# Managing the JavaScript UI Thread with the WinJS Scheduler

Apart from web workers and background tasks that run as web workers, all JavaScript code in a Windows Store app runs on what we call the *UI thread*. That code can invoke asynchronous WinRT APIs that perform their operations on separate threads, of course, but there’s yet one very important characteristic to remember: the *results* from all those non-UI threads eventually come back to the UI thread for processing. This means that making a bunch of async WinRT calls (like HTTP requests) all at once can potentially overload the UI thread if their results all come back about the same time. Furthermore, if you (or WinJS) add elements to the DOM or change styles that trigger a layout pass on the UI thread, there’s even more work that’s competing for the CPU. As a result, your app can become sluggish and unresponsive.

In Windows 8, apps could take some steps to mitigate this, such as starting async operations in timed batches to manage their rate of callbacks to the UI thread, and batching together work that triggers a layout pass so as to combine multiple changes in each pass. Starting with Windows 8.1, however, there is the ability to asynchronously prioritize different tasks on the UI thread itself.

Although the app host provides low-level scheduling APIs for this such as MSApp.executeAtPriority, we recommend that apps use the [WinJS.Utilities.Scheduler](http://msdn.microsoft.com/library/windows/apps/dn301978.aspx) API instead. This is because WinJS very carefully manages its own tasks through this Scheduler API, which means any work you schedule in the same way will be properly coordinated with the work that WinJS is doing on your behalf. The WinJS Scheduler also provides a simpler interface to the whole process, especially where promises are concerned.

It’s good to know that using the scheduler is not at all required—it’s there to help you tune the performance of your app, not to make your life difficult! But assuming that you want to make you app work its best, let’s first understand the different priorities that the scheduler offers, then see how to schedule and manage work at those priorities.

## Scheduler Priorities

The relative priorities for the WinJS Scheduler are expressed in the [Scheduler.Priority](http://msdn.microsoft.com/library/windows/apps/dn301907.aspx) enumeration, which I list here in descending order: max, high, aboveNormal, normal (the default for app code), belowNormal, idle, and min. Here’s the general guidance on how to use these:

|  |  |
| --- | --- |
| Priority | Best Usage |
| max high | Use sparingly for truly high priority work as these priorities take priority over layout passes in the rendering engine. If you overuse these priorities, the app can actually become *less* responsive!. |
| aboveNormal normal belowNormal | Use these to indicate the relative importance between most of your tasks. |
| idle min | Use for long-running and/or maintenance tasks where there isn’t a UI dependency. |

Although you again need not use the scheduler in your own code at all, a little analysis on your use of async operations will likely reveal places where setting priorities might make a big difference. For example, you can prioritize non-UI work while your splash screen is visible, because the splash screen is noninteractive by definition, perhaps running your most important HTTP requests at max or high while secondary requests are set to belowNormal. This help those first requests get processed before rendering your home page, at which point the user expects to interact with that content, and then allows handling of the secondary requests to occur in the background. Of course, if the user navigates to a page that requires the secondary content, you can then change the priority of that task to aboveNormal or high.

WinJS itself makes extensive use of priorities. For example, it will batch changes to a data-binding source at high priority while scheduling cleanup tasks at idle priority. In a complex control like the ListView, fetching new items that are necessary to render the visible part of a ListView control is done at max, rendering of the visible items is done at aboveNormal, pre-loading the next page of items forward is set to normal (anticipating that the user will pan ahead), and pre-loading of the previous page (to anticipate a reverse pan) is set to belowNormal.

## Scheduling and Managing Tasks

Now that we know about scheduling priorities, the way to asynchronously execute code on the UI thread at a particular priority is by calling the [Scheduler.schedule](http://msdn.microsoft.com/library/windows/apps/dn301941.aspx) method (whose default priority is normal). This method allows you to provide an optional object to use as this inside the function along with a name to use for logging and diagnostics. (The [Scheduler.execHigh](http://msdn.microsoft.com/library/windows/apps/dn301933.aspx) method is also a shortcut for directly calling MSApp.execAtPriority with Priority.high. This method does not accommodate any added arguments.)

As a simple example, scenario 1 of the [HTML Scheduler sample](http://code.msdn.microsoft.com/windowsapps/HTML-Scheduler-Sample-4b8084f2) schedules a bunch of functions at different priorities in a somewhat random order (js/schedulesjobscenario.js):

window.output("\nScheduling Jobs...");  
var S = WinJS.Utilities.Scheduler;  
  
S.schedule(function () { window.output("Running job at aboveNormal priority"); },  
 S.Priority.aboveNormal);  
window.output("Scheduled job at aboveNormal priority");  
  
S.schedule(function () { window.output("Running job at idle priority"); },  
 S.Priority.idle, this);  
window.output("Scheduled job at idle priority");  
  
S.schedule(function () { window.output("Running job at belowNormal priority"); },  
 S.Priority.belowNormal);  
window.output("Scheduled job at belowNormal priority");  
  
S.schedule(function () { window.output("Running job at normal priority"); }, S.Priority.normal);  
window.output("Scheduled job at normal priority");  
  
S.schedule(function () { window.output("Running job at high priority"); }, S.Priority.high);  
window.output("Scheduled job at high priority");  
  
window.output("Finished Scheduling Jobs\n");

The output then shows that the “jobs,” as they’re called, execute in the expected order:

Scheduling Jobs...  
Scheduled job at aboveNormalPriority  
Scheduled job at idlePriority  
Scheduled job at belowNormalPriority  
Scheduled job at normalPriority  
Scheduled job at highPriority  
Finished Scheduling Jobs  
Running job at high priority  
Running job at aboveNormal priority  
Running job at normal priority  
Running job at belowNormal priority  
Running job at idle priority

No surprises here, I hope!

When you call schedule, what you get back is an object with the [Scheduler.IJob](http://msdn.microsoft.com/library/windows/apps/dn255156.aspx) interface, which defines the following methods and properties:

|  |  |
| --- | --- |
| Properties | Description |
| id | (read-only) A unique id assigned by the scheduler. |
| name | (read-write) The app-provided name assigned to the job, if any. The name argument to schedule will be stored here. |
| priority | (read-write) The priority assigned through schedule; setting this property will change the priority. |
| completed | (read-only) A Boolean indicating whether the job has completed (that is, the function given to schedule has returned and all its dependent async operations are complete). |
| owner | (read-write) An owner token that can be used to group jobs. This is undefined by default. |
|  |  |
| Methods | Description |
| pause | Halts further execution of the job. |
| resume | Resumes a previously paused job (no effect if the job isn’t paused). |
| cancel | Removes the job from the scheduler. |

In practice, if you’ve scheduled a job at a low priority but navigate to a page that really needs that job to complete before the page is rendered, you simply bump up its priority property (and then drain the scheduler as we’ll see in a moment). Similarly, if you scheduled some work on a page that you don’t need to continue when navigating away, then call the job’s cancel method within the page’s unload method. Or perhaps you have an index page from which you typically navigate into a details page, and then back again. In this case you can pause any jobs on the index page when navigating to the details, then resume them when you return to the index. See scenarios 2 and 3 of the sample for some demonstrations.

Scenario 2 also shows the utility of the owner property (the code is thoroughly mundane so I’ll leave you to examine it on your own). An owner token is created through [Scheduler.createOwnerToken](http://msdn.microsoft.com/library/windows/apps/dn301931.aspx), then assigned to a job’s owner (which replaces any previous owner). An owner token is simply an object with a single method called [cancelAll](http://msdn.microsoft.com/library/windows/apps/dn255179.aspx) that calls the cancel method of whatever jobs are assigned to it, nothing more. It’s a simple mechanism—the owner token really doesn’t nothing more than maintain an array of jobs—but clearly allows you to group related jobs together and cancel them with a single call. This way you don’t need to maintain your own lists and iterate through them for this purpose. (To do the same for pause and resume you can, of course, just duplicate the pattern in your own code.)

The other important feature of the Scheduler is the [requestDrain](http://msdn.microsoft.com/library/windows/apps/dn301934.aspx) method. This ensures that all jobs scheduled at a given priority or higher are executed before the UI thread yields. You typically use this to guarantee that high priority job are completed before a layout pass. requestDrain returns a promise that is fulfilled when the jobs are drained, at which time you can drain lower priority tasks or schedule new ones.

A simple demonstration is shown in scenario 5 of the sample. It has two buttons that schedule the same set of varying jobs and then call requestDrain with either high or belowNormal priority. When the returned promise completes, it outputs a message to that effect (js/drainingscenario.js):

S.requestDrain(priority).done(function () {  
 window.output("Done draining");  
});

Comparing the output of these two side by side (high on the left, belowNormal on the right), as below, you can see that the promise is fulfilled at different points depending on the priority:

|  |  |
| --- | --- |
| Draining scheduler to high priority Running job2 at high priority Done draining Running job1 at normal priority Running job5 at normal priority Running job4 at belowNormal priority Running job3 at idle priority | Draining scheduler to belowNormal priority Running job2 at high priority Running job1 at normal priority Running job5 at normal priority Running job4 at belowNormal priority Done draining Running job3 at idle priority |

The other method that exists on the Scheduler is [retrieveState](http://msdn.microsoft.com/library/windows/apps/dn301935.aspx), a diagnostic aid that returns a descriptive string for current jobs and drain requests. Adding a call to this in scenario 5 of the sample just after the call to requestDrain will return the following string:

Jobs:  
 id: 28, priority: high  
 id: 27, priority: normal  
 id: 31, priority: normal  
 id: 30, priority: belowNormal  
 id: 29, priority: idle  
Drain requests:  
 \*priority: high, name: Drain Request 0

## Setting Priority in Promise Chains

Let’s say you have a set of async data-retrieval methods that you want to execute in a sequence as follows, processing their results at each step:

getCriticalDataAsync().then(function (results1) {  
 var secondaryPages = processCriticalData(results1);  
 return getSecondaryDataAsync(secondaryPages);  
}).then(function (results2) {  
 var itemsToCache = processSecondaryData(results2);  
 return getBackgroundCacheDataAsync(itemsToCache);  
}).done(function (results3) {  
 populateCache(results3);  
});

By default, all of this would run at the current priority against everything else happening on the UI thread. But you probably want the call to processCriticalData to run at a high priority, processSecondaryData to run at normal, and populateCache to run at idle. With schedule by itself, you’d have to do everything the hard way:

var S = WinJS.Utilities.Scheduler;  
  
getCriticalDataAsync().done(function (results1) {  
 S.schedule(function () {  
 var secondaryPages = processCriticalData(results1);  
 S.schedule(function () {  
 getSecondaryDataAsync(secondaryPages).done(function (results2) {  
 var itemsToCache = processSecondaryData(results2);   
 S.schedule(function () {  
 getBackgroundCacheDataAsync(itemsToCache).done(function (results3) {  
 populateCache(results3);  
 });  
 }, S.Priority.idle);  
 });  
 }, S.Priority.normal);  
 }, S.Priority.high);  
});

In my opinion, going to the dentist is more fun than writing code like this! To simplify matters, you could encapsulate the process of setting a new priority within another promise that you can then insert into the chain. The best way to do this is to dynamically generate a completed handler that would take the results from the previous step in the chain, schedule a new priority, and return a promise that delivers those same results:

function schedulePromise(priority) {  
 //This returned function is a completed handler.  
 return function completedHandler (results) {  
 //The completed handler returns another promise that's fulfilled  
 //with the same results it received...  
 return new WinJS.Promise(function initializer (completeDispatcher) {  
 //But the delivery of those results are scheduled according to a priority.  
 WinJS.Utilities.Scheduler.schedule(function () {  
 completeDispatcher(results);  
 }, priority);  
 });  
 }  
}

Fortunately we don’t have to write this code ourselves. The WinJS.Utilities.Scheduler already has five pre-made completed handlers like this that also automatically cancel a job if there’s an error. These are called [schedulePromiseHigh](http://msdn.microsoft.com/library/windows/apps/dn301938.aspx), [schedulePromiseAboveNormal](http://msdn.microsoft.com/library/windows/apps/dn301936.aspx), [schedulePromiseNormal](http://msdn.microsoft.com/library/windows/apps/dn301940.aspx), [schedulePromiseBelowNormal](http://msdn.microsoft.com/library/windows/apps/dn301937.aspx), and [schedulePromiseIdle](http://msdn.microsoft.com/library/windows/apps/dn301939.aspx).

As pre-made completed handlers, simply insert the appropriate *name* in a promise chain where you want to change the priority, as highlighted below:

var S = WinJS.Utilities.Scheduler;  
  
getCriticalDataAsync().then(S.schedulePromiseHigh).then(function (results1) {  
 var secondaryPages = processCriticalData(results1);  
 return getSecondaryDataAsync(secondaryPages);  
}).then(S.schedulePromise.normal).then(function (results2) {  
 var itemsToCache = processSecondaryData(results2);  
 return getBackgroundCacheDataAsync(itemsToCache);  
}).then(S.schedulePromiseIdle).done(function (results3) {  
 populateCache(results3);  
});

To be clear, you only ever use these function by inserting their names into a promise chain: you don’t call them directly as a discrete function. (This is unclear in the present documentation.)

## Long-Running Tasks

All the jobs that we’ve seen so far are short-running in that we schedule a worker function at a certain priority and it just completes its work when it’s called. However, some tasks might take a much longer time to complete in which case you don’t want to block higher priority work on your UI thread. To help with this, the scheduler has a built-in interval timer of sorts for tasks that are scheduled at aboveNormal priority or lower, such that a task can check whether it should cooperatively yield and have itself rescheduled for its next bit of work. Let me stress that word *cooperatively*: nothing forces a task to yield, but because all of this is affecting the UI performance of your app and your app alone, if you don’t play nicely you’ll just be hurting yourself!

The mechanism for this is provided through a *job info* object that’s passed as an argument to the worker function itself. So we’re clear how this fits in, let’s first look at everything a worker has available within its scope, which is best explained with a few comments within the basic code structure:

var job = WinJS.Utilities.Scheduler.schedule(function worker(jobInfo) {  
 //jobInfo.job is the same as the job returned from schedule.  
 //Scheduler.currentPriority will match the second argument to schedule.  
 //this will be the third argument passed to schedule.  
}, S.Priority.idle, this);

The members of the *jobInfo* object are defined by [Scheduler.IJobInfo](http://msdn.microsoft.com/library/windows/apps/dn255148.aspx):

|  |  |
| --- | --- |
| Properties | Description |
| job | (read-only) The same job object as returned from schedule. |
| shouldYield | (read-only) A Boolean flag that is typically false when the worker is first called and then changes to true if the worker should yield the UI thread and reschedule its work. |
|  |  |
| Methods | Description |
| setWork | Provides the worker for the rescheduled task. |
| setPromise | Provides a promise that the scheduler will wait upon before rescheduling the task, where the worker to reschedule is the fulfillment value of the promise. |

Scenario 4 of the [HTML Scheduler sample](http://code.msdn.microsoft.com/windowsapps/HTML-Scheduler-Sample-4b8084f2) shows how to work with these. When you press the Execute a Yielding Task button, it schedules a function called *worker* at idle priority that just spins within itself until you press the Complete Yielding Task button, which sets the taskCompleted flag below to true (js/yieldingscenario.js, with the 2s interval changed to 200ms):

S.schedule(function worker(jobInfo) {  
 while (!taskCompleted) {  
 if (jobInfo.shouldYield) {  
 // not finished, run this function again  
 window.output("Yielding and putting idle job back on scheduler.");  
 jobInfo.setWork(worker);  
 break;  
 }  
 else {  
 window.output("Running idle yielding job...");  
 var start = performance.now();  
 while (performance.now() < (start + 200)) {  
 // do nothing;  
 }  
 }  
 }  
  
 if (taskCompleted) {  
 window.output("Completed yielding task.");  
 taskCompleted = false;  
 }  
}, S.Priority.idle);

Provided that the task is active, it does 200ms work of work and then checks if shouldYield has changed to true. If so, the worker calls setWork to reschedule itself (or another function if it wants). You can trigger this while the idle worker is running by pressing the Add Higher Priority Tasks to Queue button in the sample. You’ll then see how those tasks are run before the next call to the worker. In addition, you can poke around elsewhere in the UI to observe that the idle task is not blocking the UI thread.

Note here that the worker function checks shouldYield first thing to immediately yield if necessary. However, it’s perfectly fine to do a little work first, and then check. Again, this is all about cooperating within your own app code, so such self-throttling is your choice.

As for setPromise, this is slightly tricky. Calling setPromise tells the scheduler to wait until that promise is fulfilled before rescheduling the task, where the next worker function for the task is provided directly through the promise’s fulfillment value. (As such, IJobInfo.setPromise doesn’t pertain to handling async operations like other setPromise methods in WinJS that are tied in with WinRT deferrals. If you called IJobInfo.setPromise with a promise from some random async API, the scheduler would attempt to use the fulfillment value of that operation—which could be anything—as a function and thus likely throw an exception.)

In short, whereas setWork says “go ahead and reschedule with this worker,” setPromise says “hold off rescheduling until I deliver the worker sometime later.” This is primarily useful to create a work queue composed of multiple jobs with an ongoing task to process that queue. To illustrate, consider the following code for such an arrangement:

var workQueue = [];  
  
function addToQueue(worker) {  
 workQueue.push(worker);  
}  
  
S.schedule(function processQueue(jobInfo) {  
 while (work.length) {  
 if (jobInfo.shouldYield) {  
 jobInfo.setWork(processQueue);  
 return;  
 }  
 work.shift()(); //Pull the first from the FIFO queue and call it.  
 }  
}}, S.Priority.belowNormal);

Assuming that there are some jobs in the queue when you first call schedule, the *processQueue* task will cooperatively empty that queue. And if new jobs are added to the queue in the meantime, *processQueue* will continue to be rescheduled.

The problem, however, is that the *processQueue* worker will finish and exit as soon as the queue is empty, meaning that any jobs you add to the queue later on won’t be processed. To fix this you could just have *processQueue* repeatedly call setWork on itself again and again even when the queue is empty, but that would be wasteful. Instead, you can use setPromise to have the scheduler wait until there is more work in the queue. Here’s how that would work:

var workQueue = [];  
var haveWork = function () { }; //This function is just a placeholder  
  
function addToQueue(worker) {  
 workQueue.push(worker);  
 haveWork();  
}  
  
S.schedule(function processQueue(jobInfo) {  
 while (work.length) {  
 if (jobInfo.shouldYield) {  
 jobInfo.setWork(processQueue);  
 return;  
 }  
 work.shift()(); //Pull the first from the FIFO queue and call it.  
 }  
  
 //If we reach here the queue is empty, but we don't want to exit the worker.  
 //Instead of calling setWork without work to do, create a promise that's fulfilled  
 //when addToQueue is called again, which we do by replacing the haveWork function  
 //with one that calls the promise's completed handler.  
 jobInfo.setPromise(new WinJS.Promise(function (completeDispatcher) {  
 haveWork = function () { completeDispatcher(processQueue) };  
 }))  
});

With this code, say we populate workQueue with a number of jobs and then make the call to schedule. Up to this point, and so long as the queue doesn’t become empty, we stay inside the while loop of *processQueue.* Any call to the empty haveWork function so far is just a no-op.

If the queue becomes empty, however, we’ll exit the while loop but we don’t want *processQueue* to exit. Instead, we want to tell the scheduler to wait until more work is added to the queue. This is why we have that placeholder function for haveWork, because we can now replace it with a function that will complete the promise with *processQueue*, thereby triggering a rescheduling of that worker function.

Note that an alternate way to accomplish the same goal is to use this assignment for haveWork:

haveWork = completeDispatcher.bind(null, processQueue);

This accomplishes the same result as an anonymous function and avoids creating a closure.

## In Conclusion

The WinJS Scheduler API enables apps to schedule different tasks on the UI thread at relative priorities, including different tasks within a single promise chain. In doing so, an app automatically coordinates its tasks with those that WinJS is running on the app’s behalf, such as performing data binding and rendering controls. With thoughtful application of the available priorities, an app can take meaningful steps to improve its overall responsiveness and its user experience.

Kraig Brockschmidt  
Program Manager, Windows Ecosystem Team  
Author, [*Programming Windows Store Apps with HTML, CSS, and JavaScript*](http://aka.ms/brockschmidtbook2), Second Edition