Examples

A simple download

It isn't really what Oboe is for but you can use it as a simple AJAX library. This might be good to drop it into an existing application so you can refactor later to make it progressive.

```
oboe('/myapp/things.json')
   .done(function(things) {

      // we got it
   })
   .fail(function() {

      // we don't got it
   });
```

Using objects from the JSON stream

Say we have a resource called things.json that we need to fetch over AJAX:

```
{
   "foods": [
      {"name":"aubergine",    "colour":"purple"},
      {"name":"apple",        "colour":"red"},
      {"name":"nuts",         "colour":"brown"}
   ],
   "badThings": [
      {"name":"poison",       "colour":"pink"},
      {"name":"broken_glass", "colour":"green"}
   ]
}
```

In our webapp we want to download the foods and show them in a webpage. We aren't showing the non-foods here so we won't wait for them to be loaded:

```
oboe('/myapp/things.json')
   .node('foods.*', function( foodThing ){

      // This callback will be called everytime a new object is
      // found in the foods array. Oboe won't wait for the
      // download to finish first.
```

```
      console.log( foodThing.name, 'is', foodThing.colour);
   })
   .node('badThings.*', function( badThing ){

      console.log( 'Danger! stay away from', badThings.name);
   })
   .done(function(things){

      console.log(
         'there are', things.foods.length, 'things to eat',
         'and', things.nonFoods.length, 'to avoid');
   });
```

Hanging up when we have what we need

We can improve on the example above. Since we only care about the foods object and not the non-foods we can hang up as soon as we have the foods, reducing our precious download footprint.

```
oboe('/myapp/things.json')
   .node({
      'foods.*': function( foodObject ){

         alert('go ahead and eat some ' + foodObject.name);
      },
      'foods': function(){
         this.abort();
      }
   });
```

Duck typing

Sometimes it is more useful to say *what you are trying to find* than *where you'd like to find it*. In these cases, duck typing is more useful than a specifier based on paths.

```
oboe('/myapp/things.json')
   .node('{name colour}', function( foodObject ) {
      // I'll get called for every object found that
      // has both a name and a colour
   };
```

Detecting strings, numbers

Want to detect strings or numbers instead of objects? Oboe doesn't care about the types in the json so the syntax is the same:

```
oboe('/myapp/socialgraph.json')
   .node({
      'name': function( name ){
         // do something with the name
      },
      'friends.*.name':function( friendsName ){
         // etc etc
      });
```

Reacting before we get the whole object

As well as `.node`, you can use `.path` to be notified when the path is first found, even though we don't yet know what will be found there. We might want to eagerly create elements before we have all the content to get them on the page as soon as possible.

```
var currentPersonElement;
oboe('people.json')
   .path('people.*', function(){
      // we don't have the person's details yet but we know we
      // found someone in the json stream. We can eagerly put
      // their div to the page and then fill it with whatever
      // other data we find:
      currentPersonElement = jQuery('<div class="person">');
      jQuery('#people').append(personDiv);
   })
   .node({
      'people.*.name': function( name ){
         // we just found out that person's name, lets add it
         // to their div:
         currentPersonElement.append('<span class="name">' + name + '</span>');
      },
      'people.*.email': function( email ){
         // we just found out this person has email, lets add
         // it to their div:
         currentPersonElement.append('<span class="email">' + email + '</span>');
      }
   });
```

Giving some visual feedback as a page is updating

If we're doing progressive rendering to go to a new page in a single-page web app, we probably want to put some kind of indication on the page as the parts load.

Let's provide some visual feedback that one area of the page is loading and remove it when we have data, no matter what else we get at the same time

I'll assume you already implemented a spinner

```
MyApp.showSpinner('#foods');

oboe('/myapp/things')
   .node({
      '!.foods.*': function( foodThing ){

         jQuery('#foods')
            .append('<div>')
            .text('it is safe to eat ' + foodThing.name);
      },
      '!.foods': function(){

         // Will be called when the whole foods array has
         // loaded. We've already wrote the DOM for each
         // item in this array above so we don't need to
         // use the items anymore, just hide the spinner:
         MyApp.hideSpinner('#foods');
      }
   });
```

The path parameter

The callback is also given the path to the node that it found in the json. It is sometimes preferable to register a wide-matching pattern and use the path parameter to decide what to do instead of

```
// JSON from the server side.
// Each top-level object is for a different module on the page.
{  "notifications":{
      "newNotifications": 5,
      "totalNotifications": 4
   },
   "messages": [
      {  "from":"Joe",
```

```
         "subject":"blah blah",
         "url":"messages/1"
      },
      {  "from":"Baz",
         "subject":"blah blah blah",
         "url":"messages/2"
      }
   ],
   "photos": {
      "new": [
         {  "title": "party",
            "url":"/photos/5",
            "peopleTagged":["Joe","Baz"]
         }
      ]
   }
   // ... other modules ...
}


oboe('http://mysocialsite.example.com/homepage')
   .node('!.*', function( moduleJson, path ){

      // This callback will be called with every direct child
      // of the root object but not the sub-objects therein.
      // Because we're coming off the root, the path argument
      // is a single-element array with the module name like
      // ['messages'] or ['photos']
      var moduleName = path[0];

      My.App
         .getModuleCalled(moduleName)
         .showNewData(moduleJson);
   });
```

Deregistering a callback

Calling `this.forget()` from inside a callback deregisters that listener.

```
// We have a list of items to plot on a map. We want to draw
// the first ten while they're loading. After that we want
// to store the rest in a model to be drawn later.

oboe('/listOfPlaces')
```

```
   .node('list.*', function( item, path ){
      var itemIndex = path[path.length-1];

      model.addItemToModel(item);
      view.drawItem(item);

      if( itemIndex == 10 ) {
         this.forget();
      }
   })
   .done(function( fullJson ){
      var undrawnItems = fullJson.list.slice(10);

      model.addItemsToModel(undrawnItems);
   });
```

Css4 style patterns

Sometimes when downloading an array of items it isn't very useful to be given each element individually. It is easier to integrate with libraries like Angular if you're given an array repeatedly whenever a new element is concatenated onto it.

Oboe supports css4-style selectors and gives them much the same meaning as in the proposed css level 4 selector spec.

If a term is prefixed with a dollar sign, instead of the element that matched, an element further up the parsed object tree will be given instead to the callback.

```
// the JSON from the server side looks like this:
{"people": [
   {"name":"Baz", "age":34, "email": "baz@example.com"}
   {"name":"Boz", "age":24}
   {"name":"Bax", "age":98, "email": "bax@example.com"}}
]}


// we are using Angular and have a controller:
function PeopleListCtrl($scope) {

   oboe('/myapp/things')
      .node('$people[*]', function( peopleLoadedSoFar ){

         // This callback will be called with a 1-length array,
         // a 2-length array, a 3-length array etc until the
         // whole thing is loaded (actually, the same array
```

```
         // with extra people objects pushed onto it) You can
         // put this array on the scope object if you're using
         // Angular and it will nicely re-render your list of
         // people.

         $scope.people = peopleLoadedSoFar;
      });
}
```

Like css4 stylesheets, this can also be used to express a 'containing' operator.

```
oboe('/myapp/things')
   .node('people.$*.email', function(personWithAnEmailAddress){

      // here we'll be called back with baz
      // and bax but not Boz.

   });
```

Streaming out HTML from express

Generating a streamed HTML response from a streamed JSON data service.

```
app.get('/foo', function(req, res){
   function writeHtml(err, html){
      res.write(html);
   }

   res.render('pageheader', writeHtml);

   oboe( my_stream )
      .node('items.*', function( item ){
          res.render('item', item, writeHtml);
      })
      .done(function() {
          res.render('pagefooter', writeHtml);
      })
});
```

Using Oboe with d3.js

Oboe works very nicely with d3.js to add content to a visualisation while the JSON downloads.

```
// get a (probably empty) d3 selection:
var things = d3.selectAll('rect.thing');

// Start downloading some data.
// Every time we see a new thing in the data stream, use
// d3 to add an element to our visualisation. This basic
// pattern should work for most visualistions built in d3.
oboe('/data/things')
   .node('$things.*', function( thingsArray ){

      things.data(thingsArray)
         .enter().append('svg:rect')
            .classed('thing', true)
            .attr(x, function(d){ return d.x })
            .attr(y, function(d){ return d.x })
            .attr(width, function(d){ return d.w })
            .attr(height, function(d){ return d.h })

      // no need to handle update or exit set here since
      // downloading is purely additive
   });
```

Reading from Node.js streams

Instead of giving a url you can pass any ReadableStream. To load from a local file you'd do this:

```
oboe( fs.createReadStream( '/home/me/secretPlans.json' ) )
   .on('node', {
      'schemes.*': function(scheme){
         console.log('Aha! ' + scheme);
      },
      'plottings.*': function(deviousPlot){
         console.log('Hmmm! ' + deviousPlot);
      }
   })
   .on('done', function(){
      console.log("*twiddles mustache*");
   })
   .on('fail', function(){
      console.log("Drat! Foiled again!");
   });
```

Because explicit loops are replaced with declarations the code is usually about the same length as if you'd done JSON.parse:

```
fs.readFile('/home/me/secretPlans.json',
   function(err, plansJson){
      if(err) {
         console.log("Drat! Foiled again!");
         return;
      }
      var plans = JSON.parse(err, plansJson);

      plans.schemes.forEach(function(scheme){
         console.log('Aha! ' + scheme);
      });
      plans.plottings.forEach(function(deviousPlot){
         console.log('Hmmm! ' + deviousPlot);
      });

      console.log("*twiddles mustache*");
   });
```

Rolling back on error

The fail function gives a callback for when something goes wrong. If you started putting elements on the page and the connection goes down you have a few options

- If the new elements you added are useful without the rest, leave them
- If they are useful but you need the rest, make a new request
- Rollback any half-done changes you made

```
var currentPersonElement;
oboe('everyone')
   .path('people.*', function(){
      // we don't have the person's details yet but we know we
      // found someone in the json stream, we can use this to
      // eagerly add them to the page:
      personDiv = jQuery('<div class="person">');
      jQuery('#people').append(personDiv);
   })
   .node('people.*.name', function( name ){
      // we just found out that person's name, lets add it to
      // their div:
      var markup = '<span class="name">' + name + '</span>';
      currentPersonElement.append(markup);
   })
   .fail(function(){
```

```
      if( currentPersonElement ) {
         // oops, that didn't go so well. instead of leaving
         // this dude half on the page, remove them altogether
         currentPersonElement.remove();
      }
   })
```

Example patterns

Pattern	Meaning
`*`	Every object, string, number etc found in the json stream
`!`	The root object. Fired when the whole response is available, like JSON.parse()
`!.foods.colour`	The colours of the foods
`person.emails[1]`	The first element in the email array for each person
`{name email}`	Any object with a name and an email property, regardless of where it is in the d
`person.emails[*]`	Any element in the email array for each person
`person.$emails[*]`	Any element in the email array for each person, but the callback will be passed t
`person`	All people in the json, nested at any depth
`person.friends.*.name`	Detecting friend names in a social network
`person.friends..{name}`	Detecting friends with names in a social network
`person..email`	Email addresses anywhere as descendent of a person object
`person..{email}`	Any object with an email address relating to a person in the stream
`$person..email`	Any person in the json stream with an email address