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 before iteratively refactor towards progressive loading. The API should be familiar to jQuery users.

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

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

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

Extracting objects from the JSON stream

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

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

On the client side we want to download and use this JSON. Running the code below, each item will be logged as soon as it is transferred without waiting for the whole download to complete.

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

      // This callback will be called everytime a new object is
      // found in the foods array.
```

```
      console.log( 'Go eat some', foodThing.name);
   })
   .node('badThings.*', function( badThing ){

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

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

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 patterns that use location:

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

Hanging up when we have what we need

If you don't control the data source it might give more information than the client actually needs.

In the example above if we only care about the foods 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();
```

```
      }
   });
```

Detecting strings, numbers

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

```
oboe('/myapp/things.json')
   .node( 'colour': function( colour ){
      // (colour instanceof String) === true
   });
```

Reacting before we get the whole object

As well as `node` events, you can listen on `path` events to receive notification when paths are found, even if we don't yet know what will be found there. In the example below we eagerly create elements before we have their content so that the page updates 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 = $('<div class="person">');
      $('#people').append(personDiv);
   })
   .node({
      'people.*.name': function( name ){
         // we just found out their name, lets add it
         // to their div:
         currentPersonElement.append(
            '<span class="name">' + name + '</span>');
      },
      'people.*.email': function( email ){
         // we just found out their email, lets add
         // it to their div:
         currentPersonElement.append(
            '<span class="email">' + email + '</span>');
      }
   });
```

Giving some visual feedback as a page is updating

Suppose we're using progressive rendering to go to the next 'page' in a dashboard-style single page webapp and want to put some kind of indication on the page as the individual components parts load.

Let's give visual feedback that when an area of the page is loading and remove it when we have data.

```
// JSON from the server side:
{
   'progress':[
      'faster loading',
      'maintainable velocity',
      'more beer'
   ],
   'problems':[
      'technical debt',
      'team drunk'
   ]
}
```

```
MyApp.showSpinnerAt('#progress');
MyApp.showSpinnerAt('#problems');

oboe('/agileReport/sprint42')
   .node({
      '!.progress.*': function( itemText ){
         $('#progress')
            .append('<div>')
            .text('We made progress in ' + itemText);
      },
      '!.progress': function(){
         MyApp.hideSpinnerAt('#foods');
      },
      '!.problems.*': function( itemText ){
         $('#problems')
            .append('<div>')
            .text('We had problems with ' + itemText);
      },
      '!.problems': function(){
         MyApp.hideSpinnerAt('#problems');
      }
   });
```

Implying meaning through node location

Node and path callbacks receive the location of found items as an array of strings describing the descent from the root. It is sometimes preferable to register a wide-matching pattern and use the item's location to decide programmatically what to do with it.

```
// JSON data for homepage of a social networking site.
// Each top-level object is for a different module on the page.
{  "notifications":{
      "newNotifications": 2,
      "totalNotifications": 8
   },
   "messages": [
      {  "from":"Joe",
         "subject":"Wanna go fishing?",
         "url":"messages/1"
      },
      {  "from":"Baz",
         "subject":"Hello",
         "url":"messages/2"
      }
   ],
   "photos": {
      "new": [
         {  "title": "Birthday 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
      // resembling ['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, the node matching that term is explicitly selected, even if the pattern as a whole matches a node further down the tree.

```
// JSON
{"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.
         // Putting this array on the scope object under
         // Angular re-renders the 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 `oboe` 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 pattern-based declarations, the code is usually about the same length as with JSON.parse:

```
fs.readFile('/home/me/secretPlans.json',
   function(err, plansJson){
      if(err) {
         console.log("Drat! Foiled again!");
         return;
      }
      var plans = JSON.parse(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 event notifies when something goes wrong. If you have 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. For example, in a web-based email client it is more useful to show some messages than none. See dropped connections visualisation.
- If they are useful but you need the rest, make a new request. If the service supports it you need only ask for the missing items.
- Rollback any half-done changes.

The example below implements rollback.

```
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 = $('<div class="person">');
      $('#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 descendant 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