(data);
});
```

Callbacks are fine as long as the functionality you want is confined in the callback context. What if you want to evaluate the data in our GET example outside of the callback function? Since jQuery implements the Promise interface, you can make the following.

```
var objGet = $.get( "test.php");
```

So now you can evaluate objGet like so.

```
objGet.done(function (data) {
    alert("Data Loaded: " + data);
});
```

The variable objGet enables us to escape callback hell by encapsulating the value we want into an object that can be evaluated later.

Deferred and Promises

Now since we know that functional programming enables us to encapsulate state (or whatever data you want to track) into an object for later evaluation, we will differentiate between two objects that are central to promise-based programming.

1. Deferred - an object that we can trigger as a result of an event (success, error and notify) and pass the result to a Promise object
2. Promise - an object that holds the result of a Deferred object operation (either resolve/reject or notify

operation)

For example, in a relational database a transaction is either committed or not.

- when a transaction is committed, we can say that it resolves a successful operation
- when a transaction is rolled back, we can say that it rejects the operation

Consider the following code.

```
var i = 1;
var dbTxnDeferred = when.defer();
var dbTxnNotifyDeferred = when.defer();

dbTxnNotifyDeferred.notify('Transaction pending');

if (i === 1) {

    dbTxnDeferred.resolve('Transaction committed');
} else {
    dbTxnDeferred.reject('Transaction rolled back');
}

dbTxnNotifyDeferred.notify('Waiting for transaction... ');
dbTxnDeferred.reject('Transaction rolled back');

var dbTxnNotify = dbTxnNotifyDeferred.promise;
var dbTxnPromise = dbTxnDeferred.promise;  //returns a promise object

dbTxnNotify.progress(onProgress);

dbTxnPromise
.then(onFulfilled)
.catch(onRejected)

function onProgress(value) {
    console.log(value); //Transaction pending
}

function onFulfilled(value) {
    console.log("Success: " + value); //Transaction committed
}

function onRejected(value) {
    console.log("Error: " + value); //Transaction rolled back
}
```

Output:

```
"Transaction pending"
"Waiting for transaction... "
"Success: Transaction committed"
```

From WhenJS documentation:

A promise makes the following guarantees about handlers registered in the same call to .then():

1. Only one of onFulfilled or onRejected will be called, never both.
2. onFulfilled and onRejected will never be called more than once.

3. onProgress may be called zero or more times.

A couple of notes:

- notice that dbTxnDeferred.reject was never called since it was already resolved
- resolve/reject are mutually exclusive. If there is resolve/reject call, notify will be ignored
- if you want to use notification, create a new deferred object separate from the deferred that resolves/rejects

jQuery and WhenJS

Look at the following code:

```
    $("#username input").on('keyup', function () {

        var user = $(this).val(); //string
        username = when(user); //promise object that holds a string

        if (nonEmpty(user)) {

            usernameEntered = when(true);

            //simulate availabilityPending
            setVisibility(usernameAjaxIndicator, true);

            //jQuery AJAX calls implement a promise interface
            jqAvailabilityRequest = $.get("/usernameavailable/" + user);

            usernameAvailable = when(jqAvailabilityRequest);

            usernameAvailable.then(function (bool) {

                setVisibility(usernameAjaxIndicator, false);

                //when usernameavailable is false, show unavailabilityLabel
                setVisibility(unavailabilityLabel, !bool);
            });

        } else {
            usernameEntered = when(false);
        }
        setButtonEnabled();
    });
```

Since jQuery AJAX calls also implement the Promise interface, now we are stuck with a jQuery promise object. Fortunately, WhenJS has a when() function that accepts any promise that provides a thenable promise, that is, any object that provides a .then() method, even promises that aren't fully Promises/A+ compliant, such as jQuery's Deferred. It will assimilate such promises and make them behave like Promises/A+.

This is important so we can convert a jQuery promise into a WhenJS promise that can be evaluated later on. If all we need is to evaluate the data returned by the jQuery promise object, we can do so using jQuery's then or done method on the spot.

```
//jQuery AJAX calls implement a promise interface
jqAvailabilityRequest = $.get("/usernameavailable/" + user);

usernameAvailable = when(jqAvailabilityRequest);
```

Learn Bootstrap 2

Bootstrap, from Twitter

Bootstrap is a toolkit from Twitter designed to kickstart development of webapps and sites.
It includes base CSS and HTML for typography, forms, buttons, tables, grids, navigation, and more.

Nerd alert: Bootstrap is **built with Less** and was designed to work out of the gate with modern browsers in mind.

> Note: If you are into jQuery UI, Foundation or any other user interface framework, then by all means use it. However, if you are totally new to CSS, I will recommend using Bootstrap from Twitter. The built-in components enable you to use them out of the box and manipulate them using jQuery.

Since we are dealing mostly with the Web version of desktop applications, Bootstrap 2.3.2 which is the last version in the 2.0 series is more than enough to get us up and running in no time without learning advanced CSS. The goal is to build the GUI as fast as possible with Bootstrap 2 components. The wording in this chapter has been copied verbatim from the Bootstrap documentation.

For a more detailed treatment regarding Bootstrap, I recommend the book "Bootstrap" by Jake Spurlock (O'Reilly).

Without further ado, here are some of the most important concepts in Bootstrap 2.

Container

A fixed container layout provides a common fixed-width (and optionally responsive) layout requiring only class="container" on your page. If you want a fluid container layout, use class="container-fluid" instead.

```
<div class="container">
<div class="row">
    <div class="span2">
        <button type="button" class="btn btn-block btn-inverse">Span 2</button>
    </div>
    <div class="span10">
        <button type="button" class="btn btn-block btn-success">Span 10</button>
    </div>
</div>
</div>
```

Fixed grid

The default Bootstrap 2 grid system utilizes 12 columns, making for a 940px wide container without responsive features enabled. With the responsive CSS file added, the grid adapts to be 724px and 1170px wide depending on your viewport. Below 767px viewports, the columns become fluid and stack vertically.

For a simple two column layout, create a .row and add the appropriate number of .span* columns. As this is a 12-column grid, each .span* spans a number of those 12 columns, and should always add up to 12 for each row (or the number of columns in the parent).

```
<div class="row">
    <div class="span4">
        <button type="button" class="btn btn-block btn-primary">Span 4</button>
    </div>
    <div class="span8">
        <button type="button" class="btn btn-block btn-danger">Span 8</button>
    </div>
</div>
```

Offset columns

Move columns to the right using .offset* classes. Each class increases the left margin of a column by a whole column. For example, .offset4 moves .span4 over four columns.

```
<div class="divBootOffset">
    <div class="row-fluid">
        <div class="span4">
            <button type="button" class="btn btn-block btn-primary">Span 4</button>
        </div>

        <div class="span8">
            <button type="button" class="btn btn-block btn-danger">Span 8</button>
        </div>
    </div>

    <br>

    <div class="row-fluid">
        <div class="span4">
            <button type="button" class="btn btn-block btn-primary">Span 4</button>
        </div>
        <div class="span5 offset3">
            <button type="button" class="btn btn-block btn-warning">Span 5 offset 3</button>
        </div>
    </div>

    <br>

    <div class="row-fluid">
        <div class="span4">
            <button type="button" class="btn btn-block btn-primary">Span 4</button>
        </div>
        <div class="span2 offset2">
            <button type="button" class="btn btn-block btn-success">Span 2 offset 2</button>
        </div>
        <div class="span3 offset1">
            <button type="button" class="btn btn-block btn-info">Span 3 offset1</button>
        </div>
    </div>
</div>
```

Nesting columns

To nest your content with the default grid, add a new .row and set of .span* columns within an existing .span* column. Nested rows should include a set of columns that add up to the number of columns of its parent.

```
<div class="row-fluid">
    <div class="span12">
        <button type="button" class="btn btn-block btn-danger">Span 12</button>
        <div class="row-fluid">
            <div class="span8">
                <button type="button" class="btn btn-block btn-info">Level 2 span 8
                </button>
                <div class="row-fluid">
                    <div class="span3">
                        <button type="button" class="btn btn-block btn-success">Level 3 span 3
                        </button>
                    </div>

                    <div class="span9">
                        <button type="button" class="btn btn-block btn-default">Level 3 span 9
                        </button>
                    </div>
                </div>
            </div>
            <div class="span4">
                <button type="button" class="btn btn-block btn-warning">Level 2 span 4
                </button>
            </div>
        </div>
    </div>
</div>
```

Fluid grid

The fluid grid system uses percents instead of pixels for column widths. It has the same responsive capabilities as the fixed grid system, ensuring proper proportions for key screen resolutions and devices.

Make any row "fluid" by changing .row to .row-fluid. The column classes stay the exact same, making it easy to flip between fixed and fluid grids.

```
<div class="row-fluid">
    <div class="span4">
        <button type="button" class="btn btn-block btn-primary">Span 4</button>
    </div>

    <div class="span8">
        <button type="button" class="btn btn-block btn-danger">Span 8</button>
    </div>
</div>
```

Navigation tab

All nav components —tabs, pills, and lists—share the same base markup and styles through the .nav class.

```
<ul class="nav nav-tabs">

    <li class="dropdown">
        <a class="dropdown-toggle" data-toggle="dropdown" href="#">jQuery 101
            <b class="caret"></b>
        </a>
        <ul class="dropdown-menu">
            <li><a href="#">Action</a></li>
            <li><a href="#">Another action</a></li>
            <li><a href="#">Something else here</a></li>
            <li class="divider"></li>
            <li><a href="#">Separated link</a></li>
        </ul>
    </li>

    <li class="dropdown">
        <a class="dropdown-toggle" data-toggle="dropdown" href="#">Bacon.js
            <b class="caret"></b>
        </a>
        <ul class="dropdown-menu">
            <li><a href="#">Merge and Scan</a></li>
            <li><a href="#">Filter EventStream</a></li>
            <li><a href="#">Publish/Subscribe with Bus</a></li>
            <li><a href="#">EventStream and Property</a></li>
            <li><a href="#">Filter Property-based EventStream </a></li>
            <li><a href="#">Combine EventStream</a></li>
        </ul>
    </li>
</ul>
```

Navigation bar

To start, navbars are static (not fixed to the top) and include support for a project name and basic navigation.

Place one anywhere within a .container, which sets the width of your site and content.

```
<div class="navbar">
    <div class="navbar-inner">
    <a class="brand" href="#">Title</a>
    <ul class="nav">
    <li class="active"><a href="#">Home</a></li>
    <li><a href="#">Link</a></li>
    <li><a href="#">Link</a></li>
    </ul>
    </div>
</div>
```

Breadcrumb

Breadcrumb provides context when you are in a specific page

```
<ul class="breadcrumb">
<li><a href="#">Home</a> <span class="divider">&raquo;</span></li>
<li><a href="#">Software</a> <span class="divider">&raquo;</span></li>
<li><a href="#">Linux</a></li>
</ul>
```

Pagination

Simple pagination inspired by Rdio, great for apps and search results. The large block is hard to miss, easily scalable, and provides large click areas.

```
<div class="pagination">
    <ul>
    <li><a href="#">Prev</a></li>
    <li><a href="#">1</a></li>
    <li><a href="#">2</a></li>
    <li><a href="#">3</a></li>
    <li><a href="#">4</a></li>
    <li><a href="#">5</a></li>
    <li><a href="#">Next</a></li>
    </ul>
</div>
```

Labels and badges

```
<span class="label">Default</span>
<span class="label label-success">Success</span>
<span class="label label-warning">Warning</span>
<span class="label label-important">Important</span>
<span class="label label-info">Info</span>
<span class="label label-inverse">Inverse</span>

<br><br>

<span class="badge">1</span>
<span class="badge badge-success">2</span>
<span class="badge badge-warning">4</span>
<span class="badge badge-important">6</span>
<span class="badge badge-info">8</span>
<span class="badge badge-inverse">10</span>
```

Alert

Wrap any text and an optional dismiss button in .alert for a basic warning alert message.

```
<div class="alert alert-block alert-error">
    <button type="button" class="close" data-dismiss="alert">&times;</button>
    This is an alert-error with extra padding (alert-block)
</div>

<div class="alert alert-error">
    <button type="button" class="close" data-dismiss="alert">&times;</button>
    This is alert-error
</div>

<div class="alert alert-success">
    <button type="button" class="close" data-dismiss="alert">&times;</button>
    This is alert-success
</div>

<div class="alert alert-info">
    <button type="button" class="close" data-dismiss="alert">&times;</button>
    This is alert-info
</div>
```

Table

```
<table class="table table-striped table-bordered table-hover table-condensed">
<caption>Table Caption here</caption>
<thead>
<tr>
<th>Country</th>
<th>Capital</th>
</tr>
</thead>
<tbody>
<tr class="success">
<td>Philippines</td>
<td>Manila</td>
</tr>

<tr class="error">
<td>United States of America</td>
<td>Washington, D.C.</td>
</tr>

<tr class="warning">
<td>Japan</td>
<td>Tokyo</td>
</tr>

<tr class="info">
<td>Canada</td>
<td>Ottawa</td>
</tr>

</tbody>
</table>
```

Buttons

```
<p>
<button class="btn btn-large btn-primary" type="button">Large button</button>
<button class="btn btn-large" type="button">Large button</button>
</p>
<p>
<button class="btn btn-info" type="button">Default button</button>
<button class="btn" type="button">Default button</button>
</p>
<p>
<button class="btn btn-small btn-success" type="button">Small button</button>
<button class="btn btn-small" type="button">Small button</button>
</p>
<p>
<button class="btn btn-mini btn-warning" type="button">Mini button</button>
<button class="btn btn-mini disabled" type="button">Mini button disabled</button>
<button class="btn btn-mini btn-inverse" type="button">Mini button inverse</button>
</p>
<button class="btn btn-block btn-danger" type="button">Block-level button</button>
```

Toggle

```
<!-- courtesy of http://getboilerstrap.com/bootstrap-styles/ -->
<a class="btn" data-toggle="collapse" data-target="#toggle-demo">Toggle Section</a>
</p>
<div id="toggle-demo" class="collapse">
  <div class="hero-unit">
```

```
    <h1 class="center">Toggle Demo</h1>
  </div>
</div>
```

Modal

```
<!-- courtesy of http://getboilerstrap.com/bootstrap-styles/ -->
  <a href="#demo-modal" role="button" class="btn btn-success" data-toggle="modal">Launch Modal</a>

<div id="demo-modal" class="modal hide fade in" role="content" aria-hidden="true">
  <div class="modal-header">
    <button type="button" class="close" data-dismiss="modal" aria-hidden="true">&times;</button>
    <h1 class="modal-h1-title">Demo Modal</h1>
  </div>
  <div class="modal-body search-modal row-fluid">
    <p>Oh hello!</p>
  </div>
  <div class="modal-footer">
    <a href="#" class="btn btn-primary" data-dismiss="modal" aria-hidden="true" title="Click to dismi
  </div>
</div>
```

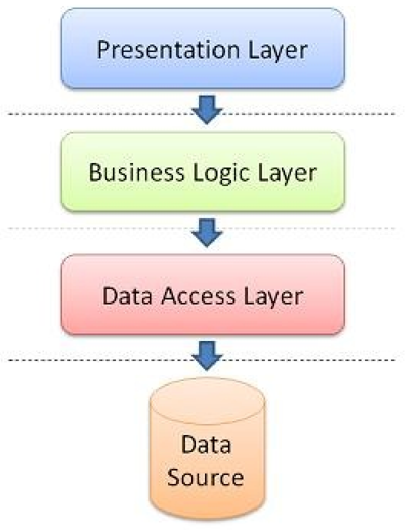
Part III. Back-End Development

Back-end development refers to server-side programming which comprises

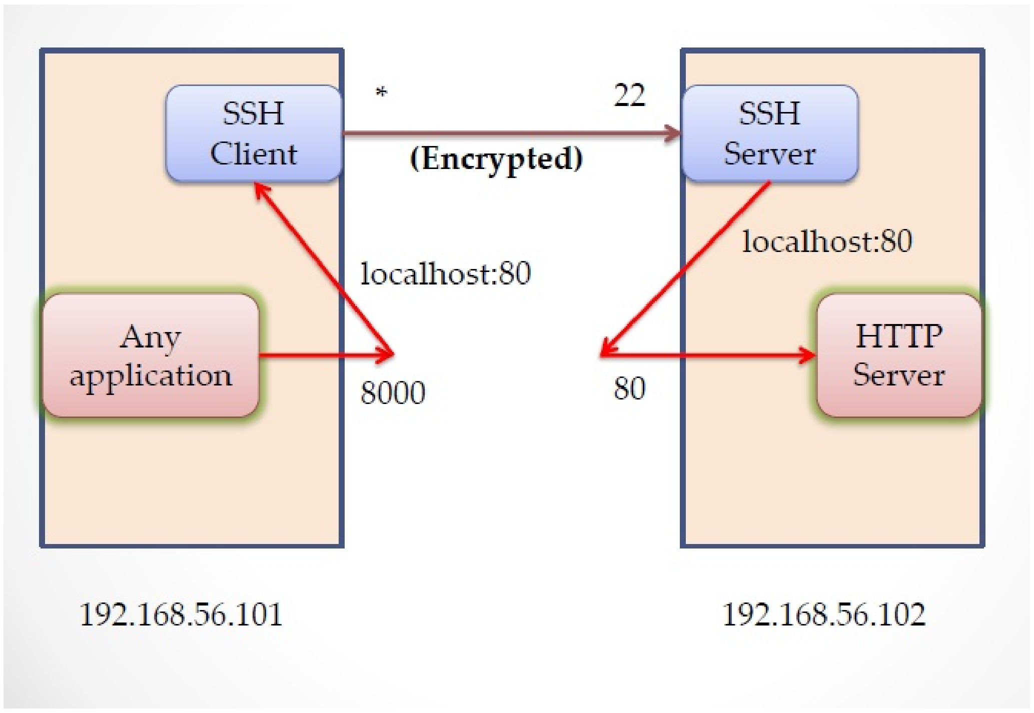
1. the Web application (logic tier) and
2. the relational and/or non-relational NoSQL database (database tier).

The simplest setup is the so-called LAMP (Linux, Apache, MySQL and PHP/Python/Perl) installed on a single computer (or its equivalent in Windows or other operating system).

Logic Tier



Rule 1: Export services via port binding



This is Rule VII from The Twelve-Factor App. That is, Web apps need to use a reverse proxy like nginx or the like. To see why this important, let me give you a brief background:

Historically, Web apps are executed inside a Web server container. For example, PHP apps might run as a module inside Apache HTTPD, or Java apps might run inside Tomcat (courtesy: Twelve Factor App). The Web apps are tightly coupled with a Web server.

Why you should decouple a Web app from a Web server?

1. Efficiency

From Peter Smith, author of "Professional Website Performance":

Nginx (pronounced “engine X”) appeared on the scene in the mid-2000s, and although initial uptake was slow because of lack of documentation, it has quickly gained in popularity. Many surveys now show it to be the third most used web server (behind Apache and IIS), and it is particularly popular for high-traffic sites. Nginx’s structure is notably different from Apache’s. As you have seen, Apache uses processes or threads (depending on the MPM used), each of which use significant resources (not to mention the overhead involved in creating and destroying child processes). By contrast, Nginx uses an asynchronous, event-driven model, which removes the need for each request to be handled as a separate child/thread. Instead there is a single master process with one or more worker processes.

The performance impact of this change is huge. Whereas each additional Apache process consumes an extra 4 MB or 5 MB (if the likes of mod_php have been loaded, this figure will be more like 15+MB) of memory, the impact of additional concurrent requests on Nginx is tiny. The result is that Nginx can handle a huge number of concurrent requests — much more than Apache, even with the
newer threaded MPM.

2. Separation of concerns

- Static apps (client-side)

When you decouple the presentation layer of a Web app from its business logic layer, the former can be served simply as static files and the latter can be architected as Web services (RESTful). Better yet, front-end development (HTML/CSS/JavaScript) can be written by one team and back-end development by another. Static files are downloaded once and can be cached resulting in faster performance compared with server-side pages (like ASP/JSP/PHP) that are dynamically rendered on the client-side. See staticapps.org for more information.

- Web services (back-end)

With Web services, you are no longer confined with the constraints of a Web server. You are no longer thinking in terms of Web sites. Instead, you design your back-end such that you can easily replace your HTML front-end without breaking functionality. Web services go beyond Web sites. Just look at Amazon Web Services. It redefined IT infrastructure by exposing compute/network/storage functionality as programmable Web APIs. Its AWS management console is just a window to what lies underneath.

Now if you're thinking Web application development has gotten more difficult with Web services, think again. You don't have to assume the role of a Web server because there is a reverse proxy. You can delegate serving static files to a Web server to offload tasks with your backend Web apps. However, you are now concerned with health checks of your backend apps (like restarting crashed apps). Fortunately, there are open source tools to do just that (see Hipache for example).

Rule 2: Minimize control logic, maximize application logic

Computing is basically I-P-O.

I = input

P = process

O = output

The process part is composed of two components:

1. Control logic - examples include the main program in C, Go or Node.js
2. Application logic - libraries, plugins, packages, modules

The control logic controls the overall state flow of the system so it is imperative to minimize it as much as possible (less moving parts). On the other hand, the application logic or modules should function as black boxes at the program level. You don't care how it works internally, you only care about input parameters and its corresponding output. Modules are simply being invoked by the control logic, no more no less.

A Tale of Two Programming Paradigms

As far as programs and systems are concerned, there are only two programming paradigms: imperative and declarative programming.

1. Imperative programming is a low-level programming paradigm in contrast with declarative programming (like functional programming or flow-based programming) which lets you think in terms of more high-level abstractions

2. Imperative programming focuses on the how, declarative programming focuses on the what. As such, they tackle modularity (breaking into parts) and composition (gluing the parts together) from two different perspectives.

3. Declarative programming does not completely eliminate state (only keeps it to a minimum)

4. Turing machine is the basis of imperative programming while lambda calculus is the basis of functional programming

5. Programming paradigms all exhibit structured programming and they all promote modularity. The differences are in the implementation.

Structured Programming

- Sequence - with imperative programming, you call functions either by value or by reference. With functional programming, it is call-by-need (aka lazy evaluation).

- Selection - "if" statement is executed in imperative programming. In functional programming, selection is an expression (which is converted to a value)

- Loop - with imperative programming, there are "for", "while" statements and the like. With functional programming, the functionality in a loop is abstracted away (see function as abstraction at

EloquentJavaScript.net). For example, Underscore.js is an FP library for data while Bacon.js is an FP library for events.

Why Functional Programming Matters (courtesy of John Hughes)

Functional programs contain no assignment statements, so variables, once given a value, never change. More generally, functional programs contain no side-effects at all. A function call can have no effect other than to compute its result. This eliminates a major source of bugs, and also makes the order of execution irrelevant - since no side-effect can change the value of an expression, it can be evaluated at any time. This relieves the programmer of the burden of prescribing the flow of control. Since expressions can be evaluated at any time, one can freely replace variables by their values and vice versa - that is, programs are “referentially transparent”.

Control Flow as a Graph

Declarative programming (like flow-based programming and functional programming) lends to a programming design akin to a graph in a whiteboard. To quote Henri Bergius,

> the logic of the software is designed as a graph — much like a flowchart — and stays as a graph.

Rule 3: Concurrency is not parallelism

Do not confuse concurrency with parallelism. The Go language blog has an excellent post:

In programming, concurrency is the composition of independently executing processes, while parallelism is the simultaneous execution of (possibly related) computations. Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once.

In other words, **concurrency** refers to a **single program that executes asynchronous processes**

- regardless whether it is single-threaded or multi-threaded and
- regardless whether it runs on a single-core or multi-core CPU

Concurrency is not about threads or CPU. It is all about whether you execute processes in an asynchronous fashion. In the case of Node.js, since it is using a single-threaded event loop, you are advised not to block the event loop for input/output operations or network calls. In the case of Go language, concurrency can be achieved using its built-in high-level constructs called goroutines and channels. Go multiplexes user-level goroutines onto OS-level threads.

But here is the difference between concurrency in Node and Go. With Node, you are **manually** managing concurrency through the use of callbacks, promises, async library or what have you. With Go, the Go runtime

automatically manages concurrency so you can write blocking operations without callback as long as you put it in a goroutine and optionally use a channel for its output if need be.

Either way, asynchronous programming is easier to reason about than thinking in terms of multithreading.

On the other hand, **parallelism** is about **executing multiple identical processes**

- regardless whether it runs on a single computer with multi-core CPU or multiple computers with single/multi-core CPU
- regardless whether it is concurrent process or not

Of course, parallelism does not apply to a single computer with a single-core CPU but it does apply in the context of multiple computers regardless of number of CPU as long as the above-mentioned condition is met.

Parallelism is best described with N-tier architecture where programs/applications are deployed in parallel at the application/logic tier to achieve high throughput and high availability.

In short, concurrency is about asynchronous processes while parallelism is about executing multiple identical processes simultaneously.

Rule 4: Use interface, not leaky abstraction

The key to program modularity and extensibility is interface, not leaky abstraction. Software must be either a kernel or a component in the form of plugins, modules, libraries or packages.

To illustrate, I will give examples of interface:

- operating system - interface to the physical machine
- hypervisor, container (Docker, Solaris Zones) - interface to the operating system
- SQL - interface to relational databases
- NPM (Node Package Manager) - module interface in Node
- jQuery - interface for DOM, AJAX, etc
- Browserify - module interface to client-side JavaScript
- composition - interface inheritance (see composition vs implementation inheritance below)

Examples of leaky abstraction

- client-side frameworks - Angular, Ember - leaks to JavaScript
- ORM (object/relational mapping) - leaks to SQL
- .NET, JVM - leaks to operating system

- implementation inheritance - ask the Go language designers

The problem with leaky abstraction is that they only provide a facade instead of a well-designed interface to the underlying software. There are two issues with this scenario.

- the underlying software is a moving target so the leaky abstraction must keep up with the changes, so now you have to track and maintain two different sets of software
- backward incompatibility is a fact of life in the software world but that is not the point. The issue is, leaky abstraction won't solve all use cases so for edge cases, you have no option but to dig the underlying software

But **leaky abstractions are not the same as tight-coupling though they are related**.

The direction of "eventual functionality" between these two are just polar opposites.

For **leaky abstraction**, its eventual functionality is **downward**.

To illustrate, when a leaky abstraction cannot provide the functionality you want and it does not offer a way to write an extension, you are forced to dig deeper below the layer of abstraction. For example, when an ORM does not cover your use case, you will eventually resort to SQL, so why not just use SQL in the first place? The notion of abstracting away many relational databases is a fallacy. That is not going to happen because your application is tightly coupled to your data store. In case of NoSQL, why not just use its native query language? The choice of data storage software is a high-level architectural decision that is left to those who know what they are doing.

Another example of leaky abstractions are client-side frameworks (CSF) like Angular and Ember. Ok, there are extensibility avenues but plugins designed for one framework are not going to work with another. You have to port them. When there is porting involved, there are two issues:

1. lock-in - now you are tightly coupled not just programmatically but mentally as well
2. interoperability - when plugins are not interoperable, turf wars are inevitable

In contrast, consider the elegant solution of interface compared with leaky abstraction.

For **interface**, its eventual functionality is **upward**.

> I will give two examples why an interface trumps leaky abstraction.
>
> Ansible, a configuration management tool

For starters, Ansible is written in Python and was originally designed to manage Linux machines. Management is being done from a control host machine running Linux (which is akin to military command and control). When Linux functionality stabilizes, it began to consider managing Windows as well. Instead of using the Windows port of Python, it retained the native Linux OS for the control host. So managing Windows environment is being done from the same Linux control host that manages Linux machines.

Here is the crucial decision: Ansible did not abstract away the differences between Linux and Windows versions of Python. Instead, it adapted Windows by utilizing Windows native PowerShell to talk to the same Linux control interface.

So Ansible standard modules that work in Linux also work seamlessly with Windows. Of course, there are going to be Windows-only modules but that is not the point. My point is, a well-designed interface is the key to avoid lock-in while maintaining program modularity, extensibility and interoperability.

The benefits of interface trickle down to Browserify, jQuery, NPM, SQL, Docker container, Xen, KVM, etc. Modules or plugins can be written once and deployed anywhere, no porting required although with minor differences among operating systems.

If you think there is tight coupling among component versions, you are right. It is called API versioning but it has nothing to do with issues given above.

CommonJS as implemented in NPM

Building large-scale JavaScript applications requires a module system that manages dependencies. But JavaScript was designed in a hurry so a module system is just an afterthought.

The community came up with two major specifications:

- AMD
- CommonJS

According to Addy Osmani, AMD adopts a browser-first approach to tackle the problems of module system and dependency management. CommonJS on the other hand takes the server-first approach.

Node.js adopted the CommonJS specification and implemented it with NPM (Node Package Manager). NPM solves the problem elegantly and has an ever growing number of modules (called packages). At the time of writing, the NPM repository has over 90,000 packages. The fact that Node included NPM as a built-in module says a lot about its architecture and implementation.

On the other hand, these are the experience of those using AMD.

- Journey from RequireJS to Browserify

- Why I Stopped using AMD
- AMD is not the answer
- Browserify: My new choice for modules in a browser

The list goes on and on. As Einstein said, "In theory, theory and practice are the same. In practice, they are not". Well, he is damned right about AMD.

Let's dissect the real issues.

First, dependency management is hard. You should not do it manually. Let the algorithm do it for you automatically using a well-defined interface (syntax) and a few rules
Second, compare NPM repository with Jam
Third, consider the case of using the actual module. If there is too much ceremony as Tom Dale pointed out, do not count me in

Now, consider how NPM solved those issues intuitively.

- NPM is a package manager with a few rules
- NPM has a bustling repository
- NPM has a simple syntax when importing modules

So back to the issue of interface versus leaky abstraction.

Both CommonJS and AMD are specifications. NPM and RequireJS are implementations of CommonJS and AMD respectively.

So what's the problem?

The problem is that AMD is a leaky abstraction because dependency management is based on configuration. With NPM, you can segment your business logic into folders under node_modules and it will resolve the dependencies automatically.

So how can you use those NPM modules in the browser?

Simple. Through Browserify!

You see, Browserify does not have to reinvent the 3-in-1 awesomeness of NPM (package manager, repository and clean interface).

> Since Browserify implements the Node.js module resolve algorithm, you can use the same NPM modules in the server and export them for use in the browser.

This is what I mean by interface. An interface is not concerned with the underlying implementation. If the syntax or interface being used by NPM modules works in the server, it should also work in the browser. And that's what Browserify does.

> Composition vs implementation inheritance

With composition, you talk to an interface, not the object itself.

Excerpt from Golang:

Object-oriented programming provides a powerful insight: that the behavior of data can be generalized independently of the representation of that data. The model works best when the behavior (method set) is fixed, but once you subclass a type and add a method, the behaviors are no longer identical. If instead the set of behaviors is fixed, such as in Go's statically defined interfaces, the uniformity of behavior enables data and programs to be composed uniformly, orthogonally, and safely.

Fortunately, JavaScript eschewed class-based OOP in favor of prototype-based paradigm (see problem of categorization). And Go does not have type hierarchy.

Composition lends itself to better interfaces rather than dependency injection.

Rule 5: Scale out the logic tier, scale up the storage tier

The logic tier is the business logic. In a three tier architecture, this comprises the Web applications. As such, Web apps regardless if they are deployed to physical machines or as virtual machines are expected to be decommissioned anytime. Thus, in the event of system maintenance or downtime, the continuity of online business is not disrupted.

Scaling out means deploying your apps to commodity hardware so you have three options:

- physicalization - deploy to physical machines
- hardware-level virtualization like KVM or Xen
- OS-level virtualization like Docker or SmartOS Zones

On the other hand, when I say scaling up, I am not referring to big fat machines like those being sold by the big boys. Instead, I am referring to commodity hardware that have larger memory and faster CPU compared to their little counterpart in the logic tier.

RDMBS and distributed storage system like Ceph need to be installed on dedicated bare-metal servers so these are mean and brawny machines.

Rule 6: Your tool does matter

Your choice of programming language reflects the expediency of youth versus the wisdom of maturity. If Java, Python, PHP, Ruby, JVM and .NET all led you to class-based OOP hell, it's time you break loose and onto the promised land of functional programming (FP). OOP is like taking the blue pill. OOP has led you to a make believe world, a misdirection, a sleight of hand, a reality distortion field.

But you don't have to take my word for it. You may have experienced OOP and it gets the job done. Well, that is the nature of software. It may have solved your problems in the small scale but at what cost? What are the tradeoffs? Development complexity? Software is intangible so you better look out for things that are not directly measurable but affects the bottom line which is developer productivity.

Now, if you think functional programming is a silver bullet, you are missing the point of FP. Functional programming does not eliminate imperative programming (IP). Rather, the former complements the latter. Moreover, functional programming is a mere subset of declarative programming paradigm.

If we adopt JavaScript and Go as the first programming languages to teach our students, we can surely defeat The PHP Singularity now and into the future.

Which leads us to a corollary:

> IT is 20% software development and 80% operations.

You may be using the best tools for software development but it won't shield you from the larger problem of operations. Operations include storage, networking and computing, the backend stuff that are invisible to users behind the graphical user interface. Mastery of operations is the reason why you cannot simply topple Facebook or Twitter or Google or Amazon because software development is just a small part of the overall picture.

Rule 7: Separate data structures from functions

OOP is the legacy of the Gang of Four along with JVM, .NET and Ruby on Rails. I recommend that you read Steve Yegge's article "Execution in the Kingdom of Nouns". This is the reason why you should not adopt class-based OOP to Node.js and Go language. And this is also the reason why you should not use stored procedures with your favorite relational database management system. There is a reason why computing is input - process - output. Input and output are data while code is process.

I will quote Joe Armstrong (who designed Erlang) from his article "Why OO Sucks":

> Objects bind functions and data structures together in indivisible units. I think this is a **fundamental error**

> since functions and data structures belong in totally different worlds. Why is this?

Functions do things. They have inputs and outputs. The inputs and outputs are data structures, which get changed by the functions. In most languages functions are built from sequences of imperatives: “Do this and then that …” to understand functions you have to understand the order in which things get done (In lazy FPLs and logical languages this restriction is relaxed).

> Data structures just are. They don’t do anything. They are intrinsically declarative. “Understanding” a data structure is a lot easier than “understanding” a function.

Functions are understood as black boxes that transform inputs to outputs. If I understand the input and the output then I have understood the function. This does not mean to say that I could have written the function.

Functions are usually “understood” by observing that they are the things in a computational system whose job is to transfer data structures of type T1 into data structure of type T2.

Since functions and data structures are completely different types of animal it is fundamentally incorrect to lock them up in the same cage.

Rule 8: Software development is "anything goes"

Don't subscribe to Agile Software Development or any software development methodology for that matter.

The best methodology is no methodology at all.

Why?

Because software development at the individual level is "anything goes".

> "Regression testing"? What's that? If it compiles, it is good; if it boots up, it is perfect. - LINUS TORVALDS
>
> We are tired of XP, Scrum, Kanban, Waterfall, Software Craftsmanship (aka XP-Lite) and anything else getting in the way of...Programming...We are tired of being told we're autistic idiots who need to be manipulated to work in a Forced Pair Programming chain gang without any time to be creative because none of the 10 managers on the project can do... Programming...M!@#$%^&*() -ZED SHAW
>
> Don't tell people how to do things, tell them what to do and let them surprise you with their results. - GEORGE PATTON

Of course, the top management all have their plans, their schemes. They want those accomplishment reports and other management tools. They want to show the client that the project is making progress so the client can pay on schedule. At the team level, there is a test-driven engineering practice you need to adhere to. There is nothing wrong with that.

But at the individual level, programmers don't want to be micro-managed.

Programmers are like The Joker (at least in front of their computer).

> Do I really look like a guy with a plan? You know what I am? I'm a dog chasing cars. I wouldn't know what to do with one if I caught it. You know? I just do things. The Mob has plans. The cops have plans... You know, they're schemers. Schemers trying to control their little worlds. I'm not a schemer. I try to show the schemers...how pathetic their attempts to control things really are.

Well, you cannot blame Zed Shaw against those control freaks. At least unlike the morally deranged Batman nemesis!

Rule 9: Git is for version control, not deployment

Git is for developers to manage their code, not to deploy code. PaaS (platform as a service) is a black box unless you understand that you wrestle control with the provider and trust their service.

A black box is worse than a leaky abstraction. While an abstraction lets you break the walls if you need to, a black box effectively locks you out of a particular software or service. This is the knowledge asymmetry that the provider wants to leverage since you need to go for your checkbook just to unlock what you need, instead of doing it yourself.

Rule 10: Understand the problem domain first

Understanding the problem domain is more important than the solution space.

Tim Bryce said it best:

> No amount of elegant programming or technology will solve a problem if it is improperly specified or understood to begin with.
>
> Project management is a philosophy of management, not a tool or technique.

Rule 11: Simplicity is the ultimate sophistication

Simplicity in analysis, design and implementation must be the hallmark of your application. It must exhibit simplicity in API, tools and usability. You must give ample space to your user interface so that the user can breathe. You must provide a stable application programming interface such that the kernel may change internally without breaking compatibility. Avoid clutter.

To paraphrase Antoine de Saint-Exupery, design is achieved not when there is nothing left to add, but when there is nothing left to take away.

Rule 12: Do not be afraid of the command line

For desktop users, Windows or Linux distros may be your GUI (graphical user interface). For developers, you may have your favorite code editors. System operations may have a Web-based GUI but only for mundane or common tasks. For corner cases, a CLI (command line interface) is your only tool.

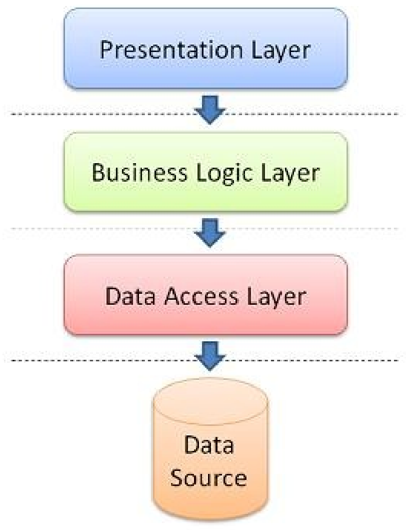
For Web developers, operations is an inevitable task. You need to know how it's working under the hood. You need to learn beyond front-end development. When I say command line, I am referring to the command line interface of the underlying operating system. Once you get the hang of it, a configuration management tool like Ansible will come in handy when the going gets tough.

The command line is your window to the underlying machine, be it physical or virtual machine. If you can't see it, you are under the mercy of schemers.

As Giuseppe Maxia has said,

> Don't be a rookie forever - Be in command! (line)

Database Development



Non-relational database development is beyond the scope of this book. Instead, we will focus on relational database since it is a fundamental technology and covers most use cases for business applications.

Basics of RDBMS

- Relational database relationships (one-to-one, one-to-many, many-to-many

- Read performance is about indexes (like reading an index in a book)
- SQL joins
- Transactions (it's all or nothing, ACID constraints)
- SQL DDL (CREATE, ALTER TABLE, etc) vs SQL DML (INSERT, UPDATE, DELETE)
- Database security (field encryption, SQL injection, etc)
- Working Dataset (Read-write) vs Archived Dataset (Read-only)
- User administration (user permissions vs role-based ACL)

Why Relational Database?

- ACID Guarantees - banks need this "all-or-nothing" guarantee to transactions. Either money was deposited or not, there is no limbo state
- The relational model is declarative - you use SQL to define, manipulate and query data
- Separation of concerns - unlike object-relational database (e.g. PostgreSQL), relational database simply store data. In my view, objects, user-defined functions, triggers and stored procedures should be relegated to the logic layer, not at the database layer. Moreover, blobs or objects can be stored and retrieved more efficiently with REST-based services like S3, Swift and the like
- Easy to learn and use - just look at MySQL, the world's most popular open source database

Isolation Levels

From High Performance MySQL, Third Edition:

Isolation is more complex than it looks. The SQL standard defines four isolation levels, with specific rules for which changes are and aren't visible inside and outside a transaction. Lower isolation levels typically allow higher concurrency and have lower overhead.

READ UNCOMMITTED

In the READ UNCOMMITTED isolation level, transactions can view the results of uncommitted transactions. At this level, many problems can occur unless you really, really know what you are doing and have a good reason for doing it. This level is rarely used in practice, because its performance isn't much better than the other levels, which have many advantages. Reading uncommitted data is also known as a dirty read.

READ COMMITTED

The default isolation level for most database systems (but not MySQL!) is READ COMMITTED . It satisfies the simple definition of isolation used earlier: a transaction will see only those changes made by transactions that were already committed when it began, and its changes won't be visible to others until it has committed.

This level still allows what’s known as a nonrepeatable read. This means you can run the same statement twice and see different data.

REPEATABLE READ

REPEATABLE READ solves the problems that READ UNCOMMITTED allows. It guarantees that any rows a transaction reads will “look the same” in subsequent reads within the same transaction, but in theory it still allows another tricky problem: phantom reads. Simply put, a phantom read can happen when you select some range of rows, another transaction inserts a new row into the range, and then you select the same range again; you will then see the new “phantom” row. InnoDB and XtraDB solve the phantom read problem with multiversion concurrency control.

REPEATABLE READ is MySQL’s default transaction isolation level.

SERIALIZABLE

The highest level of isolation, SERIALIZABLE , solves the phantom read problem by forcing transactions to be ordered so that they can’t possibly conflict. In a nutshell, SERIALIZABLE places a lock on every row it reads. At this level, a lot of timeouts and lock contention can occur. We’ve rarely seen people use this isolation level, but your application’s needs might force you to accept the decreased concurrency in favor of the data stability that results.

Limitations of Relational Database

Don't abuse the relational model - here is Rule 14 from *"Scalability Rules" book by Martin Abbott and Michael Fisher*:

- **What**: Use relational databases when you need ACID properties to maintain relationships between your data. For other data storage needs consider more appropriate tools.
- **When to use**: When you are introducing new data or data structures into the architecture of a system.
- **How to use**: Consider the data volume, amount of storage, response time requirements, relationships, and other factors to choose the most appropriate storage tool.
- **Why**: An RDBMS provides great transactional integrity but is more difficult to scale, costs more, and has lower availability than many other storage options.
- **Key takeaways**: Use the right storage tool for your data. Don’t get lured into sticking everything in a relational database just because you are comfortable accessing data in a database.

Of course, you may use NoSQL for non-relational data models:

1. Document-oriented - MongoDB, CouchDB
2. Graph - Neo4j
3. Column-oriented - HBase
4. Key-value - Riak, Redis, DynamoDB

Basics of Relational Database

- Relational database management system - this is the software like MySQL that manages relational databases
- Database - this is the container that houses the relational tables
- Table - like in a spreadsheet, you may visualize a table as the container of vertical columns and horizontal rows of data. Think of a table as an entity
- Field (aka column) - like in compiled programming languages, a field is a kind of data type in a table, an attribute of an entity. Think property of an object in OOP. You may visualize a field or column as the vertical set of all data in a spreadsheet
- Row - you may visualize a row as the horizontal set of all data in a spreadsheet
- Null - according to Wikipedia,

The database structural query language SQL implements ternary logic as a means of handling comparisons with NULL field content. The original intent of NULL in SQL was to represent missing data in a database, i.e. the assumption that an actual value exists, but that the value is not currently recorded in the database.

Please refer to MySQL documentation on problems with NULL.

- Primary key - from Wikipedia, a field that uniquely identifies the characteristics of each row. For now, suffice to say that a primary key is auto-increment (that is, auto-increment automatically assigns an integer as unique key in a table). A primary key optionally can be a foreign key to a related table
- Foreign key - the primary key of a related table
- Index - like index in a book, an index is a data structure that facilitates searching for rows more efficiently
- One-to-one relationship - fields (or attributes) belong to one table only. For example, first name, last name, date of birth and so on belong to one entity only (person)

Customers
Firstname
Lastname

DOB

- One-to-many relationship - field in one table is recurring in a related table. For example, a state has many cities.

States Table	StateID
California	1
Portland	2

Cities Table	StateID
Los Angeles	1
San Francisco	1
San Jose	1
Palo Alto	1

- Many-to-many relationship - fields in table A and other fields in table B are contained in a third table C. For example, suppose we have table A (customers) and table B (products). To track which product a customer has subscribed, we need to create table C (accounts).

Customers
Peter
Paul
John

Products	ProductID
Savings	1
Checking	2
Housing Loan	3
Auto Loan	4

Accounts	ProductID
Peter	1
Peter	3
Paul	2
John	4
John	2

Without going into the technical detail of database normalization, we could summarize the basics of a sound relational model as follows:

1. Know your one-to-one relationship
2. Know your one-to-many relationship
3. Know your many-to-many relationship
4. That's it!

Use SQL or NoSQL where applicable, not ORM (object/relational mapper).

Why SQL?

1. First and foremost, ORM is a leaky abstraction because it won't solve all the data retrieval and data manipulation that SQL natively provides.
2. Second, it guarantees ACID transactions. If you don't need ACID guarantees, you can use NoSQL
3. Third, SQL is the single source of truth for relational databases, not multiple ORMs from multiple frameworks or libraries
4. Even Google marries SQL and NoSQL with its Spanner database
5. Ceph RBD puts an end to myth of RDBMS scalability (or any NewSQL product out there)

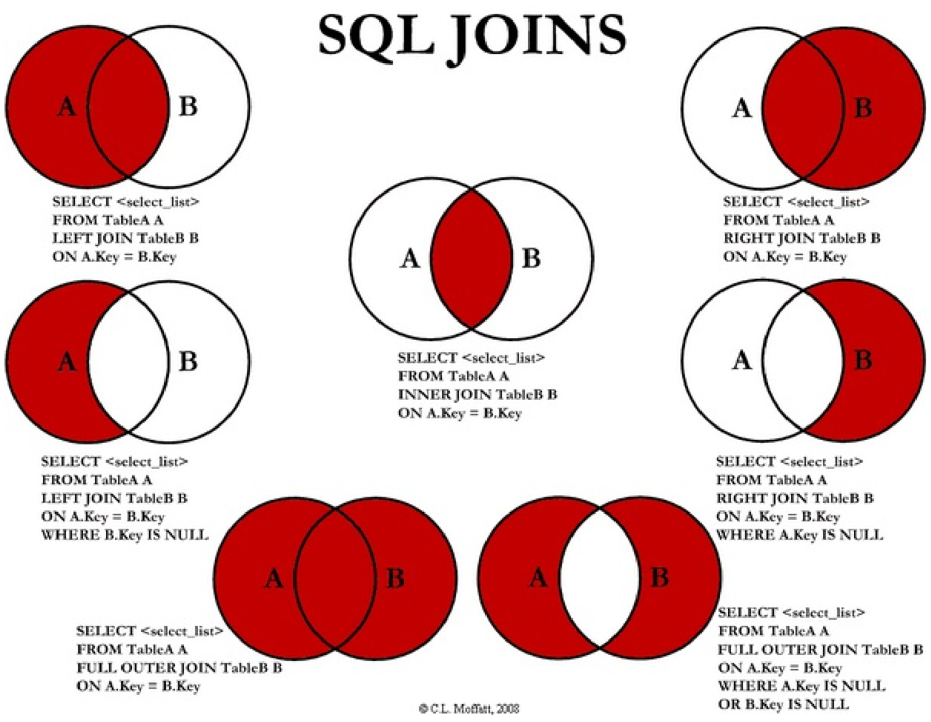
SQL Caveats

1. A basic knowledge of SQL indexing would squeeze more performance out of your queries
2. SQL is just a specification. The implementation varies with RDBMS software
3. A poor craftsman blames his tool (SQL)

MySQL Join

The following are my references on SQL joins.

- From CodingHorror by Jeff Atwood
- From CodeProject by C.L.Moffatt



```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```

```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

Here is my version derived from those above-mentioned articles with examples from MathIsFun.com

Suppose we have two MySQL tables of the same structure with id and name fields:

- Soccer Table
- Tennis Table

Now let's say that alex, casey, drew and hunter play Soccer:

- Soccer = {alex, casey, drew, hunter}

And casey, drew and jade play Tennis:

- Tennis = {casey, drew, jade}

INNER JOIN

```
select * from soccer s
inner join tennis t
on s.name = t.name
```

or you can issue this SQL statement and get the same result.

```
select * from soccer s, tennis t
```

```
where s.name = t.name
```

Result:

```
    id  name        id  name
------  ------  ------  ------
     3  casey        1  casey
     4  drew         2  drew
```

LEFT OUTER JOIN OR LEFT JOIN

```
select * from soccer s
left outer join tennis t
on s.name = t.name
```

Result:

```
    id  name        id  name
------  ------  ------  ------
     1  alex    (NULL)  (NULL)
     2  hunter  (NULL)  (NULL)
     3  casey        1  casey
     4  drew         2  drew
```

LEFT EXCLUDING JOIN

Soccer − Tennis = {alex, hunter}

```
select * from soccer s
left outer join tennis t
on s.name = t.name
where t.name IS NULL
```

Result:

```
    id  name        id  name
------  ------  ------  ------
     1  alex    (NULL)  (NULL)
     2  hunter  (NULL)  (NULL)
```

> Note: Left outer join simply means you retrieve all the rows from the left table **regardless** if there is matching record on the right table or not. Right outer join is basically the same thing as left outer join if you reverse the order of the table.

RIGHT EXCLUDING JOIN

```
select * from tennis t
right outer join soccer s
on t.name = s.name
where t.name IS NULL
```

Result:

```
    id  name        id  name
```

```
------  ------  ------  ------
(NULL)  (NULL)       1  alex
(NULL)  (NULL)       2  hunter
```

FULL OUTER JOIN OR UNION

> Note: There is no FULL OUTER JOIN syntax in MySQL but you can use UNION.

```
select * from soccer s
left outer join tennis t
on s.name = t.name

union

select * from soccer s
right outer join tennis t
on s.name = t.name
```

Result:

```
    id  name        id  name
------  ------  ------  ------
     1  alex    (NULL)  (NULL)
     2  hunter  (NULL)  (NULL)
     3  casey        1  casey
     4  drew         2  drew
(NULL)  (NULL)       3  jade
```

OUTER EXCLUDING JOIN

Outer excluding join = Full outer join - Inner join

```
select * from soccer s
left outer join tennis t
on s.name = t.name
where s.name IS NULL
or t.name IS NULL

union

select * from soccer s
right outer join tennis t
on s.name = t.name
where s.name IS NULL
or t.name IS NULL
```

Result:

```
    id  name        id  name
------  ------  ------  ------
     1  alex    (NULL)  (NULL)
     2  hunter  (NULL)  (NULL)
(NULL)  (NULL)       3  jade
```

MySQL to JSON

The GROUP_CONCAT function is useful to construct MySQL data into JSON (courtesy of Thomas Frank)

```
SELECT CONCAT("{",
GROUP_CONCAT('"',id,'":',
    CONCAT('{"firstname":','"',firstname,'",'),
    CONCAT('"lastname":','"',lastname,'"}')),
"}") as json
FROM client
```

Result:

```
{
    "4": {
        "firstname": "Peter",
        "lastname": "Cross"
    },
    "5": {
        "firstname": "Steve",
        "lastname": "Klein"
    },
    "6": {
        "firstname": "Mary",
        "lastname": "Smith"
    }
}
```