JQUERY PROMISES AND DEFERRED

Many operations you perform in both JavaScript and jQuery can take a non-deterministic amount of time.

Some operations, such as animations, take place over a specified amount of time. While you will frequently be responsible for specifiying the amount of time an amination will take, there will be times when the length of time will be variable.

Also, when creating rich web applications, you'll frequently access server resources from your scripts. When you add such functionality, you don't know how long the server is going to take to process your request and return a value.

When those types of operations take place, you don't necessarily care how long they're going to take, but you do need to execute code when they complete. This is where promises come into play.

A **promise** is an object returned by functions in jQuery that take a long or variable amount of time. By using a promise, you can ensure your code executes whenever the operation completes, be notified of its success or failure, or potentially receive updates about an operation's progress.

Besides the built-in functions that return promises, jQuery also offers you a **deferred** object. A deferred object allows you to create your own long running operations, allowing developers to use the same patterns provided by the promise object, and be updated when your operation completes.

We're going to start our exploration of asynchronous programming in jQuery by introducing promises. We'll then see how you can create your own functions that return a promise through use of the deferred object. As part of this, we will also discuss the concept of a **web worker**, which is an HTML5 feature allowing web developers to simulate threads in a web browser.

## JQUERY PROMISES

A **promise** is a programming pattern in which a long running operation "promises" to let you know when it has completed its work.

### Long running operations

Any jQuery function that runs over a long period of time, such as an animation, or communicates with a remote server, such as Ajax calls, returns a **promise**. The promise object offers several events that are raised when the operation is completed, or if there is a progress update.

### done

done is raised when the operation completes successfully.

done accepts one or more event handler functions.

The event handler can accept one or more parameters, which will contain whatever data the promise object has returned. For example, when making Ajax calls, you will be able to access the data returned by the server in the event handler's parameter. The data returned is determined by the operation.

// code to obtain promise object

promise.done(function(data) {

// data will contain the data returned by the operation

});

### fail

fail is raised when the operation has completed with an error.

Like done, fail accepts one or more parameters. The parameters' values will be determined by the operation.

// code to obtain promise object

promise.fail(function(data) {

// data will contain the data returned by the operation

});

### progress

progress is raised when the operation raises an alert about its current state. Not all operations raise progress events.

Like done and fail, progress allows you to specify one or more event handlers, each optionally accepting parameters. The parameter values are set by the operation.

// code to obtain promise object

promise.progress(function(data) {

// data will contain the data returned by the operation

});

### Chaining

You can add done and fail (and potentially progress) event handlers by chaining the method calls as demonstrated below.

// code to obtain promise object

promise.done(function(data) {

// success

}).fail(function(data) {

// failure

});

### then

then is a single function allowing you to register done, fail, and progress event handlers in one call. The sample below is identical to the chaining demonstration above.

// code to obtain promise object

promise.then(function(data) {

// success

}, function(data) {

// failure

});

## INTRODUCING WEB WORKERS

This section introduces the concept of HTML5 web workers. If you're already familiar with web workers, you're free to skip this section. If you're not already familiar with how web workers are implemented, or just want to brush up, this section is for you!

### threading

Threading is a basic programming concept that allows developers to execute code on a separate process. Threading is extremely helpful when working with operations that either require additional processing power, or may take a long amount of time.

Applications typically start with a single thread that is used to execute code and update the user interface. If an operation is long running, and it executes on that thread, the user interface isn't able to be updated, and thus freezes. This provides a bad experience for the user. By using separate threads, you can execute your long running code elsewhere, allowing the user interface to still be responsive to the user.

As mentioned above, threading is an extremely powerful tool. Unfortunately, this tool can easily be mismanaged or abused, leading to degraded performance or potential security risks. This poses a challenge when working with web applications, in which users execute code (JavaScript) without knowing the developer of that code. Allowing threading in a browser could create an undesirable experience for the user. As a result, browsers don't allow JavaScript to use threads.

This is where web workers come into play.

## WEB WORKERS

A web worker is made up of two components, the parent or calling script, and the worker or executing script. The worker runs in an environment similar to a separate thread, and **does not** have direct access to the calling environment or the UI. Web workers use a messaging system to pass information to and from the worker.

## CREATING THE WORKER SCRIPT

To create a web worker, you create a separate JavaScript file. This file will contain the code that will execute in the worker environment. The code in the file **will execute immediately** when the worker object is created from the calling script. As a result, if you wish to defer execution in a worker, the code will need to be contained inside of a function.

### self

The web worker environment provides an object named self, which represents the worker. self has one function and one event.

The worker provides a function named postMessage that is used to send data to the calling environment. postMessage accepts most data types, including JavaScript objects.

// send a signal back to the calling script

self.postMessage('hello from the worker!');

The worker offers one event, message. message is raised when the calling script has sent a message to the worker. message is raised when the calling environment calls postMessage, and thus almost any type of object can be received. The data passed into postMessage is available by using the data property of the event object.

// Receive a message from the calling environment

self.addEventListener('message', function(e) {

// the data property will contain the data passed from the calling script

});

## CALLING A WEB WORKER

To call a web worker, you create an instance of the HTML5 Worker object. Because web workers are a relatively new development, it is a best practice to first check to see if the browser supports web workers. This can be done by testing if Worker is equal to null, meaning it doesn't exist. If Worker is null, you know the browser doesn't support web workers.

// Test if the browser supports web workers

if(Worker == null) {

alert('You need to upgrade your browser!');

} else {

// do your work here

}

### Creating an instance of the Worker object

The constructor for Worker accepts one parameter, the location of the script it will load into the worker space. Remember, the script will execute immediately, so unless you're certain it's been built to allow you to start it manually, don't create the instance until the last possible moment.

var worker = new Worker('script-location.js');

Similar to what we've already seen, the worker object offers a postMessage method to send data to the worker space, and an event message that is raised when the worker sends a message back to the calling page. The parameter you pass to postMessage is retrieved by using the data property of the event object in the message event handler.

// Register event handler

worker.addEventListener('message', function(e) {

**$**('#output').append('<li>' + e.data + '</li>');

});

worker.postMessage('Started!');

## DESIGNING YOUR WEB WORKERS

### Creating a web worker that accepts status messages

As you may have noticed, the web worker doesn't provide a built-in structure for handling common events, such as start and finish. However, the worker's simple messaging system allows you to easily build your workers to perform the operations you need, and add your own system for managing start and stop events.

Quite frequently, you will want to delay execution of the worker script until the caller sends a signal to start. Remember, when your worker script is loaded, the script is run immediately. You can change this behavior by adding a simple check to the worker for a start message.

Because JavaScript is weekly typed, the data property of the event object passed by the workers doesn't need to be set in advance. You could set it to your status strings, such as START and STOP when you're sending those types of messages, and use a JavaScript object in data when you need to send other payloads.

The script below is one simple implementation of the behavior described, using simple strings for event management. You can use other objects as you see fit, depending on the complexity of your needs.

// worker.js

self.addEventListener('message', function(e) {

if(e.data === 'START') {

// Start message received.

// Begin work

startWork();

} else if (e.data === 'STOP') {

// Stop message received.

// Perform cleanup and terminate

stopWork();

} else {

// A different message has been received

// This is data that needs to be acted upon

processData(e.data);

}

});

function startWork() {

// code to start performing work here

// send a message to the calling page

// worker has started

self.postMessage('STARTED');

}

function stopWork() {

// cleanupp code here

// stop the worker

self.postMessage('STOPPED');

self.close();

}

function processData(data) {

// perform the work on the data

self.postMessage('Processed ' + data);

}

### Calling a web worker that accepts messages

One of the great advantages to having a worker that's been built to accept status messages, such as start and stop, is it makes it very easy to get everything set up, and then start the worker process when you're ready to have it run.

If you were using the worker that's been designed above, you would use it by following a couple of basic steps.

1. Create an instance of Worker, passing in the script.
2. Add the event handler for the message event. Ensure the event handler can respond to the status messages and normal data.
3. When you're ready to start the worker's work, call postMessage('START');
4. When you're done, send the stop message by calling postMessage('STOP');

// inside of HTML file

var worker = new Worker('worker.js');

worker.addEventListener('message', function(e) {

if(e.data === 'STARTED') {

// worker has been started

// sample: update the screen to display worker started

**$**('#output').append('<div>Worker started</div>');

} else if(e.data === 'STOPPED') {

// worker has been stopped

// cleanup work (if needed)

// sample: update the screen to display worker stopped

**$**('#output').append('<div>Worker stopped</div>');

} else {

// Normal message. Act upon data as needed

// Sample: display data on screen

**$**('#output').append('<div>' + e.data + '</div>');

}

});

// When you're ready, send the start message

worker.postMessage('START');

// Send data as needed

worker.postMessage('sample data');

// Stop worker when you're done

worker.postMessage('STOP');

## RETURNING PROMISES

If you are creating a long running function, you can return a promise, allowing the caller to be alerted to your operations status, or when it completes. You return, and manage, a promise by creating an instance Deferred.

### Deferred

Deferred and promise seem very similar, and they are. The difference between the two is who uses which. Deferred is used to create, and manage, a promise object. A promise object is returned by a long running operation, and only allows you to register event handlers.

To put this another way, Deferred is the server side. When **create** a long running function that will be called by other developers, you'll use Deferred to return a promise. You'll use the Deferred object to update clients when your function completes (or you want to send a progress signal).

Continuing the analogy, promise is the client side. When you **call** a long running function, it will return a promise. You will use the promise to be alerted, and execute code, when that long running function completes (or sends a progress signal).

## WHEN TO USE DEFERRED

If you are creating a function that may take a long time to execute, it's best to return a promise. This makes it easier for developers who call your function, as they can use the promise events.

One nice thing about jQuery is the developers of the API follow their own best practices. As a result, if you execute an operation, such as an Ajax call, the function will return a promise. If you are creating a function that will be wrapping such a call, you can simply return the promise returned by the function.

For example, consider the following jQuery. We create a function that calls slideToggle. slideToggle can take a couple of seconds to execute, depending on how long you tell the operation to take. As a result, it returns a promise, as we saw in an earlier section. Because slideToggle returns a promise object already, we can just use that, rather than creating a Deferred object on our own.

function displayMenu() {

// just return the promise object

return **$**('#menu').slideToggle(500);

}

However, if we are creating a function that will take an unusual amount of time, say one that will be working with graphics, we need will want to use Deferred to return a promise to the caller.

### Breaking down using Deferred

The basic steps are as follows.

1. Create an instance of deferred: var deferred = $.Deferred();
2. Start your asynchronous operation, typically using a worker
3. Add the appropriate code to detect success and send the success signal: deferred.resolve()
4. Add the appropriate code to detect failure and send the failure signal: deferred.reject()
5. Return the promise: return deferred.promise();

function beginProcessing() {

// Create deferred object & make sure it's going to be in scope

var deferred = new **$**.Deferred();

// Create our worker (just like before)

var worker = new Worker('./Scripts/deferred.js');

// Register the message event handler

worker.addEventListener('message', function (e) {

// simple messaging - if the worker is ready it'll send a message with READY as the text

if (e.data === 'READY') {

// No UI code

// Progress notification

deferred.notify('Worker started');

} else if(e.data === 'COMPLETED') {

// processing is done

// No UI code

// Completed notification

deferred.resolve('Worker completed');

worker.terminate();

}

});

return deferred.promise();

}

## SERIALIZATION AND JAVASCRIPT

One of the most powerful capabilities JavaScript offers is the ability to make calls to the server. This allows us as developers to create web pages that behave like locally installed applications, and access server-based resources, such as a database, without having to refresh the entire page.

The catch, of course, is how do we take our in-memory data and send it to the server, and how do we convert the server's response into an in-memory object? This is where JSON comes into play. JSON, or JavaScript Object Notation, can be used to represent the object as a string. The process of converting an in memory object to a string representation, including JSON, is known as serialization.

### JSON basics

A JSON object uses a notation that's similar to that of a JavaScript object. If you wanted to create a JavaScript object with two properties, firstName and lastName, you'd use the following code:

var person = {

firstName: 'Christopher',

lastName: 'Harrison'

}

When using JSON, the property names are strings, rather than variable names. The only difference between the syntax above and below is the quotes around lastName and firstName.

var person = {

'firstName': 'Christopher',

'lastName': 'Harrison'

}

If you wish to create a more complex object, one that contains another object, you simply contain the second object in a new set of curly braces ({}).

var person = {

'firstName': 'Christopher',

'lastName': 'Harrison',

'email': {

'address': 'charrison@adventure-works.com',

'type': 'business'

}

}

And, if you want an array, use square brackets ([]).

var person = {

'firstName': 'Christopher',

'lastName': 'Harrison',

'emailAddresses': [

{

'address': 'charrison@adventure-works.com',

'type': 'business'

},

{

'address': 'charrison@fineartschool.net',

'type': 'home'

}

]

}

## SERIALIZING OBJECTS

JavaScript provides a native utility for converting objects to JSON (serializing), and converting JSON strings to objects (deserializing).

### Serializing an object

To serialize an object, use JSON.stringify.

// Create an instance of Object, a basic JavaScript object

var person = new Object();

// add properties

person.firstName = 'Christopher';

person.lastName = 'Harrison';

// serialize

var jsonString = JSON.stringify(person);

### Deserializing JSON

To deserialize JSON, use JSON.parse. Because JavaScript manages objects dynamically, you don't need to worry about casting the returned object to a class, like you would in other programming languages.

// Deserialize a JSON string

var newPerson = JSON.parse(jsonString);

// access properties as normal

**$**('#first-name-display').text(newPerson.firstName);

**$**('#last-name-display').text(newPerson.lastName);

## MAKING CALLS TO THE SERVER

Typically, when we make a call to the server, we need to refresh the entire page. Not only can this impact performance, it can change our user's perception of our pages. In addition, as developers, we'd like to be able to incorporate server-side resources into our pages, allowing us to update individual portions of the page with new data, rather than updating the entire page. This is where the XmlHttpRequest object comes into play and, Ajax.

### Asynchronous JavaScript and XML (Ajax)

Ajax is a set of technologies that act together to make it easier for us as developers to make calls to server resources from JavaScript. Breaking down the three words that make up the acronym, you'll notice we have asynchronous (which jQuery simplifies through the use of promises), JavaScript (which we already know), and XML. XML is probably the one that doesn't fit, as XML is typically not a preferred mechanism for serialization. As we've seen, we typically want to use JSON, as it’s more compact and native to JavaScript.

### Basic data retrieval

The most basic Ajax operation we can perform using jQuery is get. get contacts the URL we provide, and passes the string the server returns into the parameter we'll use for our event handler. get accepts multiple parameters, but the two you'll most commonly use are the URL you wish to call, and an event handler that will be executed on success.

**$**.get(

'some-url', // The URL to call

function(data) { // Success event handler

// The data parameter contains the string

**$**('#output').text(data);

}

);

### jQuery Ajax and promises

All jQuery Ajax calls return a jQuery promise. This means you can use done for your success event handler, and fail to catch any errors. The two code samples perform the same operations.

// Option one (pass the success function as a parameter)

**$**.get('some-url', function(data) { **$**('#output').text(data); });

// Option two (use the done function of the promise)

**$**.get('some-url').done(function(data) { **$**('#output').text(data); });