Beyond jQuery

It all started so quietly...

One line at a time.

No avoiding it now...

JavaScript is a first class citizen.

What does that mean?
How do we create modern web applications?
Applications are changing.

How do we embrace that?
How’d we get here?

In the beginning...

“I think there is a world market for maybe five computers.”

— Thomas J. Watson (attributed)
Computers were monsters.
Expensive.

Centrally managed.

Looked like this:
And we programmed using something like this:

Anyone remember those days?

Good times.

How did we access all that power?
Everyone’s favorite UI, the green screen.

The dumb terminal. If you’ve ever traveled...

Some advantages to this model though.
Terminals were cheap. Easy to maintain.

Didn’t have “compatibility” issues. But it was lacking something.

Beyond taste. The UIs weren’t exactly rich.
Not user friendly.

Eventually...

Computers got smaller.

They were dismissed as toys.

“First they ignore you, then they laugh at you, then they fight you, then you win.”

— Mahatma Gandhi (attributed)
And they weren’t very powerful. But the future was here.

And we started to create... Client server apps.

VB, PowerBuilder, Access. Richer UIs!
Take advantage of all that *power*.

But there were downsides.

DLLs.

Distribution.

Compatibility issues...
Ever experience that?  

But, change was afoot.

In the 90s, we discovered...  
The Internet.

And a revolution was born.  
We created thin clients...
That, ah, talked to powerful servers.

Remind you of anything?

But they were better than a green screen right?

Hmm.

We even kept the submit button...

And, we didn’t have client side issues to deal with.
Other than IE at least.

But, the UIs weren’t very rich.

Forms with holes.

#winning

Again.

Request/response paradigm.
And many people had experienced rich UIs.

Sorry.

Good for us... less good for users.

But then what happened?

And of course...
Ajax.

Enter the era of RIA.

Best of both worlds?

The “husky” client.

Easy distribution, rich (enough) interfaces.
Sensing a theme here...

Back and forth.

And we’re heading there again.

Now browsers are the PC.

They’re getting more powerful.
V8, Nitro, JägerMonkey.

And we’re asking them to do more.

HTML5.

Web Workers, Web Sockets, Offline.

REST.

JavaScript.
First class citizen.

We’re writing more of it too!

New possibilities.

And we have options.

We can break out of the request/response approach.

Click and wait?

http://alexmaccaw.co.uk/posts/async_ui
Enter asynchronous UIs.

Why?

Performance matters.

Well, perceived performance at least.

Milliseconds matter.

Amazon: 100ms delay reduces sales by 1%. 
400ms on Yahoo!?

5-9% drop in traffic.

500ms extra on Google?

Drops traffic by 20%.

It matters.

Embraces what we’re doing.

http://www.slideshare.net/stubbornella/designing-fast-websites-presentation
Provide structure to all that JavaScript.

I know what some of you are thinking...

Anything but JavaScript.

“Bad developers will move heaven and earth to do the wrong thing.”
— Glenn Vanderburg

Embrace it.

Partial refreshes, JSON, services.
Takes it a step further.

MVC.

Non-blocking UI.

Decouple requests from the UI.

Render view on client.

Push state to client.
Talk to server asynchronously.

Update the UI, then tell the server about it.

Wait. What?

Things will go wrong!

Yep.

What about validation?
Server could reject the change.

Client side validation.

Need to validate on the server too...

What if the server pukes?

Error handling.

Parallel requests?
Pipeline ajax requests.

Try to navigate off gmail with update pending...

It isn’t perfect!

But there are answers to many issues.

Why should we do this?

Better user experience.
Some things should be synchronous!

Need *some* feedback.

Gives us another tool.
How do we do it?  The server.

REST.  FTW.

Not request/response...  Finer grained.
May need support for web sockets.

Jetty, Node, Socket.IO...

The client.

State and view.

Preload data.

Server communication is asynchronous.
Update the client then tell the server what happened.

Opposite of what we’ve done for years.

Hmm, managing state on the client sounds hard.

Can be.

JavaScript is often... lacking in structure.

Probably want to use a library!
Typically built around MVC or MVP.

Backbone.js, Spine.js, Sammy.js, KnockoutJS...

List grows daily.

How do I know which one to use?

Play with them.

Compare them.

https://github.com/addyosmani/todomvc
So what does this look like? Let’s look at Backbone.js.

Backbone.js Very lightweight.

As in ~5 KB compressed. ~1300 lines.
Fully documented. Isn’t a “UI” framework.

Built for MVC JavaScript applications. Models, events, collections, views.

Controllers, persistence. Influenced by Ruby on Rails.
Data lives in models.

Not the DOM.

Changes to models trigger change events.

Views are notified of said changes to the model.

Update accordingly.

No more find stuff and change it - it just updates.
You’ll be coding to events.

Create models that extend Backbone.Model.

Your models inherit a ton of behavior.

- get/set
- has
- clear
- toJSON
- save
- validate
- clone
- changedAttributes
- previous
- ...

Backbone.Model
And much more!

Provides an empty validate method.

You provide implementation.

set() and save() halt on invalid data.

Provides a way of setting default values.

Backbone.Collection
Includes some fancy collection magic.

Sets of models.

Usually of a single model type.

Events fire when items in the collection change.

Also when items are added or removed.

Borrows from Underscore.js as well.
Gain some nifty iteration functions.

- add/remove
- get
- sort
- pluck
- parse
- fetch

And more.

Retrieve models via client IDs or model’s ID.

Collections can be ordered.

Provides a richer comparator concept.
<table>
<thead>
<tr>
<th>Also adds a <code>fetch</code> to retrieve collections from server.</th>
<th>Provide a URL endpoint.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backbone.View</td>
<td>Convention.</td>
</tr>
<tr>
<td>Not templates.</td>
<td>Often used with a template library.</td>
</tr>
</tbody>
</table>
Such as Mustache.js, Haml-js, or Eco.

Handle presentation.

Linked to a DOM element.

this.el

Can bind directly to an existing element.

Defaults to an empty div.
Bind a view’s render object to the change in a model. Instead of a series of queries and DOM updates.

Extend Backbone.View. Implement render.

Return the right HTML. Update el with said HTML.
Again, probably using a template library. Model has toJSON() to feed data to template.

Also gives an event hash. Easy way to bind to interesting events.

`"eventType selector": "callback"` Selector is optional.
Leave it off? Binds to e/.

Backbone.Router

The controller.

Web apps should be linkable & bookmarkable.

Backbone.Router helps.

Connects state to URL hashes.
History API can handle much of what we’d want.

Backbone fills in where browsers fall down.

Connects and routes pages.

Shocking.

- routes
- navigate

Works in conjunction with Backbone.history.
saveLocation()

Backbone.sync

Model updates need to get to the server.

When a model changes, Backbone informs server.

By default, makes a RESTful JSON request.

sync(method, model, [options])
If call succeeds, client side model is updated. Based on jQuery.

method - CRUD.

<table>
<thead>
<tr>
<th>CRUD</th>
<th>HTTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td>POST</td>
</tr>
<tr>
<td>read</td>
<td>GET</td>
</tr>
<tr>
<td>update</td>
<td>PUT</td>
</tr>
<tr>
<td>delete</td>
<td>DELETE</td>
</tr>
</tbody>
</table>

model - the thing that changed.

options - additional callbacks, ajax options.
Expects server to return updated attributes as JSON.

Save is asynchronous.

Free to bind to any of the ajax callbacks.

Can override to use local storage, WebSockets, etc.

There's an existing local storage adapter.

The server side.

https://github.com/jeromegn/Backbone.localStorage
Expects certain RESTful endpoints to exist.

<table>
<thead>
<tr>
<th>Method</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>/collection</td>
</tr>
<tr>
<td>GET</td>
<td>/collection</td>
</tr>
<tr>
<td>GET</td>
<td>/collection/id</td>
</tr>
<tr>
<td>PUT</td>
<td>/collection/id</td>
</tr>
<tr>
<td>DELETE</td>
<td>/collection/id</td>
</tr>
</tbody>
</table>

Backbone handles model serialization.

Endpoints should return model as JSON.

Putting it all together.

Several sample apps.
todos.js

http://documentcloud.github.com/backbone/docs/todos.html

window.Todo = Backbone.Model.extend({
  defaults: function() {
    return {
      done: false,
      order: Todos.nextOrder()
    },
  },
  toggle: function() {
    this.save({done: !this.get("done")});
  }
});

window.TodoList = Backbone.Collection.extend({
  model: Todo,
  localstorage: new Store("todos"),
  done: function() {
    return this.filter(function(todo) { return todo.get('done'); });
  },
});

window.TODOView = Backbone.View.extend({
  tagName: "li",
  template: _.template($("#item-template").html()),
  events: {
    "click .check" : "toggleDone",
    "dblclick div.todo-text" : "edit",
    "click span.todo-destroy" : "clear",
    "keypress .todo-input" : "updateOnEnter"
  },
  initialize: function() {
    this.model.bind('change', this.render, this);
    this.model.bind('destroy', this.remove, this);
  },
  render: function() {
    $(this.el).html($(this.template(this.model.toJSON())));
    this.setText();
    return this;
  }
});

Works like this:
Also an “app” level view.

Notice it uses some jQuery! Not so hard!

Wine Cellar. But we use jQuery.

You bet!

How many of you use jQuery today?

So do we.

Rocking good library.

Some would say it’s a bit heavyweight.
Like, say, the jQuery core team.

1.7 deprecated a lot.

https://groups.google.com/forum/#!topic/jquery-bugs-team/r7rGk6eAAxI/discussion

Are you using all of it? Probably not.

Dragging along a lot of excess baggage.

Is that an issue?
It depends.

For desktop users?
Probably not.

But what about mobile?
Extra bits matter.

Especially on certain cell networks...

Movement today towards micro frameworks.
Do one or two things... Really well.

Unix model! Small, loosely coupled.

Pros: If you’re not using it, you don’t need to push it.
Smaller learning curve.  

Constraints shall set you free.

Cons: 

May need multiple libraries.

Reinvent the wheel.  

Existing skill sets.
Three rough categories.

jQuery alternatives.

JavaScript MVC.

JavaScript alternatives.

Some libraries build on jQuery.

Does anyone use Backbone?
Yes!

LinkedIn Mobile, DocumentCloud, Flow.

Foursquare, Khan Academy, Do, Posterous.

Groupon, Basecamp Mobile, Stripe, Pandora.

Soundcloud, Code School, SeatGeek, Kicksend.

Decide, Trello, QuietWrite.
More and more.

Your competitors?

They won’t tell you why they’re beating you.

This isn’t for everyone.

You will write JavaScript.
Sorry.

Requires a rethinking of your application.

Probably can’t “port”.

It is different.

It is new.

As in 0.9.1.
Evolving.

JavaScript Web Applications.

Start thinking about it.

Where would it fit for you?

It can be done!
Be aware of the alternatives. What are they good for?

What shouldn’t they be used for? How might they fit in your world?

Image Credits

- http://www.flickr.com/photos/marc_smith/6246956530/
- http://www.flickr.com/photos/marc_smith/6246433643/
- http://www.flickr.com/photos/sylvar/73589278/
- http://www.flickr.com/photos/bobjagendorf/5493860578/

Thanks!

Nathaniel T. Schutta
@ntschutta