Promises Recap

Recap

- AJAX
- HTTP Verbs - GET, POST
- Callbacks, “out of order” code

Today’s Outline

- Define and understand promises
- Converting callbacks into promises

Promises - Why?

Looking back at the same code to introduce AJAX and callbacks, what's wrong?

```
<div id="newsFeed"></div>

$.get('/newsfeed', function (response) {
        $('#newsFeed').html(response);
});
```

What are we missing?

Promises - Why?

In order to handle server error, we need to invoke the more generalized jQuery ajax function $.ajax. Our code then becomes:

```
$.ajax('/newsfeed', {
        success: function (response) {
                $('#newsFeed').html(response);
        },
        error: function (err) {
                $('#errorSection').html('Something bad happened');
        }
});
```

Promises - Why?

This code is perfectly fine. It is slightly more complex, but necessary and easy to read.

Now, what do we do while the page is loading?

Promises - Why?

```
$('#wrapper').css('display', 'none');
$.ajax('/newsfeed', {
        success: function (response) {
                $('#newsFeed').html(response);
        },
        error: function (err) {
                $('#errorSection').html('Something bad happened');
        },
        complete: function () {
                $('#wrapper').css('display', 'inherit');
        }
});
```

Promises - Why?

Again, this code is straightforward and is perfectly reasonable to be found out in the wild.

Finally, we need to add another AJAX call to the mix once the newsfeed call is finished. We need news “tags” to be displayed on the same imaginary section and don’t want to show anything on the screen until this is finished.

There are two* ways to go about this.

*Three if you count async: false, don’t count it though.

Promises - Why?

```
$('#wrapper').css('display', 'none');
$.ajax('/newsfeed', {
        success: function (response) {
                $('#newsFeed').html(response);
                $.ajax('/tags', {
                        success: function (tags) {
                                $('#tags').html(tags);
                        },
                        error: function () {
                                $('#errorSection').html('Failure to load tags.');
                        },
                        complete: function () {
                                $('#wrapper').css('display', 'inherit');
                        }
                });
        },
        error: function (err) {
                $('#errorSection').html('Failure to load news.');
                $('#wrapper').css('display', 'inherit');
        }
});
```

Promises - Why?

You could also solve the problem by storing a bool variable after each ajax call for “finished” and checking after each one before displaying the screen.

The symptoms of the problem are clear in either case, the code becomes nested to the right, and the code is more difficult to reason about.

We can begin to use promises to make this code a little bit cleaner.

Promises - What?

“A promise represents the eventual result of an asynchronous operation.”

- This is an inversion and abstraction for callbacks, allowing flow of control to be more easily managed.
- A literal “promise” to return a value at some point in the future.
- Promise libraries should (don’t have to) follow the Promises A+ spec guidelines.

Promises - What?

When we call *$.get('/newsFeed')* with or without a callback function, we are returned an object that behaves like a *promise*.

```
var myPromise = $.get('/newsFeed');
```

This promise follows certain rules to allow developers to construct code that flows from top-to-bottom instead of left to right.

Promises - Rules

- A promise must be *“thenable”*.

myPromise has a method *“then”* that takes two parameters, a success callback and an error callback. Just like a regular callback, when the server is finished, the success callback runs, if there is a problem the error callback runs.

```
$.get('/newsFeed').then(
    function (articles) {
        $('#newsFeed').html(articles);
    }, function (error) {
        $('#errorSection').html(error);
    });
```

Promises - Rules

You could rewrite the previous example as...

```
var success = function (articles) {
      $('#newsFeed').html(articles);
};

var error = function (error) {
      $('#errorSection').html(error);
};

var myPromise = $.get('/newsFeed');

myPromise.then(success, error);
```

Promises - Rules

- The *“then”* function wraps the success result in a promise so that functions can be *chained* together.

```
myPromise.then(..., ...)                    // omitted for brevity
    .then(function () {
        console.log(“I’m running”);
    })
    .then(function () {
        console.log(“I’m running too.”);
    });
```

Notice the result of the *then* function does not need to be an explicit promise, the library should wrap the function call.

Promises - Rules

- The *then* function will return the result of the previous function call.

```
$.get('/newsFeed')
    .then(function (articles) {
        $('#newsFeed').html(articles);
    }, function (error) {
        $('#errorSection').html(error);
    });
```

In the call above, "articles" are the return value because the $.get('newsFeed') promise asks the server for articles and returns those articles to the promise.

Promises - Rules

```
$.get('/newsFeed')
        .then(function (articles) {
                $('#newsFeed').html(articles);
        })
        .then(function (result) {
                console.log('result is undefined');
                return 5;
        })
        .then(function (number) {
                console.log('Number is equal to ' + number);
        });
```

Promises - Rules

- If an error is thrown or a promise is explicitly rejected, the next error function registered in the chain will be called and any methods in between will be skipped.

```
$.get('/newsFeed')
    .then(function (articles) {
        throw new Error('Why would you do this?');
    })
    .then(function (result) {
        return 5;
    }, function (error) {
        console.log('This was the error');
    });
```

Promises - Rules

```
$.get('/newsFeed')
        .then(function (articles) {
                throw new Error('Explicit error');
        })
        .then(function (result) {
                console.log('result is undefined');
                return 5;
        })
        .then(function (number) {
                console.log('Number is equal to ' + number);
        }, function (err) {
                console.log('Do not pass go.');
        });
```

Promises - Convert

Using this new information, how do we convert the news to tag example to promises-style instead of callback style?

Promises - Convert - Old

```
$('#wrapper').css('display', 'none');
$.ajax('/newsfeed', {
        success: function (response) {
                $('#newsFeed').html(response);
                $.ajax('/tags', {
                        success: function (tags) {
                                $('#tags').html(tags);
                        },
                        error: function () {
                                $('#errorSection').html('Failure to load tags.');
                        },
                        complete: function () {
                                $('#wrapper').css('display', 'inherit');
                        }
                });
        },
        error: function (err) {
                $('#errorSection').html('Failure to load news.');
                $('#wrapper').css('display', 'inherit');
        }
});
```

Promises - Convert - New

Let's rewrite the following code to use the built-in jQuery promise.

```
$('#wrapper').css('display', 'none');
$.get('/newsfeed')
        .then(function (articles) {
                $('#newsFeed').html(articles);
        })
        .then(function () {
                return $.get('/tags');
        })
        .then(function (tags) {
                $('#tags).html(tags);
        })
        .fail(function () {
                $('#errorSection').html('Something bad happened');
        })
        .always(function () {
                $('#wrapper').css('display', 'inherit');
        });
```

Promises - Convert - New

If we named and abstracted the functions, it would read similar to this...

```
blockScreen();

getNewsArticles()
        .then(renderArticles)
        .then(getTags)
        .then(renderTags)
        .fail(handleErrors)
        .always(unblockScreen);
```

Promises - Convert - New

Two “newer” concepts used in the conversion…

- fail - shorthand for writing a then function without an error handler. Below is logically equivalent.

```
promise.then(null, function (error) {
        console.log(“Notice theres no success callback.”);
});
```

- always - Will run no matter what happens in the function chain. Useful for things like removing loading icons or cleaning up the screen in other ways. Reassigning variables etc...

Promises - Convert - New

Both *fail* and *always* are unique to the way jQuery wants to name its functions.

Other frameworks might use words like catch and finally.

Promises vs Callbacks?

When to stick with callbacks?

- One-off scenarios
- The word chain doesn’t come to mind
- Never, frameworks will leave you behind.
- Don’t worry, promises are just fancy callbacks anyways.

Promises - My Two Cents

- It’s easier to read things top to bottom, that’s how humans read.
- You are forced to create the smallest possible chunk of code for each “step” in the promise pipeline.
- The smaller the function, the easier it is to test and the less likely you break other crap by modifying it.
- You can still clean up callback style functions by naming functions and passing them by reference instead of anonymously. Can’t avoid the nesting though.

Workshop

Build a Cap