|  |
| --- |
| PluralSight |
| Modern Async JS |
| Full Course Transcript |

|  |
| --- |
| Nag, Somjit (Cognizant)  [Date] |

# Callbacks

## Callback Nesting Nightmare

Back in the early days of Node.js we took a stab at using it to write one of the back ends for one of our web applications. We were excited about the cross-platform nature of Node.js and the wonderful ecosystem of packages provided by MPM; however, we hit a snag in the process that ultimately led us to abandon this effort. Since then I have had many opportunities to try this again and I've had much success using JavaScript outside of the browser. Welcome to my course, Modern Asynchronous JavaScript. My name is Wes Higbee and I want to help you avoid the problems that I had and that my team had when working with Node.js and JavaScript. I want to teach you what I learned along the way. Here is what this snap looks like. Let me give you some background of this example. This is an Express application written in Node.js that uses MySQL to store data. The snag that we hit, a snag you've probably encountered yourself, is something that's not unique or specific to Node.js; it's common in all JavaScript environments including the browser and I'll show you many examples of this in the browser later on in this course, but for now we'll start with Node.js. Let me explain a little bit about this code that demonstrates the snag. First off, I just want to talk about the code on lines 44 to 51; you can ignore everything else on this page. On line 44, app.post registers a route in the web application, the /users route, as an endpoint that can receive a form submission. So when a form is posted to the server, this function will be called. If you're more familiar with front-end development, this would be a lot like wiring up a click handler to a button, so this function is fired off any time a request comes in for the /users route. This function receives a request and a response object so it can look at what was requested and send back a response. Inside of here is the logic that's executed. Now this application uses a MySQL database so when somebody posts a user, that user will be inserted into the users table and then once that insert is complete, there is yet another function nested inside of here, a function that's commonly referred to as a callback function that will receive the result of that database operation. There will either be an error or if the result is successful there will be some information about the successful result. In the case of an error, this callback function, this example app, just re-throws that errors and that's rather interesting. However, if things are successful, the ID for the new record is sent back to the browser. In explaining this code, can you take a guess at what the snag we hit was? This code functionally works and we wrote a lot of code like it, but I just don't like this code. It's extremely verbose. I really only see two lines of code here and yet there are actually six lines to insert the record into the database. So there are six lines here for what I think are actually only two lines of code. Here's the first line of code, here's the second line of code; the rest of this is all boilerplate. Of course, this might be manageable, but let me scroll up and show you another example of what things often ended up looking like. When this app spins up, it has a little bit of initialization code to create the database if it does not yet exist, to use that database and then to create a user's table. Now as I'm looking at this, this could actually all be combined into one SQL statement, but it's plausible that we could have a chain of SQL statements that need to be called sequentially because there might be dependencies between them. Because of the asynchronous nature of firing off a query and the fact that we need a callback then to handle the response, the duplication and verbosity in that first example explodes in this one. We have three callback functions that we have to write with three levels of nesting and we have three duplicated error handlers that yet again, do something rather interesting that we'll get to later in this course. Of course, we have to close up all of these functions as well. When I'm looking at this, I really only see three lines of code. I see at least six lines of code I'd like to eliminate. Let me show you what I mean. Let's start off with that first example. I copied this into WebStorm here and then I rewrote it down below. So there are two different ways to look at this chunk of code. The first is this style using callbacks with Node.js and the second is a hypothetical style that I'd like to have. The second only involves the two pieces of code that I care about. The boilerplate has been removed. First we insert the record into our users table. If anything goes wrong, and exception is thrown, just like we're throwing the exception here, but if everything is okay, they we go ahead and send back the new ID. So if you had to write and support this code, which style would you prefer? I'm definitely in the camp of the later and if you've worked with other languages, I'm sure you're much more familiar with this later style than the former. Here's that second monstrosity, setting up the database. Here is what I prefer this code to look like. The only code that really matters are these three queries that we fire off, all the error handlers and closing up the callback functions and declaring the callback functions; that's all noise. That's something that should be handled by the language or the platform that we're running on it. It shouldn't be something we have to religiously type out every time we fire off an asynchronous operation. In this course, I want to show you how you can make the second style a reality. Because if you have to write code like this, whether it's on the front end or the back end, you're not going to enjoy doing it if you have to and you'll probably try to avoid using JavaScript as a result and I don't want you to do that because there are many, many more viable ways to write your code like this second style. And the best part is, as you'll see in this course, you don't need any fancy tools to pull this off. We'll be using constructs that are already available as a part of ES6, constructs that are implemented in most JavaScript engines already.

## What You Will Get out of This Course

In addition to learning how you can write code in a style that's much more readable and easier to maintain and much less verbose, you'll also learn a lot about using Promises. Promises are a first step on the journey to writing code in this new style that you can see in the bottom here of this MySQL example. So Promises are a first step to improving our asynchronous code to overcome many of the challenges that we face with the callback style. So walk away with a good understanding of what Promises are. Perhaps you have used them a little bit. I'm sure you have if you've worked with jQuery. Promises have been a part of jQuery for quite a while now. In addition to understanding how to use Promises, we'll step behind the scenes and see how we can write a Promise framework. I think this is the best way to wrap your mind around how Promises work, to step behind the scenes and be the producer of a promise framework in addition to being the consumer of Promises. Once we've tackled Promises and you've got your mind wrapped around that, we'll take a look at generator functions because generator functions afford the opportunity with Promises to write code in this more concise format, so we'll start out by looking at how we can use generators to build a generator control flow function and we'll see what that means later on, but ultimately that will allow us to write our code in a more concise style. After we've done that, we'll talk about some frameworks that already exist that you could be using instead of rolling your own. It's great to roll your own to learn how a generator control flow function works, but you'll probably want to use something more robust and reliable in reality so I'll show you examples that you can use in your projects and then we'll wrap up this course talking about a new feature that's coming in the JavaScript language. Unfortunately, it didn't become a part of ES7 or ES2016, but is very likely to be a part of ES2017. This new syntax involves asynchronous functions and we'll talk about those at the end of the course. Async functions will formalize much of what we're doing with generator control flow functions and provide some syntactic sugar built into the language itself.

## What You Need to Know

I designed this course for a variety of people. I don't really believe in skill levels. What I do believe thought is that you should have a desire to learn. You will need a basic understanding of writing JavaScript code, so this is not a beginning JavaScript course. I also need you to be able to reason about the asynchronicity of JavaScript code. In other words, I need you to understand when things execute and truly understand the single-threaded nature of JavaScript. I'm going to review the key takeaways that you should be familiar with from my previous course if these resonate with you and if you think you understand these already they you can proceed with this course. However, if you're uncertain about any of these, then I'd encourage you to start out with my course, Reasoning About Asynchronous JavaScript. Watch that first. It's a pretty short course. Work through the examples and then come back to this course so that we can get down to business understanding how to build out Promise frameworks, generator control flow functions, things that will require you to be able to reason about asynchronicity, otherwise it's possible you'll be lost with examples in this course. So here is what I'm going to assume you know coming into this course as far as understanding when things execute in JavaScript. First and foremost, JavaScript engines are single-threaded. This is a lot like having a bank with only one teller or a bar with only one bathroom. This means that only one thing can happen at a time. JavaScript engines have what's known as an event loop and that event loop has a queue to hold work that needs to happen. That queue is a lot like the line that forms at a bank or the line that forms outside of a bathroom. While somebody is using the bathroom, everybody else has to wait. Same thing with our JavaScript code. While a piece of our code is executing, all the rest of our code has to wait. One of the key benefits of JavaScript, in order to support concurrency, JavaScript relies on what's known as non-blocking code. Often you'll hear about non-blocking I/O. This is a common use case to not block the application when you're waiting for a request to come back, say a request to read a file from the disk or a request to a remote web service. So if the one thing that could be happening in the bathroom waited until the web request came back, then nothing else could execute and that would not be a performant environment to be executing code in. Just because JavaScript has the concept of not blocking for things like making a web request, doesn't mean that you can't accidentally block in JavaScript. So when you're writing code, you need to make sure that that code isn't blocking other work from happening. In other words, you need to make sure that your code isn't hogging the CPU because there's nothing that will inadvertently stop you from blocking the CPU. Another feature of JavaScript engines is that whatever code is executing will run to completion. So if a function is executing, it can't be preempted like many other languages. This stands in stark contrast to multi-tasking languages where code can be preempted and something else can start running. This is not possible with JavaScript. Whatever code is running will run to completion. This is a lot like in a bank, a customer has to be satisfied before the next customer can be serviced. There is no interrupting a customer that's taking a long time. A customer taking a long time will block the rest of the line, hence why it's important not to block the line. All of this is because JavaScript supports cooperative concurrency. That stands in stark contrast to what you may be familiar with other languages known as preemptive concurrency. What this means is, the people waiting in line, so our code waiting in the event loop's queue to execute, that code needs to play nice with the rest of the code that needs to execute; in other words, you don't want to hog the CPU. If you have a long-running operation, you might want to look into something like web workers, which will spawn a separate JavaScript engine and thread to execute some code behind the scenes. It's like opening up another line at a bank with another teller to service customers. As a result of these features of JavaScript engines, I think that a healthy mentality when looking at JavaScript code and reading it, a healthy mentality entails starting to think about the little programs that our programs are made up of. Start to slice and dice programs around the asynchronous seams so you can understand which parts will execute at different times. This is like looking at the different customers in line at a bank or in line to use a bathroom. If we can understand where these seams are at, then we can reason better about when these little programs will execute, and of course, finding these little programs entails looking for asynchronous seams. A callback is a great example of an asynchronous seam in a program. We had a few other key takeaways. One is that timer delays are not guaranteed in JavaScript. Keep this mind as we work through this course because we'll be using timers to execute code later on, but remember, those timers, the delay is not guaranteed. That's a minimum and that's because when the timer elapses, the work that was scheduled to be executed will be pushed onto the queue and will have to wait for whatever is in front of it. Also, just because code looks like it's asynchronous, for example, just because code involves a callback, doesn't mean it's actually asynchronous, so you have to be very careful that you're not assuming something is async when it's actually synchronous. A good example of this is a button click event handler. When you add a listener for a button click event, all the listeners are fired off synchronously; they're not fired off asynchronously. The last important thing you should understand coming into this course is that you can still have race conditions in JavaScript, so even though a piece of code that's executing can't be preempted, even though that helps with a lot of issues around concurrency conflicts, you can still have issues with the order of operations in the queue. In the previous course we saw how this could be a problem if we fired off two web requests, say one for the current weather of a location and the other for the 5-day forecast for that location. If we start to update the UI of our application and we assume that the current weather will come back first, that may not always be the case and so we can have issues if the 5-day forecast comes back and our UI might be messed up if we're depending on the current weather coming back first. If you understand these concepts, even as I was explaining these and you feel comfortable with these, then go ahead and move forward with this course, but if you'd like to really learn more about these or if you're a bit uncertain, then check out my previous course, Reasoning About Asynchronous JavaScript. Start there and then come back to this course.

## How to Use This Course

There are two paths that you can take through this course. The first part and the bulk of this course will focus on helping you really understand Promises. If you think you have a really good grasp of Promises already, in other words, you think that you could write a Promise framework yourself already, then you can go ahead and skip to the later part of this course where I talk about merging Promises with generators. Specifically, you can jump to the module called Building a Generator Based Control Flow Function, which is module 7. In either case, I created this course for you to follow along, step by step. That's the best way for you to learn and that's the best way to absorb the content of this course.

## Callbacks

In this module we're going to talk about callbacks. I assume you have a pretty good understanding of what a callback is, but let's review this just to be certain because it will be the foundation of understanding the rest of this course. So in general terms, a callback is a piece of code that we pass along when we're invoking some function that we'd like to be invoked once that function is done. So, for example, we have some type of application program, our program, and that main program calls some library function. It doesn't have to be a library function, but this is often the case and this library function might be something like reading a file off of a disk. Along with that function call specifying what file to read, we pass a function that we want to be called once the contents are available. The contents of that file are then passed to that callback function that we pass and that callback function is a part of our application program, so callbacks are a way we can call some other function and then resume our program later on; that's one you could think about them. Let's take a look at an example of this. This is a Node.js example of reading a file off the disk. In this case I form a path to access a file by looking at the current directory and looking for a file called /english-words.10. I'm specifying this is utf8 encoded and then I'm passing a callback function here that I'd like to be invoked once the contents are read from that file. You can see that callback function receives the contents, or if something goes wrong, it receives an error. I have a little code in here to check if there's an error and print the error out if so and abort the rest of this callback, but if there's no error then we go ahead and just print out the contents of the file. So we can run this program and you can see in the terminal here we have the various words that are read out of this file, so this a file with English words in it, so we print out those words. However, if something in this file name is changed, say a dash-dash, which would be invalid then, and we run the program, we now get an error instead, which says that there's no such file or directory on disk. So this very last function here is a callback. Now one thing specific about callbacks in Node.js, there's a common convention to pass the error first and then whatever result second. You don't see this so much on the front end in browser development, but this is prevalent across Node.js APIs. It's a convention that everybody has agreed upon. Here's an example from my previous course and if you run this, you'll see we get the current weather and interestingly, we ran into a race condition here so the five-day forecast isn't showing. If we load this again though, you could see the five-day forecast shows. Check out my previous course if you're interested in that race condition. In this example, we have two callbacks so they don't quite look like the other example. Our callbacks are specified when we're setting up the requests here, so we have two XML HTTP request that we're firing off. If things are successful, then our success functions are called. If there's a failure though, our failure functions are called. In this case, the callbacks look different because the callback for success is split apart from the callback for failures, but nonetheless these are callbacks because we're passing programs that we'd like to be executed once our request has completed, just like reading a file off of a disk and these callbacks are down here below. This style of splitting up success and error callbacks is somewhat common in front-end development, though there aren't lot of conventions. So again, callbacks are functions that we pass to other functions. Those callbacks are invoked when those functions are done doing their work to resume the execution of our program. You could think of our program as this entire block here and it's like we suspend the operation of our program here waiting for some other function to complete. Now I hope you can see how callbacks relate to the asynchronous nature of JavaScript because they become a mechanism for us to suspend the operation of our program. Right in here we're not doing anything if you think of this as a timeline from left to right. In the middle here, our program is doing nothing while we're waiting for something else to complete like a web request. So we're suspending our operation here. This is a good example of using cooperative concurrency with callbacks. Now this assumes that callbacks are asynchronous. Now that we've discussed what callbacks are, let's get into some of the pros and cons of using callbacks. One of the cons is that what looks like an asynchronous callback isn't always asynchronous.

## Double-edged Sword: Explicit Async Seams

One of the features of callbacks when callbacks involve asynchronicity is that they're a good indication of an asynchronous seam in your application. They're an explicit seam. They're easy to see. Now this is a double-edged sword. This both a blessing and a curse. The blessing is that we can see these. The curse is that the rest of our program is nested inside of a function, a callback and certain this isn't a problem with one level of nesting, but as we have more and more asynchronous operations involving callbacks, we then get into these situations, like in the MySQL database example where we have three levels of nesting. Our program is being split up along these asynchronous seams and that's easy to see, but seeing those seams is not something we always want. It can become very difficult to reason about this code. As I said, this is the important part of this code, but we have all this other noise and boilerplate here, opening and closing functions, checking for errors, etc. So one of the problems with callbacks is that the explicitness can get in the way of understanding your code.

## Con: Seams Rip Across Program

A consequence of the explicit nature of the seam involved with callbacks is that those seams tend to rip across your entire program. If you've ever tried to factor out code that involves callbacks, you'll have run into this problem. Let me show an example of what this looks like. Let's go back to the MySQL example we were working with earlier. This is a gist you can grab if you want to follow along. This is a script that starts up, a web server, that listens for a couple of requests to come in, first to get this index page and then also to post a user. A part of posting a new user to create a user in the database requires setting up the database to make sure it actually exists, so there is some bootstrapping logic here that makes sure that the database is created if it does not yet exist, and then it uses that database and then it goes ahead and creates some tables in that database. Now I'm making the assumption that this query function that's called three separate times is asynchronous. It's very possible that it's not, but I don't know why they would use the callback pattern for this function if it wasn't, so let's just assume that it is asynchronous for the purpose of this discussion, because that's something that you will really run into when bootstrapping a database. The problem is, this function that's passed as a callback that executes the rest of this code will happen later on. It will happen after this first query executes to create the database, which can take some time. Same thing with the rest of the callbacks here. We have another callback that will happen after we set the database on the connection and this code will be executed later on. Same thing again, once this last query is done, we have yet again one more callback and this code will execute after that. The problem is, not with this piece of code here so much as if we scroll down and we take a look at the part that inserts a user into the database, it's possible that a web request could come in to create a new user before that bootstrapping takes place and if that happens, though unlikely, we'll have an issue because the users table will not yet exist and so this INSERT INTO statement will fail when it shouldn't actually fail. This is what I mean when these seams rip across our application. We need the database to be bootstrapped before we can actually use it, which really means we need to nest our code somehow to set up the routes for posting users, which means if we want this program to not have an issue with the order of bootstrapping and inserting users, we would need to take this code here that registers a route handler for /users. We have to move all of this code up inside of this callback, if we wanted to avoid any issues with timing. Now in this case this person has taken the approach of assuming that this is so fast to bootstrap the database that it's not a problem and while I would agree with this as a perfectly reasonable approach to take. There are situations where you can't allow this type of assumption to take place; you could actually run into problems, especially with systems that are automated where users aren't involved in submitting a form, for example. It would take a long time for a user to pull down the user creation form, type all the values in and submit it. There's probably no way the database would bootstrap after that, unless of course the database was extremely slow, but this is problem. As we want to modularize and break up our application to have bootstrapping logic, sometimes we have this problem of separating it from the rest of our code that depends on that having actually been executed and we can run into cases where we have to make assumptions like this that can actually cause problems. Could you imagine how ugly this code would be if we had to take everything down here and move it inside of this callback? That would be really, really hard to understand and that's why this person may have taken this approach of just assuming that the timing will work out. So the key takeaway here is that these seams have a tendency of ripping across our entire program.

## Con: Another Error Mechanism

There's another problem we have with callback and that's that they introduce another mechanism whereby we have to deal with errors. I think this is best illustrated with the Node.js examples, but it's also present in the front-end examples we looked at. When we're reading a file, if something goes wrong, we receive an error object instead of the contents of the file, so that means we have to write some additional logic to see if we get an error object. If there's no problem, there is just no error object and thus we can continue the execution of our program. This stands in addition to thrown exceptions. So it's possible this readFile function might throw an exception, for whatever reason. Let's just say we pass a malformed path and it has the ability to check that. We might then have to write a try catch to handle any exceptions thrown from the read file and deal with the error appropriately. So now you can see we have two different types of errors to deal with. We have thrown errors, which we have to handle with a catch block, and then we also have the error object as a part of callbacks, which we have to then write if statements for and have branching code as a result. This branching is also present in the front-end example we have here. We happen to have one nice feature here and that's that the callbacks are split up so we don't need an if block to deal with this, but we do have a separate failure function and that would once again stand in addition to try-catches to catch thrown exceptions that might be thrown as a part of initiating this web request. So either way we have two mechanisms now whereby we have to worry about exceptions or errors happening and more likely than not, you've run into scenarios where you didn't deal with exceptions from both mechanisms and you had unexpected problems with your applications. It's hard enough to get errors right with one mechanism, let alone two, and of course the other problem of this additional error mechanism, especially with Node.js code, is that it tends to repeat itself. Every time we have an asynchronous seam we have to check for an error and in this case this person is throwing the error, which is very likely to bring the program down if there is any problem interacting with the database. Now maybe that's desired, but probably not. It seems to be the pattern in this example. If we scroll up to the other example of posting a new user, we throw an error here as well. This will bring the program down more likely than not, though I would have to actually look at this query function to see what it does to invoke our callback to see if that's the case, but very likely this will bring the program down and I'm sure that was not was intended and that will happen because this thrown exception from the callback won't be handled anywhere by our application and thus Node.js will terminate the process because we have an unhandled exception. So not only do we have two error mechanisms, that new error mechanism is not just another mechanism, it's difficult to work with and as a result in many situations, there's quite a bit of boilerplate involved in this error handling.

## Con: Hard to Reuse Error-handling Logic

This might not be so obvious, but it's hard to reuse the error handling logic. Back over in the bootstrapping code, if for some reason the database bootstrapping fails, we have three separate points where we check for an error to have happened and so three times we have to repeat the same logic, which in this case is to throw the error. Now if we wanted something more elegant like maybe logging that error out to disk or maybe sending a notification to somebody, assuming email works, we would have to repeat that logic or try and put it in a separate function and make that function reused throughout these if statements. However you look at it, it's difficult to reuse our error handling logic. We're used to, with try-catch blocks, being able to define one catch block that can handle the same type of exception that might come from three different lines of code like in this case where we're firing off three queries so it's very likely the same error could come back from all of three of these. It might be nice to just handle that same error with one catch block. Of course, the caveat here as this example is not that great because we could roll this all up into one script, but assuming we had three separate queries that truly were distinct, it'd be nice to roll up some of this error handling logic and reuse it; unfortunately, we can't. In the best case scenario we might be able to do something like throw and have this one line of code here to deal with error handling.

## Con: Difficult to Understand

For a variety of reasons, code that's written with callbacks can be difficult to understand. Now in the case of one callback here to read the contents of file off disk and to asynchronously then process the results, this isn't hard to understand. In fact, this is probably pretty awesome compared to mechanisms you might have worked with with other languages to perform non-blocking I/O. However, when we get to that MySQL example, obviously this is a bit difficult to understand. Aside from just the nesting involved, it's hard just at a glance to understand that there's a series of steps here involved in bootstrapping a database. So one of the reasons it's difficult to understand callback-based code is that there's all this nesting involved and some verbosity, but this isn't the real problem. The real problem is that we're making a bunch of assumptions about the code that we've passed our callback to. It's possible that that readFile function that we're looking at, that readFile function may never call our callback, or it could call our callback multiple times. This could become really problematic if we're doing things that can only be executed once and that we expect to be executed once, but perhaps one of the biggest problems about the assumptions we make is that in many cases we assume that a callback or any time where we pass a function that continues the rest of our program's execution, we assume that that happens asynchronously, but we don't actually know that unless we go look at the implementation of this readFile function and all of the functions that it uses. So we're making an assumption that this function is called later on when in fact it might be called synchronously. This readFile function might be blocking and it might read the file while blocking or hogging the CPU and if it reads the file by blocking, then it may call our function synchronously and this changes the entire game. Go back to that MySQL example. If for some reason this query function is blocking I/O, it may actually call our callback synchronously in which case the database then would be bootstrapped before we ever get to the code later on to register the route handlers for web requests that come in. So then we wouldn't have the problems that I was talking about earlier, but we don't know any of this unless we dig into the code for querying and it's tempting to believe that this is asynchronous given the fact that callbacks are involved. In fact, it's almost mean or evil to write code for Node.js that has APIs that take in what looks like a callback when it fact it actually isn't, but it's possible somebody could even make a mistake and call your function your callback synchronously, and then that changes the entire game.

## Challenge: Synchronize Multiple Callbacks

A while back I said that learning is about doing so here is your first task to try something out on your own, and this will help illustrate another problem we have with callbacks. Right now, pull up this Plunker and if you've followed along with the prior course, this is the same example that we were using in that course, and go ahead and get this Plunker up and running. Now make sure that you put in your own API key for open weather map; this one will not work for you. Then see if you can run the Plunker so that as you're going, you can try things out. What I want you to do is change this code. Right now when we load the weather, if I load this a couple of times I'll see if this raise condition happens. Okay, here we go. Here is an instance of what can go wrong when we have things executing potentially out of order. Most of the time you can see, because we've fired off the weather request for the current weather first, which is this code here, most of the time that completes and so our weatherSuccess function is called first and that function calls showResults. Our five day usually comes back after because we requested after; however, we've just hit a situation where the current weather request took longer than the five day, way longer such that the fiveDay success function was called first and then our weatherSuccess function was called and that's a problem because the fiveDay success function appendResults to add its results assuming that the results were already there for the current weather and then current weather overwrote the fiveDay success results. So we have an issue with ordering of operations. If you come and look at these showResults and appendResults function you can see show overwrites the innerHTML with whatever result you pass, whereas append adds it onto the end. Now one way we could fix this is to use append for both, but that's not without its own problems. For example, now we have the loading… because we call showResults with loading before we fire off our request and we probably don't want to get rid of that, but we probably also don't want to show that and now if we run this a couple of times, it's possible and maybe I won't get this to happen, but we should be able to see the request come back. I'm not getting it here and I don't want to overwhelm my API key for the day, but it's possible this five-day forecast could be at the top and the current weather could come in at the bottom and maybe you'll see this as you're working through this and that's possibly an unpleasant user experience as well if most of the time people expect the current weather on top. In fact, that's definitely a problem because you can see we have to scroll a while to get through the five-day forecast so users may not know that the current weather is at the bottom and might get really frustrated, so even this doesn't work and we could do things to have separate elements on the page where we write the weather success into one div and then right below it we have another div for the five-day success, but we could also have problems with that. I mean, it's possible we don't want to show anything until both requests come back. I'm sure you've dealt with that in your apps. You don't want to maybe route to another page in a single-page application until you have all the data ready to show that page, because you don't want a jarring user experience as the interface slowly comes to life with all the pieces of data coming back. So for whatever reason, I want you to do something with this code on the left-hand side to synchronize and not call showResults until both of the responses come back. So you need to use callbacks so you can't change from using the xmlHTTP request, and by the way, you can go grab the docs for this out on MDN if you are curious about how this works. Likely you use jQuery for making requests so this is the plumbing underneath jQuery. So you can't change from this. You can't use Promises. I want you to continue to use callbacks, but I want you to synchronize the callbacks so that you don't show anything until you get both responses. Now to simplify this to make this tractable without spending several hours, you can ignore the failure case. You don't need to synchronize failures. You can go ahead and just leave this code as is. If you really want an added challenge, you could synchronize that as well. So this is a challenge for you. I'm not going to give you any hints. I just want you to think about this. How can you synchronize these two callbacks so that you only call showResults after you get the response back for both of the requests? Join me in the next clip where I'll go over how I solved this problem. Oh and by the way, it probably will help to click the Stop button and not have the Plunker run every time you change your code on the left-hand side and then click Run when you're ready to test out your solution. That will also help from overwhelming requests for your API key, though I haven't really had problems with the open weather map API limiting my requests.

## Solution

Alright, I'm going to talk you through how I thought about this problem. It's important to point out there that there is no right way to solve this problem. I'm sure you came up with a different way and that's perfectly fine, so long as you came up with a way to synchronize and call showResults once you have everything back, well, that's a solution that works and is perfectly acceptable, but let me show you what I came up with. So, I was thinking in my mind, let's keep these callbacks the same, so we'll still call these, but what we need to do is put off appending the results or showing the results. So let's comment those out for now, and then I was thinking, let's create a global variable, global in the local scope anyways, for the weather data. We'll declare it, but we won't define it so that will be undefined. Same thing for the fiveDay data, declared but not defined. We can print that out just so that you can see that. Okay, so what we'll do then instead here is we'll set this global variable when we get back the weather data and we'll also print that out. The same thing with the fiveDay data. So we've captured those in these global variables, which means that it doesn't matter which order these complete in, once we get both responses back, assuming no failures, we'll have both of the results available to us in this global scope, this local global scope. So we can do then is check here when one of the requests comes back, to see if the other one is ready yet. So we can say if the weatherData variable is defined, in other words it's truthy, and the fiveDay data variable is defined, in other words it's also truthy, which means we assume that it has the object of the response back, then if that's the case we can go ahead and show the results. So we can take this line of code here and yank it out and put it up in here, take the appendResults as well, yank it out, put it up in here, uncomment both of those and instead of referring to the old local variable, we can use weather data and we can use fiveDay data. Now there's a slight problem with that and that's that we need to call this function down here as well, but instead of duplicating that, how about we extract a function for that and we could just show this showResultsIfReady, and we could put that function down below. Go ahead and paste in our code here and now we can call this from both of the other two functions and then how about we print out some information to know when we get both results and if we don't have both results yet we can print that out as well. Okay, this is how I was thinking about this problem. I do want to explain this one more time just to make sure it makes sense, but to do that I want to step through this code so let's throw a debugger statement in here and let's do that right here after we set up our declared but undefined variables to hold the responses. Let's open up the debug tools. We'll go to the console tab and clear this out just so we start from a clean slate and then we'll run our program. Okay, so I've launched the program here and we've got the debugger hit on the line where we put the debugger statement. By the way, if you have any trouble with this example, I'd encourage you to follow along in Chrome. You can see this statement here paused in the debugger and we're on that line with our debugger statement. If you scroll over to the right, actually we did need to log things out because the debugger here is showing us that these two variables are undefined. You can check the console though and see that we've got one, two, three in undefined, which means we've sent off our requests and we've declared these two variables, but we haven't defined them. Okay, so back here on the Sources tab, which you might have to look here in the double arrows to find, and I'll pull this over so we have a little more room to work with. Let's go ahead and throw in a few breakpoints just so that we can easily step through our program by using the Resume script execution. So I'm going to throw a breakpoint in each of the callbacks and then I'm also going to throw a breakpoint into the showResultsIfReady function. Okay, let's go ahead and resume the execution of our program and you can see, actually we land in the five-day success function first, so that must have come back first. So if we then step over, we'll define the fiveDay data. You can see that has the result there with the five-day forecast and we'll go ahead and print out to the console and of course, we're going to then step into our showResultsIfReady function and at this point we don't have the weather data, but we have the fiveDay data, so if we go and hit the Resume button, that will complete out the function here to showResultsIfReady and in that case we shouldn't be showing the results and if we go to the console, we can see we don't have both results yet so we can't show any. Now that we have the weatherSuccess function called, if we step over these lines of code and actually step over this one and into the showResultsIfReady, you can see at this point we now have both the weather data and the fiveDay data, which means if we step into this we'll actually hit our code block that says that we have both results, step over that, and then watch the UI here as we step over this code. First we'll show the current weather and now we'll append the results, so we do indeed synchronize showing the results to our users; we only show them when we have both results back and console will verify this as well. We get the fiveDay data. We don't have both results yet. We get the current weather and now we've got both results. Then we can resume script execution. We'll go ahead and close this tab here. We're done with the debugger. I hope that helps you understand how this code works. Next we'll talk about why this is a problem.

## Parallelism: Wouldn't This Be Nice?

The reason I wanted you to work through this example is that I think it helps illustrate how difficult it is to synchronize working with the result of parallel operations. It's possible, but it's not pretty. The code we have to write to synchronize multiple callbacks so that we handle the responses together is custom to our specific situation. We have to define our own variables. We then have to check those variables to see if they're both defined, indicating that we both results back. Now you could extract this out into a framework, but that would be work and many of those frameworks aren't pretty. What we're ultimately trying to do here is fire off our requests first. When the weather success completes, that hit our weather success function and then when the five-day completes that should hit our five-day success function and then when each of these are done, we'd like to fire off this bit of code here to show the results. So we have this diamond that represents the order of operations, okay it's kind of a skewed diamond, but you can see we start at one place, we fork into two separate parallel requests and then we want to combine those requests back together and we pulled that off here, but this is quite a bit of code to it and if you ask me, it's way too much code to do it. Let's take a look at what this code could look like if we were a little bit liberal with the style and syntax we can use. So let's come up to the top here and let's stop the program and we'll leave this part alone and then hypothetically what would be nice right here is to just have something that says let weatherData = and then somehow it would be nice for the response to come back and then when that came back it would nice if it showed up in this weatherData variable. Same thing with the fiveDay data. It would be nice if we could wait for that response as well and let's just assume that somehow we know that these responses are correct here. Maybe it'd be nice if we had something like the weatherRequest and the fiveDay request. Again, this is all hypothetical, but wouldn't it be nice if it looked like this and then right after, we could go ahead and show the results and then we would assume any problems that happen would raise exceptions. Wouldn't this be nice if this is what this could look like? This is really all our code is doing and yet we have like 25 lines of code to pull this off. That's ridiculous! This is what our code should look like, even with non-blocking I/O happening behind the scenes. Again, this style is something we'll see how to pull off in the rest of this course.

## One-shot Events

I may have made it seem like callbacks are somehow evil and should never be used; that's not the case. I'm trying to set up the impetus for this course and show you the value you can get out of new techniques. In the process I had to beat up callbacks to help you understand why these new techniques exist. However, there are plenty of situations where callbacks are perfectly fine, for example, with streaming data. This is an example of using streams with Node.js and I don't expect you to understand all of it, but I want to point out when data comes in, and this is the web server in this case, when data comes in, we have a callback to handle that data and process it. This could fire off multiple times, so it makes sense to use a callback. You have to specify some function that will be called each time data comes in. We also have eventually the end of a stream of data and we'll need to handle that differently so we have another callback in that situation, so callbacks work well here. Over in the Plunker we've been looking at, this load function, it may not be obvious, but if you look at the indexHTML page, this load function is called when somebody clicks on the button. This button onClick is specifying a callback here that gets executed when somebody clicks. It may not be so obvious, but we could have wired that up in code as well with something that looked more like a callback like we're familiar with, but nonetheless we have a callback here and with buttons, people can click on those multiple times and each time we click on the button, we'll have to execute something and a callback function is a nice way to package that up. In these situations, I don't think there's anything wrong with callbacks. I think they capture the intent and they don't really cause that many problems, but there's a big difference between these situations and the ones we'll spend most of our time with in this course. The situations where I'm mostly okay with callbacks or that wouldn't be solved well with the techniques that we're talking about in this course, deal with a stream of events, so events that can fire off multiple times. The situations we'll be dealing with in this case you could think of as one-time events. For example, firing off a web request. That only will complete one time so the event for completion will fire off only one time, so you can think of the examples in this course as focusing on one-shot events, events that fired off one time, so something is going to happen in the future and it's only going to happen one time. It will either be success or failure and that's it. That's what we're focusing on in this course and that's where callbacks tend to fall apart. If you don't like callbacks in the context of events that fire off multiple times, any number of times for example with button clicks, or streaming data, then you can take a look at reactive frameworks because these are an alternative approach to dealing with streams of data, event streams, for example, and because I only have a limited amount of time in this course and I want to focus on one thing, I won't be covering this in this course so this is actually a great topic for a subsequent course.

## Takeaways

Because JavaScript is single-threaded, we know that we have to avoid blocking or hogging the CPU and one of the ways of doing that is to use callbacks whereby we can pass off responsibility to some other code that's happening behind the scenes in the case of making a web request or reading a file off disk, and then that code can call us back later on and we can resume the execution of our program. This way we're not hogging the CPU waiting for the file to come back from disk or waiting for the web request to complete and this is absolutely essential if we don't want to obliterate the performance of our applications and since we don't have multi-threading, the only way that we can have some level of concurrency is to do things quickly and to be able to switch contexts rather rapidly and so callbacks are one way to do that. Unfortunately, as we've seen, callbacks don't come without their share of problems. While it's nice that the seams of asynchronicity are explicit and it's easy for us to reason about them, for example, it's easy to understand when we're reading a file off disk that we have this asynchronous seam where processing the contents of that file will happen later on, so we know we're not blocking there and hogging the CPU. While that's nice in the simple case of doing one asynchronous operation, it becomes a nightmare when the complexity grows and those complexities are a reality of developing software. So callbacks can literally become a nightmare and we're one of the main reasons why I said that we just stop using Node.js for the time being. Fortunately, there are plenty of good alternatives and we'll take a look at those in this course. We saw how it can be difficult when these seams tend to rip across our program, for example in initializing the database before and then inserting data into it. We saw how we can have problems with timing or we can have problems with insane levels of nesting. It's hard to compose code that involves asynchronous callbacks. We also saw how callbacks introduce yet another mechanism whereby we have to deal with errors and it's not as if we don't have enough problems with try/catch, now we have yet another mechanism to deal with errors and that mechanism has quite a bit of boilerplate involved in it and also it's very difficult to reuse the logic we set up for error handling; we tend to have to repeat ourselves. As a result, it's very difficult to look at code that involves asynchronous callbacks, especially when there are multiple asynchronous operations. There's a lot of boilerplate to look at and that leads to a lot of nesting, but that's not the primary problem, though nesting is a huge drawback. The real problem is, that code we're passing off to, for example to read a file or to make a web request, we don't know if that code will ever call our callback. We have no control over that, so we might have to code up defensively to handle the scenario where, say a third-party library doesn't call our callback. That's extra work we have to do. We also don't know if our callback might be called multiple times. Usually the convention is that you don't call callbacks multiple times, but who knows? Somebody might introduce a bug and that could happen. Again, we'd have to defend against that whenever we're using callbacks. We also don't know if callbacks or code that looks like it's using callbacks actually fire off asynchronously or if they fire off synchronously and this can drastically change what the code is doing and thus drastically change our understanding of it and without getting under the hood of these functions that we're calling and passing callbacks to, we can't know any of this and even if we can figure it out, can you imagine trying to remember all the nuances of every function you use in regards to how it may cause problems with callbacks? It's a nightmare. Callbacks are not predictable. And of course, the last thing we saw is that synchronizing the result of parallel operations is difficult and this is just one of many different flows that could happen in your program. What if it's more complex than what we looked at? What if your program fires off four separate requests and then has to wait for all of them and then let's say one of those fails and the other succeeds and then you have to do something because of the failure and you also have to do something with the result. As you can imagine, handling that would be insane. This list is by no means exhaustive of the problems we'll run into with callbacks and we'll cover more throughout this course, but it's a good starting point for understanding how the alternatives address these issues. One of those alternatives is Promises, which is what's up next.

# Lab Setup

## Cloning the Starting Point from GitHub

Before we dive in and create a Promise framework, let's get your computer set up for you to be able to follow along. If you look me up on GitHub under g0t4 and then search for the repository pluralsight-modern-async, if you open that up you're going to find two branches inside of here. First is the master branch, this is the starting point for this course. So you want to clone this to your local computer. There is another branch on here as well called step-by-step. This is the ending point for this course. So this contains all the individual comimits, and if you look at the commit history, there's a rich history here for you. This contains all of the individual commits for you to work through step by step everything throughout this entire course. I committed just about at the end of every single video. So you can come out here if you ever get stuck and copy and paste to pick up where I left off if you're having a problem with a particular scenario. So feel free to do that. But for now go back to the master branch and go ahead and clone this down to your computer. So come over to the right hand side here, click clone and download, make sure you're on https and then copy this URL. Once you have that copied, then hop over to the command line or whatever git client you prefer and run a git clone and paste in that repository URL. That will pull everything down to your local computer change into that directory and then if you list the contents here you will see the individual files that are out on the starting point for this GitHub repository. And again keep an eye on this repository because I have all the individual commits out there for you if you ever get stuck throughout this course.

## Operation Examples

I thought about having a Plunker that you could use throughout this course to follow along, but the reality is, we're going to be writing a lot of code and it will be nice to organize things. It will be nice to have an IDE experience as you're working through things, especially if you want to do some advanced debugging. I'm going to be using WebStorm in this course; you don't have to use it, but if you want to follow along, I'll give you some tips and tricks to help out and make you more productive. So what I'll do first is go ahead and open up that repository I cloned with WebStorm. In the dialog that pops up, I've navigated to the location where I clone that repository. Do the same thing if you're following along and then click OK. This will open up the example inside of WebStorm and will create a new project for you. If you hop back to the terminal and list the contents of that directory again, actually in ls -a, you'll see there's a new folder called .idea and this is the project files for WebStorm. Right now I've set this up to be excluded from check-in so you won't have any of the settings I have for WebStorm, nor will any of your settings be checked in. You could modify the gitignore if you want to change that, but I don't typically check that in. So what I'll want to do is help you set up a few things in WebStorm to make you productive with the examples we have in this course. First things first. Go ahead and expand in the project view and open up the source folder. This is where all the example code is at. The rest of this is boilerplate to use a Mocha test runner to be able to run some tests on the examples we're working on. I'm using Mocha because I'm building out the examples for this course and I realized there are quite a few examples I want to work through and it will be nice to have separate test cases for those to isolate those scenarios and that way we don't have to work on one scenario and then comment it out to be able to start on another one and not have them interfere with each other. So we use Mocha for testing mostly just so we can have multiple scenarios and isolate those and I'll show you what that looks like in a minute. Anyway, those files are at the root of the project as well as a package.json file for some dependencies we'll be using, but we will be running these tests inside of Chrome so we will be doing the examples inside of Chrome. We're not using Node.js in this case, just because I have WebStorm and I'm working locally with a package.json; I'm just using that to fetch some dependencies that we can use in the browser. We will be running these examples as if we are writing this code for front-end applications, though the nice thing about Chrome, behind the scenes it uses v8 and Node also uses that, so what we develop here will also work with Node.js without any changes and what we develop here will work just fine in other browsers as well, unless they're really old browsers. Okay, so come up to the source folder and go and open up this operation file. Inside of here are a few functions we'll be using throughout this course to talk about issues we have working with callbacks. I have one function that involves getting the current city and you could imagine this uses someone's GPS location or it uses some location service to figure out the city that I'm in right now so there's no data we have to pass to this other than a callback that we'd like to be executed once the current city is available. So you can imagine this uses the phone's GPS unit to figure out where the phone is at and returns a string that is passed along to this callback with the city name. This is something you might use if you're building an app to show the weather because you'll need the city before you can figure out the weather for a given city. So you could get the user's current city and then get the weather and then also get the forecast, so both getWeather and getForecast take a city name and then hypothetically look up and return back the weather and forecast respectively, and if you look at these functions, if we expand any of these out, all these do, all three of these have the same pattern inside. They all call setTimeout so that basically we call the callback later on and there's a delay specified. You can control this is you want. It's 1 millisecond right now. I just want to make sure that these functions are returning results later on in the future asynchronously so that they're not synchronously returning. That way we have the foundation of examples that are asynchronous to work on as we look at Promises and generators and generator control flow functions throughout this course. As you can look through these, these are just stubbed out methods; they don't do anything, they don't interact with any real APIs, but they are methods that you might find in an API. So I wanted them to look realistic in terms of what functionality they provide, but I wanted them to all be fake so that we're just running locally and you don't have any issues with interacting with a remote web service throughout this course, and so things are fast; as you're tinkering you can get quick results and not have to wait for a web API, so anyway getCurrentCity just has a hard-coded city of New York, NY and that's because that's where I live so of course, I'm going to use New York. After the delay, this function will fire off. This function will basically call the callback. There will never be any error cases and it will just return the city. If we look at getWeather and getForecast, the same pattern, though the function that is executed a millisecond later in both of these cases, make sure that a city was passed. If no city is passed, then an error is returned to the callback in both cases. If, however, a city is provided then this hardcoded weather or forecast is returned. So we have a weather with a temp of 50. Pay attention to this because as we're working on examples, you'll see temp:50 in the console a lot when we lock information out. So you can see we just passed the weather result back to the callback in this case and in the getForecast case we just pass back the hardcoded fiveDay forecast. Again, pay attention to the hardcoded data here; you'll see this in the console a lot. So these are the operations that we'll be working with. For the most part you could just close these up and imagine these are a real API and you don't need to know how they work, you just need to know that these are all asynchronous. They're all going to take some time to process; they're not something that we're going to block and wait for synchronous results from and they all take a callback and specifically they all take a callback using Node.js callback convention. So the callback, we always pass the error if there is any first and then we pass the result. So in this case where there is no error, we just pass the result and null for the error. If you look at the getWeather function, when we have an error, that's the first thing we pass to the callback and we pass nothing else so there's no result; however, if this is successful, we pass again null for error because there's no error and then we pass back the result object. So this is using Node.js callback conventions and I did this because these are probably the most difficult callback conventions to deal with and the browser in a lot of cases, we see separate success and error callbacks. For example here I have this JSFiddle and you can see there's a lot of code involved here and you can pull this up if you want, but more or less we have the geo location API we call getCurrentPosition, which by the way, this was my inspiration for the fake examples I have and to this getCurrentPosition function, this is a real API. We pass a success handler and you can kind of see this behind the word JavaScript here and we pass an error handler, so we pass separate callbacks in the front end. This is pretty common in the front end, but in the back end, the Node.js convention of one callback that receives both errors and successful results, that's pretty typical and it's, I think, the more complicated scenario so we're going to start from that position because it's very likely you'd do some Node.js development as well and it can't hurt to have a little broader perspective. By the way, if you want to see what this example does, you can load it up on your own computer. It uses the location API on your browser to figure out where you're at and then it shows a map for you with Google Maps, so you can see where I'm at. You will need your own API key though if you want to use this.

## Running the Tests

Okay, in addition to the operations that we'll be using I have some examples pre-canned here. First, a set of callback examples that relate to the concepts we talked about having problems with in the first module, so you can open this up and these are examples using the operations I just went over of many of the problems we have that we discussed in the very first module of this course. I also have a set of Mocha examples and we'll walk through a few of these in this lab setup just to help make sure you understand how to use Mocha and if you want to follow along, a few tips about debugging. So I have these tests that are written in Mocha and you might be wondering, well how in the world do we run these? Well, what you need to do is open up a terminal and you can do that inside of WebStorm if you want, or you can use the terminal outside of WebStorm. On Windows you can use a command prompt or a PowerShell session, either of those are fine, and from this prompt you'll want to run npm install first to install the dependencies we're using, just a few dependencies here. We pull down live-server to run the test and watch for changes and rerun the tests in the browser so we have an automatic rerun for our tests as we're making changes and then I also have this expect assertion framework. So, two dependencies we need. Once those are installed, then you can run npm test. This will open up a browser window and it will run the test for you and it will show the results. If I split the screen here, we can see our code on one side and our browser on the other. Now a few things, I'm going to minimize the terminal because we don't need to see the output there; we'll focus on the browser for that and if you double click on a tab in WebStorm the project view will hide. Okay, so the very first thing that loads up are just the callback examples and that's because I'm skipping the Mocha suite, so in the callback examples, these are examples that relate to problems we had in the first module and one thing you might notice, these are all garbled up. Join me in the next clip where I'll show you how to set some settings in WebStorm so that you don't have all this garbage and yellow and red all over the place.

## WebStorm Settings

So when you first open up the test, very likely you're going to have a bunch of yellow all over the place and that's because the language version is not set correctly by default, unless of course you've changed your default, but you can go into the preferences for WebStorm and you can set the JavaScript language version to ES6. By the way, that is under Language Version if you want to search for that and then Language and Frameworks JavaScript and you can pick the version there. Go ahead and click Apply and click OK. WebStorm will update and you'll see the yellow is gone, but now we have this banner across the top asking us if we want to use Babble to transpile this code. We don't want to do that so click dismiss and then just close and hide that message. Another thing that I find helpful in WebStorm is to open up the preferences again and look for the JavaScript libraries and you can search for this. Under JavaScript there's a Libraries option and under here are libraries that are loaded that will help you with auto completion. If you click Download, we can add a few that are relevant to our project. First off, you should grab Mocha so just type in mocha and go ahead and click Download and Install. This is actually giving some IntelliSense from the typescript definitions for Mocha. This will still work with just vanilla JavaScript as well; you don't have to use typescript to get the advantages of this with WebStorm. In addition, click Download and also look for Expect. This is the assertion framework that I'm using in my examples and by the way, if you're wondering what this will also do, you'll notice some dots under things and some of the colors for the keywords involved in the testing frameworks and some of those warnings will go away once we add these definitions. Go ahead and click OK and then if you go to the Mocha example, this will probably be the starkest contrast here. See how toBe is colored as if it's a known method? If we just go ahead and delete off what's there and we do a .to and then we type out Be, you can see we get some IntelliSense with some of the methods that are available to us. So we can auto complete, which is helpful when we're writing out expectations to not have to know exactly what this assertion framework uses, though if you want, I'd encourage you to look at the docs for this assertion framework. It's pretty lightweight and simple; that's why I'm using it. It has methods like toBe, which we'll spend a lot of time using, just expecting something to basically equal something else, toBe using stricter quality whereas we have toEqual for using non-stricter quality. Anyway, you can come here and look through the docs for that as we're working through the examples. Another site I'd encourage you to look at is the mochajs.org website that has examples of how to use Mocha that we'll talk about using Mocha briefly here in the next clip.

## Auto Running Tests

Let's take a few minutes here and let's see how we can run the tests that we have in addition to using the doc side. I want to walk you through a few of the features of Mocha so that you're familiar with this as we work on this throughout this course; if you're already pretty familiar with Mocha, you can skip this. Okay, first things first. I'm going to split the screen here. On the left-hand side will be my code and you can see right now that these Mocha examples, this sweet Mocha with QUnit interface, shows nowhere inside of the results and we only have callback examples, which are coming from this suite of tests and that's because right now I'm skipping the Mocha suite of tests, but what I can do is replace the word skip with only, so if you have suite.only you can define the only suite.only, you can define the only suite you want to run, which is the inverse of skipping a suite. Now when we refresh the browser, you'll see different examples show up. A lot of red because I wanted to show a lot of cases where failures happen. Now one thing I want to point out, I told you I have live reload going on behind the scenes. Inside of WebStorm, if you make any changes to the tests, like for example, renaming this, WebStorm will automatically save the changes to disk, but if you want to trigger the watcher to pick up those changes and reload the browser automatically, you can come over to the browser and click on it. The watcher will be tripped. Something behind the scenes with WebStorm actually sends out a save signal of some sort, but switching tabs to switch windows is not exactly that friendly. Instead what you can do and you can see the exclamation points, by the way, and so what you can do, if we get rid of these exclamation points, inside of WebStorm if you just run Save All, which Control + Shift + S is the shortcut for me, you'll notice the test on the right-hand side automatically rerun there. Look for Save All in the action search, which by the way, the action search or Control + Shift + A in WebStorm is how to access that.

## A Few Pointers About Using Mocha

Please take a minute if you're not familiar with Mocha and look at these Mocha examples and make sure you understand them. I do want to point out a few things. Down below in addition to writing an assertion for a passing test, down below I have some examples of throwing errors because we'll be working on a lot of asynchronous scenarios and I want to make sure you have an understanding of what happens when we throw an error asynchronously and nothing handles it. So the second test case here is called throw something, but we have an expectation that something will be thrown so we're using the expect assertion library and we're saying that this should throw something, so we expect it to throw "oh noes," and then we pass a function that when invoked will throw "oh noes," and it does so you can see on the right-hand side that that test is passing, which by the way, in the browser you can see the test by clicking on the name and it will open up and show the code here. The next test is synchronously throwing an error that's uncaught. So if we just throw an error in our test and we don't catch it and write some type of assertion about it, then the test runner will catch it for us and it will show us the error. You can see "synchronous throw, uncaught." It shows Error: oh noes. Pretty straightforward. Then I have an example of asynchronously throwing an uncaught exception, so we set a timeout to do something in the future asynchronously and then we throw an error in this context. Now this will look different though. You can see "asynch throw, uncaught" on the right-hand side and you'll see it shows "uncaught error" in addition to showing the error message "oh noes." So these two look different so just because of the context within which we throw an exception, the result will look different, so keep that in mind; don't be shocked if you see if this uncaught error. What that means is you likely have an exception thrown asynchronously. Okay, next we have a synchronous assertion and I want to point out that when an assertion fails that's also an error. Sometimes we gloss over this, but assertion frameworks work by throwing errors so this line here, expecting true to be false, will obviously throw an assertion error and you can see over on the right-hand side that's indeed what it does, so synchronous assert is an uncaught error as well. Expected true to be false. So both throwing an exception ourselves or having an assertion that fails will both result in an exception. Keep that in mind and the key pivotal piece of this and why I wanted to show you this is here is an asynchronous assertion and when it fails, it's also an uncaught error and this may not be so obvious and this can really confuse you as you're working through examples. So here we use setTimeout and we have the same assertion after a millisecond and this will be executed asynchronously so this will also be an uncaught error, but it will look completely different then the synchronous counterpart of these assertions and this might throw you off and lead you to believe something is awry, but again, this uncaught error is an indication that something asynchronously has failed and in this case it's an assertion. A few other things with Mocha. If you want to isolate and run just one test, you can come right after the word test and type .only. We saw that work on the suite level; that also works on the test level. Control + Shift + S will save the files, which will trigger a live server to reload the page and you can see on the right-hand side we have the results for just one suite. You can also skip a test with skip. Control + Shift + S will run everything, but this test. You can see at the bottom this test is blue. If you hover over it, it will say pending, which means the test is ignored. So if you have a troublesome test you could turn it off with skip. Keep in mind, only and skip work on the level of a suite, so if we want to turn off these Mocha tests, we can come up to the suite level and just type in skip. Now when we Control + Shift + S to save everything, you can see the callback examples run instead and you can go ahead and do that or you could delete this Mocha examples file or remove it actually from the index.html page where we're referencing this file. So in the index.html page we load up these files. So by the way, if you want to have other files with tests in them, just add them to this list here so that they get executed as well. Okay, join me in the next clip where we'll talk briefly about those callback examples.

## Callback Examples

Before we move on and get into promises, I wanted to point out the callback examples I have pre-setup here. You should be familiar with most of these already; these are things we talked about earlier in the course, but I want to point out a few of these. First off, the first scenario shows nesting, serial asynchronous dependencies, so we have getting the current city followed by then getting the weather. By the way, with Mocha if you have an async test, you'll receive done to the function and you just call done once all your async callbacks complete. If you don't call done, the test will time out and you should see that right now on one of the tests. We have a timeout on this second test here. You'll see timeout of 2000ms exceeded if you don't call done. Here's another example. I'll close that one up. We get the current city, then we get the weather, and inside of here the key is the error handling is rather verbose. Four lines of code here, we could try and collapse this down, still verbose, it's hard to reuse. We have to have the error handling here and in here and in this case I've commented this out to point out that we can easily forget this and also remember, this error handling is yet another error handling mechanism we have to worry about. So this suite in the browser is failing; it's timing out and that's because we need to actually add in the error handler, so if you uncomment this block of code and save that file, you'll see it reruns now and the verbose test about error handling is now passing because I intended it to fail here to also show you that if you want a test to error out in an particular undesirable situation like this example here of failing to get the weather, you can pass an error to done and that will fail the test as well. We could fix this by actually passing the city here. Save that. Now the test is passing, except that I'm not calling done down here, which I should be doing. I'll save that. Now it's passing. I have some more examples. The only other one I really want to point out right now is this parallelism with result synchronization. We saw this when we were talking about fetching weather and forecast data in parallel. I've rewritten that for the operation samples here. It will look a lot like what we had before, except I have a finish if ready function that checks if we have both of the results back. Obviously, both of these when we call getWeather with the city and getForecast with the city, these fire off in parallel. When the callback is called, we cache the result here globally in this global local state and then we see if we're ready to finish. This will only work after we have both results, though and in that case we'll call done to finish out the test. By the way, I have console logging in here. With Mocha if you pull open the console in the browser, clear out the console, you'll be able to see output, though right now with all of these tests going on, this will be really confusing to look at so what we can do, since this test is called parallel result sync, we can come to that test right here and we can click the arrow on the right-hand side to open up just this one test, so that's another way you can run just one test. You'll see the URL changes. There are query string parameters that specify the test we want to run and only that test runs, which means that the output here makes a little more sense, and actually it looks like suite skip is not supported; it just works because it breaks that file, so just a heads up. Let's go ahead and actually remove the Mocha examples because we don't need those. Okay, now this reruns, the console makes a little more sense. We have our weather object that comes back, so getWeather completes first. We're not done yet though when we call finishIfReady. So not done yet gets called. Then we get the forecast logged out and then we get both done because we have both results back, so I just wanted you to see that because I'll refer to this example and I want to refer to this version of it and not the version we built out in Plunker earlier in this course.

## Debugging

One more thing I want to point out. If you'd like to debug these samples, you can do that right inside of Chrome. I'm going to maximize the window here. Right now I'm honed in on just this parallel result sync example. Probably the best way to debug is to make sure you only have one test selected. Again, the way to do that, the whole list here is loaded. You can click the arrow on the right-hand side corresponding to the test you want to look at. Click that and you'll notice it reruns it and it only shows the console output for that test, but in addition if you want to debug, go to the sources tab. Click on the relevant file under source, so in this case we're in the callback examples. Probably make this a little bit bigger so you have some room to work with. By the way, you can Command + or Control + on a PC to zoom in on the debugging experience if that's not big enough for you. I'll try and zoom in whenever possible so that you can see what's going on on my screen. Okay, once the source is pulled up for the relevant files, so callback examples here, you can scroll to wherever you would like to have a breakpoint at. In this case we're looking at the parallel test so let's scroll down to that and let's just throw a breakpoint in here on one of these lines. We could also put it here in the callbacks and let's put one in the finishIfReady function, so let's put one in there. So now we'll step through and see the ordering of these log outputs, so let me move this down to the log line. Okay, so now when we run our test again. Click Play. By the way, that arrow button is really a Play button in addition to opening up just that one example. You see we're paused in the debugger and we are on that very first line and we can just step through with the Resume button here and this will hit each of our breakpoints. So we hit our first breakpoint. We're getting the weather. We have the weather back now. We're logging out the weather. By the way, you can hop over to the console if you want and you'll see nothing is logged out. If I resume, in the console you'll see the weather. We hit our second breakpoint and we're not done yet. Look at the console, not done yet. Sources. We're back here on our next breakpoint. We're logging out the forecast. Resume. So you can see the forecast in the output now and we're hitting the last breakpoint, both done, and we can go ahead and hit the button one more time to finish this out, but one thing I want to point out. Check this async box. This will give you asynch call stack history. If you don't know what that means, watch my previous course for more on that, but basically you can debug what goes wrong when you have asynchronously executed code.

# Building a Promise Framework

## Why Build a Promise Framework?

Let's turn our attention to Promises now as a first alternative to callbacks. It's very likely that you have worked with Promises already; perhaps you have and you didn't even recognize it. jQuery has had Promises for quite a while now as an alternative means of requesting data from web services. The jQuery.getfunction to request data from a remove URL and then process that data takes advantage of Promises and it's used Promises for quite a while now. Here's an example of using get to fetch this example.php resource. I guess this returns some data that we want to use. Now in an older version of get, we could've passed a callback that would be called upon success, but instead of passing a callback, with Promises we can chain our callback on instead with a call to done, so when we call jQuery.get, an object is returned back, a promise, and on that promise is a method called done that allows us to pass our callback. This is really interesting. One of the key benefits is that it allows us to pass our callback later on. Now we've seen styles of passing callbacks later on, for example with the XML HTTP requests so this is not new, but this definitely is a big feature and we'll talk about that in this module. Along with success handlers, Promises split apart success and error handler so we have a fail function that allows us to specify a callback that's called when there's a problem and there are more features in jQuery, for example, in always and you could think of this like a try-catch and then a finally block. We'll get into more of this later. Now I could explain Promises to you from the perspective of using them, for example, with jQuery, but I think it's much easier to understand Promises and really benefit from them as you use them by first building out a Promise framework from scratch and as we go, I'll show you why we're building each piece of that Promise framework, tying it back to the problems we have with callbacks, tying it back to the problems we also have with asynchronous programming in general and thus you'll understand the impetus of all these features, not just how to use them.

## Challenge - Split Success and Error Callbacks

The best way to start building a Promise framework is with a challenge and I want you to come in to the operation.js file and right here after the operations that are defined, I want you to take the getCurrentCity function and I want you to change it and I want you to change it and have a new name for the changed version; I want you to change this function so that instead of taking one callback that receives both an error and a success result, I want you to be able to pass two callbacks, like that geo location API we saw in the browser. So you figure out how to do this, but inside of a test case so that we can isolate this, so call test with parentheses, but quotes to describe the test, we'll describe this test as fetchCurrentCity which is the name of the function I want you to use instead of getCurrentCity and we want this to have separate success and error callbacks, so that is a description of the test. We'll open up a function which will contain the test body. I want you to implement this. Now go ahead and pause the video and do that, unless you want a hint. If you really want a challenge, pause the video and don't listen to this hint. If you feel like you need a hint, then here's one. Okay, so here, I'll show you how I want you to use this function. So hypothetically, we're going to have a fetchCurrentCity. I won't show you how to implement it, but I will show you how to use it, so we'll call fetchCurrentCity when we're done and we'll be able to pass a success and error callback, so something like success and error here and we should have two functions, one for success and that will receive a result and let's just say we log out the result and then we also have a function for errors. That will receive the error and let's just also log out the error in that case, and we'll call this errors instead of error. Perhaps better names are on onError and onSuccess. So that's what this looks like to use this function, now I need you to implement this function and the only other hint I'll give you is you can have it call the getCurrentCity function that exists.

## Solution - Split Success and Error Callbacks

Okay, so here's how I solved this and by the way, I'm going to make this fetch function available globally because we'll work on this with other test cases as well. So I'll come up here and I'll write the function, fetchCurrentCity. Now this will take both a success and an error callback, so how about we keep the names onSuccess and onError and then what I'll do is I'll call getCurrentCity and I'll create a callback here and of course, we get the error or the result, and then all I need to do in here is say if the error is set then we know we have an error so we'll call onError and I can return then so I stop executing because then I'll want to call success otherwise and pass the result. So that's what this challenge entails. It's just rewriting the signature of this getCurrentCity function so that we have two callbacks. What we've done here is effectively removed some of the insanity of error handling because we now have two separate functions. We as the consumers of this function don't have to write the if statement anymore. Now you might look at this and go, well, we have two separate functions now; that's more overhead and you're right to a degree, though we didn't have to create these functions separately, we could've wrote these inline and in fact, we could have used really concise syntax as a part of ES6. We could use arrow functions instead, so instead of onSuccess we could've taken the result and just console.log the result and then instead of onError we could get rid of that and we could get the error object. We could log that out as well. This is more concise and shows the benefits of having just two separate functions for these two different cases. We don't have to write the if statement and now we have everything in one line of code, though that might be a little bit difficult to understand if you're not familiar with arrow functions; even if you are, it's a bit terse, so what we could do is break this out over separate lines here and that looks pretty readable, pretty easy to understand, so we fetched the current city. If we get the result, then we log it; if we get an error, we log that too. One thing of note is that this is purely a solution based on callbacks, although we're going to be working with this thing called Promises, Promises are really just using callbacks behind the scenes, so I also wanted to in this example show you that everything we do with Promises is just a new way of working with callbacks. By the way, if you're not familiar with ES6 syntax, I'd encourage you to check out some of the updates to the JavaScript language. I'll be using these interchangeably throughout this course; I'll expect you to look up some of the basics, for example, an arrow function. If you want a good resource, this ES6 feature site is pretty good. You can click through here and see examples of some of the new features, for example, arrow functions. You can see how these work and you can see both the new ES6 format and the equivalent ES5 format. So in this case, an arrow function is just another way to write a function, so if you're not familiar with them, just tell yourself arrow functions are just another way to write a function and specifically the function declaration, though in my case I'm also using a very, very terse version of this arrow functions where I don't even have to have blocks, so I don't even have to have the braces around this, though I could have them if I wanted them. That allows you to write a function very concisely in one line of code. One last thing I'd like to say before we move on, this is how I solve this problem. There is no right or wrong way to solve this problem; this is a challenge I wanted to throw at you; however you came up with it is fine. I'd encourage you to type up what I have here though because we'll be building on things from this perspective, but whatever you came up with is fine as long as you solved the challenge and even if you didn't solve the challenge, it's okay too. I'm not here for the correct solutions, I'm here to stimulate your thinking. There's no such thing as a correct solution.

## Add an Operations Suite

Next up, it might help to see this running in the browser and step through the logging we have here. So we can come back to the browser and you can see right now we just have the callback examples and that's because we left off with just looking at one test in the browser. If we go to the root of the server though, we can get all of our tests and in this case we can see our new fetchNewCurrentCity test, so let's do a few things here. First off, let's add a suite and we'll call this suite operations and then let's just make this the only suite we run right now; we don't need to be executing those callback examples every time we make a change. If I pull up in the browser then and refresh here, you'll see the tests run and now we just have our operations suite with the fetchCurrentCity with separate success and error callbacks and of course, I can open up the debugger tools here, go to the console and clear that out. I can rerun it with the play button if I want. It's a bit hard to click on with the screen so small there so I'll make that a little bigger and click on that and you can see we get New York in the output here. If I wanted to, I could come up here and I could pass an error instead. Save everything and you'll see the test reruns; now we show error in the console, but let's put that back to null because getCurrentCity doesn't need to throw an error for the examples we'll work on. If we want to throw errors in tests, use these functions for getWeather and getForecast or write a special function just for the test you're working on.

## Using Done to Assert

Let's briefly talk about assertions here with asynchronous code. Right now we're logging out the result or the error object and that's great when we only have one test, we can pull open the console in the browser and take a look at what comes through, but that's not so helpful when we have a lot of tests, so what we can do instead is come to the end of our test declaration and receive a done function from Mocha and that will indicate to Mocha that we are working on an asynchronous test and the test should not complete until we call that done function. So if I save this code here, you'll see the browser refreshes and the test times out so it's failing now. So whenever we ask for that done function, Mocha will wait for that done function to be invoked. So in the case of a successful result, we can just call done; that's a good enough indication that our code is working here. Here we just wanted to make sure that we appropriately split out the behavior of success from errors. If we save that again, now the test is passing because we've called done. In the case of an error, we can just call done and pass the error. Now this shouldn't happen in this test case, so if it did start happening, we'd know our test is broken and we'd know because Mocha would pass the error along and show it in the browser. Let's trip this to work though. Let's go up and modify this new fetchCurrentCity function we have. Let's modify the callback and let's also call onError at the start of this callback regardless of what comes back and we can just put something like regardless in this string. What do you think is going to happen now? Do you think that things will work okay? Let's save this and see what happens. Ah! It looks like we've got something odd going on here. Our test duplicates in the browser. There are two instances of it and Mocha tells us that we called Done multiple times down at the bottom of the output, so one thing that can indicate a failure if is we call Done twice, so you can use Done and multiple places in your test case in order to catch scenarios where callbacks are called inappropriately, like in this case we shouldn't be calling both error and success. Of course, one way to fix this would be to come to the start here of calling onSuccess and just commenting this out, saving that, and now we don't have the duplicate call to Done. Instead we have the error regardless here that we passed back and you can see that that hit our error case then and if we look over in the browser and scroll over to the right here, you'll see our message here, regardless. So when you pass an error to Done, that's also the same thing as failing a test. So we can use this instead to indicate to Mocha and get automatic feedback on success or failure instead of needing to look at the log all the time. I still will use console log statements for looking at things as we're going along, maybe with debugging as well, but we should also have some assertions in here so that we can rely on these past test cases to make sure we don't break something as we add more features going forward.

## Challenge - Pass the Callback Later

Okay, it's time for another challenge. Let's write a new test case to isolate this functionality. We'll open up a new function body for this test case and here's a description of what I want you to do. I want you to modify the fetchCurrentCity function that we have here. I want you to modify it so that we pass the callbacks later on. I don't want you to pass them when this function is invoked anymore. Now that's all I'm going to give you if you want the supreme challenge here. If you want a hint, stick with me and in a few seconds I'll give you a hint, otherwise go ahead and pause the video. If for whatever reason you want a hint, for example, if my explanation of that doesn't make sense, that's perfectly fine; let me show you how we might use this function after we make this change. So before we add fetchCurrentCity and we will call that when we pass success and error, we don't want to do that now, so we should be able to call it and then later on, we should be able to pass the success and error handlers. Pause the video now if that makes sense to you now, if you want a more difficult challenge. If you'd like another hint, I'll give you another hint here. The next hint is, if we want to pass something later on, we're going to have to capture some variable here that comes out of calling fetchCurrentCity. Fetching something is an operation so how about we just call this operation? That's a nice generic way to think about this, so we're going to capture some representation of an operation and here's the other part of this hint. We can then, on the operation object, do something to pass in the callbacks. There are many, many ways we could do this. I'm going to add a setCallbacks method and in here I will pass the callbacks. So result again will do the same thing, just Done and in the case of an error, we'll call Done and pass the error along and then let's go ahead and break this up like we did in the last situation just to make this a little more readable. Okay, so this is how we would use this and I would expect that this test case passes successfully, also assuming we ask Mocha for that Done function. Okay, now go ahead and make this work and then join me in the next video where I'll go over how I approached this problem.

## Solution - Pass the Callback Later

There are many ways that you could have solved this particular challenge. I'll go through my solution and once again I'll encourage you to copy down what I have here, but there's nothing wrong with what you came up with, especially if you got something working that's pretty awesome. This was definitely a mind bender and I did that intentionally. Now hopefully you had a chance to work through this because this will really help you understand how Promises work. This is one of the key pieces of a change in mentality of how we approach callbacks. Instead of passing the callbacks when we initiate an operation, for example fetching the current city, what if we could pass those callbacks later on? Perhaps at any point? What if we could pass them back an hour later, in that much actual time? In other words, what if we could somehow cash the result of an operation and keep it around indefinitely? Or what if we could just pass the callbacks right away after we initiate an operation? Or maybe a couple hundred milliseconds later when some other operation completes? If we have this separation, some really interesting things open up for us, so let's take a look at what this looks like though, just in case that still doesn't make sense to you, let's go through the implementation. So fetchCurrentCity is up here and here's our test case. You may have figured out that we're going to destroy this test case here, so let's just go ahead and get rid of that, because there's no reason for that version anymore, these tests are duplicative of what they're testing. Okay, so we have fetchCurrentCity. We call that and we want to get something back so we have to have something to return here, so we need to return something and how about we call it the same thing here that I recommended we call the object we get back, which is operation? So we need to return an operation here so we can get the operation back when we use this function. This means we'll need to define operations somewhere. How about we define that up at the top of this function and we'll just make an object that we return back. Now with this in place, we have an object we're sharing between the fetchCurrentCity function and the consumer of that function and we would like this object to have a method on it called setCallbacks, so let's copy that and let's come up here and anywhere in this function we can define this function that we'd like to have. So let's operation.setCallbacks and let's make that a function. For debugging purposes let's call that setCallbacks as well so we don't have an anonymous function and then inside of here, we can somehow get those callbacks in, but let's first put those in here onSuccess and onError. Now we can take these out of constructor. They'll no longer be passed when we initiate the request. Instead they'll be passed with this function. Now how do I go from having them here, which will be called later on, to getting them in here? If you didn't solve this, take a minute and pause the video and think about how we can finish out the solution. Well, what we can do is cache these functions on our operation object. Just like that. And once we've cached these on the object, then we can use these over here and we can just make an assumption right now that they'll be available at the time that the getCurrentCity function's callback here is executed, so by the time this executes, we assume that these callbacks have been provided. We'll deal with the reality of that not being the case later on, but for now, we can make that assumption because we know calling getCurrentCity is asynchronous, we know this will not happen until the code that calls fetchCurrentCity runs to completion so that means that we're okay here. SetCallbacks will be called for sure before this callback can ever complete and that's because this is an asynchronous callback and right now this code is executing so this code gets to run to completion. If you're confused at all about that, again that's why I have that previous course so refer back to that, refer back to the section on running to completion. So we're okay here. So all we've done is passed these callbacks later on and then we've cached them on this object and used references from that object, instead of passing them directly into this fetchCurrentCity function. In a way we've opened up the private scope of this function to be able to pass in data after the fact. That's all we've really done here, but in the process, we've completely separated initiating an operation, or in other words, firing off the operation, starting it. We've separated that from registering our callbacks. Let's go ahead and test this out to make sure it works. Let's save this file and it looks like right now we have nothing in the browser and that happens to be because I must have clicked on that one test case earlier to hone in on it. Let's go back to all our test cases and we got rid of the one we are drilled in on so be careful of that when you click the Play button to look at one specific test. Okay, so we have our fetchCurrentCity passed as a callbacks later on and that looks like it's working. To test this out, let's come up and mess up the getCurrentCity function and let's have it return an error. Save that and our test now fails. You can see on the right-hand side we get the error that we just returned back from the getCurrentCity function, so it looks indeed we are able to pass our callbacks later on.

## Question - Why Might This Separation Be Helpful?

I've got a question for you. We've got this separation now. Why might this be beneficial? Take a minute, pause the video, and think about this. What does this open up for us? And if you're stuck, think back to that slide about the problems we have with callbacks. Think back to this slide, this might help you a little bit. Think of these problems and think about how they might not be such a big deal now that we have this separation or maybe they're worse. Pause the video and think about that.

## Timing Safety without Coupling and Nesting

Do you remember that MySQL example earlier in the course where we had all that database initialization that was pretty nasty to look at and do you remember how I said that we had other code that was dependent upon it, but that code was just assuming that the DBinit would complete before anybody would send in a request? Yes, so what if we thought about that situation, but with this design instead? What if we had initiateDB? What if we had an initiateDB function just like our fetchCurrentCity function and we have the result here then in this operation variable, and what if we called this initDB? That will be our initDB operation and then later on we could have separate callbacks registered. For example, and we don't have this working yet, but you could imagine we could make it so we could register multiple callbacks so we could have another callback here initDB.setCallbacks and we could pass additional callbacks. Let's have a function here taking in the result, which let's just say the result is the database object itself and it's an object we can connect to the database with. So on success then we can do things with the database like db.InsertUser and maybe on this first one, we have a function and this one is db.InsertPayment. Can you think of where I'm going with this? Each of the controller actions in our website, so when a request comes in, each of the route handlers we have could rely upon this initDb object that represents the operation of initializing the database and once that's done then, we could go about doing our work like inserting a payment, which might be one route in our application and another route in our application might insert a user and of course we could use this in other places. Every time we want to do something with the database, we first could call initDb.setCallbacks. Now that's not exactly elegant at this point, but it will get better later on in this course, but for now at least we'd have a convenient way to refer to a previous operation and easily depend upon it without needing to register those dependencies when we initiate the operation. It would be really ugly if we had to have all of these functions ready to go when we called initDb, but we could capture that object and cache it for the entire lifespan of our application and use it every time we need to make an insertion into the database and then we'd never worry about timing because we know this code will never be called before the database is initialized. This ability to separate initiating an operation from registering callbacks helps us deal with these seams that callbacks introduce that tend to rip across our program. We no longer have to just depend on luck and assume the timing will work out to be able to write code that isn't insane to deal with and isn't nested in 800 layers of callbacks. We can now have one piece of our application initiate an operation like bootstrapping a database, and then we can have another part of our application react to the database being bootstrapped, for example, inserting a record in the database and that second piece of code doesn't have to know anything about the first piece of code and the first piece of code doesn't need to know anything about the second piece of code. Those two parts of our application can be completely independent, other than passing that operation object between the two. By the way, before we wrap up this video, I went ahead and took that example that I typed up because it's not realistic and just put it inside of a comment here and reverted back to what we had before so we can continue evolving our implementation of fetchCurrentCity here.

## Challenge - Register Multiple Callbacks

An operation like fetchCurrentCity is something that could take a while, especially if it were an operation being performed on a mobile device that needed to reach out to a GPS unit, find coordinates, and then turn those coordinates into the name of a city, so it stands to reason that this fetchCurrentCity function is something that we might like to cache the result of and then be able to use that cached result throughout multiple parts of our application. Now we could go ahead and nest everything inside of the callback once the fetchCurrentCity operation completes, but we've seen how that can turn into a rather unruly situation and that probably won't work if our application is modularized. Say, for example, we have a UI that's based on widgets that stand alone and multiple widgets need access to the current city; there might be a weather widget and there might be a forecast widget. Each of these need the current city to be able to get their respective information to display. If we tried to couple these together, that could lead to a mess inside of our application. It would be nice if we could call this fetchCurrentCity once and then be able to hook up multiple callbacks to the operation, callbacks that each operate on their own part of the app. So in this next challenge, I'm going to ask you to do that; modify the setCallbacks function to be able to be called multiple times and keep track of all the callbacks that are passed and then make sure you execute all those callbacks once the operation completes. If you'd like a hint, stick around. So the first thing let's do, let's collapse some of these tests here and I'm going to add this new test above the existing test so we keep it close to the fetchCurrentCity function. I'm just going to make a copy of the existing test and then let's go ahead and change the name here and we'll call this a test where we're going to pass multiple callbacks and we expect all of them to be called, and then let's do this. We know we shouldn't be getting an error in this test case, so let's only pass the success callback. That allows us to collapse this down and make this a little more concise and let's just duplicate this line of code. So this is what the test will look like to validate that our code is operational, at least as far as registering multiple success callbacks. Now we're going to have one slight issue with this test case right now. We can't call Done twice and expect things to succeed and there's no way to configure Mocha to allow Done to be called two times instead of one. We could try and couple these tests together, but I don't want to do that. In other words, we could try and nest validations, but that actually changes the intent of what we're testing here and I think it pollutes what we're testing, so instead of what I've done, I've added a few files to this project. I've added a multiDone file and a multiDone.tests.js file and you can get these files out of my GitHub repo, the same one you cloned from, but change to the branch step by step and look for those two files. Go ahead and pull those down and add them to your project. Actually, you really only need the multiDone file. The multiDone test demonstrates how to use this, so let's take a look at the test real quick and I'll show you how to use this. You can look at the implementation of this if you want to later on; that's not important. What this allows us to do though, it allows us to be able to call Done multiple times and I do that by wrapping our calls to Done with a multiDone, so just a clever way to say, hey, we could have multiple calls to Done and the wrapper in this case I'm setting up to basically call Done after we call the multiDone twice. So you could specify the number of calls you want. I have this convenient method called afterTwoCalls because we'll be using that a lot. So let's copy this and bring this over. Then instead of calling Done, we call multiDone instead. So what will happen here, we pass in Done. Done is behind the scenes with multiDone. Once we call multiDone twice then Done will be called. If we don't call it twice though, our test will time out. Let me hide the project view and let's split the screen here and then let me open up the test in the browser. Okay, so our tests reload in the browser and that's happening because we need to add the multiDone file into our test runner, so this callDone is not defined and that's what's tripping up an error here, so let's fix that first. Go the index.html page. We can duplicate the script block here and then come over and reference the multiDone file. If we save everything, our test should rerun and now we only have one test failing. Our fetchCurrentCity should still work and it does and you can see that the passing of multiple callbacks is timing out and that's to be expected because, well, we haven't implemented multiple callbacks yet, so there you go--there's your hint. Go ahead and hop to it and see what you come up with.

## Solution

Okay, so here is how I thought of this particular challenge. So if we're going to call this setCallbacks function multiple times, instead of just tracking one callback and overriding it then, let's come up to when we define our operation and let's create a couple arrays to hold as many of these as we want and as I was thinking about the name of this, I kind of like the idea of calling this success reactions, the reactions to a successful operation. This will hold all the things we'll want to fire off on success. So this will be an array and then we'll also have error reactions. So we'll have two separate arrays that we can push elements into and that we can do instead is use these arrays. So we have our successReactions array. It's initialized so we can just push a new element into it for this new success callback. Likewise we can take our errorReactions, even though we are not testing this at this point and we can push into our errorReactions the error handlers as well. This idea of a reaction is kind of neat. You can think of it as a piece of code that would like to react to the result of this operation. Of course, we'll need to come up here then inside of our callback. Once the operation completes, we'll need to call all of our reactions, not just one. So in here we'll do operation.errorReactions and for now we can use forEach on this array and then forEach reaction and I'm using an arrow function here, r is standing for reaction, we'll call that reaction and pass the error. Same thing on success, forEach reaction we'll call the reaction and pass the result. Notice our tests rerun in the right-hand side and our test is now passing, so we have successfully implemented the ability to have multiple reactions. That means we could have multiple parts of our application registering callbacks to the same operation, only call that expensive operation one time and reuse the result in multiple places. In a way this is like caching the result. If you want an additional challenge, go take a look at implementing this same functionality with callbacks, so go over to the callbacks challenges, look through the examples, copy one of those, and set it up so that you can register multiple callbacks. You'll find that this is a lot of work, especially if you had to do this by hand every time you wanted to register multiple callbacks, and you have to register those callbacks up front. We can register these callbacks here after the fact, which means we can split up our application and we don't have to couple things together.

## Splitting Registration of Success and Error Callbacks

Another benefit of this separation is that we can split apart code that handles the success of an operation from code that handles the error. Remember we have boilerplate error handling that was all in line with the nested callbacks and that turns into a mess? Well, simply by passing the success handler separate from the error handler, now that we can call setCallbacks multiple times, means that we could break apart the code that registers the success handlers from the code that registers error handlers, which might make it a bit easier to tame our error handling code. Let's take a look at what this looks like. So let's duplicate the test we have here of passing multiple callbacks. So it's possible on our second call here, we might not pass a success handler and instead only pass an error handler and then we can do whatever we'd like in the case of an error. So you could imagine, we could split apart these parts of our application and if we have several operations that would have errors, we can go ahead and put all the error handling code down here and all the success handling code up here. It starts to give us some separation. It's not perfect yet, but it starts to get us some separation. Now as I'm looking at this, I don't like that I have to pass null just to pass an error handler. What if we also had an extra convenience method up here called operation.onFailure and this is a function as well. This just takes an error handler only and then we can call behind the scenes here operation.setCallbacks and pass on that error handler, much like we're doing down here in the consumer code. Now we've made this a responsibility of this function so consumers can simply call this instead. Much more concise and readable and I think something that people would appreciate if this were a library we were authoring. Now to make this onFailure symmetric with the setCallbacks function we have right now, I think we could go ahead and rename setCallbacks at this point. I chose that name because that's something you could relate to as the operation we're performing, but I think you know that that's what's happening know so I could say onSuccess, but that's not entirely true because we can also pass an error handler to this one as well, so how about we just call this on completion? You'll notice WebStorm pops up a little refactoring preview because there are a lot of usages of this and it's just asking us if we would like to go ahead and rename this in all these places. Go ahead and click Do Refactor. I've already reviewed the snippets here to make sure that these are okay. By the way, here is a diff. Sometimes I'll pull up a diff and get what we've changed thus far and this is just with this video, so you can see what was renamed here if you're uncertain about what was renamed. You can scroll here and see all the renamings of onCompletion. So now we have two methods that work together that give us the ability to register concisely a success handler and then separately an error handler, though we can also in the case of the completion, register the error handler as well, but now we can decouple these pieces and it looks really nice. Okay, with this syntax in place, let's get test to something meaningful and for this first test, we'll only register an error handler and if that happens, then we should ignore success and if you look in the browser right now, we have a failure, r is not a function, and that's because we're passing null in when we call onFailure here and registering a success reaction that's null and of course, we can't invoke that as a function, so first we'll fix this use case. Now in this case we don't need a multiDone. We can just call Done in here, which will complete this test and let's delete this and put this after so it gets registered second; that will be a convenient way right now to make sure our test actually runs through to completion and then on an error, which should never be called, even in this case, we would actually fail the test. So we'll save that again and the test works, so this is one of our test scenarios and we'll also need a second scenario. In this case, the opposite, when we only register a success handler we'll ignore the error handler. Okay, so we have two tests here. Now in this first test we're calling fetchCurrentCity, which is an operation that we'll complete successfully so we'll never have a failure; however, in the second test we need a failure and in that case, that will be our indication that the test succeeds, so let's put that last. If for some reason our success handler fires off, that's actually an indication of a problem so we'll say that this shouldn't succeed because we need to call an operation that fails here so that we can make sure that when we don't pass the error handler and we only pass a success handler like we're doing here in this first line, we'll make sure that that error handler of null is not injected. Of course, to get this test to even begin to work we'll need to have some operation that can fail. We'll get to that in a minute. First, let's go ahead and get this test working. So let's rerun our tests and we have both of them failing. If we come up here and in the case of only registering an error handler, our success handler will be null. So one thing we could do is we could just coalesce with the or operator and if success is not truthy\_\_\_\_\_, in other words it's not defined, it's null in this case, then we could just inject a noop, which for a noop we could just have a function that does nothing and if we want this to be more meaningful we could even extract this out and call this noop. Okay and you can see on the right-hand side our test is now passing. In the next video, we'll get this second test wired up because we need to convert one more of our functions to this fetch operation style to be able to have something that can error out for us.

## Convert getWeather to fetchWeather

I took the liberty here to rename these tests and to add a comment to each. I like the test name noop if no success handler passed and noop if no error handler passed. This gets the intent across I think a little more clearly and I also added a comment to indicate why we have this call to onFailure and onCompletion at the end. These are our triggers to make sure that the noop is registered and if we wanted to be more complete with our comments we could say, noop should register for error handler and likewise for the success handler. Okay, I think that test naming makes a little more sense. Now we'd like to get the second test to work and that will necessitate having some asynchronous function that can fail. Our fetchCurrentCity function doesn't support failing, but if we come way to the top and we look at the other two functions we have, these can fail if we don't pass a city so how about we convert one of these? How about we convert the getWeather function and make a fetchWeather function? That will take a city, but in this case there will be no callback because we'll make this work with operations as well. Now we could come up here and cop this giant text block here of everything inside the fetchCurrentCity function and paste that here and then all we'd need to change right here is this getWeather function. Call that instead of calling fetchCurrentCity. We need to pass a city also and then we could come down here to the tests that we're working on and call fetchWeather and this time don't pass a city that will make this test fail. We can save this then and there we go. We've got the failure we want. R is not a function and that's because we've registered a success handler without an error handler and we have an error so that null error handler is trying to be invoked, so we can come up here inside of the fetchWeather function and we can fix this and pass in our noop instead and save this and our test is now passing and of course this works, but we've got a little bit of a problem here. I'm sure you've noticed this, we've only fixed this in one place. We really need to fix it up here too so we'd need a test for this, but we can't test this because our fetchCurrentCity function can never fail. We also have a lot of duplication here. What we should do is extract out the duplication we have between these two functions because we're forming something here, which is really what we're testing. We're not really testing the current city function or the weather function; we're really testing this operation construct that we're building up here, this operation abstraction is what we're testing.

## Extracting the Operation Type

Let's factor out this duplication and let's create an operation abstraction because that's really what we've been building up inside of this fetchCurrentCity function. So let's yank out this first part where we're creating this operation. Let's create a function down below called operation, paste that in, and return that operation as well. Now what we can do, we can come up here at the top of fetchCurrentCity and our operation can equal a new operation and we can do the same thing inside of the fetchWeather function. We can yank this piece of code here and paste this over the top. Okay, so we've changed nothing and our test should still pass. Let's move on. So we have these two methods on the operation type, let's grab those and move those out and paste those in here and then we should be able to just get rid of these on the fetchWeather function. Scroll up here and let's go ahead and save to run our tests again. Ah! And we get a failure and that's because we took the version of onCompletion from our fetchCurrentCity function and that didn't have this new noop so let's put that in, a noop in the case of no error handler. Save again. Now that's the reason we have tests for this course, because we're going to add more and more methods here to this operation type and I just want you to focus on what these are doing. I don't want you to have to worry about if you've broken your code; I want you to have tests to fall back to. Okay, let's finish this up. So I'm looking up here at fetchCurrentCity. Things look pretty good, but I feel like this part could be removed as well. This is more an implementation detail. It'd be nice to yank this out and instead have something like operation and in the case of an error we can say fail, so we'd like to fail the operation in case of an error and fail it with this particular error and then what does that entail? Well, down in the operation we can set that up. So operation.fail = and that's a function and it will take the error and it will just run through our error reactions and now likewise in the case of success, let's have that delegate to a function that's hidden away inside of our operation abstraction. So how about operation. and then in this case let's say succeed and we'll pass along the result. So we're succeeding the operation saying that it successfully completed and again we can come down here then, operation.succeed, make a function, pass in our result, and just call that code that we snipped out above. Save everything and our tests are passing. Let's come down here into the fetchWeather function and we'll want to make sure we call the same operation so let's come up and grab those, the same body here and paste that in. Now if we wanted to, we could be a little more zealous here. This is duplication so we could actually yank out and let's just say we have something like operation.nodeCallback, and then we could come in here and have operation.nodeCallback and make this function reusable. It's almost like pre-building an adaptor for any of our node functions and then we could use this above here and our fetchCurrentCity as well. There we go. If we save our tests, everything is still passing. Now that we have this abstraction set up, here's a quick challenge. Go ahead and pause the video and use this operation abstraction to make an adaptor for the forecast. So we have getForecast right now that takes a city and a callback. Adapt this like we've done with the other two operations and make a fetchForecast function. Okay, I hope you did that because I hope you can see how easy this is. Let's just copy the fetchWeather function, paste it and the only thing we have to change here is call getForecast. Then we can have this be our fetchForecast function. So within three lines of code we've adapted a function that takes callbacks into this function that works with operations. The only other thing we might do here is grab these adaptor functions, yank these out, and let's paste these up here. That way we can collapse these down because once we've adapted these functions, we won't need to work on these much; we can focus more on this operation type so let's keep that close to our test cases. Let's run our tests. Everything still passes so we're good. Let's do a quick recap and then we can move on to the next module and build out more of this operation abstraction.

## Operation Type Benefits

In this module we saw first that if we split apart registering success and error callbacks and make those two separate callbacks, we can reduce a lot of the boilerplate in those if checks that are common in Node.js applications. After we did that then, we asked the question and I threw the challenge to you to figure out if you could pass the callback later on. This may not have seemed like a big deal, but it unlocked some really interesting scenarios. I created this quick summary of what we've accomplished here and in this summary I've fetched the current city and then on completion I called this callback. I fetched the weather then and on completion here, I called this callback. I did this because I wanted to show you that syntactically this almost looks identical to what we had before with callbacks, the key difference being that we pass a callback later, though that's really not that different than just passing it up front, but this unlocks some really interesting use cases. For example, we could fetch the forecast and then we could register multiple callbacks. We also have the ability to delay registering the callback, which means we can decouple the part of our application that initiates an operation from the part that consumes the result of that operation. A great use case there is initializing the database when you're app boots up. We then further refined passing the callback later and set things up so we could just pass in a success function or just pass in an error function and then we'd inject a noop for any handlers we don't register, that way we can choose to only register the success handler or only register the error handler, which means we can split these two pieces apart, which means we could consolidate our error handling separate from the rest of the behavior of our application, which gets us much closer to having less insanity and duplication and error handling, and of course, out of this we breathe life into this new operation type and while we're calling this an operation, this type that we've created is what everybody else refers to as a promise. A promise is just another name for this. You could think of a promise as promising that an operation will complete sometime in the future and then a promise will also have the error if it fails or the result if it succeeds. We'll stick with our operation type for now, but eventually we'll see how these names relate to the naming conventions with Promises. By the way, another couple of names for referring to this type that we've created. Sometimes you'll hear this referred to as simply a task, sometimes you'll also hear this referred to as a future. All of these names connote the fact that something is executing and there will be some result in the future representing whether or not that execution completed successfully or if it failed. So one way to just demystify Promises is to look at tasks and futures because those are used in some other languages for the same concept, maybe you're even familiar with those; maybe in the .NET space, for example, you've worked with the task type. Well, that task type is pretty much a promise. Join me in the next module where we build out more functionality of this operation type.

# Chaining Operations (Promises)

## Get Me a Cup of Coffee

In the last module, we brought to life this new operation type. In this module we'll further flesh this out, specifically focusing on chaining operations. We still have some issues with callbacks that we haven't addressed yet, for example, nesting is still a problem we haven't addressed. We haven't yet talked about how we can easily synchronize parallel operations and perhaps you've noticed this, but the implementation of the operation type right now has a big glaring error in it. We can register callbacks right now synchronously, but we can't register callbacks asynchronously. If we do, we may run into a race condition with the operation that we're registering a callback for. We'll deal with all this in this module, but first, I want to take a minute and just talk abstractly on a high level about operations and then I also want to give some analogies here because I think it's important to step back and think at a little bit higher level. We've been in code for a while. Let's make sure that we conceptually understand what's going on here and I think the best way to do that is just to start with this idea that we have an abstraction right now to represent an operation and an operation starts out in a pending state. So once we initiate an operation, it's pending or in other words, we're waiting for the operation to complete. So you could think when we initiate or fire off a function that takes a callback, initially it's in a pending state and then eventually the operation completes and it either succeeds or fails. When it succeeds, we call that succeed function and pass in the result to our operation abstraction and once that happens then we have an operation that has succeeded and we also have the result for it. We have that piece of information about our operation now and we forward that information on to the success reactions, so all the callbacks that are registered when things go okay are then fired off and we pass a result to them, so this is one possible pathway for the lifecycle, if you will, of our operation abstraction. The other path if something goes wrong, well we call that fail function and pass in the error, which means our operation transitions into a failed state and we also have the reason why in that error object. We take those then and forward those on to the error reactions instead, so anybody that said, hey, I'd like to know when something goes wrong, will be notified and they'll be given the reason for the failure. So this is the other pathway through the lifecycle of our operation abstraction. These two states succeeded and failed, represent when the operation is complete, so that's another way we can look at operations; they start out pending and eventually they complete. There was a lot of code in the last module. If you only take away one thing from that module, take away this slide right here and specifically, take away the fact that operations can only be in one of three states--pending, succeeded, or failed, and keep in mind that operations are synonymous with promises so promises can also only be in one of these three states. As we get into more complex scenarios here with chaining, I'll be referring to these three states to help you understand what's going on as we add some more complexity here to handle some more of the problems we have with callbacks. Work is a great analogy for promises and operations. Let's say you're the boss and you've asked somebody to do something for you; let's say you've asked somebody to go get you a cup of coffee because wouldn't we all love to have somebody that can get us coffee. Asking somebody to get the coffee is initiating a request. It's initiating an operation or initiating a promise. At that point, you don't have the coffee yet, so the work is in a pending state. The person you've asked to do this will head off and go seek out some means by which to get a cup of coffee and you can go about your work. You might do some paperwork, you might pay some bills, and then eventually, hopefully, the person comes back to you successfully with a cup of coffee. The cup of coffee is akin to the result in an operation and handing the coffee off to you is akin to calling the success reactions and passing off the result. So that's one possible pathway. It's likely though something else could have happened instead; something could've gone wrong and at some point the person may come back to you and tell you that something went wrong and of course, hopefully they give you a reason for this. Let's say they come back to you and tell you they went to the coffee shop, but their credit card was declined. In doing this, this is much like calling error reactions and passing along a reason. Now that you know about this you might give them your credit card and ask them to go purchase a cup of coffee with your credit card and that's like firing off a new operation, which starts the whole cycle over again. Now it's possible that there might be other people that are interested in this. There could be other people that like to react to this operation. For example, if you're not supposed to be telling people to go get you a cup of coffee, H.R. might like to know if you asked somebody to get you a cup of coffee and so the operation may fail because the person goes and tells H.R. and H.R. tells them they don't need to go get you a cup of coffee and then of course at some point that comes back to you in the form of a failure as well. Interestingly there, there's also probably a chain of operations happening for that failure to finally get back to you. Sometimes it's helpful to think of real-world analogies outside of programming and relate what we're doing to those ideas so we can think about some of the complexities involved with the code we get into in terms of more simple ideas. In fact, just right now in describing this analogy to you, I've realized that this analogy serves well to understand chaining, which was not my intent; I just wanted to give you an analogy, but wow! Think about chaining here. They didn't have money to pay for the coffee so they came back. That starts a new operation. They go off and get the coffee, hopefully they come back and that succeeds, but it's possible something could go wrong there as well. Maybe they get in an accident on the way back with the coffee. So again, there's another operational kickoff to do something about the accident and help them out. As we're going through this module and the rest, come back to this analogy if you're getting lost in the code and see if this can help you out.

## Challenge - Register a Callback Asynchronously

Sometimes when I go out to eat, if I'm in a hurry, I'll get my credit card before the waiter brings the check and that way I have it ready when they get there and I can hand them the card as they're handing me the check. Usually the check will be put on the table and the waiter will walk away as the people figure out who's going to pay for what and what payment method they're going to use--cash, credit, etc., but you know if you're in a hurry you'll need to have your card ready and you can hand that to the waiter. This represents a problem we have with our code right now. We make the assumption right now that we don't get the result before we register the handlers and that assumption has served us quite well because thus far in the course we've registered the handlers first and then the operation completed, but if things go the opposite way and the operation completes before we register the handlers, we're going to be in trouble because they won't be here to be called when the operation completes. If they're not in this array of reactions they won't be called, so if you call onCompletion after the operation succeeds or fails, those callbacks will never be called. Let's go ahead and fix this. Because in addition to being able to pass a callback later, shouldn't we be able to pass a callback any time in the future? So let's say we have a method called register success callback async. It's reasonable to assume we might have some code on our application when our app boots up to fetch the current user's location so we call fetchCurrentCity when the app starts up and we print it out for debugging purposes, but then it's also reasonable to assume that later on we might use for another purpose. So we might have something like this mock scenario where later on and to indicate that I want something to run later on I'm using setTimeout and setting a timer in the future of 1ms and then once this timer elapses, I'll go ahead and register a callback. So I'm registering this callback asynchronously here and we might do something like fetch the weather for the city. This could be another part of our app. Remember, this operation abstraction gives us the ability to cache the result of the operation or at least it will once we implement and fix this test scenario. Remember, we don't know when this could complete so in this other part of the app we should be safe and register a handler just in case it's not done yet. We don't want timing problems, but right now you can see over in the browser that this test is timing out. We never are calling Done here and that's because onCompletion is called after fetchCurrentCity completes. So Succeed here is called before we call onCompletion the second time. So what we need to do when we register a callback is just check and see if the operation is already done, because if it's already done, we don't need to push the reaction into the array, we should just call the operation right then and there. If you would like a challenge, go ahead and pause the video right now and implement this yourself. You don't have to, but this is an optional challenge if you'd like to implement this yourself first and then join me back and I'll explain how I implement this.

## Solution - Register a Callback Asynchronously

Somehow we need to track the state of the operation if we're going to be able to determine if it's already complete. So what we could do is come up here then inside of the completion methods. We could a state property and set its value, in this case to failed, and in this case we could set the value of the state to succeeded. If we do this then, down here we can write an if statement and check that state. So we can check if the state is equal to succeeded first. If so, we could go ahead and call the onSuccess callback, but then we also need to pass something to onSuccess so we'll also need to capture the result up here and likewise in the case of a failure, we'll need to capture the reason why. Now we can pass the success result here and just immediately invoke on success. Now if we go ahead and save these changes, our test passes. Of course, it's pointless to push the reactions into this array anymore; we don't need to do that. It's not going to hurt anything, but let's put those in an else block. So basically if the operation is completed already, we'll just go ahead and call the callback right away, that makes sense. Otherwise we will put it into our array to be called once the operation completes. Now to round this out we should have another test case for the error callback. Before I copy and paste this though, this is a bit verbose. Let's just go ahead and get rid of this line right here. We don't need this part of the test, nor do we really need to call fetchWeather. I was just putting that in there for illustrative purposes and also setTimeout with a value of 1, that's a bit redundant. Why don't we create a function called doLater because that's all we really intend in our test here is to do something later on. In this function we'll call setTimeout, which will then after a millisecond call the function that we provide. Now we can get rid of this part right here and change setTimeout to just doLater. Not a huge savings, but a little bit more clear in intent and a little less code to write, which definitely isn't a bad thing. We can even shorten this up a little bit here. Alright, let's paste that test in. We'll add the error version of it. In this case we need to call something that will throw an error so we'll call fetchWeather without passing a city. So we'll have operation that errors and in this case, we'll just call onFailure directly. Let's save this and you can see we get a timeout on our new test case and that makes sense. Before I go up and fix that though, let's just rename this as well to match. This is operationthatSucceeds. It's nice to convey the intent in our variable names and tests. Okay, so let's come up here and fix this last scenario. Let's copy our first case, paste it in here. We'll put and else/if on that and this time we want to check and see if we fail; if so, we'll call onError instead. This time we'll pass the error value and if we save this, our tests are now passing and there we have it. We now have code such that we could hand the credit card to the waiter before they ever even bring us the check, which is sometimes what I do when I really want to get out of a restaurant.

## Challenge - Parallelism

In addition to passing callbacks asynchronously, we've got something else with the change we just made, but to figure this out, I'm going to give you another challenge. In this challenge, I want you to go back to the problem we have with callbacks in performing callbacks in parallel. Here is the result synchronization example to try and get these callbacks to work in parallel. You can refer back to this. Remember, this is in the callback examples file. So we're firing off two callbacks and then we're synchronizing the result so that we can have both results available before say, we set up the UI in our application. This was difficult with callbacks. What I want you to do is set this up with the operation abstraction we have now with these new fetch methods. So see how you can implement parallelism in terms of the result synchronization with the fetch methods we have. If you want a hint, stick around. Alright, so here's your hint. Here's the new test name I'm using for this. I'm calling this test name lexical parallelism. I made this up myself, but if you look up lexical scope, this might give you the hint you need. One more hint, one more hint. If you want it, stick with me, otherwise pause the video. Keep in mind, we went to a lot of trouble to be able to separate initiating a request from registering the callbacks. That's a really big hint.

## Solution - Parallelism

As I mentioned, we got something for free. That was actually another hint so we don't need to make any changes to our operation type, we already can easily synchronize the result of parallel operations. So let's do this. We have two functions right now that we can reasonably run in parallel; it makes sense, but in order to do that we need a city. Those two functions are fetchForecast and fetchWeather. It makes sense in our app. We might want to load those in parallel, so we can pick one of them to start with, fetchWeather; we can pass our city and then we can pick the other one, fetchForecast and pass the city. Now at this point we've initiated two requests, but we haven't done anything to handle the results yet. What I'm going to do to help you reason through this is create a weather op variable that represents the weather operation and then I'm going to create a forecast op variable. So these two variables represent the two operations that are now running; we've initiated both of these at this point in our code; now we just need to bring the results back together. So I can pick either of these, it doesn't really matter, they're both processing right now. We have two web requests going on right now, so I'll just start with the weather op and I'll say hey, on completion, run this code. That means I'll have the weather here and then inside of here I can say forecast op on completion, so I can wait for the second and there we go. I've synchronized waiting for both results and I'm not going to do anything until I have both results. It doesn't matter the order I specify here with the callbacks because we already have both requests running. I could flip these around and it wouldn't matter and this is what we got for free by allowing ourselves to register a callback asynchronously and then we can round this out by printing out the current weather information. So we could say, hey, it is currently, grab the temperature in the city, with a five-day forecast of, and then the five-day forecast. It reminds me of listening to the weather when I was a kid. Of course, we can call Done to know that we're done and if I go ahead and save this code now, our tests are all passing, which is not surprising. We didn't need to make any changes. If we drill in on the lexical parallelism case and we pop open the console, we can see the string here that we just added and we can rerun this just to see that that's actually happening. We couldn't possibly build up the string like this unless we have both results, which means we have to have them at the same time, which means we've synchronized the results while still running the operations in parallel. If you want, you can go all the way up to the top here, come into the getWeather function, and we could log out that we're getting the weather, copy that, come into getForecast, log out, getting the forecast, go back the browser and refresh and you can see getting weather, getting forecast. Of course, maybe you don't believe that that actually is happening in parallel so let's do this. So I'll put a line in before we register any callbacks and indeed we are firing off both requests before we attach any completion handlers. Now the reason I named this test lexical parallelism is that we can read the code quickly and understand the scope easily based on literally the words that are right in front of us. I think this method of registering two completion handlers after we fire off the ops is also plain as day just based on the words we have in here, just based on the code itself you can understand that we are synchronizing the results right here. There's just no way we can build up this string if we don't have both results right here and it didn't require a lot of hoops to jump through like in the case of a callback. Remember the callbacks? Only right here do we have everything together and we have to create this global state to do it and we have to stitch together the callbacks. This is not pretty. Now we have this nice, easily understandable way of chaining together our callbacks. We still have some perhaps undesirable nesting in here, but that's something we'll address throughout the rest of this course.

## Challenge - Unnesting

It's time for another challenge so that we can do something about the nesting we've been putting off for quite a while now. I've got an example set up here where we fetch the current city once we get it and then we fetch the weather, a typical scenario where we need to wait for one async op to complete getting the city and then we can execute the next async op and of course we then have nesting as a result and we have two levels of nesting here, but just like with fetching the current city, it's possible when we fetch the weather that we'd like to use that somewhere else in the app as well. We'd like to cache that. So you could imagine we have some other piece of code somewhere else where we'd like to use the weather again, so those are two reasons alone. Number one, maybe get rid of some of this nesting; number two, reuse this weather information somewhere else in our app. That's two reasons why we might want to do something about this nesting because I'll ask you right now. How at this point can we go about accessing the result of this operation down here? So that's a challenge for you. Go ahead and pause the video and see if you can figure out how we can access the result of this operation down here. How can we hook up another completion handler?

## Solution - Unnesting

Okay, so let's talk about this together here. Maybe you came up with the solution and maybe not. As I'm looking at this though, one thing I could do, I could go ahead and introduce a variable here, extract a variable, and we'll call this weatherOp and then I could try to use this variable down here, but I can't access that, so even if I extract a variable, I can't use that down here to set up a completion handler because what I'd like to do maybe is use the weather in another part of the app and to simulate that we'll finish off this test by calling Done if we can pull this off. So this is what we need to get working for this test to be passing so that we call Done here to indicate that we can access the weather somewhere else in our app, not just inside of this nested callback. So how do we do this? Maybe you got this far, maybe you didn't get this far. Pause the video again and think about this and see if you can come up with how we can actually pull this off. Oh and by the way, I don't want you modifying our operation abstraction to make this possible. Think of a way of pulling this off without modifying the operation abstraction. Well, one thing we could do is try and bring the variable to this outer scope so we could have a weather op here and then we could assign the variable instead of declaring it inside the callback, but we could have timing issues here. This is exactly the case we saw with that MySQL example way at the beginning of the course. We would have to assume the timing just happens to work out and we don't want to do that in our applications unless we want them randomly failing so this won't work. What we really need is something to represent when this weather operation is complete, we need access to some operation that can tell us that it's ready, but the problem is that we need that reference to the operation before we get into the callback. We need it right here, but I'll ask you, what if we just did this? What if we make a new operation object that represents when this weather operation is complete, so this is a brand new operation. There's nothing stopping us from creating these, in fact, this is exactly what we're doing up in those fetch functions. Up in those fetch functions, all we're doing is creating a new operation and we're returning it and by the way, this operation.node callback, remember we extracted this code for convenience, all we're doing there is delegate to fail or succeed. We can do exactly the same thing down here. So we create this weatherOp right here and then inside of our weatherOp's completion, we can complete this other weatherOp as well. Now that's getting confusing because I left some variable names here. I'm going to get rid of this. There we go. That should make more sense. So we have a weatherOp operation that represents this weather op and we create that before we get into the callbacks so we have access to it later on, so inside the callback, all we need to do then to make this work is call that succeed function and forward on the result. We just need to forward the operation result and there we go. We've basically cloned the weather operation here. We've made another copy of it that will resolve with the same value. It will complete with the same value, this weather response and so now we have it somewhere else in our application. I pulled up the tests here. Let's go ahead and comment this line out right here though. Let's run the tests real quick. It'd be nice to see this fail. Indeed it times out and all we have to do to make this pass is forward on that result. Save that and our tests are now working. So just like that, we can create an operation that represents the result of some future operation we have yet to even initiate.

## Forwarding Completion

Now as I'm looking at this code here, if we don't need this console log, in other words if we don't need to perform anything right here, we could just get rid of that and then really what we have here, we're just passing this function right here to our onCompletion handler. So we could write this a bit more concise like this a bit more concise like this and we could say weatherOp.succeed; that's exactly what we had before. I can save the tests and rerun them and they still work and for completeness we should also pass weatherOp.fail in the case of a failure. So in doing this, we are now forwarding the completion of this operation onto this operation and this is just a very concise way to write this and we're forwarding both success and failures because we're registering this operation's succeed and fail methods to be wired up to this operation's completion callback reaction so they'll wire together and this operation is now locked into the result of this operation. Of course, you could imagine needing to reuse this code so why not roll this up into a part of our operation abstraction? This code right here could be something that we could write a little bit differently. Let's copy this and let's just add a method called forwardCompletion. We pass the operation we want to forward it on to and then we come up here, paste in the code we had before, and of course we get the opIn and it's not a weather op, it could be any type of operation, and we'll just call operation.onCompletion here. Register both of the forwarding functions, thus locking in this second operation to the first one. We pass success and we pass failures just like that. Stick with me. We have a few more changes to make to this.

## Waiting for the Completion of the Completion

This is a pretty elegant solution at this point. What we've set up is an operation here that represents the completion of this callback, so it's the completion of the completion, and specifically we're interested in the completion of the completion of the completion. We're interested in this fetchWeather function's result. We could use this as our solution, but this would get pretty repetitive whenever we needed to rely on something like this. Wouldn't it make more sense if we didn't have to new up this operation and instead on completion could just proactively make one? I mean, doesn't it make sense that we might want to wait for the completion of the completion, especially if this callback involves async work as well, which we have in this case. So yes, let's just start out with having onCompletion new up the operation for us. So that means we can yank this out and we can just set the weatherOp to be equal to the result of calling onCompletion. Of course, if we save our test it won't work right now; we'll have to come up and fix this. So what we'd like to do right here is just new up and return an operation object and for consistency we'll also add a return keyword here in the onFailure since we're forwarding the call to onCompletion. So now we're just newing up an operation so we don't have to do that down here. We don't have to use this if we don't want to. All the rest of our tests will keep working just fine, but hey, if we want it, we've now got it.

## Operation Does the Forwarding

Now that we've set up gives us a new operation object to work with, but it's kind of weird. That operation object on its own won't do anything; we have to do something with it in the case of forwarding the completion here. It might be nice if somehow this were also done for us. Why do we have to call forwardCompletion? Isn't that something we could ask onCompletion to do for us as well, in a way? What if we could return something from this callback function that represents the result of this operation and then that is what we use to forward completion. In other words, what if we did this? So we're basically saying hey, we're going to fetch the current city and then on completion we're going to fetch the weather which itself is asynchronous. We're returning this back so what we're saying is we'd like to automatically forward the result onto this operation. In other words, we're saying that the completion of the completion is this right here, which is an operation so we want to wait for that to complete then and automatically forward the result to this operation that was created for us. I mean, that kind of makes sense if onCompletion is going to return an operation that represents the result of its completion, shouldn't it also go ahead and actually trigger that operation for us when things are done? It makes sense to me. So let's come up to the top here. Now bear with me, I'm going to give you another recap of this once we get done because I know this is going to be a bit mind bending; it's still mind bending for me. That's okay, but what we're doing is we're just wanting our operation abstraction to take care of forwarding the completion if the callback returns an asynchronous operation. Again, we're doing all this just so our operation framework does the heavy lifting for us so that we don't have to do this as consumers, we can just reap the benefits as consumers. So what we really need to do in the onCompletion method is somehow access the result of calling our onSuccess method, but the kicker is we have to do it no matter if we're doing it right now or in this case later on. So what I'm going to do is just rewrite that function, the onSuccess function, let's just rewrite it so that no matter when it gets called, it incorporates our new behavior in it. So let's make a function here called success handler and we'll take advantage of the closure here to access this private state. First thing first, we need to call the success function that was passed to us and we'll want to pass the operations result to that. That means right here now, we can just call the success handler instead and then down here we can just reference the success handler. It's also worth mentioning now this noop check doesn't make any sense and that's something we'd want to move up into our success handler and for now we could do that with a quick check to see if the method is even defined. We don't have to use our noop function. So we've preserved the existing behavior by just creating this new function that's nothing but a proxy to call the original function, not too fancy, but what this has opened up for us is the opportunity now to get access to the result of calling that callback. Let's come down here and see what I'm talking about. We now have access to the result of this function, which in this case we're returning an operation that represents the weather operation, not to be confused with this weatherOp variable. Two separate operation objects. So that means we could grab the result here. We could say the callback result and now we can take a look at this callback result and if this callback result has an onCompletion method, then we know it's also an operation, which means we can go ahead and do the forwarding that we set out to do a little bit ago. How will this forwarding work? Let's be careful here. So we were forwarding on to the operation that we had created that weatherOp. That's this operation down here, so that means we need access to this operation object long before the end of this onCompletion function. So what I'll do is I'll create a variable called completionOp to represent the operation that represents the completion of this promise and then we'll go ahead and create this at the top here. Now we have access to this here to go ahead and forward it on here. Now keep in mind, be careful about what we're forwarding from. We are forwarding from the callback result, which in this case is fetching the weather. So if we scroll down here, the callback result in the operation that represents actually fetching the weather. That now comes through the callback result. We forward the completion of that onto our completion op, which we have right here, which is the operation we created before we even fired off the weather request. It's pretty neat. A bit mind bending, I know, because we're manipulating time here and creating these operations that represent things that are happening at different times. I know that's a little bit confusing, but bear with me and if you go ahead and rerun the tests, it looks like most are passing. We've got some issue with the lexical parallelism test. If we come over and take a look at this, we can't read property on completion of undefined. So, yes, okay, what's going on here, we are calling on the callback result in that case and trying to access a property on an undefined callback result, so we can just add another check in here, callbackResult and make sure it's actually truthy or defined before we actually try to access the onCompletion property. Save the test again and now everything is running.

## Where Did the Nesting Go?

There's something that might not be so obvious right now but that is really, really neat. Let me make a few changes to this code we have here in the test case that involves what we just set up with forwarding. Let's get rid of this part right here. We can do that because we are just capturing a local variable. Let's get rid of that as well. What do you think of this? And if you want we can actually clean this up a little bit more. Of course, cleaning up is subjective, but we could use an arrow function instead to call fetchWeather. Perfectly valid to write this. Do you notice that the nesting, the two levels of nesting are gone? We still have one level of nesting. Don't be confused by these arrow functions; that's still one level of nesting, but we don't have two levels of nesting anymore. Let's go look at the callback examples. This very first example, nesting serial async dependencies, getting the current city and then getting the weather, there's two levels of nesting. This is one level, this is two levels. We don't have that anymore with Promises. Simply by allowing ourselves to implement this forwarding functionality and creating operations that represent completion of completion and completion of completion and completion, we now have this really neat way to chain together promises in such a way that we've removed quite a bit of the nesting, all but one layer at this point.

## And Then

While we're at it reflecting on things, I think now might be a good time to consider the naming of some of our operation methods, specifically this onCompletion method. We started out with setCallbacks a long time ago and then we renamed it to onCompletion to indicate that this is what will be called when the operation completes. Makes sense, but now I think it's time to maybe consider renaming again now we have this terse syntax for chaining together asynchronous operations, we have fetchCurrentCity, onCompletion, fetchWeather, onCompletion, we're done or print out the weather or whatever. That's not bad, but what if we had the word then instead? How does this read now? FetchCurrentCity, then fetch the weather, then print out the weather and be done? I like it. Let's go ahead and add that in. Now we could go ahead and rename the onCompletion method. Let's just leave that for now and let's add an alias and we'll need to put that in down here so we can actually reference that method. Save our code. Everything still runs, so that's good and I'm noticing if you really want to make this terse and concise, look what we can do now. We don't need a function to invoke a function, we can just pass the fetchWeather function directly. Save our code. Everything still works. So fetch the current city, then fetch the weather, and you could even imagine we would have a method called printTheWeather, and so we could create that method, put our code in here to call Done instead and we could actually log this out. Though the implantation of that is not too interesting, we could collapse that down, but look how nice this reads now. The nesting is still there, but we can do some really neat tricks with that nesting by virtue of passing concise function expressions, in this case just functions we have. We need pass nothing else. This reads really nicely. So we've taken care of the nesting problem and in chaining together these operations or promises, we have very expressive flow control as a result.

## Chaining Recap

We've covered a lot of ground in this module and we have some really cool new features as a part of our operation abstraction. Let's take a minute and let's review these though just to give you another way to look at these new features we've added and to help you out just in case some of this is overwhelming, which if it is overwhelming, that's not a big deal, we've actually done some rather advanced things with this abstraction at this point. It's going to take some time to become familiar with these ideas and really internalize them, so let me help you out by walking through and explaining things again from a different angle. So I've brought up four different code samples here. In the upper left, I have an example of nesting callbacks and using the operation interface to do this, but this is still our classical nesting of callbacks when we have asynchronous dependencies that rely on each other. Now at the beginning of this module I asked you the question, what if we have some other piece of code that wants to use the weather for our current location? How do we give it access to the results of this fetchWeather operation? Of course, we realize that we can't just create a global variable because we could have timing issues, so what we did to work around this over in the upper right quadrant we created a proxy operation. So before we even call for the current city, let alone call to get the weather of the current city, we create a proxy operation that represents the operation we will fire off when we fetch the weather and then once we get the weather back, we go ahead and we pass that on, so we pass the weather result on to that initial weatherOp that we created and we call that result forwarding, so we forwarded the result of fetching the weather onto this weatherOp that we created, so we forwarded that to our proxy operation, which means we can now use that safely throughout the rest of our application, for example down here we can register an onCompletion handler and there won't be any timing problems because this code down in the bottom will just wait until the operation completes either successfully or if fails and if it completes successfully then its work can be done and in this case that means printing out the weather and then calling Done so that we finish our test successfully. At times you're going to be confused about remembering the rules for how promises work or operations in our case and that's okay. One of the things you can come back to is this example when you're confused. Come back to this example of proxying operations because I think this is simple to understand in this format. This is pretty easy to reason about because it's explicit here right in the code, but the kicker is, this is not code we want to have to write every day. Many times we'll need to rely upon future operations or promises that we have not yet even created that are nested inside of calls to other asynchronous operations and we won't want to have to write out all of this forwarding code and that's why in this module we rewrote this upper right-hand quadrant into what we have in the lower right quadrant, so we added some functionality to our operation abstraction to be able to write things this way and have the same result and of course, the key here, now when we call then we get a promise back, immediately get that promise back and that's that proxy promise just like the manual proxy promise we created up here called weatherOp. Now keep in mind we also have the promise that we created when we call fetchCurrentCity and we also have a promise that we eventually create when we call fetchWeather and there's one more promise in this example and we call this then here as well. We don't use that promise, but it's there also, so these black squares represent the promises in this example. We call fetchCurrentCity and get a promise. When we call then on fetching current city, we get a promise that represents the completion of this success handler that we're registering here that will then eventually call and get the weather and of course, a part of that fetchWeather returns a promise as well and that's returned back so something interesting happens as a result. When we return a promise from one of our success handlers, that becomes then eventually forwarded on to the proxy promise that was created. Those two promises become locked together so the result of fetchWeather will also be the result of the proxy promise or proxy operation. By the way, I will use operation and promise interchangeably going forward. Now that you can see what promises are involved or what operations are involved, let's erase this and start over and I want to walk through the timing of this and talk about how these promises resolve. Resolve being another word for complete. So we've been using complete for our operation type. Resolve is another word to refer to completing an operation or completing a promise. So here's how this code executes right now. First fetchCurrentCity is called and that returns a promise and that promise is pending at this point in time so an empty box here will represent a pending promise. On that promise we call then, which generates another proxy promise, again empty because it's pending at this point. Now keep in mind, fetching the current city is an async op so the promise there cannot complete until all the current code has run to completion and we're still in a block of code here; we're still in a little program. So this program has to complete and the next operation in this program is to call then again, which will generate another proxy promise that represents the completion of the second success handler that prints out the weather and calls Done to complete the test. So right now we have a promise for fetching the current city and we have two promises that are proxy promises that represent the completion of two separate success handlers. Now this first little program is done and at this point in time it's reasonable to assume that the queue is open in the event loop and eventually the current city operation will complete, push some code into the queue that will then be handled that will ultimately lead to the resolution of the fetchCurrentCity promise. So when the fetchCurrentCity operation completes successfully then that promise is resolved, its state turns to succeeded and then the success handlers for it will be fired off. Now in this case this promise has one success handler, the middle handler, the call to then here, so that one success handler will be fired off. The result of fetching the current city will be passed into this success handler and that success handler will use that to call fetchWeather and of course, calling fetchWeather will generate a new promise as well and that will represent the completion of the weather operation. Of course, the fetchWeather op is asynchronous so that promise will complete until later on. Now one thing to be careful of, the second call to then, that's chained to that proxy promise here. This second call to then is not attached to the fetchCurrentCity promise, it's attached to the second one because we chained it on there. So this is not a success handler; the second success handler here is not a success handler for fetching the current city; it's a success handler for after the weather has been fetched and that should intuitively make sense to you looking at the chain of control flow here. Okay, so once again, there is no work on the queue and eventually the fetchWeather operation will complete and assuming that it completes successfully, then it's promise will turn to success as well and the result of that promise will be forwarded on to the proxy promise and its result will also go to success and it will also cache a copy of the weather, which can then be used to trigger its success handlers and its success handlers includes this last success handler because we chained a call to then, so the weather response will be then forwarded on to this success handler, which will then use it to print it out and then this success handler will call Done to complete our test. Now once this success handler completes, our program is now done. However, you might be wondering, well what about that promise we have there that represents the completion of that second success handler, that proxy promise? It's just sitting there pending and it will sit there pending forever because we haven't wired up anything to happen. In a more full-fledged promise implementation or operation implantation, we would've also completed this promise, this proxy promise would've been completed even if there is nothing returned from the success handler. We don't have that right now and we could easily add that and in fact, that's a challenge you could take on if you'd like, so in a real promise implementation we would also succeed this then, once this success handler is complete. The last quadrant I have here on the screen in the lower left corner shows a very concise way of writing what we have in the lower right-hand corner. That's what we ended with, just showing how concise we can write things now with our operation/promise framework, but I did want to illustrate what we're talking about with the lower right example because there are more moving parts visible for me to be able to point out and I think it makes more sense to you; however, the lower left is probably how you'll eventually start writing your code once you get comfortable with promises, because it reads really nicely. So that's it for this module. We've seen how we can easily achieve parallelism with promises now. We've seen how we can remove nesting here in this example. We've seen how we can chain together operations and we've also seen how we can expose asynchronous operations that depend on other asynchronous operations, how we can expose those with proxy promises and then access those anywhere else in our application without having timing issues that we might have with callbacks. In the next module we're going to continue to work on our operation framework and we're going to see how the operation/promise framework we're building can add a lot of protection and safety when working with callbacks.

# Cushioning Callbacks

## Shifting Perspective

In this module, I want to turn our attention to some protection mechanisms we can add to our Promise framework. We'll spend quite a bit of time talking about errors. If you remember back with callbacks, callbacks introduce another error mechanism whereby errors are passed as function results and so now we have to deal with this in addition to catching thrown errors, so we have two error mechanisms that we have to work with. In this module, we'll see how we can add in additional features to our Promise framework to help simplify error handling and we'll also see how we can consolidate both catching errors and returning errors into one error mechanism with our Promise framework. Another challenge we had with callbacks was knowing whether or not they are called synchronously. Something may look like it's asynchronous and in fact, it's synchronous instead and that can cause big problems if you're making assumptions based on the timing of things in your application. We'll see how we can address this and force all callbacks to be executed asynchronously with our Promise framework. In this module we'll start shifting gears. We've been spending quite a bit of time thinking from the perspective of building this Promise framework, so being the producer of a Promise framework and we're going to keep that mentality, but we're also going to start thinking a lot about being the consumer of our Promise framework as well. So we'll start with a lot of examples in this module where we look at things from the consumer perspective. For example, I have a test case up here and I could ask you a question about this and this is from the perspective of a consumer, so we're using our Promise framework here. What will happen here if we throw an error in one of our success handlers? How is that going to affect our program execution? And one additional question. How might you like this to affect the program execution. We'll answer this later in this module, just bear in mind that we'll be starting to think about things more from the perspective of the consumer, so we'll start a shift from producer to consumer because now that you know how things work behind the scenes, you can take the knowledge into how things work from the consumer perspective, which is where you'll spend quite a bit of time working with promises. You'll spend quite a bit of time using promises in your applications. You probably won't spend that much time writing promise frameworks, so it's time to start shifting that mentality.

## Challenge - Error Recovery

Let's say that our GPS is currently offline so fetching the current city isn't possible. We can simulate this with a new function called fetchCurrentCityThatFails and this version of fetchCurrentCity will always fail the operation with the message that GPS is broken. Let's go ahead and add a test case that uses this. This test case is called error recovery and in this test case we call our fetchCurrentCity function that we know will fail. We hook up a success handler to it and since we know we're not going to get a city because GPS is broken, we want to do something somewhere to inject a default city in the case of a failure. So I've coded up the assertion for this test, but as you can see in the browser, this test is failing at this point because our success handler will never be called and that's because the operation fails and so obviously we wouldn't call the success handler in that case. We don't have any error handler in this case either, so our test is simply timing out. Now my question to you is and I want you to pause the video after this and think about this, how might we want to handle the case where an error occurs and then do something so that we can recover from it? So how might we want to set up error recovery? If you want a hint, stick around, otherwise go ahead and pause the video now. If you want a hint, think back to what we did to support being able to chain together calls to different asynchronous operations, for example fetchCurrentCity followed by fetching the weather. Look at we did to be able to support fetching the current weather while simultaneously avoiding nesting and you'll have a hint about what we could do for error recovery.

## Solution - Error Recovery

Here's how I thought through the error recovery challenge. So we have a second function we could pass here to specify an error handler, so we could receive the error here and then do something about it. Unfortunately, by the time we get here, we can't do anything that would affect our success handler here. We need to do something before we register our success handler to register some sort of error recovery. Now we know we have this onFailure function right now where we can specify an error handler as well, in addition to passing it as the second parameter to then. Now at this time the only thing we can do in an error handler is perhaps log out information or send some sort of notification. Basically we can react to an error, but we can't do anything to recover from an error at this point. What if we some ability to just return something else from this function that would replace the result in the case of an error, but what if we could just return default city here and that would be used as the result of the fetchCurrentCity operation. Of course, we can't replace the result of the fetchCurrentCity operation because that promise has settled at this point and we don't want to change its value because something else might be reacting to this failure as well in another part of our application, but we can modify the value of the operation that's returned from onFailure. Remember, we have that proxy promise we're generating when we call then and we're also generating that when we call onFailure, hence why we can chain then after calling onFailure. If I scroll up here, you'll see an onFailure returning the result of calling onCompletion, which is our then function. How about we replace that just for sanity? So we passed the proxy option back in this case too and actually instead of completionOp like we used here, let's call this proxyOp. Proxy operation or proxy promise, either works. They reflect the result of the success or error handler that's invoked. It's a proxy for that future operation, so we have that here too. So what if we could return default city as the result of that proxy operation, which means the proxy op could succeed while this original operation had failed and then we would be able to chain on to the successful result of our proxy operation and execute the success handler that we intended for this original operation. So that would provide error recovery and then it would also be nice if we bypass error recovery in the case where we don't need it. We'll get to that in a minute, but in other words, it would be nice if fetchCurrentCity works, if we called the overload that didn't fail. It'd be nice to just bypass on failure and we can wire that up as well. In many ways this becomes like a catch block with try catch. In fact, instead of onFailure, in promise frameworks, in the ES6 Promise spec, catch is used instead for this name. So let's come up and alias that as well because I actually like that name. So all we're doing is aliasing onFailure to be accessible via catch. So it's like having a catch block. A catch block is just skipped if there's no error; however, it will be invoked if there is and we can replace the value then in this case with the default value and once we've done that, once this works, then our test should pass because the city should come back with default city. I've pulled up the test here and you can see we still have a timeout because we have yet to implement this functionality. Let's scroll up and talk about this though. So in the last module, we created the success handler, which would look at the callback result of calling the passed in onSuccess function. We can do the same thing with our error handler case. So we can create an error handler and add some additional logic that will be invoked before we call whatever was passed to us with onError. Remember, onError comes from calling onFailure or catch and passing onError, which then flows through then up to our onCompletion function. So that error handler we passed down below where we're returning a default city, that default city will come out of invoking onError than in this case of having an error and of course, we can say if onError. Protect ourselves in case one wasn't passed in because it's possible somebody only passes a success handler, so if someone does pass an error handler though and in the case of an error then we can go ahead and invoke that and get the callback result. In this case we'll want to pass the error object. We can then take the callback result and use that to succeed our proxyOp, so our proxyOp.succeed and we'll pass a callback result. So whatever is returned from error is used to recover from error and more or less continue as if the original promise had completed successfully with this value as the result. Then we'll need to come down into here and use this instead of calling the onError function directly, just like we did with the success handler. Now if we go ahead and save this, our test is now passing. Now we have a few more Edge cases to deal with in this situation of an error handler and next we'll take a look at how we can recover from an error with another asynchronous operation, in other words how we can return a promise in the error handler and wait for that to resolve, just like we did up in the success case when we return a promise because we might have an async op that perhaps tries again, so we'll need to wait for that to complete before we can then recover from the error.

## Async Error Recovery and Challenge - Forwarding Success

One thing I like to do when I'm behind the scenes here with the promise framework is to look at the symmetry between success and error cases. This often illustrates opportunities to add additional functionality. It's easy to see, for example, that in the case of success be allowed to return a promise that represents an asynchronous operation, but in the case of error, we don't have that so we're just assuming that error recovery is always synchronous, but hey, why should it be? Error recovery may be asynchronous as well, so it makes sense then to mirror this functionality. In fact, the reverse is true as well. In the success case we are only allowing for the return of asynchronous operations. What about synchronous ones for success? So we'll see how both of these play out here in this module; however, while we can see these asymmetries and come up with additional functionality, I still want to continue to switch gears and think more about the consumption of promises and less about being the producer, again because you'll be the consumer more often than the producer and the implementation of the promise framework is going to get rather complex while I think the examples of understanding how to use things like error recovery will intuitively make sense. So in this case it's possible we want async error recovery. Let's create a test for that and then I'm going to paste in the above error recovery code in the synchronous case and in fact, let's call that synchronous error recovery. Now instead of returning something like a string, what if we just return the result of calling the fetchCurrentCity function that doesn't fail? Let's clean this up to simplify a little bit. So in the case of a failure here we'll call the actual function that doesn't fail and then we should assert that the city is not the default city. Save this. Of course, that test passes because the result, the city is not the default city, so how about we use toBe and let's go look up the value. New York, NY. Let's go ahead and copy that. Let's make a constant for it, actually, and use that expected current city and save this now. Okay, now we have a failing test. We expected this object here which is a promise. You'll see all the properties of that promise. Scroll way to the right. We expected it to be New York, so obviously it isn't and that's because right now we're succeeding with a promise. We're just passing the promise back here of our proxyOp. That doesn't make sense though. We should unwrap that promise if you will, which means waiting for its result and then complete our proxy option when it's done, which will represent the completion of this async op, just like we did in the last module when we returned promises and waited for them complete to unnest the nesting insanity. To fix this we just need to come up to our promise implementation and we need to mirror this functionality right here. Paste that in. So if the callback result is defined and it has an onCompletion function and how about we switch to then because that's what we'll be using with promise frameworks. So if it has a then property, then we'll go ahead and assume that it's a promise and we'll forward the completion of that promise onto our proxyOp, otherwise we'll succeed immediately. Now in the case that we have a promise come back, we want to bail out then and not succeed immediately. We can go ahead and save this now and our tests are all passing, so we have both synchronous and asynchronous error recovery now. I want to throw a little challenge your way. Right now we go ahead and recover from errors and set a new result, but what happens if there isn't an error? So let's call fetchCurrentCity which we know doesn't error and then let's simplify this error handler here, so again we return the default city, but in this case we should expect it to be the current city, the expectedCurrentCity that we know comes out of the fetchCurrentCity function, so let's write this test and just see what happens here. It looks like it times out. So go ahead and copy down this test and fix this test so it passes and then join me in the next clip and I'll show you how I approach this.

## Forwarding Success and Challenge

So right now this test doesn't work. The test times out, which means our success handler is never being called and that's because we've set up an error handler here and then we've chained onto it another operation that should be invoked if there is no error. In other words, success should fall through our proxy operation so that we can define our success handlers after our error recovery handlers, but we need one critical piece to pull this off. Up in our error handler, if no error handler is provided, then we don't do anything. Same thing with success cases. If a success handler is provided, we won't do anything. That becomes a problem then for chaining promises because our proxy operations just sit there pending forever unless we passed a handler; that doesn't make sense. We should do something with that proxy if a handler isn't defined and most of the time a handler isn't defined because remember, we pass two handlers every time we call then. There's a proxyOp generated and that proxyOp could succeed or fail, only in the case where we passed both handlers will we be able to continue to our chain, but when we don't pass one, we should be able to do something like bypass. Now before we had that noop, but bypassing will be a bit different, so in the case of providing an error handler like catch, like we have down below, this call to catch generates a proxyOp. In the case of an error we'll be okay, but because we didn't provide a success handler, this proxyOp just sits there pending forever; however, if we go up to onSuccess and if there was no onSuccess function, in other words else here, then we could go ahead and complete the proxy op by hand and in the case of a success we would want to succeed the proxyOp and we'll go ahead and just forward the result on to this promise. So we'll just forward the success result and that kind of makes sense. If we don't pass a success handler, go ahead and just let the successful result trickle on down the chain of promises. This allows us to bypass our error handlers in the case that there is no error. Now you can see our tests are passing. Now this asymmetry between the success and error cases is interesting because we might also want to do something here if no error handler was provided. So if we call then, but we don't pass an error handler, we might still want to do something with our proxyOp in the case of an error, so you tell me. What will I add here in the case of an error if there is no error handler? Go ahead and think about this. Write a test for this and just see if you can wrap your mind around what exactly we're providing functionality-wise here. What functionality does this provide? Join me in the next clip where I'll give you an answer to that.

## Solution - Forwarding Errors

Okay, before I give you an answer about what to put here, let's come down and let's write a test case for this and let's talk logically about what type of functionality we're creating. So let's copy one of these test cases and modify it to fit the functionality we're trying to add here. We'll come up with a name in a minute. Okay, so I'm interested when we register a success handler and not an error handler so let's start with that. So what if we have a then call here, assuming we're going to get the city, and let's just say we log out the city in that case and then we go ahead and complete by fetching the weather then with the given city. So a reasonable set of operations here, assuming that fetchCurrentCity passes, we'll then grab the forecast for that city and then we can go ahead and just get rid of this catch handler for right now and chain on a call to then here which we would assume gets the forecast and then we could expect that the forecast is equal to some value and let's come up and get that value so we can make an assertion on it. We'll do the same thing as we did in the case of getting the current city. We'll copy this value here, paste it here, and then we can actually use it inside of this function. We could do the same hard coding here with the expectedCurrentCity. Okay, so this is our expected forecast coming out of the getForecast function. So if we come down to our unnamed test, we should be able to assert at this point in time that the forecast matches the expected forecast and if I call a version of fetchCurrentCity that works, this test should complete successfully. Let's name that so we can see it and just run the test and you can see it's passing over in the browser. But what happens if we call fetchCurrentCityThatFails instead? Now you can see our test just times out and that's obviously because our operation at the top here, fetchCurrentCity is failing, which means none of our success handlers will be called, which means Done will never be called so our test will time out. Now when we talk about bypassing in the case of a success handler being defined, what we're really talking about is forwarding errors and letting errors fall through, just like we let the success value fall through up here to bypass error recovery if it's not needed, we should error values fall through as well so we could in one place, write a catch handler that gets the error and then can go ahead and handle the error. Now in this case we expect there to be an error here so let's put Done in there and delete that instance of Done and let's pass error to Done so we can see that error. Let's go ahead and save this now. You can see our test is still timing out, but what we're trying to pull off here is allowing errors to fall through, so when fetchCurrentCity fails, we want that to fall all the way through our success handlers and invoke our error handler here at the end that we registered by calling catch. This is like having a global error handler for a function. If fetchCurrentCity fails, none of this will fire off, which is really neat. That means the rest of our chain of operations that are dependent on that first failed op will just be aborted more or less, they'll never be invoked, and instead we'll invoke our error handler. Okay, let's get our test working now. Let's scroll up here. Remember, I eluded to this because of the asymmetry between our success and error handlers, so perhaps the solution is simpler than the test case we set up, but I hope that test case helps you understand the value of this and if it doesn't, stick with me because we've got another consequence of this that's really neat. So we take our proxyOp and we'll just fail that operation, if an error happens and there's no handler. So if there's no error handler, obviously we can't recover from that, so we just want to forward the error on so that the rest of our program can continue to execute and maybe we have an error handler later on that can take care of this then. This is a lot like try catch blocks that can catch exceptions thrown from inner functions. So in this case we pass operation.error then. We just forward it along like we forwarded the success result along. So we fail the proxyOp, passing along the same error, which mean it's like the proxyOp failed for the same reason. So what will happen then is fetchCurrentCity will fail. This success handler will never be called, but the proxyOp returned by calling then will fail with the same reason as fetchCurrentCity. Then down here, this success handler will never be invoked. The proxyOp returned by this call to then, remember we have separate proxyOps for each call to then and catch, this proxyOp will fail as well and finally, our error will arrive where we've defined an error handler; that error handler will be invoked and our test should now pass if we save this. We are still getting a red message over here, though if we make this a little bigger, you can see the error actually here is that Done was invoked with a non-error. Calls to Done expect an object of the error type, so our code is actually working. Let's go fix this though by coming to our fetchCurrentCityThatFails and let's just make an error object here. Now if we save this, you can see in the browser, we get error fallthrough and that shows our error message. Now actually what we want to do to make sure that this test case passes, in other words expects an error, we don't want to pass the error along, so it's expected we hit this block with an error. If anything, we could write an assertion about what we expect this error to be, but now this should make our test pass.

## Reusing Error Handlers

So I have a question for you. What happens if I take out fetchCurrentCityThatFails and replace it with a fetchCurrentCityThatWorks and then I come down to fetchForecast, take out the city here so that fetchForecast fails instead. What's going to happen here with our program execution? Will our test complete successfully? Let's run it and see what happens. As you can see, I re-ran the test there and the test still completes. The really neat thing about forwarding errors is that that error could've come from anywhere up the chain of promises. Think of this operation down here where we have three stages. If anything went wrong in any of these stages, we could just slap on a catch handler and write some code to deal with it. We don't have to have a catch handler for each asynchronous operation, which stands in stark contrast to callbacks where we have to write that if statement every time a callback is called, otherwise we might miss an error. Here's that callback example again. You can get to this in callback.examples. Remember we have to have these if statements every time to check for an error, otherwise something could go wrong. Well now, not only can we separate these out with a catch block and keep them separate from success so we don't have this if statement at all, now we can just write this one time and have one chunk of code that catches errors for multiple asynchronous operations and we can chain that on right at the end and in a lot of cases when errors happen, we don't care where they happen from, we won't be able to do anything about them; we just need to show a message to the user or fail an operation and move on with the day, but we don't need to duplicate that code 15 times and we don't need to plug that code into 15 different places either. We should be able to just write it in one place and be done with it.

## Synchronous Result Transformation

There's still one last piece of asymmetry between the success handling and error handling logic we have in our promise framework now. It's this line right here. In the case where the callback result is not a promise in our success handlers, we don't do anything about that. What I want you to do is pause the video and think about this last case, think about how you might reconcile this asymmetry, but more importantly, right down a test case of this and try and figure out what features this would provide in a promise framework. Go ahead and pause the video and then join me back in this video when I'll show you how to make this symmetric. Okay, the first thing let's do is let's go ahead and add a test case for this. In the case of error recovery, we could simply return some value and that would be used to resolve our proxyOp. We referred to that as synchronous error recovery and we also had async error recovery where we could return a promise instead. In the last module, we talked about returning promises from our success handlers so that we could unnest the complexity of nesting when a success handler called an async operation. So the last piece here then is on the success side of things, if we return a result synchronously instead of asynchronously, we would want to forward that as well. That would match then and make those two error handler and successHandler functions symmetric. So what we could do is come up here and copy one of these simpler tests and let's call this synchronous result transformation and what I mean by that, if we put in some function that will complete successfully, we would get back some city here. It's possible we get a city when we really need a zip code and let's just say we have a synchronously available in-memory cache of cities to zip codes and ultimately what we could do is look up, not asynchronously but synchronously, the zip code. So let's say we get back this zip code. We could transform this value then and this will be the value then that resolves our proxyOp so the zip code is available in the next step, not the city. So we'd have the zip code here. So let me come down here and we'd expect the zip code to be 10019. So this is a completely reasonable thing we might want to do. We might have some synchronous operation instead of an asynchronous operation that forms the next step in our control flow so we could use a success handler to perform that transformation and if you will, we keep inside of our promise continuation. We're still passing around promises, we're just mapping the value of a promise, so if you've ever worked with functional frameworks, this is like implementing map for promises and in this case it's the synchronous map operation. So we can go ahead and save this and run our tests and of course, right now this times out and that's because we don't have symmetry, so what we want to do in the case of success, if we have a success handler defined and things are successful and the result that comes back is not a promise, meaning we don't have an async op, we have a synchronous transformation instead, let me just go ahead and take that result and forward it on to our proxyOp. So we succeed it as well and this time we just pass the callback result, which in the case of our code below, will be a zip code. So we can save this now and of course, it's a good thing we have test case. You can see a bunch of tests errored out here and what's popping into my mind is the last bit of asymmetry and we need that return statement in here to bail out so we don't do both of these pieces of code if we have a promise. Now let's save our tests and run them again and it looks like we still have an error. Ah! It looks like it's a type issue. Let's come down to our test case. So this should be 10019. Save that. And of course, we should also call Done. Okay, now our test is working. So as you can see here, we could have asynchronous or synchronous transformations of what is a promise pipeline. So you can start to think of promises as a pipeline of operations that will execute in your program and of course then, if you're thinking in that mentality, the idea of transformation starts to make sense too, so that's another way to look at the callbacks are involved. They're just transforming values for you. We get a city and then we turn that into a zip code. A zip code might go into a weather API that works on zip codes and give us the forecast and then if we have any problems, we have some error handler at the end of that pipeline. Now before we move on, I do want to point out one thing, a reason why I typically don't use intermediates here to transform values synchronously, and that's because if you just go ahead and get rid of those two lines of code and just declare a variable instead, this is the same thing. One less step in the process. So most of the time when you have synchronous transformations, you would just do those inside of the same function that you're going to then consume the result in, but the option is there if you want to transform the result synchronously with a separate step in your promise pipeline. Separating things out might be useful if you already have some existing functions that do all this and you don't want to rewrite those functions or you want to have something like this that's really readable where you could do something like fetchCurrentCity, then fetch the zip by city, then fetch the weather, etc. So this reads really well. This might be a reason to do this as well.

## Recover from Thrown Errors Too

I'm going to paste in another test case here. This tests involves throwing an error in a success handler. I want you to think about what will happen when we run this and then I also want you to think about what you'd like to have happen when we run this. Go ahead and pause the video and then join back so we can talk about this. We know when we're working with callbacks that we need to check that error object that comes in, but a very common mistake with callbacks is then to just assume that there are no errors thrown when it's very possible a function could throw an error, like when you're firing off a web request if you pass in a malformed URL, that might be an error that's thrown instead of being passed to the callback. So we can miss these situations where we could have a thrown error and if we really wanted to handle both errors from the callback and throwing errors, we would need a try/catch block in addition to that if check. So if we really wanted to avoid any mistakes, we would need to come in here and wrap this call to getWeather with a try/catch block. Something like this. Now could you imagine needing to write both these four lines of code and this code? And in fact, don't forget, we need a try/catch around getCurrentCity because, well, it could throw errors as well. If we have well documented functions that work as they should, we might not need to do this, but come on. We know mistakes happen even with the best of intentions. So if we truly want our program safe with callbacks, we need both of these error checks, otherwise we'll get into situations where an uncaught exception happens in our program, which by the way, is what happens when we run that test, so let me run it. I saved everything and we're getting an uncaught error. An uncaught error will crash a program. So your program is going down if you don't do anything about this. I hope you have something to bring it back it up and I really hope you have some way of getting some error handlers in there. Now we've already seen how we can make error handling easy from the errors that come back from the async operation; these errors right here in the callback. We also can take care of these synchronous errors as well and we can do that because guess what? This function, this callback is being invoked by our promise framework. All of our success handlers are invoked by our promise framework. Remember, we've been working with this quite a bit in this module. Right here and right here are the two places where we invoke these functions. So we could put the try/catch block here and here and make it reusable. So let's go ahead and do that. Now we already have a test case set up, in the case where a success handler throws an error so let's deal with that first. Let's go ahead and select this line of code and then let's use a Surround With in WebStorm which you can get with Control + Shift + A and then let's pick try/catch. Now we'll need to move the declaration above and then this needs to be a variable we can change. Okay, so we have a try/catch now around invoking our success handler and if something goes wrong, we can do something about it. Let's take a look at that test we have right now. In this case if something goes wrong, we'd like to just pass that along as an error like any other error and we could reuse our catch handler then, making even more reuse out of these reusable catch error handlers. So in that case then, if something goes wrong, we need to fail our proxyOp and that makes sense. Our success handler fails by throwing an exception, that means our proxyOp should fail. So we can pass that exception along. Note we're passing the error that was thrown. Don't get that confused with the asynchronous operations error result and then in this case we need to bail out of this function at this point because we don't want to continue on. And then if we go ahead and save all this, our test should now pass. So we now have error recovery from thrown errors as well as asynchronous, so both async and synchronous error handling all wrapped up in one. That means we can get back to just having one error handling mechanism with asynchronous operations; that's invaluable.

## Recover from Error Errors

Just as it makes sense to recover from errors in success handlers that are thrown, we also might like to recover from thrown errors in catch handlers. It's possible the code we're trying to recover with could throw an error. I've pasted in a test case here to illustrate this called error, error recovery and we throw an error here called "oh noes" and then we expect the message on that to be "oh noes" and then we throw another error here with a different message and that should be then caught by this next catch handler and that message should be the new errors message and then at that point we'll go ahead and complete the test successfully. Now right now when I run this, you can see we have an uncaught error, so let's go fix this and wire this up. Now if you're astute, you'll know that we just have some asymmetry now between our success and error handlers. Specifically we have this try/catch block up here that we'd like to move down below, so let's go ahead and just copy this and let's come down here and paste that in right above and let's take a look at what we have here. We need to take this part right here where we call onError, passing the error result and put that right here. Now we'll catch thrown errors out of our error handlers. Save this and now our tests are passing, and of course, we've restored the symmetry between our success and error handlers, which is always seemingly a good thing here.

## Operations Should Only Complete One Time

We spent a good chunk of time talking about how promises can provide a lot of protection for error scenarios, but there are other scenarios as well that we can protect ourselves from. For example, let me paste in a different implantation of fetchCurrentCity and this one is indecisive. It can't decide what result it wants and instead it tries to complete the operation or the promise twice, so it tries to call succeed twice. This represents a case where we pass a callback to some library we're using. That library could actually call our callback multiple times. We normally don't think that way because we expect callbacks to only be invoked once, but it's a function, so it could be invoked repeatedly and if that were placing an order or charging a credit card, you can imagine that that would cause some problems. Take a minute and pause the video and think about what you think should happen if we invoke this fetchCurrentCity function. What should happen if we hook up a success handler, and also think about what will happen right now with our promise implementation. Right now with our promise implementation the success handler will be called multiple times, just like if we had callbacks. So here's a test of this and this one's pretty straightforward to test. So we can protect from doubling up on success because we only want to invoke a callback once. That's expected behavior for callbacks so how about we enforce that? So the very first call to succeed is the only one that we'll complete. We'll ignore subsequent calls with additional values. Now you might have thought there's a different way to handle that, maybe throwing an exception here; I'm sticking with ignoring because that's what the ES6 promise spec goes with, so that's something you'll need to become familiar with. So in this case we just want to ensure that Done is only called once, but if I save this right now, you can see Done is called multiple times. So take a minute if you want an additional challenge, pause the video and go fix this. Fix this so it only calls Done with the first value. Okay, to fix this, we need to come up to our functions that complete our operation, the fail and succeed functions and in this case we're talking about success so we'll start with that one. What we could do when we call the succeed function is create some variable that lets us know if this operation is complete, so how about just a complete variable? We'll set that to true when succeed is called and then we'll check that and see if that's true and if so, we'll bail out except we'll need to do that before we set this to true. So when we come into succeed, we'll check to see if the operation is already complete. If it is complete, we'll bail out of trying to complete it again, otherwise we'll set this complete to true. Now when we save this code, you can see our test is now passing.

## Block Multiple Failures

In addition to not succeeding twice, an operation should not be failed twice either, so here's some code to check that out. So instead of succeeding twice, this function fails twice and then we perform the same check, just make sure Done is not called multiple times. So if we save this, we can see this test fails as well for the same reason. We can come up here and duplicate our check in the case of a failure. Save our code and now our tests are passing. One thing of note here is that we share one flag complete between both succeeding and failing a promise and that makes sense because an operation either succeeds or fails; it doesn't do both and it doesn't do either of them more than one time.

## Surrepticious Synchronicity

I added in a new test case here called what does this print out? This is a test case that simulates what a user interface might go through as an application is loading up in the browser. We might show this loading message to our users if we have some data we need to fetch until that data is available and then once that data is available, we can then update the UI and show the information. For example, in this case, showing the user what city they're from, if it takes us a little bit of time to fetch their location from GPS and get a city name to show. So this is a common pattern. We show loading so that users don't think our application is broken. By the way, this uses a new fetchCurrentCity2 function, but I don't want you to worry about what that does just yet. We'll get to that in a minute. For now, just focus on the consumption code here and tell me what you think will happen to this UI variable over time. What values will it hold and when? Maybe more specifically, assume that fetchCurrentCity will complete within 100ms, answer the question, what value will UI have after a second? So if we're a user and we turn away for a second and we turn back, what value will we see in the user interface? Okay, so let's walk through this. Now what I've done is I've coded up an assertion that would help us simulate what it's like to be a user that maybe looks away and then comes back and looks at the program in a second and if I were that user, I would expect, and if I were the application developer behind this app, I would expect that they would see in a second, they'd see what city they're from. So I added that and then I complete off our test after a second and this is all under the assumption that we get back the current city within a second as well. You can see though on the right-hand side, this test has actually executed now and you can see we have an uncaught error. We expected loading to be you are from New York City. So what's going on is we wanted to see this, but we're actually seeing loading… Now if I change fetchCurrentCity to the original and save this, you'll see our test case passes. It takes a bit of time. You can even see Mocha is telling you that there is quite a bit of delay here so our 1 second is working, but you can see in this case, it passes whereas using fetchCurrentCity2, it doesn't pass. Now both of them involve promises, asynchronous operations. Why is it that these work differently? Without looking at fetchCurrentCity2, take a stab at why you think there's a difference. What did I do differently in fetchCurrentCity2? So here is what I did differently. Let's open this up. I didn't make a call to do later. I succeeded the operation right away. So in other words, I completed the operation synchronously and right now, our promise framework is set up to fire off success handlers and error handlers as soon as an operation completes and then right away once an operation completes. So what happens here in the case of fetchCurrentCity2, we call this function which makes an operation. We log out getting the weather, we succeed the operation right away, so we complete it and pass the value, New York, NY and then we return that completed operation so at this time the operation is already done. That means when we call then, remember our code for then? Let's collapse a little bit of this here. We don't need to see the error handler and success handler wrappers we have, so our call to then calls onCompletion and if the state is succeeded or it's failed, we go ahead and fire off the handler instead of pushing it into that array and we did that so that we could support registering callbacks after an operation completes. So what that means then for our test case down here is that we immediately invoke this function as if it were synchronous, but when we're working with callbacks, we tend to think that callbacks are executed asynchronously because any good callback design would include actually executing the callback asynchronously. So we get in the correct habit, or should be correct habit of assuming that callbacks are asynchronous, but we have no guarantee of that and if you've watched that previous course, Reasoning About Asynchronicity in JavaScript, then you've seen some other examples of this as well and we know we shouldn't assume that things are asynchronous because they may not be and that's one of the big problems with callbacks, passing those around, we assume asynchronicity, but we don't know unless we actually look at the implementation of the code we're passing our callback to and we really don't know unless we look at every version of it because if we perform an update, there could be a bug that switches from asynchronous to synchronous and then we could have complete chaos in our applications because the timing is off. So one of the protections we can provide with promise frameworks is the ability to ensure that callbacks are executed asynchronously; in other words, do things later on. Remember our doLater function? We can ensure that we always do callbacks later on so the current code that's executing can run to completion, even if later on is only a millisecond later or as fast as possible. In other words, we can always allow the current code that's executing in the event loop to finish before we ever trigger a callback from promises and that will guarantee we always execute callbacks asynchronously. So if you want a challenge, go ahead and pause the video now and implement that yourself or just follow along with me and I'll show you how to do that. So if we come up and look at our promise framework, there are a couple of places where we invoke success and error handlers. Here, here, here, and here. So we either do them right away if we're already done, or we do them later on. The laterOn actually comes into play then in the failing and succeeding functions where you can see we go through our reactions, we call those reactions, and we execute each of them, but this is going to be executed synchronously. So these are the places we'll need to update our application. Conveniently, all of these happen to refer back to these success and errorHandler functions we created earlier. So we could just patch these up to take care of this as well. These error handler and success handler wrappers, if you will, have been really convenient to dump some functionality and make sure it happens everywhere inside of our promise framework. So let's modify these to support always executing callbacks asynchronously. Join me in the next video where we'll pick up with this and if you really want a challenge, I'd encourage you to go ahead and implement this yourself.

## Ensuring Asynchronicity

Now this test I set up to help you understand the timing issues; this is a bit too convoluted for what we need so I'm going to create two new tests. We have to cases to handle, one for success handlers and one for error handlers and we want to make sure that these are both called asynchronously. Now if you look at the basis of this test I had set up, we have this fetchCurrentCity2 function that basically just creates an operation and immediately succeeds it, so we will use this in the success case, and then we do the opposite; we'd fail this in the error case. Then we can use these completed operations or promises and immediately change success and error handlers to them and then use a quick little trick here where we alias the Done method to end the test and we alias it after we set up these handlers, so that way this will fail here, this call here to doneAlias will fail if the handlers are called synchronously, so we can copy this and use this down below as well. The only difference here is we're wire up a catch handler and then after we add that in, let's make sure we rerun our tests and of course, you should be getting timeouts as expected. Okay, we have two tests in place. We can now scroll up and we can deal with making our handlers asynchronous. So again, we can just add this to our successHandler wrapper function and errorHandler wrapper function and remember we have that doLater function we defined? Let's use that. We'll open up a function to pass in and we'll just wrap up all the code inside of our successHandler. This might not be the most glamorous way to do this right now, but this will work for now. If we go ahead and save this, then our tests should rerun and the success case should complete and it does. We still have a timeout on the error side of things though, so let's go ahead and just copy this call to doLater and that should take care of the error side of things and you can now see all of our tests are passing, even that funky UI test that we had set up, but in the interest of not having slow tests, let's go ahead and just get rid of that UI test. We don't need that anymore. Okay, so we're now ensuring with our promise framework that the callbacks we register will be called asynchronously. This is yet another layer of protection that we've added to our promise framework.

## Summary of Protections

As we've seen, when things go wrong, we have quite a few new mechanisms built in to our promise framework to help us out, to make our life easy, to help us avoid getting into troublesome situations that we very likely are getting into with pure callbacks. We started out looking at error recovery. How can we take a situation where a promise fails and transform that promise into perhaps a default value and how can we do that and easily forward that onto our success handlers as if the original promise never failed and we set up error recovery to work both with synchronous recovery and async. Say a service fails on you. You made a web request and the service fails. You might want to retry that request so that might be a subsequent asynchronous error recovery. We've got that now. Imagine what that would be like with callbacks. If there is no error, we bypass our handlers for errors and just forward the result along as if we never had a call to catch in our promise pipeline. This allows us to inject catch handlers anywhere we'd like in a pipeline of promises as we've seen in this module. We even saw how we can recover from errors that happen when we're trying to recover from an error and that work seamlessly as well and perhaps best of all, we've seen how we can leverage error fall through. So if we have several asynchronous operations that are serially chained together, we can have one handler for all of them if something goes wrong. This is in stark contrast to needing a special handler inside of each callback at every stage of the pipeline. Now we can just leverage one, which is often the case what we'll do because if something goes wrong, we usually can't do anything about it, so we just need one place to detect that happened and take the appropriate corrective actions, so promises give us this ability to get back to reusable error handlers. In this module we also saw how we can get back to just one error handling mechanism, which is enough of a problem as it is. Our promise framework doesn't care if an error comes from something thrown synchronously or an error coming back asynchronously. All of these errors are folded up into the same mechanism and we can register error handlers with the catch method at any point in our pipeline to deal with the error appropriately. We spent quite a time in this module focusing on error handling, which is the majority of the problems we have where the protections provided to us by promises really help out, but these protections extend to other problems that we're often unaware of. For example, our promise framework allows us to ensure that our callbacks are only called once. Have you ever had a library you're using call your callback more than one time and cause you some grief? Well, now we know with our promise framework that that's never going to happen. Those subsequent calls to complete the promise will just be ignored. The last protection mechanism we saw was how we can ensure that our callbacks are fired off asynchronously because these lead to an even nastier problem if we're making one set of assumptions about timing and the reality of timing is entirely different because a callback is being invoked synchronously. Join me in the next module where we'll finish bridging the gap between the operation type, our promise framework that we've been building, and ES6 Promises, which are what you'll use going forward for Promises.

# ES6 Promises and Beyond

## Promises are Not New

It's time to bridge the gap between what we've implemented thus far in our Promise framework and what the ES6 specifications provide for the behavior of a Promise, so I'll make a few small changes yet to our Promise framework, but we'll mostly focus on some of the features we haven't had time to get to and we'll take a look at a lot of these from the perspective of a consumer, in other words, how we use them. Promises have been around for quite a long time now. One of the first places you likely use them is in the jQuery framework. A simple search of the jQuery docs for the word Promise turns up many results and that's because Promises are baked into jQuery. jQuery helps us interact with the web page and make web requests so naturally there are lots of asynchronous operations happening and thus Promises are beneficial. One primary difference with the implementation of Promises in jQuery is that they're referred to as deferreds. Some Promise operations use the word defer to represent the idea that an operation is potentially deferred for some future point in time and so a deferred is a way of abstracting or modeling that operation that will eventually complete. Bluebird is another Promise library that's rather popular and so is the q library. Both of these work with Node.js and inside the browser and for a long time now these have been providing us with Promises in both environments. Because Promises are something that can be implemented by a library, they don't really require any special language syntax nor is it necessary for the environment to be aware of them, thus over time quite a few Promise libraries have accumulated and as you can imagine, people had different opinions about how a Promise library should work and the terminology involved. Of course, there have been attempts to standardize on what a Promise framework should entail and one of those attempts was the Promises/A spec upon which the Promises/A+ spec was built and the Promises/A+ spec is what was used to formalize Promises as part of the ES 2016 specification or ES6, and if you look in the ES spec doc, you'll see a section 25.4 on Promise Objects. Now as of right now at the time of recording, ES 2017 has been released so I'm looking at the most recent version of this, but Promises were introduced in ES 2016. So there has been a history involved with Promises and there are a lot of Promise frameworks that have different opinions and this plays into some of the features we have yet to roll into our Promise framework that we'll talk about in this module. Promises are by no means unique to JavaScript. This is a list of implementations out on Wikipedia of futures and Promises; remember futures are another way to refer to the same concept of Promises or deferreds or operation in our case. This is a list of languages and implementations of Promises. Promises are a pervasive concept and that's because they're very beneficial and now that Promises have been added into the JavaScript language spec, we're starting to see many APIs being created that use Promises instead of callbacks. For example, there's a new fetch API in the works to replace an XML HTTP request. This new fetch API is entirely based on Promises. I've got an example of this where I rewrote that weather plunker and took advantage of the fetch API instead. Now in this case I'm just fetching the weather. I'm not fetching the forecast. We'll take a look at parallel requests later in this module, but for right now, this is an example of using the new fetch API. On the right-hand side I've pulled up our old example using an XML HTTP request so you can compare these. We now just have a simple fetch operation and pass the URL to fetch and the request will be initiated, as opposed to needing to new up a request and then eventually call open and send, but most importantly, we use success handlers with then and error handlers with catch, like we've been working on throughout this course and the nice thing is, all of this is built into the browser so you can run this example right now. The fetch API uses native Promises in the browser provided as a part of the ES6 spec. If you pull open the console in the browser, you can type in the word Promise and you'll get completion for this new Promise type that's baked into the browser for you. Okay, so let's turn to talking a bit about some of the differences in functionality and terminology between what we've built thus far and this Promise type that's baked into the browser.

## Terminology

ES6 Promises just like our operation type, start in the pending state, and then once they transition out of the pending state, they transition into one of two final states, either fulfilled with a value or rejected with a reason. These final states we called completed when we were dealing with operation type to indicate the idea that an operation has completed. These completed states are referred to as settled and the best way to avoid any confusion is to keep in mind that settled means that the Promise is not pending anymore. Now just like with our operation type, there are few control functions that can transition a Promise from the pending state into one of these final states. First, there's an aptly named reject method to match the rejected state. We had fail and failed with operations so this makes sense; reject would turn a Promise from pending into rejected and of course you provide the reason as well. Now on the flip side, on the success side of things, we had succeed and we'll get to why this is called resolve and not fulfill, which would match the fulfilled state. It has to do with result forwarding when we were chaining Promises, but we'll get back to that. Now just where our operation Promise framework is at right now, there are then and catch methods that register, fulfill reactions, and reject reactions and of course, these names just mirror the new names for the state so instead of success reactions, we have fulfill reactions and instead of error reactions we have reject reactions. As you work on becoming familiar with this terminology, if anything is confusing, refer back to the exact same set of terminology we had for operations. This is a second set of terminology to relate to, a set that I think is honestly a little easier to relate to than some of the terms like reject versus fulfilled. Either way, you have two sets of terminology to work with and keep in mind, if you're trying to help somebody else understand Promises, you could show them this set of terminology and it might help them as well and you could make comparisons between this set of terminology and the ES6 Promise terminology and in that comparison process you can learn a lot.

## Why Is it Called Resolve?

Let's take a look at why we have resolve instead of fulfill as one of the control functions for ES6 Promises. If you open up our operation framework and you take a look in the success and error handlers, we have this piece right here where we check to see if the callback result is itself a Promise and we do that by looking for a then method and if we do detect a Promise, then we go ahead and forward the completion of our callback result onto our proxy operation and that's what gave us the support for chaining Promises together instead of having to nest Promises and this also gave us support for asynchronous error recovery in the case of errors, so the same code exists right here as well and then if the callback result is not a Promise, we just immediately succeed our proxyOp. Now if we look at this forwardCompletion method, it's pretty basic; we just wire up one Promise to trigger the completion of the other Promise. We could use then in here instead just to avoid any confusion and then you could see, we're calling the control functions for the Promise that we're forwarding the result onto, so in a success we'll forward the success and in a failure we'll forward the failure. The combination of this check and this check is what forms the resolve method. So resolve method doesn't just succeed a Promise, it also can take in another Promise and lock in the current Promise to the Promise that was passed in. In other words, it sets us up for forwarding results between Promises and that's why we have a function called resolve and not just succeed because the resolve function might wait for the result of a Promise, which could either succeed or fail, so it wouldn't make sense to call this function succeed. Let's take a look at quick test case though to really understand what's going on here. So let's add a test called what is resolve and then let's do this; let's just work with Promises directly. So let's create a fetchCurrentCity Promise like we might create if we called that fetchCurrentCity function and we'll succeed it with the value of NYC. Then let's create a Promise that we're calling fetchClone and then you could imagine trying to succeed this fetchClone with fetchCurrentCity. So you could imagine trying to succeed one Promise with the result of a second Promise and of course, the reason that really doesn't make sense is it's possible that this original Promise failed and so it wouldn't be success, but it'd be a failure in that case. We'll get to that in a moment. If we wanted to test that fetchClone gets the result of fetchCurrentCity, we could take fetchClone then, attach a success reaction, and it'll get the city value and then we can go ahead and paste in an expectation to check the city value and then complete our test by calling Done. Right now when we run this test, obviously this fails because our succeed function doesn't do anything to check if a Promise is passed in; it just takes the result and uses that to complete the Promise, so right now the result is a Promise. By the way, for completion sake, we could go ahead and alias our fail function as reject and then we'd like something in here called resolve and that needs to be a new function, except instead of a result it really gets a value and that value could be a Promise or a result. So what we can do then with this resolve function is come down here and grab this code that checks the callback result, yank all of this out, and bring this up here. Then what we'll do when we get the callback result is take our proxy operation and resolve it with whatever the callback result is, so we don't care anymore inside of our success handler. We don't care what comes back. We'll let resolve take care of unwrapping the Promise, if there is one. It means we can come down here into our error handler as well because this is the exact same code. Just to show you that those are indeed the same chunks of code, I've pasted them into a dif tool here and you can see the left and the right-hand side match. Alright, we can go ahead and paste in the same proxyOp.resolve call in the errorHandler case and again, same thing there. We'll let resolve take care of dealing with whatever the callback result is and unwrapping in the case that it's a Promise, representing an asynchronous error recovery scenario. Okay, now we need to patch up our new resolve function because we don't have callback result here; we have value instead, so let's just put that there so that I can rename that variable to value. So here is how we'll check things. First off, value could be a Promise. If value is a Promise, we'll know that because it'll have a then method on it. In Promise parlance this is referred to as a thenable, something with a then method and anything that's a thenable is assumed to be a Promise. We'll talk a bit about that more later on in this module. So if it is a Promise, then we want to forward the completion onto, in this case, the operation. The proxyOp was a particular operation we had down here, so when we call proxyOp.resolve, proxyOp will be the operation because we're calling its resolve method. So we'll forward the completion onto whatever operation we pass and we'll bail out then. Otherwise we'll go ahead and succeed the operation with the given value because it's not a Promise so we'll use the value as a result of the operation. Okay, let's go ahead and make sure our test case is working and to make it work now, we want to use this new resolve function, it's a special enhanced version of succeed that will unwrap Promises and forward the result, so we're taking the fetchClone here and we want to resolve it with this Promise, fetchCurrentCity so the result of fetchCurrentCity will be used as the result to fetchClone. Save this now and now you can see our tests are passing. Join me in the next clip where we expand upon this resolve method.

## Reworking Resolve

So we have this new resolve function and all it does right now is add a bit of additional functionality in front of calling succeed, so what we're doing here is we're replacing succeed with resolve and to keep things simple, we should start using resolve instead of using succeed and there are a few places where we're using succeed right now; let's take a look at some of those. So we have it here in one of our test cases when we are setting up an operation that we wanted to immediately succeed. We can go ahead and just switch that over to resolve, that should work fine. We have it here when we're testing out resolve so we can switch there as well and this time we're passing a string value so that will succeed immediately. We have it inside of our onCompletion method inside of our operation framework here, our Promise framework, so definitely we can just call resolve here. And then next here, in fetching the current city that was indecisive, we'll use it here. We can go ahead and replace that in both of these instances. Next inside of forwarding the completion, so this is a part of our Promise framework again, we call succeed and we can just pass resolve here. In fact, let's go ahead and yank this line. We don't need this forwardCompletion function anymore and let's put this line right inside of our code up here. I'll paste that in, we'll fix that in a moment, but let's get our renames done before we come back to that. So the nodeCallback. Remember this is the wrapper function we used earlier, just a convenience method. We can call resolve in that case as well. Remember, we use this to build up our fetch functions. NodeCallback is a callback we use to turn a callback function into one that resolves with an operation into one that uses this operation we've created, that's convenience method again. So in the case of a failure, we can just call actually reject here or in the case of the callback succeeding, we resolve them with whatever result came back. And the last case we have here is in operation.resolve itself and it's fine to leave that because we do want to delegate over to this other function. The only thing I'm going to do here, I'm going to take this function away from being accessible outside of the Promise, so we'll make a private function here to encapsulate and hide away this succeed function and that means we just call succeed directly now and pass the value. Now the interesting thing about doing this update to use resolve and to hide away succeed is we can save our tests now and we should have some guidance now from our tests about what we might have forgotten to do when we switched over to this new resolve method instead of a succeed method. You can see the first failure is from this forwardCompletion change that we didn't complete so let's go ahead and do that now; let's comment out what we had before. Okay, so if we get a thenable, we take that thenable and we wait for it to succeed or fail and when it does, we have it call back and resolve or fail this operation. Okay, so we can go ahead and delete this line above and now when we save our tests, you can see that they're succeeding, so we're now wiring to this public resolve method and we've hidden away succeed and it looks like our tests are all passing so that's a good sign unless of course we have some missing tests, but we do have one problem here. It's not a huge problem, but right now our check to see if the operation is already complete is inside of our succeed function and that's tucked away, so that will be checked before we ever complete this Promise, but it is possible we could resolve the Promise multiple times right now, if we pass in multiple Promises. Of course, eventually only one will get through to the succeed function, but we really don't know which one that will be, so if we want that to be predictable, we shouldn't allow people, and if I save the tests here we should have some broken tests; we shouldn't allow people to resolve with a Promise multiple times and then have some random Promise finally resolve this Promise. In other words, we shouldn't have randomness there. We should only resolve from the very first call to resolve. So if we put this operation.complete check inside of the resolve function now, then we know we'll only succeed this Promise or fail it based on the result of the first call to resolve, so that will be predictable. Now when we save this and run this, we have some problems with our tests. Join me in the next clip where we talk about this problem.

## Challenge - Fixing Resolve

Okay, so in moving this operation.complete check so that we can't call resolve multiple times and also so we can't call fail multiple times, in moving that we've caused some of our tests to break. If you would like a challenge, pause the video and figure out how you're going to rewrite this code to be able to accommodate for the problem we have here. So you'll have to figure out what's wrong with these tests and then fix our Promise implementation to accommodate for what we just moved. If you want a hint, stick around. I'll give you one in a few seconds. The hint is that we call resolve on our operation once the Promise we're waiting for completes. Okay, pause the video if you would like to work on that challenge, otherwise continue on with me and I'll show you how we address this. What's happening? First we call resolve and let's take a look at an example of this in our what is resolve. First we call resolve on the fetchClone. We call resolve and pass fetchCurrentCity, so that's our first call to resolve. So now let's come back up to our resolve function and you can see inside of here what we set up then, fetchCurrentCity is the value so it will be fetchCurrentCity.then and then we'll call resolve again on the same operation so that causes a problem. So our guard here is causing us a problem when we have a Promise involved because we need to call resolve again to be able to succeed the operation. Now one quick workaround for this is to just call succeed here instead of operation.resolve. So we can just call our private succeed function. Keep in mind, I'm passing this here as a function. I could've written this like this, but I'm just shortening this up and passing succeed. Now when we save our tests, you can see that all but one of our tests are passing now and if we were to look at this one, we're using error handlers. The problem here is fetchForecast is set up in this case to fail. So we're resolving with the Promise that's going to fail, so let's go back and look at in our resolve function. So it's the second case here. When we call operation.fail we hit our guard check again and have a problem. So the reason our tests were timing out is because these subsequent calls were just ignored. What we need to do is create some function to bypass our check here if we know it's safe to do so. So let's yank this out and let's create a function called internalFail to which we can pass an error and we'll paste in the code we removed and then we'll use that internal fail here and actually let's call this internalReject to keep in line with using reject now. Now we can use this down below as well. Now we'll bypass the check and conditions where it's safe, so our test should now pass and our tests are now passing. To maintain symmetry, let's rename the succeed function and we'll call it internalResolve. So what we have here are public facing functions, fail or actually reject is what we care about now, and resolve, so those are public facing and for those, we'll go ahead and protect those functions with these checks here. We then have internal functions where it's okay to bypass those checks and conditions where we know that's okay to do and the only condition really where we have that is right here when we have to wait for the result of another Promise. By the way, if we want, we can take this piece of code here and we can move it up into internalResolve as well. And the reason we'd do that is we could have Promises that return Promises that return Promises and this would allow us to unwrap any number of levels of those, which you probably won't see in any code initially, but it could be something you get into in advanced scenarios. That doesn't change anything other than basically our public facing function just adds these security checks and then they delegate off to the respective internal function. When we move that, we do need to rename our variables here. Let's stick with value in the case of resolve because we don't know if we actually get a result. Let's run our test just to make sure everything is okay and all of our tests are passing.

## Defining Resolved

I want to take a minute and talk about the usage of the word complete here. Before I do that, one thing that's bothering me is that we're going to talk a lot about reject going forward in this course, so let's flip this over to reject and instead let's patch in and have fail be the alias that we define. So I'm just flipping that around so that by default we talk about the reject function and I'm actually going to yank this out and get it out of site for now. Let's just put it way down here at the bottom. Okay, so now we can talk about resolve and reject. So we have this check in here and we use the word complete right now to guard from calling multiple times into resolve and to reject. And complete was a good word here when originally we just had succeed and fail methods that immediately succeeded or failed the Promise, but now we have resolve and resolve will unwrap a Promise so this is really not succeeding immediately anymore. Reject still works immediately, but resolve doesn't. So even though we've called resolve, it still may be a while before our Promise transitions from pending to either fulfilled or rejected, depending on that other Promise that we're now locked into and by the way, lock-in is what's referred to when a Promise is used to resolve another Promise. It just so happens that in the ES6 spec, this case where we've called resolve, but the Promise is still pending another Promise to complete, well actually the word resolved is used to indicate that state, so it's almost a transitionary state where the Promise is still possibly pending, but this state also represents once the Promise settles, so once it actually has a value. So that's nice because we can use resolved now instead of the word complete. We use that for our guard because that's actually a piece of parlance from the ES6 spec. You can see here in the spec, resolved means that the Promise is either settled, so it is rejected or it is fulfilled, or it's locked in to match the result of another Promise and here's the clause that specifies that we want to prevent a second call to resolve or to reject and ignore those. So now our code matches the specification. Keep an eye out for the word resolved. It might take a little while to remember that resolved represents either a pending Promise that's locked in or a settled Promise. Let's just make sure we're all on the same page here when it comes to understanding what I mean by resolved. So let's go back to having a Promise that we just created and it's in the pending state. We can refer to that as unresolved. At some point a call to reject could lead to this Promise being rejected and as before, a call to succeed could succeed this Promise, but now we do that with resolve. So a call to resolve, passing something that's not a thenable, not a Promise, will result in a fulfilled Promise then, and keep in mind, both of these two states, fulfilled and rejected are referred to as settled, or the easier way to think of that is just not pending. So once a Promise is not pending, obviously it's in one of these two states. The tricky part here is when we resolve now and we resolve with a Promise. In this case our current Promise locks into the Promise we resolved with, so if we fetch the current city and then we want to create another Promise that clones whatever fetchCurrentCity comes up with, then that fetchClone locks into fetchCurrentCity. This lock-in scenario is really simple. The key is that the current Promise will either be rejected or fulfilled based on the result of that locked-in Promise. Now unfortunately, one thing I can't illustrate is that this lock-in process is recursive. So we could have one Promise that locks into the result of another Promise that locks into the result of another Promise, on and on and on and on and on. Now that's highly unlikely, but Promises are built to support that. Now the key additional piece of information that I want to make sure we understand is that everything on the right-hand side here that I've thrown this box around, is in a state that we can also refer to as resolved. The second that either resolve or reject is called, the Promise transitions into a resolved state where any more calls to resolve and reject will just be ignored and the resolve state, of course, entails both locked-in and settled states. This resolve state means that the value of a Promise is set in stone. It may not be known explicitly, but the dependency to another Promise is known or the value is known.

## Executor - the Revealing Module Pattern

Let's talk briefly about how we've been creating new Promises. So we've been newing up our operation type and then some of our tests we've just directly been resolving or rejecting the Promise. So we've been relying on these methods being publicly accessible right on our Promise. If you take a look at the documentation though out on MDN of Promises, this will mirror what's a part of the ES6 spec and that's that there's no method called resolve and there's no method called reject that's available on the Promise prototype. There are global methods available; we'll talk about those in a little bit. But there is no method available on a Promise that you create using ES6 Promises. Let's take a look at this in the browser. So I'll follow along and I'll use this syntax here to create a Promise. Let's actually just copy this and paste this and then we can get rid of the … . So that will create a Promise. Let's capture this in a variable. So now we have this variable p and if I look for completion here, there is no resolve method and there is no reject method, so how do I complete this? Well, as you can see in the example text above, when we create a new Promise we also pass a function to it. That function is called an executor. That function gives us access to the resolve and reject methods just for the scope of that function. So resolve and reject are rather protected in the ES6 spec. You won't find this in all Promise frameworks. In most Promise frameworks you'll find access to these two methods, but in the ES6 spec they're protected. The only time you'll have access is inside the scope of this function that you pass when you create a new Promise. So, for example, I could come up here and let's make a new variable a. I could come up here and inside of this function I could immediately resolve with the value of 1. Now take note that p up above, if I access it, it's in a pending state. Now let's access a, but before I do that, you tell me, what do you think the status of a will be, and also notice the value of p is undefined; what do you think the value of a will be? So a's status is resolved and its value is 1. If I want to create a Promise that's immediately rejected, I can call reject instead and of course, it would help if I didn't redefine a. So let's do this. Now this is interesting that we get an uncaught exception from rejecting this Promise. We'll talk about that later on, but let's just look at a again. You can see it's in the rejected state with a string value of 1. So if we want to resolve and reject our Promises that we're creating for test cases, we have to do it inside of a function that we pass to the Promise constructor. Let's get this functionality mirrored in our implementation so that you can see this in action. So right now, all the tests are passing. That's good. So in our operation type, what we'd like to be able to do is pass a function here, an executor and that function, when it's invoked will receive the secretive resolve function and reject function, just for this Promise. So this is dependency injection injecting in these dependencies to resolve and reject the Promise we're creating. Now inside of the scope of this function we have access to these, so if we need a Promise that's resolved right away, we can call resolve right away and we can pass the value. So this is the equivalent setup using the ES6 format of creating a Promise. By the way, if the word operation is bugging you at all, you can alias that or rename that. It's not bothering me so I'll stick with it. Okay, so let's save all this again and we should have one test case that doesn't work and that's the case we're working on right now. Let's go ahead and copy this first though and we can move this down to this second case below and then in this case we want to reject the Promise right away, so let's copy this line, paste this in right here. Now we have two cases, one where we're using the executor to reject a Promise and another to resolve it right away and both of these are synchronous resolution of this Promise. They're happening right away. We'll talk about asynchronous in a minute. So let's scroll up and implement this. So when we call our Promise constructor, we'll accept an executor function and we'll make this optional at this point in time; in the ES6 spec it's required and then I'm going to scroll down to the bottom here and I'm going to write some code to work with this executor function. I don't know why I want to put this here, but I just want to put this right here at the bottom. It's like the last thing that happens before everything else is created before it. I will take a moment and remind you, if you'd like a challenge, you don't need me to implement this. Pause the video and implement this yourself. If you're not working through this though, you can just follow along with me and not pause the video. Okay, right here at the end, let's go ahead and invoke our executor function and then we need to pass resolve and reject to it and let's scroll up and see what we have to pass along here. We have this reject method on our Promise type and we have the resolve method on our Promise type and since we're using this reference to operation everywhere and not using the this keyword, we're okay at this point just to pass things using operation and then .resolve and then operation and .reject. So we're just passing these methods along as functions and again, we're not using this anywhere so we shouldn't have any binding issues, so we can pass things this way. If you are not using this referenced operation for whatever reason, make sure you deal with the binding of this. Okay, this will work though so we're passing along what should be private functions that should never be exposed to the world; we are going to leave these exposed to the world for now simply because there's no value in rewriting all of our tests. If you want to make yours private, we've already done this before. We have two internal versions of this function that bypass the resolve check. These are already private; nobody can access these, so you could use this same pattern if you wanted to make these private as well and then go update all the tests. Okay, so what's going to happen now, in our test case we're using this, our Promise is constructed and then right at the end of the executor function is synchronously invoked, just like we had called whatever function that had a callback to begin with. In these two tests we synchronously resolve our Promises. Let's go ahead and save this though and see if these two test cases work now. And it looks like everything blew up and that's because we also need to add here a quick check just to make sure that we have an executor function before we try and invoke it. For now we could just assume that truthy means we have a function here, if somebody is naughty and passes an object or something that's not a function, well, they can deal with the problem for right now. There we go. When we save that, now all of our tests are working. By the way, this idea of passing a function in and then allowing our Promise type to inject dependencies, this pattern is referred to as the revealing module pattern.

## Deferred Versus Thenable

This executor function is getting to something really important when it comes to understanding Promise frameworks and that's that there's this line in the sand in most Promise frameworks separating the operations on the left-hand side, resolve and reject, from the operations on the right-hand side, then and catch; and just remember that catch is an alias to then. So in most Promise frameworks, the stuff on the left-hand side is grouped into what's often referred to as a deferred and on the right-hand side the functionality is grouped into what's called a thenable. Now a Promise encapsulates all this functionality. This separation is about limiting access to who can see what. It's really ultimately about keeping the control of the Promise, which is on the left-hand side; in other words to dictate the outcome of the Promise. It's really about keeping that separate from consuming, so it's separating producer and consumer is another way to look at this and so that's why the executor function is the only function that can have access to the resolve and reject methods. We wouldn't want a consumer dictating the fate of a Promise. That could mess up the entire operation. Now as you can see in our Promise implementation from earlier in this course, we can make these resolve and reject methods accessible publicly, so it's not like that's impossible to do; it's just something you won't often see and that's why in Chrome we couldn't access resolve and reject on the Promise we created, but once we created that Promise object we could access then.

## Executor with Callbacks

Let's take a look at what that executor function looks like when we have an asynchronous operation, which is really what we intend with Promises. Let's scroll way up to the top here and let's take a look at some of our fetch functions we created earlier in the course. So now we pass some executor. They will receive the resolve and reject methods and then this function will be invoked right away so we can move this code up here to call the function that took a callback and then let's just get rid of the noCallback for now and inline that. This will make more sense if you see this. So I'm going to grab this here and I want to bring this up so you can see this inlined. Okay, so this is what this looks like with an asynchronous operation, in this case wrapping a function that takes a callback. Now you probably want to use this noCallback style, but for learning purposes, it's easier to see it inline and then now instead of using reject, we'll assume that's not available on our Promise and we'll just invoke the respective function that was passed to us. So here's how this works. We create our Promise, we pass our executor. Our executor is invoked right before the Promise is returned to us, so it's invoked synchronously. Our executor has access to resolve and reject, but this is the only place we have access to this. As our executor is being called before the Promise comes back, we go ahead and call getCurrentCity, which takes a callback, which we've created here and that callback will be invoked once the operation completes, so that will happen later on. So later on then, this callback will be executed and if there's an error we'll reject the Promise, otherwise we'll resolve the Promise with the result. Notice conveniently, if for whatever reason this getCurrentCity function returned a Promise, we could deal with that now with the resolve call. So as you can see, we can do whatever we want in this executor function and once that code is complete, we just have to write the little program somewhere like this callback that passes the result off to our Promise and thus we can collapse all of this and just work with this abstraction of our operation now. Now you could go ahead and implement this with the rest of the fetch functions we have, but I'm going to stop here for now because I want to talk about some other convenience methods that we can use when we're creating Promises.

## Static Resolve and Reject Functions

Now in the process of switching to using this executor function for some of our cases where we need immediately resolved Promises, this is quite verbose just to create a Promise that's rejected and this is quite verbose in the case of succeeding a Promise and that's where some static helper functions come into play with the Promise type in ES6. There's a Promise.reject function and there's a Promise.resolve function and both of these are just shortcuts to remove this verbosity. If you want a challenge, go ahead and pause the video and implement these, otherwise I'm going to just paste in what these look like. Here is what the Promise.reject method looks like loosely speaking. This does not follow all of the ES6 spec, but that's beyond the scope of this course, but basically what we've done here is we've just moved the code here up into here and then we've parameterized the reason, which makes sense so now we can do this instead. We can take our reasoning, yank it out, and then we actually can get rid of all of this code here, and now we can call Operation.reject, pass our reason, clean that up. Okay, so this is what we can do now instead. If we want a rejected Promise, all we have to do is call Operation.reject and pass the reason. Alright, let's take a look at resolve now and here we go. Here is the resolve version. We create a new operation and we just call resolve with the parameterized value. We can now call this from our test just like that, much more straightforward. So these are convenience methods and of course, just to be safe we can make sure all of our tests are still running and they're all passing so we're still okay.

## Testing with Promises and ES6 Promise Backward Compatibility

I've been holding off on showing something for a while now and I'm glad because now I get to kill two birds with one stone. We've been grabbing access to this Done function from Mocha and using that to indicate that our test is complete and making sure that that's invoked only if the right path of execution happens with our Promise operations, so in this case here with the fetchCurrentCityIndecisive, this naughty function tries to resolve a Promise twice. So we use Done there just to make sure that Done wasn't called multiple times because if Done is called multiples times then we know we're not protecting ourselves from doubling up on success. So for example, if I scroll up here into the resolve function, and I comment out our return here and then run the test again, you can see we get protect from doubling up on success; error done is called multiple times. This works, but it's kind of been a hassle. It would be nice if we had some help from our testing framework. Take a minute and think about this. What could our testing framework do for us so that we don't have to call Done; if you want a hint, stick around. The hint for you is that Done is a callback. The other hint is to go look at Mocha's documentation. So it turns out, Mocha is smart. It knows how to work with Promises. Specifically it knows how to work with thenables. Remember that slide I just showed where the right-hand side, the then method is treated separately from the control methods? Well it turns out we can get rid of this Done call here. We don't have to call then that then calls done. We can just create our Promise and we can return it. So just like this, we can create a Promise and pass it back to the Mocha test framework and this is using our operation type, so it's our custom Promise framework that we've built and if we save this, our test will reexecute and we still get the test failure, the same test failure right there, so the Mocha framework registers a success handler and wires up done to it for us. It also registers an error handler and registers done so that it will be invoked with the error if something goes wrong, so we can just pass a Promise back and to prove this works, let's go fix this and let's rerun our tests and now our test is passing. So we can actually return our Promise. Specifically it's the thenable portion of our Promise that matters. Because we return an object back that has a then method on it, Mocha knows that it's a Promise or thinks it's a Promise and of course, it chains up to listen to the result of that Promise to dictate whether or not the test passes or fails. This is pretty sweet. Think about how cool this is for testing things that are asynchronous. Now the second bird that I want to kill with the same stone is talking about this idea of thenables with ES6. There is some additional functionality we have not yet implemented for our Promise framework to play well with other Promise frameworks and if you do some digging into the ES6 specification, which you're going to find out in order for ES6 Promises to work with the history of Promise implementations we have, in other words so we don't have to rewrite all the libraries we have out there, Bluebird, q, RSVP, jQuery, etc., so we don't have to rewrite all those Promise implementations and also so we don't have to rewrite all the code that uses them, the ES6 Promise spec will work with basically any object that has a then method on it, so the ES6 Promise specification can work with all these other Promise types so long as they have a then method and that's all they need. Now that doesn't mean that it can work with everything, but there's a lot of scenarios that it can work with and you're seeing that basically in action in a different context here by being able to pass our own Promise implementation to Mocha. Mocha is doing the same thing the ES6 spec does. It's allowing us to return an object with a then method and it will treat that as a Promise.

## Why Third Party Promise Libraries Are Still Useful

Now that Promises are a part of the ES6 spec, you might be thinking that there's no use for all these different Promise libraries that we've built up historically and while that's true in many scenarios, if you're going to write an app that uses Promises, your best bet is probably just to use native ES6 Promises but there are scenarios where you might want additional functionality that didn't make it into the ES6 specification for a Promise, so this extra functionality that you might desire is something you can still find in Promise libraries; let's take a look at a few examples. Right now we're looking at the API documentation for Bluebird and one of the good examples with the Bluebird library is cancellation. Let's say in our example where we're fetching both the forecast and the weather for a city and we fire off both of those requests in parallel, let's say that one of those fails. It's possible if one of the two fails we don't want to show anything to the user, but an error message. We don't want to do anything with one of the results because we need both of the results to build our user interface. So if one of them fails, it makes sense it might want to cancel the other and not waste precious resources continuing with that request that's no longer needed. Now you'll especially know the value of cancellation if you've come from the .NET Framework because you know that the task abstraction, which is more or less a Promise in C#, comes with cancellation and you may have used this before and so you might desire to have it when you're working with Promises in JavaScript. This cancellation becomes instrumental when you're writing systems that process lots of requests. If you want to bring that system down, you may have a lot of operations that you need to cancel and you may need that cancellation to be cooperative. In other words, you may not want to just shut down a process, but you may want to give all those operations that are executing time to gracefully shut down. So cancellation can be a way then to provide graceful termination of asynchronous operations. Now this is something we don't have with callbacks at this time, though you could find a way to hack it in, but this is definitely something that can be added to the Promise abstraction and can be yet another feature that's provided to you by this Promise framework to enhance working with asynchronous operations. It's possible at some point, a future version of JavaScript will support cancellations in the native Promises as well. There's chatter about adding this to the Promises/A+ spec, which likely means there will be some interest then in maybe getting this into native Promises as well, but for the time being, this would be a great use case for Bluebird.

## Finally

We know that Promises consolidate error handling into this ability to register error handlers with catch. You also probably have experience working with try/catch and then finally blocks. So you might be wondering, well, what if I want some handler to run regardless of success or failure? Of course, you could hack that in to both of the functions you passed or levers like Bluebird often provide a method that allows you to register a finally handler. A handler that will be called regardless of the outcome of the Promise and there's a great example here in the documentation. If you fire off a web request, you might want to have a little loader animation showing while that web request is executing and then when you're done, you might want to hide that loader animation. So here is what that would look like. When you fire off the request, you could register a finally handler and notice this is a part of your Promise pipeline because the finally handler will return a proxyOp just like then and catch, so you can compose this into a Promise pipeline. Once the web request is done, you can go ahead then and hide the loader animation because that loader animation is something that you'll want to hide regardless of success or failure.

## Promisifying Existing Librarires You Love

Bluebird has another helper function that's pretty common to see in Promise libraries and that's a method called Promise.promisify. To understand this, let's hop back to the code we were working on. Inside our code, in three different places we created these fetch functions that transformed our callback based API using the get functions above into a Promise-based API and I'm sure you've seen in all three of these examples we have a lot of duplication. Now keep in mind, in this example here, I rewrote to use the executor function, which is a part of the ES6 spec for creating a Promise. These fetchForecast and fetchWeather functions should look exactly like this fetchCurrentCity function and as you can imagine, this would be a lot of work to type this out if we have APIs that already work with callbacks, APIs that we'd like to use with Promises, but we may not like Promises enough to justify manually converting all the functions and as you can see, there's some redundancy in here, quite a bit of it, and that could all be factored out. In fact, if you want a challenge, write that down on a to-do list after this course is done or pause the video now if you'd like, and factor out this duplication and see if you can create a method that you could just pass getCurrentCity to and it would generate the fetchCurrentCity function for you. Because that method that you would ultimately come up with would be what this Promise.promisify method does. This promisify method will take a function, for example, the readFile function as a part of the Node.js fs API; it will take that function in and it will spit out a new function that uses Promises instead. Now this is done with the assumption that this function you pass in is a Node style function where the callback comes last and that callback function would be using the node convention of the error being passed first and the result being passed second. So if you have Node.js APIs you're already working with that you love, you can convert them rather rapidly with promisify. Now this is a method that's used one-off. There's also a promisifyAll that helps you batch convert methods to help you quickly transform an entire API from one that uses callbacks to one that uses Promises. Now that's one route you can go to get access to Promise-based APIs in existing libraries you love. An alternative to doing the transformation yourself is to just look and see if somebody has already done it and published a package with a promisify version of the library. For example, if you search for the mz package out on npmjs.com, you'll find a whole suite of conversions of the Node.js core APIs from using callbacks to using Promises, so it's possible somebody has already converted a library for you and published a subsequent package. My only warning here is to make sure if you're going to use this that this library is being maintained. You don't want to be stuck with a promisified, but outdated API because nobody is making updates as new versions of the callback-based API is released. So really what you want to do is look for and maybe help contribute to converting callback-based APIs into Promise-based APIs in the actual libraries you're using themselves. A great example of what you're likely to see with many libraries the MongoDB Node.js driver. This API used to just be callback based and then it was ported to support Promises as well. If you pass a callback then you'll work with the API as if it were a callback-based API. However, if you don't pass a callback then you get a Promise back from this function and you can see that down in the documentation here; Promise if no callback passed, so this function will return a Promise and thus work as a Promise-based API. So watch for this in libraries. Quite a few libraries support both callbacks and Promises because conveniently both APIs don't step on each other's toes.

## Promise Collection Ops

One last set I'll point out are these collection operations that work on collections of Promises. These are extremely powerful and we haven't talked about the two that are provided with the ES6 spec yet, we'll get to those; those are race and all. Long story short on these; all allows you to wait for the result of parallel operations to complete. You can pass to it, three Promises or 10 Promises or 100 and it will give you back a Promise that represents the completion of all those Promises. Once all of them are done, the Promise it gives you will be called, so this gives you the ability to wait for a set of Promises to settle. Race is exactly like all, except it's designed to tell you when the first Promise completes. You might use Promise.all to fetch two different requests in an application. You might send off a request for the weather and for the forecast and then you use all to wait for them both to complete. It gives you the ability to join back together and wait for both to complete and then continue the execution of your program, for example, updating the UI. Race on the other hand, this will be something you would use if you perhaps have multiple servers around the world and for performance reasons you want to fire off requests to each of those servers for the exact same information; you don't care where it comes from, you just want it as fast as possible, so once you have one response back you can use that for the rest of your program execution and you can just ignore getting the other responses back, so race helps you do that. It's like a race to see which Promise finishes first. We'll have some examples of using Promise.all later in this course. The rest of these though, these are added and these are functional operations. If you're used to functional programming, you'll know what many of these do in a synchronous context, just imagine these working in an asynchronous context and being just as awesome with async ops as they are with sync ops.

## Unhandled Rejection

It's sage advice in life when somebody rejects you or you're turned down for something just to ignore it and move on with your life, move on to the next opportunity; don't get held up with somebody turning you down. However, when it comes to errors with Promises, which are referred to as unhandled rejection, there can be catastrophic consequences for ignoring errors in Promises. Just like an exception can bubble up to the top of your application and likely lead to the termination of your application, unhandled rejections can cause problems as well. They're a bit tricky though because the nature of working with errors and Promises is a bit different than thrown errors. Let's take a look at why this is. So I've pulled up one of the tests we have here and you could do this with any of the tests, but I've pulled up this sync result transformation test because it has two success handlers and it doesn't have any error handlers. So this is a situation where we can get into trouble. For example, just take the assertion here and change the value and then go ahead and save the tests so they run again and you'll see there's a timeout on this test now. You've probably run into this before. We've run into this a few times. We saw how we can use catch handlers to deal with exceptions. I even pointed out how assertions throw exceptions and thus can cause problems. Let's glue all these pieces together now. The trouble here is that we get a timeout as the reason that our test failed and that just doesn't make any sense. If this happened as we were refactoring our Promise framework, we'd be pretty confused because if we look at the code here, we have a call to done so why in the world is this timing out? So this timeout is almost useless, except that it's telling us that somehow done isn't called, but why is that the case? Take a minute and pause the video and just think about this. Postulate why this might be failing. Maybe even dig into it if you want and see if you can get at the underlying reason why this is failing and by that I mean see if you can get the test runner to report the error. The real error is coming from our assertion because we modified that so the expected value is not matching the actual value, but we don't get that over here and that's because the assertion here, the toBe check, will throw an exception that'll be caught by our Promise framework, but then we don't have any catch handlers to do anything with it, so we've just met the two conditions necessary to have an unhandled rejection. First we have a Promise that's rejected with the exception here, our assertion exception. So we have a rejected Promise, that's the first thing and then the second thing is that Promise doesn't have any reject handlers. When that's the case, you have an unhandled rejection. It's not like the JavaScript engine somehow knows what to do in that situation. Now when it comes to testing frameworks, one way to deal with this is to just not call done explicitly and instead, return a Promise to the Mocha test framework so that it can watch the result of that Promise and then if we save our tests, they'll run again and you'll see we get an assertion error instead of a timeout exception. Now this just happens to be a convenient way when you're working with a testing framework to not have this problem and to be notified to fail your test if you have an unhandled rejection. This is not what you'd do in a real application. What you'd need to do in a real app is make sure you're always registering catch handlers or reject handlers appropriately. That doesn't mean that this won't happen, so you can take a couple extra steps to figure out when it does happen much like when an exception bubbles up to the top of your application, it's usually a good idea to know when that happens, even if it leads to your program being terminated. In the queue Promises library, you'll see a method called done, aptly named I guess because we had that done function inside of Mocha and just like I told you not to use that done function in Mocha, don't use this to try and get around the problem of unhandled rejections. Instead take a look at the environment you're executing in because many environments, for example, the Chrome browser and Node.js, have events that you can listen to that will tell you if you have an unhandled rejection. And for sake of time, I'm just going to point you at links where you can learn more. You can come out to this guide here called Promise Rejection Events Sample. This will tell you how you can handle unhandled rejections inside of Chrome and then in the Node.js API, pull up the process API and then look for the word rejection in here. You'll see there are two events you can subscribe to. First is the unhandled rejection and then second is this rejectionHandled and these two events mirror what you do in the browser, at least inside of Chrome. Chrome has an unhandledRejection event and a rejectionHandled event. So both environments here have these same type of events. Now you'll have to look at the browser you're using to find out the nuances of this because this is not part of the Promise spec. This is part of the environment that the JavaScript engine is running in, trying to help you out. Now the reason there are two events is because detecting unhandled rejection is not simple. Unlike exceptions, with Promises, you could put a reject handler onto a Promise 10 minutes after you create the Promise. Ten minutes in real time you could wire up a reject handler and do something about a rejection, but the browser doesn't know that, nor does Node.js know that. So at the time of a rejected Promise, if there are no rejection handlers, then these events are going to fire off and the first one that fires off is this unhandled rejection and that will be the only one that fires off unless later on you then add a reject handler at which point in time the rejection handler will fire off, so this is kind of like an, oh, we've possibly got a problem, but we don't really know because it could be taken care of in the future, and then this is the oh yeah, that problem we thought we had? Well, it got taken care of. That's why there's two events for this because it's possible that this problem is not actually a problem, a false positive and when these things happen it will at least be nice to know that they happened, so go ahead and log that information, send that to your logging system, especially in a production environment and then how you deal with them? Well, that's got to be up to your app. I can't tell you how you're going to deal with the unhandled rejections you have in your application. Usually you'll want to strive to not have them and get the appropriate reject handlers hooked up.

## Quizzary

Instead of a summary, how about we go through a quick quiz? A quizzary. Are Promises new to JavaScript as of ES6? Surprise! This is a trick question. Yes, Promises were added in ES6, but JavaScript has a history of many, many Promise libraries, so Promises are by all means not new; they're just natively provided in ES6. What are the three Promise states? A Promise is either pending, fulfilled, or rejected. If the terminology of Promises is confusing at all, go back to the operation pneumonic we built throughout this course. Operations start, that's like pending, and then they complete at some point and they either succeed or they fail. Succeed is fulfilling, failing is rejected. What does settled mean? A settled promise is a Promise that's not pending, in other words, it's complete and thus it's either fulfilled or rejected. If you have existing Promise-based code, do you have to rewrite it as of ES6 to take advantage of ES6? The answer is no. You don't have to rewrite existing Promise-based code. The ES6 Promise spec was intentionally built on top of Promises/A+, which has the notion of a thenable to be able to support the ecosystem of Promise libraries and not need to force everybody to only use one implementation of Promises. How do you register an error reaction? First, it's helpful to remember that a reaction is something that's registered to fire off when a Promise settles. So in the case of an error, we're talking about a reject reaction. To register a reject reaction or reject handler, you use the reject function on a Promise. Why is resolved used instead of fulfill? We have this reject function and we use that in the case of a problem, why don't we use fulfill in the case of success? Resolve is used because we can either succeed a Promise or lock it into another Promise. So, what does locked-in mean? Lock-in is what we use when we want to resolve a Promise with the value of another Promise. This is useful in scenarios where we're chaining Promises together, specifically for that proxy operation that we generate when we register fulfillment and rejection reactions. Lock-in also helps us in the case of forwarding errors so that they fall through our success handlers and we can reuse one error handler across multiple asynchronous operations. What does resolved mean? A Promise is resolved once we call resolve or reject. Resolved is important because we can only resolve a Promise one time. This helps avoid issues where a callback might be executed multiple times in callback-based code. This is a layer of protection. Here's a diagram of terminology. Remember that resolved represents everything on the right-hand side here, so what's weird is a resolved Promise can be in any of the Promise states. Just keep in mind, this means the fate of the Promise is set in stone, even if it's a pending Promise that has yet to complete; we know that we're locked into the result of that Promise. What purpose does the executor serve? In the ES6 Promise implementation we pass a function when we're constructing a Promise and that function is called an executor. In the scope of that function we have access to the Promises resolve and reject methods. So this executor gives us the ability to hide access to those functions so that there's only really one person, if you will, in control of the result of that Promise. After all, you wouldn't want you client code resolving a Promise prematurely. That could put your application in a precarious position. Explain the separation between deferred and thenable interfaces. To answer this question I want to go back to this diagram. Remember I drew this line in the sand. I pointed out that lots of Promise libraries refer to this concept of deferred that represents the control over the fate of the Promise, so resolve and reject. So in the ES6 spec, the deferred component is really that executor function we pass in. It's the only thing that has control over the Promise' fate. Everything else interacts with the Promise by subscribing to the final result of that Promise and that part we refer to as the thenable because of the then method that we used to subscribe a reaction. Keep in mind, catch is also an alias to then; catch only provides a reject reaction whereas then can provide both a fulfillment and reject reaction. Another way to look at this separation, defer refers to the producer side of this, in other words, producing the Promise and controlling it and hooking it up to the asynchronous operation. Thenable refers to the consumption, so producer versus consumer. Why are third-party Promise libraries still useful? So we have native Promises now in ES6. You might think with native Promises that we should stop using third-party libraries; why is this not the case? While native Promises work for most situations, if you need functionality that didn't make it into the ES6 spec, you can turn to third-party libraries instead, for example, to have cancellation of asynchronous operations, to convert callback style APIs in Node.js to Promise-based APIs, to use the functional collection methods we saw as a part of Bluebird, or even to get progress notifications, which is something that jQuery has had for a long time now. What is an unhandled rejection? If you can't handle rejection, get a dog, don't get a cat. In your applications, an unhandled rejection happens when a Promise is rejected and there is no handler to handle the rejection. These are a lot like unhandled exceptions; they bubble up to the top of your application because there's no code to deal with them. So that's unhandled rejection. What about rejection handled? Rejection handled is a situation where you have an unhandled rejection that later on is handled because somebody registers a handler for it later on and this is the problem we have with unhandled rejections and Promises unlike exceptions; it's hard to tell when something actually is an unhandled rejection because a handler could be coming. It could even be coming in 100ms, but we don't have it now. Depending on the JavaScript environment that you're running in, there may be events that you can listen for for both of these conditions to happen and then deal appropriately with the problem. A good way to think about rejection handled is that it's a means to find out about false-positive unhandled rejections. When you're writing tests, how can you handle unhandled rejections? How can you find out about them? If you're using Mocha, all you have to do is return the Promise that you're working with and if that Promise that you return has an unhandled rejection, well then Mocha will catch it because Mocha will go ahead and register a catch handler so it won't be an unhandled rejection anymore and then the rest results will come back showing the failure message. Each testing framework is different, though I'm sure most of them work the same way here, allowing you to return a Promise from a test case and then that test framework will hook up to that Promise and wait for it to complete to decide if the test has completed successfully or if it failed.

# Building a Generator-based Control Flow Function

## Promises Aren't Perfect

We've seen many of the benefits of promise, but promise don't come without their own problems. Take for example this test. I've set up an example where we call fetchCurrentCity. We get the city and then we call fetchWeather to get the weather, so this is a set of asynchronous serial dependencies. We have to call fetchWeather after we get the city; we can't call them at the same time and that's because we need the city to fetch the weather. Now we've seen how Promise eliminate the unruly nesting with callbacks and we've seen how Promise provide safety with all the possible types of errors we could get, but they still have added verbosity. They still have extremely explicit seams. This code here with then. The function to wrap around our call to fetchWeather. Ending the call to then. The next call to then. The function here to print out the weather or in this case, run an assertion to make sure the string we would print out is what we expect it to be. If we really think about this, the first thing we're doing is getting the city. Once we have that, then we're getting the weather. Once we have then we're running a simple assertion on building up a string. So this is the bulk of what our code is doing. This is the part we care about. These four lines though turn into eight lines below to pull this off with Promise, but you have to ask yourself, why should we have to write code so differently just because operations are asynchronous instead of synchronous? We are after all the designers of languages so why don't we design languages so that both operations look and feel pretty much the same? Another problem we have with Promise is that it's hard to pass state between different parts of our Promise pipeline, so right now we fetch the current city and then we use that to fetch the weather and then once we get the weather, the only thing we have access to is the weather object. If we wanted to do something like print out the city along with the weather, we can't do this right now. We don't have access to the city that was returned. We'd either have to nest this Promise inside of this stage so that we have access to that city, or we'd have to come and globally declare some variable. Set that variable here, introducing a side effect that can cause problems. A side effect that makes our code brittle, but a side effect that's the only way right now we can pull off getting the city name along with the weather information. If you look above at these four lines of code here, it will be no problem to access the city. That variable in this style would already be available in this entire scope and then the last problem I want to point out, this ability to chain with then and catch and having error fall through, having error recovery, being able to unnest the nesting with callbacks? While all that's really nice, it is rather complicated and I would argue that's because the plumbing for dealing with asynchronous operations is something we have to deal with with promise. In this style above, you don't see any calls to then, you don't see any calls to catch, you don't see any idea about error fall through or error recovery. These concepts are there, they're just implicit. They're behind the scenes. They're not a concern we have to bother ourselves with and it's not hard to imagine that we could create the language constructs to do this, to basically hide away the lines of code down here that are irrelevant and focus on just what matters to us. That's what we're going to look at in this module.

## Iterators

There are some things that you could know coming into this module that will help you out a lot more than not knowing these things. First it would be helpful to have an understanding of ES6 iterables and iterators, and second, it would be helpful to be familiar with generators; however, you do not need to know everything about iterators and generators, so to help you figure out if you know what you need to know, I've got a series of questions I'd like to walk through with you. If you're comfortable answering these or you're comfortable with the answers even if you don't know the answers, then you can proceed with this module. If you're really uncomfortable with the answers though, I'd encourage you to check out some other courses that cover iterators and generators in depth. So let's start out with the very first question I have. Take a look at your desk and see if you can find something on your desk that's iterable. I just took a look around my desk and I have some papers, I have some sticky notepads, and I also have some books. Books are a great example of something that's iterable. There are pages in a book that we can iterate through, thus a book has the characteristic of being iterable. Now for a more technical inclination. Can you name three things in JavaScript that are iterable? Three things you might find in your JavaScript programs? Here are some things I thought of. Files. Our programs often read files. Those files might be an Excel document and Excel document and Excel documents have worksheets. You can iterate through the worksheets and then you can also iterate through the records within a worksheet. So there's a couple of dimensions of iteration inside of an Excel worksheet. You might have an HTML file. HTML files are full of elements, elements that we can iterate through. Or you might have a CSV file. A CSV file contains lines of data that we can iterate through and then within each line of data we have values that we can iterate through that are separated by commas. I also thought of arrays or multidimensional arrays. We can iterate through the elements inside of an array, so arrays are iterable and strings also came to mind. We can iterate through the characters inside of a string. What's an example of an iterator? So this stands in contrast to something that's iterable. Give me an example of an iterator and my hint to you would be to think about the example you have of an iterable that's sitting on your desk. Think about that. What would an iterator represent with regards to your iterable? So in my example of a book, a bookmark represents an iterator. A bookmark holds a pointer to where we're at inside the contents of the pages in a book. If we open up an Excel worksheet there's usually a row color difference for the current row that we're looking at. That row is highlighted. That highlighting is an iterator. To be able to go through something, like the pages of a book requires keeping track of where we are. That iterator contains that pointer to where we're at. What's the relationship between an iterable and an iterator? This is somewhat a trick question. Here's what I thought of. So a book is iterable and an iterator like a bookmark keep track of where we're at so it's also possible that we could have multiple bookmarks keeping track of where different people are at in a book, so there can be many iterators per iterable. Forgive my example of a bible, it was the only stock photography they had where a book had multiple bookmarks. That's definitely not meant to offend anybody. In JavaScript iterators have a next method. Using the item on your desk, describe what you think this next method does. In my case I had a book on my desk, so the next method in my case would turn the page, which of course in turning the page gives me access to see the contents of that page. In the case of an Excel worksheet, the down or up arrow are akin to this next method and they move up or down a record, highlighting the context of the next record or previous record, though in ES6 iterators there is no concept of previous, there's only next. So next does two things. It moves to the next element that's being iterated through and it returns the element. Imagine you create an iterator like this. So let's say this is how we can create an iterator to iterate through the elements in our array. We get back the iterator and call it numbers. Then I have a question for you. What does next return? Just describe it and describe it not just for one call, but describe it for three separate calls. What will each of these print out? This is what the output would look like from calling next on our numbers iterator. The very first call to next would log an object that has a property called value and that property value has a value of 1. We also get another property called done and the value is set to false. So if you need to, take a minute and think again about what you think the next two calls will do. When we call next again, it's like we flip to the next page, which takes us to our next number with a value of 2 then and done is also false and then the next time we return the value 3 and done is also false. So now knowing that, what do you think this does? In this case we'll get back an object. The value will be undefined and done will be true. You don't really need to know how to create iterators for this course, but you do need to understand how to use them and I hope this gives you an idea of how to work with an iterator. I hope you can see it's just a construct to formalize moving through a set of elements, thus the word iterate. What we looked at was a rather verbose abstraction, but along with this concept of an iterator we have a new type of for loop in ES6 and I want you to take a stab at what you think this will print out. So we have an array that's iterable. Note we're not creating an iterator, we have an iterable and we're passing that into our for of loop. Notice the word of is new; it's not for in and it's not a for loop that increments a number, it's a for of loop. So what do you think this will print out? Here's what we would get. We'd just get the values 1, 2, and 3. No objects with value and done properties, just the values in the iterable. So what we saw in the previous slide was the plumbing for moving through the elements for iterating and for keeping track of where we're at in that iteration. For of loops hide all those details, just take an iterable and behind the scenes create the iterator for us and unwrap the values as they come back and in this case, put them into that number variable. So that's it for iterators and iterables. Just a new means of moving through a set of elements, that's it.

## Generators

Now let's talk about generators. So what's the first word you think of when I say the word generator? Here's a couple that came to mind for me. A random number generator, an electrical power generator, and ideas. These are all things that we can generate. Generate is the production of something. For example, as you're working through these exercises with me about iterators and generators, if you haven't seen these concepts before and you're answering the questions in a way that you find comfortable then you are actually generating learning. That's generative learning. So if you had to take a guess, what would be the relationship between generators and iterators? It turns out that generators are a way of creating iterators. It makes sense. If a generator generates values then we have something we can iterate over, like a random number generator would generate an infinite sequence potentially of numbers and we could iterate through that to grab a few random numbers. Same thing with a power generator. We could plug into it to consume power. So this is a generator function. Take note of the asterisk at the end of the word function and also the new keyword yield. Think through what you think this function will do and when you're ready, tell me what you think this code here does; what do you think will be printed out? Here is what we'll get. Look familiar? The very first line of code here creates a variable called generator, which holds an instance of a generator that's created by invoking the generator function. In many ways this is like getting out a bookmark before we start reading a book. We're now ready to read the book or in this case, iterate through the numbers provided by the generator function. So when we call console.log here and pass generator.next, you can see that generator has returned an instance of something that we can iterate through. We're calling that next method to get a value and the very first value returned is from that yield expression to yield the value 1 and we also get done as false. Here's that generator function again. Now what does this print out? Starting over. So we're creating a new generator and we're using our for of loop on that generator. What do you think will print out here? We'll get the values 1, 2, and 3, just like when we were iterating over an array that has values 1, 2, and 3. So as you can see, these yield expressions inside of our generator function yield values as we iterate and we get three lines here, one for each value, so as we iterate with this for of loop, our function numbers is returning multiple values, one at a time.

## Generators Are Suspendable and Resumable Functions

I started out with that analogy of a book as something that's iterable. Let's work through a real example of some code that uses this abstract notion of iterating through a book, but we're going to do that with a generator to create our iterator and hopefully in the process you can start to see some of the additional features of generators that we'll want to take advantage of. So I've got this generator function here called reader. You can imagine this is a function that helps somebody read a book so it needs to take in that book. The very first thing it does is create a bookmark and starts on the very first page of the book and then while the bookmark is less than the number of pages in the book, in other words until we're at the end of the book, then we go ahead and log out the bookmark, we get the contents of the page that the bookmark represents, so in this case we get page 1. We yield that page and then in yielding, the iterator we create pauses for that page to be consumed, just like once you turn the page in a book, you would then read through the page; your bookmark is still there holding your place. When you're ready to go to the next page, you turn the page and put the bookmark in the second page and so that's what we have here after. Once someone is done reading that page then eventually the code here will resume, the bookmark will be incremented, and then our while loop will kick in and we'll come back and do the same thing. We'll print out the bookmark with the page we're on. We'll get the next page, page 2 in this case, and we'll yield the contents of that page and then we'll stop, allowing the consuming code to read that page. Now let's say that we have a function called buy that more or less just creates a book object. I'll collapse that because it's really not important. We use that buy function to buy a book and that's our book. When we are ready to read that book we can then create a reader and pass our book to it and now we have our myReader. Now remember as I said before, at this point in time, it's like we have a bookmark and a book, but we don't have that bookmark inside the book yet because we have not yet opened up the book to read it. You can imagine right now the bookmark is just sitting on top of the book, which is sitting on top of our desk waiting for us to read it. Eventually we open up that book and start reading it. To do that we make a call to the next method on the reader. Take a guess of what you think will be printed out when we run this program and to help you out, I will show you the buy function so that you can see what this getPage method does. I collapsed the font down for you there so you can see all of the example. So what will happen when I push the Run button? So you can see here in the output when I push the Run button, the value 1 was printed out and that comes from this console.log statement here to log the bookmark out. So in calling next we started to execute this code right here and we executed all the way up to the yield statement and then we paused the execution of this function, after which the page comes out of the next method we called then and gets passed to console.log here and you can see the output of it. It's an object with a value of page 1 and done is set to false. The value is page 1 by the way because that's what this getPage function does. You can imagine this would be the text of the page so that we could read it. Eventually we get through the first page and we're ready to move on to the next. So I'm going to clear the output here. I'm going to paste this in. I'm going to close up the buy function just so we have a little more space here. Tell me what the output will look like here. Aside from the event logs that we get from Plunker here, what does a console.output contain when we run this code? I'm going to pull this over to the side here and make this small. Here's the output. Is this what you thought it would be? First we get 1 and the same page 1 back that we saw before. Nothing has changed there. When we make the second call to next we then restart this reader function, we restart the generator. It hits this code to bump up the page number to 2, cycles back around in the loop here, logs out 2, so that's why we get 2 next. It grabs the contents of the second page now and yields that back and then this function pauses again and now the value of that second page comes back from the call to next and flows into the console.log method here and that's why we get the second object printed out in our output with the value of page 2 and done is also false. This should be pretty trippy at this point. We're not used to functions being able to return multiple results nor are we used to this idea that functions can pause themselves and that we can then resume those functions later on. Yield is pausing this function above, next resumes that function. It's a form of cooperative concurrency. We have two components that are working together. We have the reader that we've created and then we're using that reader to move through the pages of the book. It's like the book gives us back a page, we read it, and then we ask the book for the next page. It gives us the page, we read it, and then it gives us the next one. It will keep doing this until it runs out of pages. We'll talk more about the consequences of this, but let's work through some more examples.

## When Resuming a Generator We Can Pass Errors

There are two important things we've seen thus far with generators that I just want to take a moment and point out again. First off, the yield keyword inside of a generator allows us to pass information back, in this case passing back the page contents to be logged out and we do that in several places, every time we call next, but the other important part is when we call next we can pass data into the generator, for example, to turn 10 pages and then to turn 100 pages, so there is a two-way messaging system here with a consumer of the generator, us reading the book and the reader itself, which helps us keep track of where we're at in the book. But there are more ways that we can communicate with this reader. Instead of next here, we can call return and that will immediately terminate the generator. This would be like throwing away the bookmark. Think about what this will produce when we run this. Well, returning is like going way past the end of the book so our generator is now terminated. But the really interesting one and let me stop this, we can also throw in or pass in an error. What do you think will happen when I run this? Is this what you expected? Here's the output and specifically here's the error that was thrown and if we dig into this, you can see a couple things. First, this came from script.js line 7. Here's line 7 right here. So even though we created this error right here, it was thrown up here inside of the generator. You can see that in the stack trace. It's coming from inside of our reader function. If you haven't done it already, I want you to stop right now and even if you have an only rudimentary understanding of generators and iterators right now, I want you to think back to our Promise framework. What in the world do these two concepts have in common? Our Promise framework and generators? Where's the common ground?

## Calendars

Generators are yet another type of asynchronous seam in our applications and the seam is much more subtle, so much so that maybe you haven't noticed this yet, but this yield keyword is like saying, hey, I've done as much as I can right now, here's where I'm at; let me know when you want me to pick up where I left off. That's a lot like making a web request. We might make a web request and then we say hey, I can't do anything until that web request comes back and here's the code to execute once the web request comes back. That callback is the same thing as resuming a generator function, except the callback is a much more explicit verbose seam. Promise are the same way. That chain to then is saying, hey, when everything is ready then go ahead and pick up where we left off. This last weekend when I was getting ready to go to bed, I had a thought about something that I had forgotten to do in the previous week. I had a contract that I needed to do that I had been putting off. Of course, it was bedtime so I wasn't going to do anything about it then so I went to my calendar and I wrote down for Monday to start that contract and then I went to bed. On Monday when I started work, I took a look at my calendar and eventually I got to this item to start the contract. So I started working on the contract, but as I was working on the contract, I found out that there was a piece of information that I needed that I didn't have, so I fired off an email asking for that piece of information from a colleague and then since I couldn't do anything else with the contract, I scheduled myself a reminder to make sure that I got a response on the ID that I needed and I scheduled that for tomorrow, so I was giving a day for the person to reply back to me. After I added that reminder, I went about my day. Tuesday came around and by the end of the day I remembered I needed to check my email and I had got a response in my email, but I didn't have any more time to finish up the contract, even though I had the response, so I scribbled down an entry for Wednesday to fill in the ID that I got and to send the document to get signed by Dan, a colleague. Wednesday rolled around. I saw this on my calendar. I grabbed the document and finished it and sent it off to Dan and of course at that point I'd done everything I could so I put a reminder to check up with Dan on Friday just to make sure that he signed the contract and got it back to me. Friday rolled around, I had the contract back from Dan, so everything was good to go and the contract was now complete. When we are filling out contracts we have a process that we follow, but it's not often a process that we can get done in one sitting. We oftentimes have to work on it over the course of several days as we get information and signatures from other people. So there are points where we have to pause and resume this process and there's a natural separation then between the role that the calendar plays and the role that we play in doing the work. The calendar in this situation reminds me to resume my work on this contract. It helps me make sure I don't forget about it again. It helps me make sure that I get the next piece of information I need. I'm the worker so I'm the person doing the work, but there are points in the process where I can't do anything else at which point in time I suspend what I'm doing and put a new reminder on the calendar. This workflow is cooperative. It requires both a calendar that I look at, and the diligent scheduling of appropriate reminders. Calendars in this example are like generators and we're like iterators that can work through the contents of a calendar and then add new items to that calendar dynamically whenever we need to take a break. So one last time before I give it away. What do you think this has to do with Promises?

## Debugging Generators

Quick quiz. Tell me what the output of this program will be? Let's go ahead and run this to find out. I'll pull up in the dev tools. We'll stop this, clear it out and restart it. Was this what you expected? This is a little more basic of an example of working with generators. I want to walk you through debugging this generator here to make sure you understand what I mean by a suspendable or pausable function that you can then later on resume. That's something that we're not used to unless you've worked with generators already. It's not a complicated concept, but it does take your mind a little bit to wrap around this idea because we're just not used to it; we're not familiar with it. Once you become familiar with it, it'll make sense. One way that helps me is to think of a generator as the composition of several smaller functions, so this code right here would be like one function, this would be another, and then this would be the last little function in here, so three little functions if you will. Kind of like I had little programs with asynchronicity. Let's debug this by throwing in a debugger statement. Our code will reexecute. If not, go ahead and run it and you'll notice we're paused in the debugger and we're paused on the line we added the debugger statement to. So let's take a look at the order of execution here. I zoomed in a little for you. I'll actually make this a little bit bigger and let's see if we can pull up here so we get a little more details down below. Okay, a couple things to watch out for. Watch out for the call stack and then also watch the lines that we're hitting and what order they hit in. Right now we're here in script.js on line 8. Let's go ahead and step over that line. And you can see we're now on line 9. Here is where we create the generator. Take a minute, pause the video if you need to and think about how you think this will play out here as we step through this code. Okay, let's step in to creating our generator. What does that look like? You can see immediately we hop to the end of the numbers from function and that represents the fact that when we create a generator we have not yet executed any of the code inside. We've just created the generator and at this point you can see the scope has a number value with the value of 3, which is what I passed into the generator function when I invoked it. Okay, stepping onto the next execution, you see then we move on to our next line of code here, line 10 of script.js. So in evaluating line 10, what's going to happen next? Well, we're going to need to call generator.next. So if we step into the function call, you can see we step inside of our generator function. You can see we hit our first log statement. The number values of threes that will be logged. If I step into that, well we're in some other code so let's step out of that and then let's go ahead and step into our yield and see where that takes us and at this point we're going to return back the value 3 and we're also going to increment the value of number to 4. If you check out the console, you'll see 3 was logged here. This is the previous run up here, you can ignore that. Okay, so we just yielded 3. Step into that and you can see we come back to our console.log statement and now you can see that console.log will be invoked with that value 3 that came back. Notice at the call stack here we're on line 10, so we're back to line 10. We can go ahead and step over the console log. We don't need to get into the log code. If we look at the console, you'll see the iteration result here, a value of 3 and done false. Okay, so now we call again the exact same code. So first we're going to need to step into this, which will take us back into our generator and notice we pick up where we left off. The number was incremented with 4 before we left the last time so we add that value now, so we're going to log that out, so let's step over that. Notice that line 11 is at the base of the call stack and that right now we're on line 5. This is as if we had just invoked a regular function here. We can step over the yield and notice we'll yield back and get back to line 11 and now we have that value 4 to log out and when we were yielding, once we were done, the value was incremented to 5. Take a look at the console and you'll see we now have 4 printed out and when I step over this line here, you'll see we get the iteration result with that value as well. Alright we have one more call here. Notice we're on line 12 right now. Let's go ahead and step into this. We can step over the log here that will log out end. Now our generator is complete so when we step over that we go back to line 12 and this time we didn't yield anything so when we step over this, you'll notice we just log out an object with a value of undefined because nothing was yielded at the end there and done is now true. So as you can see, this numbersFrom function is pausable. It decides when to pause itself by calling yield and then we decide when to resume it by calling next. Now let's add in Promises.

## I Would Like to Yield Promises

I put together an example here of making the weather request we were working on earlier in this app, but I cut out a lot of the nonsense and I'm just sticking with making the request and then logging it out when we're done here. More than one time I've brought this up. In this code sample, there's a lot of added verbosity just to deal with the asynchronous nature of this code. Verbosity that I personally like to get rid of. The only real value here is when we fetch the current weather, so when we initiate the web request, when we transform the result or the response to JSON, and then last when we go ahead and print out the weather information. This is the code we're writing. The rest of this, it's all plumbing, it's all ceremonial. Like when I was filling out that legal contract? I care about fetching the weather, but as soon as send off the request like sending an email, there's nothing I can do about it. I shouldn't have to be bothered with that. I should just get a reminder when I'm ready to perform the next step, which in this example would be converting the response to JSON, so then I fire off that request, but then again, there's nothing I can do so I'd like to be free again and let something else deal with that for me and just notify me when I get an object back that represents the weather and then I can go ahead and print it out. So what I'm saying is, in between these red circles I'd like to yield, so in both of those spots I'd like to yield and let someone else remind me or resume my execution when the next piece is ready. Well, guess what we have that represents and asynchronous operation that we can pass around? We have a promise. We get a Promise back. We can yield the Promise that we get from calling fetch. Something else then could wait for that Promise to settle and then resume us once the Promise is settled. So something else could do the then for us so that we don't have to and it could do that for both of the Promises we have here, the one for fetching the weather and the one for parsing the JSON response. Let's take a look at that in the next clip.

## Yielding a Promise to My Assistant

You can imagine we could add some separation here in our program and then let's say we have some function called assistant and let's just assume that this assistant function will help us out and do the nitty-gritty, not-so-fun things that we don't want to have to deal with. In other words, we'll take care of the calls to then and we'll take care of resuming us. All we need to do is fire off an async operation, yield the Promise, and then wipe our hands clean of it. Let's see what this would look like then. Let's comment this out for reference and let's just start with some code and then let's make a generator function so we need the asterisk called me which will represent me. I want to do as little work as possible. The first thing I do need to do is fetch the weather. So I need to fire off that request. Of course, this is asynchronous so I don't know when it will complete so I want to put my feet up and kick back, which means I'm going to yield the promise that we get back from the fetch API, just like the promise we built with our Promise framework. This example does happen to be using the native ES6 Promise. Okay, now to glue this together, I'm going to come down to where the assistant is at. I'll need to create a me generator and then I can call assistant and pass the me generator. Just basically asking the assistant to take care of me. Alright, we have our me generator, but that has done nothing other than create the generator. What do I need to type into the assistant to have the assistant tell me to get started with my work? I need to call generator's next method. When that's invoked, the code will execute here to initiate the weather request and then yield back the Promise, so that means I can capture that result here. Remember it's that iterator result object and let's just call that next and to see what that is, let's go ahead and log that out. Let's get this code next to each other so let's yank this out and let's put this down here for reference. Okay, so that will go ahead and print out the iterator result and on that iterator result should be a Promise, so there should be a promise if I access next.value, which means I can take the Promise and wait for it to settle by making a call to then and registering a fulfillment handler. So in this case we get the result. I'll speak generically about this. We take that result and what do we need to do next? Well at this point it doesn't really matter, but if we would like to have access to the response in here, we need to accept the result of the yield expression, which means we can resume the me generator by calling generator.next and we can pass in the result which will give us access back to the result right here and just to see that in action, let's go ahead and log this out as well. Let's pull up in the debugger tools. Clear this out and run this. Look at that. We have two log entries. First we have the iterator result, the next object here and that has a value with a Promise on it and done is set to false because that generator is not yet done and then up here in the generator, when we log out the response that we get back, you can see we have a 200 ok on our web request, so we have the response back. So take a minute and think about what we just created. Focus specifically on just this block of code and actually hone in on just these two lines of code because the rest of the stuff down here is something we could sweep away under a rug. We could put it into a library. We could get something set up so this is the only code we have to write here. Join me in the next clip where we expand on this.

## Assisting with Infinite Promises

So this is pretty cool, but we have an additional step that's an asynchronous operation transforming the response into an object. Let's go ahead and do that now. So we have some more work we need to do here before we can call it a day. We need to take the response and convert it and of course, this returns a Promise as well so we want to yield that so that our assistant can take care of figuring out when it's done. All we care about is getting back the response. In this case that will be the weather object. But right now our assistant can't handle more than one asynchronous operation so we need to update our assistant. You could imagine once we get this last weather object back we can go ahead and log that out then, which was our ultimate point. Alright, so let's come down to the assistant here and let's update this so that we can handle more than one Promise, more than one asynchronous operation. We're going to need to run the same sequence of operations again to deal with the next promise that comes through, though with one small caveat and that's that we already have some repetition in here. We already called generator.next here, but we don't capture the iteration result so we actually miss out on the second promise here. This you'll notice, this call right here, is duplicative of this call right here. So what we could do is create a function inside of here to be able to repeat this same process and let's just call this remind. So each time our assistant needs to remind us, this code will be executed. Then what we can do, into remind we can pass the previous result if there was a result that we were waiting for. So whatever we're waiting for, we can go ahead and pass that when we call next here. Then we don't need to do this right here. All we need to do right here is call remind and pass the result, because that result is what we're waiting for so then that'll be passed to us with next here and then we'll get back whatever comes back next and we can continue the process indefinitely. Now the only thing we need to do is make sure we call remind the first time here and we won't pass anything because there's nothing to wait for the very first time we kick off our me generator. Now if we pull up the output here, you'll see we're getting a bunch of objects. Let's just clear this out, stop and start this. So let's walk through what this code does. When our program fires off, we create a me generator. We pass that generator to our assistant. Our assistant then calls the remind function and the first thing that remind function does is use the generator we passed in to call its next method. That means now it's like we invoked a function here, we're now inside of our generator and we invoke the very first part of this code which is just more or less the open curly brace through to fetching the weather request, which is initiating the weather request. This gives back a promise, which we yield here. I'm going to clear the arrows off here. So that promise we yield comes down here as the result of calling generator.next. It's like the function ended and we returned a promise. We then go ahead and log out next, which is an iteration result so it's not just the promise, it's also the done property. You can see that over here, the very first log entry, and then you can see we access the value property here to get back the promise. We then hook up a fulfillment reaction to that promise by calling then, and at this point the assistant has done the initial work. The assistant will be called back when the result is ready. Let's look at that now. So eventually, the weather request will complete and a response will come back and that will come in through the success handler here. So this fulfillment handler will call remind, passing the result that came back. So the result will come up here, it's what we're waiting for, and now we'll repeat the process, except now we pass waitingFor in and let me clear this out, this will make more sense. Now we pass waitingFor in when we call next, which means waitingFor is now up in our response variable up in the generator and waitingFor in this case was the weather response. We use that response to log it out and then we also use that response and we start up another async op here to convert the response as JSON into an object. Of course, we're done with our work now because that's an async op as well so we go ahead and yield the promise that comes back here, once again, and that yielded promise, just like the one before, comes back into this next variable. So our assistant then goes ahead and logs it out and that comes out in the console right here. By the way, I missed pointing out when we log the response, that comes out right here. And you can see in this case we have another promise, done being false, so we're not done with the generator yet. Again, we grab the value which is a promise. We use that promise to hook up a success reaction, a fulfillment reaction and the assistant again has made sure we did our job for the day. Everybody can go home. Eventually that async operation to convert the response to an object will complete, at which point in time the success handler will be called, passing the result which is a weather object, into this remind function which means that the result now comes up here and is then passed back into our generator. So we have the weather result right here, at which point in time we log the weather result out and you can see that here in the console right over here. I'm going to clear this out. Now this time, once we're done logging, the generator has nothing else in it so this call to next that we're waiting on here, this call to next won't get anything because there's nothing else to do. So the value of next here is an iteration result with the value of undefined and we're now done so done is true. You could imagine here we could have any number of these async operations and we'll just continue to wait for reach of them sequentially. You will notice that we've got an error here and that's because at this point in time we don't bother to check if we're done, we just assume that we never complete. So if we were to come in here and put some sanity checks in, we would say if next.done, then we know we're done so we'll want to go ahead and bail out. Now if we clear this out and run this again, you can see we don't get the error now.

## Four Async Ops in Four Lines of Code

This is all really, really cool. I hope you're as excited as I am. Let's come in here and let's comment out this logging. We don't need to see the result coming back right now unless maybe we're debugging. So the key here, we could just sweep this under the rug, right? This is what we need to focus on here. This is our function. This is our code. What's down below could come to you from a library. In fact, as we'll see in the next module there are many libraries you can use for this purpose. Now that we have this assistant that can take care of the nitty-gritty for us, let's pull some more information. First off, let's get rid of logging out the response. We don't need to see that. We also might want to get access to the five-day forecast. So of course, we'd want to go ahead and yield then and call fetch and pass the fiveDayUrl. And then of course we're going to get back a response on that as well, so we could have a response2 and then we can go ahead and convert that and again we fire off an async op to convert it. Now how cool is that? Not only do we have four asynchronous operations here and four lines of code minus the space in between, we then have the result of both available here without any hassle of global state to get this information available in one location, so we synchronize the result of these async ops and we have a total of six lines of code. That's pretty fantastic. Now one thing I point out. There is no need to avoid promise entirely. Right here we could do this instead. This is not that bad and now we get the fiveDay back and we only have one yield. We can still stay in the world of promises when it makes sense and switch over to yield when we want to get out of that world. It's completely up to us which world we want to live in, nothing wrong with either of them.

## Safety with Promise.resolve

Right now we could get into some trouble if we tried to pass back something that wasn't a promise, say some number. Let's say we yield 5. If we pull open the console right now we'll get an error and that's that promise.then is not a function and that makes sense. If we scroll down here we're doing nothing to check and make sure that something's actually a promise. Now there are a couple ways we could get around this. First off, we could try and check to see if this is a promise, maybe check for this then function, but I want to show you another functionality of promise.resolve that we didn't implement in our framework. Promise.resolve is actually a way for you to safely make sure that you're working with a promise. So we can pass anything to promise.resolve and if it's already a promise it will just return it back, otherwise it will turn it into a promise, which means if you pass something like a synchronous result to 5, it's not asynchronous, it's not a promise, you'll just get a promise that resolves and returns a value 5. And of course, it will do that asynchronously. Again, always making sure we're doing things later with promise. Anyway promise.resolve is a safe method so we can use that just to avoid any bugs you might have if we return something that's not a promise. This could happen if for example we're working with an API that's supposed to return promise but doesn't, though if it doesn't do that, well we could probably have other problems. Nonetheless, this allows us to have a safety check and when I stop and run this again, the program executes okay; we can even log out what we get to the console. Just proving we get back the value 5 and there it is.

## Throwing Errors Back into the Generator

We haven't yet talked about when things go wrong, so let's take a look at that next. Let's change the URL here so that our weather request doesn't complete. Now if we run this, you'll see we get a couple of errors here and while this is probably enough in the console for us to figure out what's going on as we're developing our app, we might want to show something to our users instead. So what if we could come into our function here and just like any other old code that's synchronous, what if we could throw a try/catch block around this entire chunk of code? Then what if we had something that would warn the user? Something that just told them that we couldn't get the weather and maybe to try again. Now of course right now we still don't have this try/catch block kicking in and that's because if something goes wrong inside of the asynchronous operation of these promise here, we don't find out about it. Down in our assistant, we register a fulfillment reaction, but we didn't bother to register anything in the case of an error. So we want to do something when things go wrong. Do you remember how we can pass errors back into a generator function? How we can get them to be re-thrown inside the generator? What we need to do somehow is call generator.throw and then we need to be able to pass the error to throw. Now to do this, we could try to pass our error to remind, but if we did that then we'd call next and we need to call throw instead, so we could then decide to try and pass some type of flag here, maybe a false in the case of a problem and a true in the case of an error and then use that as a switch statement to decide if we use generator.next or generator.throw. I don't really like that though. Instead let's do this. Instead of getting the result we're waiting to pass back, let's just get rid of that and let's assume we get the function that we'll use to resume the generator. I say that because both next and throw are functions that resume a generator. They're virtually identical minus the fact that next passes a result and throw throws an error into the generator. So if we're going to take in a function here instead of this right here, we would just invoke this function as our resumption function. Now we can control what we call whenever we call remind. So the very first thing we need to do is call generator.next and pass nothing, but this needs to be a function so I'll use an arrow function for that. So with this arrow function we now are calling remind and specifying how we're going to remind. This is our resumption function here. So all we have to do is take this and copy this down below and use this in our other locations where we need to resume as well and just make slight modifications to it. In the case of success, we go ahead and resume and pass the result. In the case of the promise erroring out, we pass the error and we also switch to using throw. Throw works exactly like next, it just throws in the error instead of passing in the result. Now having two arrow functions in a row may be confusing to you so let's do this actually. Let's switch this over to a function. It's called this fulfillment reaction. We'll get the result in and we'll put this inside and we'll do the same thing in the case of rejection. Clean this up. Okay. I think that looks a little bit easier to read through. Here is the promise succeeding, here is the promise failing. In the case of success we remind with the resumption function that calls next and passes the result. In the case of a failure, we remind with the resumption function that throws the error. Let's clear out our console here and scroll up and try this puppy out. And there you go. Do you notice in the console we now get failed to get the weather? We've now received the error here in our catch block, which means we could do whatever we'd like here to recover from the error and we could also narrow down what this try/catch wraps around if we wanted to have different handlers for different reasons. Think about how neat this is. We moved into that world of promise and promise provide a robust means of pulling all the possible errors into one mechanism, a mechanism that we can hook onto with the rejection reaction. So anything that could go wrong with the async op will flow into here and then we pass that into generator.throw which means we can now handle those errors with try/catch statements and recover from them just like we would from synchronous errors. Now while this is cool and while it's nice to have one error handling mechanism, you may not be a fan of using try/catch blocks. If that's the case, there's nothing stopping you from chaining on here and staying inside the continuation provided to you by the promise and using a catch handler in here to recover from an error as well. So right on the end of any of these promise you could just throw on a catch and go about your merry way handling it that way. So you have options here.

## Catching Exceptions Throw from Generators

Right now we're handling the error we have, but what if we didn't have a handler for this? So let's clean this up and let's let this run again. Now we get two pieces of information back. One is the browser telling us that there's a name resolution issue and the second is the error log entry we get from the failure itself, from throwing this back into the generator function and then doing nothing about it. We get an uncaught error and a promise and that's because if you scroll down here in our assistant, remember if we throw any errors inside of the generator, then when those errors come into and are thrown in the location of one of these yield statements, if they aren't caught here in our generator function, they'll bubble up just like a regular function would bubble up and that means they'll come back through the throw function, which resumed that function so this is like the original function call of a regular function. They'll come back through here wherever we invoke this at which is up here with our call to resume. This call to resume is nested inside the call to remind though. Remember we call remind here and all of this is nested inside of a rejection reaction which means that it will be captured by a promise framework and it would be available here if we chained on a catch, but before it ever gets there, it's actually also available here as a thrown exception. We actually can wrap our call to resume with a try/catch block and with this I just want to point out that when you resume a generator, it's just like calling a regular function. Exceptions thrown inside the generator will bubble up just like you're calling a regular function, so they'll bubble up here where we're calling resume. So we have the opportunity to handle them here as well. So for example, I could log out to the console and let's stop the program here. Clear this out. We could log out to the console error and then we also need to make sure we make next accessible so let's create a next variable declaration here. Clean this up. So we've thrown a try/catch around resumption and if resumption throws an exception, that means a promise had failed or something went wrong inside of our generator and we're catching that here and we'll just log out whatever happened here. It's a first step to doing something about it. And of course when this happens we really can't continue on so we bail out and we don't continue on with anything else in our generator and that makes sense. When a function throws an exception, that function doesn't continue executing. Let's go ahead and run this. Now you can see here we get a console.log entry pointing to line 47, which is where we have this console.log statement so we indeed have caught the failure right here. Now this isn't too helpful, so join me in the next clip where we'll make something a little more robust to deal with this.

## Catching Unhandled Errors in Our Assistant

Logging out when things go wrong inside of our assistant function is a good first step just to understand how exceptions flow around between our assistant function and the generator that we're controlling, but in a real app, this is all the code we would have, this right here. So in a real app, if we wanted to do something about this, we couldn't this point because that exception is being swallowed away into a catch block that just prints it out right now. So we could ask ourselves a few questions. Could we throw a try/catch block around this code here and would that take care of it? Could we get that exception thrown at some point so that we could catch it here? It turns out we can't and that's because that exception is thrown at some later time. It's not thrown synchronously. We're dealing with asynchronous operations here that can fail so we know that these exceptions can't come back to us right now. They will come back to us later on at potentially any point in the future and that means we have a condition where we have an asynchronous error potentially. Well, we have a mechanism for dealing with that. What's our mechanism for dealing with asynchronous failures? You probably guessed it. Our mechanism is using that catch function. So when something goes wrong inside of our assistant function, we could return back a promise and then we could chain onto that promise and add some sort of rejection handler, something like that. Because this is the code we would consume in our application, the code way down here that implements the assistant, this should come to us from a library so we don't have the ability to come in here and write conditional error handling logic nor would we want to. This is a framework that we're supposed to be able to reuse. So if we want to be able to chain on a catch, we have to get a promise back from our assistant that represents the completion of the assistant helping make all the reminders and running our generator function to completion and we can do that. When we create the assistant, we can wrap everything inside of here with a new promise and then we can use that executor pattern to get access to resolve and reject and then this is what we'll return back, but we need to wrap all of our code in this. This is a lot of nesting, but hey, we'd like to get those errors back and do something about them. So let's tidy this up. So we now have a promise that represents the completion of our assistant and inside of our assistant we have access to the ability to resolve and reject that promise at any point in time. So we could come into our error handler and instead of calling console.log we could call that reject function thus rejecting the promise that represents the completion of our assistant function and then we could bail out here. So that's all we should need to do. If we come back up and run this, we should get our new error message here and there we go, here's our recoverFrom error and then we print out the error that we have that we're recovering from, so now in our application we could do whatever we'd like to react to this error and the neat thing is, we're reusing this concept of a promise to pass back our errors so we have access to this nice composition of moving between generator functions and using yield for async ops in combination and in concert with being able to take advantage of all the promise mechanisms we've built.

## Knowing When the Assistant Is Done

Now in addition to knowing if something goes wrong and being able to address it from the assistant function, we might also like to know when the assistant function is done and do something to react to that. And of course that's going to be asynchronous as well so we could in this case chain on to the completion of the promise from that assistant function and perform whatever operation we'd like after that's done. For example, we could just log out that the assistant is done. This represents the end of whatever process we're orchestrating, again with our generator as a whole, so in this case the process is showing weather. Now to be able to trip this block of code here, we need to come up and fix the weather URL, get rid of that APIs and now our weather request should complete successfully, but as I'm looking over in the console here, I can see that we're getting this error that number is not defined and that's this line here I left behind when I got rid of yielding a number. Get rid of that and now we're getting our objects back, but notice we don't get our message here that our assistant is done; we got it when we had an error and that's because our catch handler was invoked, so this then function is running as a function of the fact that we had a catch handler that then ran. This is almost like having a little error recovery in here; however, we're not running this then function because down in our assistant, when we're done, we don't resolve the promise, so we can do that now and now we get our assistant is done message. Let's clear this out and run all this again. So we get back the two objects we log out, the fiveDay and the weather, and then our message prints out that the assistant is done. Now the only last thing I'll add to this is that it's possible our generator might like to return something as a final result. That something that it returns can't be an asynchronous operation because then the generator wouldn't be done, but it could be some synchronous value. For example, we might build up some object that has the data from both fiveDay and weather. So we could return an object here and send back the data and some other part of our application could consume that then. This could become a reusable way to sweep under the rug how we go about getting both the fiveDay forecast and the weather and then other parts of our application could work with this assistant function to run this and get the results, or we could even hide away the assistant function if we want and just expose some method that returns a promise here that people can hook up to. It might be nice then for completeness sake to come in here and print out the result; however, in this case we have undefined. So we're not getting this last result out of our generator function and making it available right here. If we want to do that, when we call resolve we have to pass something and in this case it just so happens to be with a return statement in a generator, not only will that end the generator just like a regular function, it'll also contain the value of the return statement and so now you can see assistant is done and the object is right there. If we want to, we could pass the object directly and then it could be logged out and accessed. Now you can see here, assistant is done and here is the object with the fiveDay and the weather attached to it. So some other part of our application could use the results then.

## Wow

I don't think there's much I have to do to sum up this module; this is pretty cool stuff. Here is what we have on the left-hand side as the result of working through this course and on the right-hand side is what we started with. These are not identical in behavior, but close enough to be able to talk about the benefits of this new approach. My question to you is, which side would you rather be working with? With our assistant function we perform four asynchronous operations and four lines of code, which is exactly the way it should be in my opinion. Over the right-hand side here just to initiate a weather request and hook up all the appropriate callbacks takes five lines of code and then on the left-hand side to process the results, well, we just access what seems like one scope here to be able to work with our information however we see fit and we know when we print things out here that this is the order they'll be printed out in, whereas on the right side, there is no guarantee about which request will finish first so the ordering of these log statements could flip around. We could write that ordering in and synchronize those two callbacks, but we know that's a lot of code. We just have that out of the box on the left-hand side here. So you could imagine this right-hand side here should be a lot worse than it actually is. Now on the right-hand side we can react to failures in one place just like on the left-hand side, but the nice thing about the left-hand side, other parts of our application can hook into and listen to errors happening in this process whereas over here, other parts would've had to have passed some callback that eventually got into this failure function before we even bothered to initiate our requests. Over here, something could hook up after the fact. We even know when this whole entire operation is complete. We have no way of doing that over here right now. We'd have to synchronize both callbacks, wait for both to complete, and then if somebody wanted to listen to the completion of both of these, they would've had to have passed a callback for that before we started any of this. It's a mess! Over here somebody could hook up after the fact to listen to the result. These are just a few of the benefits of the left-hand side. I think the rest speak for themselves so let's move on to the next module where we'll talk about libraries that you can use instead of working with this assistant function and we'll also talk about some proposed upcoming features to the JavaScript language itself which will obviate the need for this assistant function and give us a new word instead of yield.

# ES2018 Async/Await and Other Generator Control Flow Libraries

## co and Bluebird's Promise.coroutine

In this module, I want to turn our attention to existing libraries you can use and also I want to talk about future language specifications that obviate the need for these libraries entirely. CO is perhaps the most popular of these libraries. You can find out more on npmjs.com/package/co, though I think the example here is lacking. I think you should look at the example we have in this course to really see the power of this, which speaking of, here's a nice example of using CO with MongoDB. So this uses the Node.js driver which can return promises and this is a sample application. You could imagine that this is a script to insert some records into a MongoDB database. So you more or less wrap your program in a generator function and pass that to CO. CO will then take care of the process of creating a generator from that generator function and then stepping through it as we saw and then as your generator function yields, it pauses, CO will then wait for that promise to complete and then resume the generator function. So you can see here an async call to connect to the database, another async call to insert records into the database, and then another insert call to insert multiple records into the database and all of this as if this were synchronous code, and then chained onto the end we have a nice, global catch handler in case anything goes wrong. So as you can see, this works much like our assistant function, but CO is not the only library. I wanted to point out another and just remind you that there are many of these so you might want to look around and find the one that's best for you. Bluebird, a Promise library we looked at earlier, also provides a function called coroutine and as you can see here, even I think example is a bit complex, you can see once again we have this assistant function here, Promise.coroutine. We pass a generator to it just like with CO. This coroutine function will create an instance of this generator and then step through it. While this is complex, this is actually kind of interesting. This example shows how you can have communication going back and forth between two separate coroutines. So ping and pong here can talk to each other and they can do that indefinitely. So these are a few of the libraries you can use and these are a good stop gap for now. Next, let's take a look at how we might integrate this into an Express.js app, if you're wondering what does this look like on the server side of things. Remember, this MongoDB example is a great example of say, a shell script in JavaScript that inserts some records into a MongoDB database. This would be run one time, but we might have a Node.js app that server software that's running indefinitely and has requests coming in, so how can we integrate there?

## Server Side Web Apps

If you have server-side JavaScript apps, there will be some mechanism whereby you have request handlers. They'll be attached to a route, for example, the root route of a website here, and you'll usually have some function that handles a request. You can rewrite that function as a generator function and then you can use something like co-express to automatically wrap that and pass on the requests and response objects for you, so creating an assistant function here that's knowledgeable about express. This does add one extra layer here, but then you have this goodness of working with Promises and having async ops look like serial operations. This means you can write much more maintainable server-side software and server-side software is a case where you usually have a lot of async ops going on as you're interacting with databases, caches, and other services. The other approach you could go with is to just use a framework that's opinionated and the framework itself understands this idea of working with generator functions and so when you register handlers in this application, the application takes care of making sure the assist function calls your generator function and steps through it. So it can just be baked into the framework you're using itself. While this is possible to do, if you've already invested a lot in express, then go with something that adapts express for your needs.

## Async and Await

While CO and Bluebird's Promise.coroutine will work for you, you should keep an eye out for a new feature coming in a future version of JavaScript. If you come out to the active proposals for ECMAScript, you'll see that async functions are at the top of the list here; it's been a proposal that's been around for a while. If you look at the proposal itself, you'll see that the stage 3 draft has been out since the beginning of 2016 and that stage 3 means that it's pretty close to becoming an adopted proposal in the ECMAScript standard. So just keep an eye out for this because once this is available, you won't need something like CO or Promises coroutine function; this functionality will just be baked into the language itself. Of course, it will take a while for it to get into various different environments. It will be a while before it's implemented, just like anything. So one thing you can do is consider using a tool like Babel, which means you can take advantage of these new proposed language functionalities and compile them down to vanilla JavaScript. If you're already using this in your applications, you could just take advantage of this directly or you could add this into your development process if you want. If you want to try this out, come click the tryout button and follow along with me. This is a live REPL that's running through Babel. The output on the right-hand side is the compiled vanilla JavaScript and then when we put code in down here in this tab, it's a bit confusing. We'll be able to see the console output. So for example, console.log, the value of 1. You can see 1 comes down here. Obviously, console.log doesn't require any transformation. Let's come over and copy our generator function from our previous example and then come paste it into Babel. Don't worry about the compiled output, that's not so important. What we want to do is convert this to use async functions, so this is what this will look like. Instead of the asterisk to create a generator function, we'll use the async keyword to create an async function and then you can see we've got some errors here and that's because we can't use yield inside of an async function; we instead use a new keyword called await. It does everything that yield does. It just syntactically makes a little more sense. We're awaiting the result of this fetch operation whereas yield before wasn't so intuitive that we were getting something back that we were waiting for something. Now to make these requests, I need to pull in the URLs. So let's bring those into the top here and then down at the bottom or up at the top, wherever you'd like, we can go ahead and invoke our async function and there you go. You'll see in the little console output here, it's a bit confusing, but you'll see loading… and then you'll see the two objects we get back; one for weather and one for the five-day forecast. A couple of things to point out while you're trying this out. I'm making plain HTTP calls here to this open weather map API and you'll need to make sure that you're not using the HTTPS version of Babel, otherwise you won't be able to make these plain HTTP calls. It will come up as a warning in your browser. It's one you won't see though because it will just come up in the log here, a warning about mixed content. One other thing to watch out for. Make sure you have the appropriate boxes ticked here. Now in this case, you only need stage 3 ticked because this is a stage 3 proposal at this time. Notice if I untick this, we'll get a warning here and that's because we can't write an async function without stage 3 proposals, but also watch out for the fact that you can click on any of these stages and get the rest. So I could click on stage 0 and that means include everything above stage 0. Whatever you do, make sure stage 3 is enabled. Now with those warnings aside, just take a look at this right here. The language now provides for us that assistant function. All we have to do is call this like a regular function and then we have to change nothing about our application. Asynchronicity has moved to a simple seam of an await keyword and that's it. That's exactly what we're looking for at the beginning of this course.

## Challenge - Parallel Requests with Async Await

Now it's time for a challenge. Go out to babeljs.io, load up this example and get it working with async and await functions and then I also want you to do something else with this. Right now we have two requests we're making. One here and one here. These are not firing off in parallel. To prove this to you, let's open the network tab. If I make any change over here, the two requests will come up and if we zoom in on these, you'll see that these do not overlap. So there's weather, there's forecast. They're not happening in parallel. The challenge to you is to figure out how to make them happen in parallel. Join me in the next video where I'll give you some possible solutions.

## Parallel Solutions and Catching Async Function Errors

If you're still stuck, here's a hint. Keep in mind, we're making promises and then we're yielding those and that's important because we can create this promise before we yield it. So we could have an intermediate weatherRequest variable and that's something we could capture and then hold onto it and do other things before we yield and wait for the result, which means we could come here on the fiveDay and grab it as well and have a fiveDay request and then we could create a variable for that. And now we're firing off both requests, holding onto both promise operations, before we wait for any of the results to come back. This is that beautiful thing about having this abstraction where we can start an async operation and then later on wait for it to complete. Now if the code doesn't convince you, let's pull open the network tab and just make a change so the code runs again and then if we zoom in on the requests, maybe zoom in a little more, you can see these are fired off at the same time. So with that simple change we can control the ordering of things and this again leverages what I like to call lexical parallelism; the parallelism is just apparent in what we literally wrote down here. Now if you were astute, you would've noticed that this asynchronous operation here to process the JSON data we also have an opportunity here for parallelism. There are a couple of ways we could deal with this. The way I like is to just go ahead and chain a call here and transform the result. This is where I see a benefit of staying in the Promise abstraction; I don't need to have this extra step here to do this transformation and then that means I'm going to get the weather response back right here and I'll get the fiveDay response back right here. Now if you needed to inspect that actual web response, you couldn't do that because you've now lost that transformation here, but I like this approach because really all I care about is if something goes wrong and if something goes wrong, we'll still get an exception. For example, if I pull open the console, I'll clear this out and then let's break one of the URLs, at which point you can see we get an error back in the console. One is the browser helping us out with name resolution issues; the other one is the error we could've done something about and it's an unhandled promise rejection so that means somewhere we have a promise that we can subscribe to and handle this error. Can you can guess where that's at? Well, just like with our implementation of the assistant function, this new async await construct with an async function allows us to then chain a call because a promise is returned from an async function then and we can add a catch here and capture the error and then we can do something with it. Now if we clear out the console here, run this again, you'll see we still get our browser helping us with this error, but now we have control over what's coming out. Just to prove that I'll add a little custom string around this, so you can see error: and that's the message that I added, which means we can have our own error handling code here, but more importantly what this means is async functions return promises to us, which means we could just treat these as library functions and so we can compose this complex async operation inside and just have one promise that represents the result and bring that into the rest of our application using the same promise constructs that we're familiar with. This is pretty powerful. Right now we have two calls to await, to await the results and that doesn't impact

## Awaiting Promise.all

Right now we have two calls to await, to await the results and that doesn't impact the performance of our application, but sometimes we may have multiple promises to wait for, we could have end number of promises in a collection of promises and we might like some technique to be able to wait for all of them to complete and that's where a special function called Promise.all comes in. Promise.all takes an array or any iterable. It goes through the elements of that iterable and waits for each of them to complete; they don't even have to be promises. If they aren't promises they'll be sanitized with a call to promise.resolve. In other words, they'll be promise-ified, even if they're just hard-coded values. But the important thing here is we could pass a collection of promises to Promise.all, which will return a promise that represents the completion of both of these and now we get back an object that's an array with the response and inside of that array, the very first element is the first response and the second element is the second response. If we go up and fix the error we have here with our URL, you'll see we get back our responses. So this is just an optional alternative for waiting for multiple promises to complete, probably more important when you have an unknown number of promises you need to wait for. If you have a known number, you can use destructuring to simplify access to the elements here and now we can go back to having variables that are named explicitly. Destructuring just pulls these first two elements out of the ray into new variables named weather and fiveDay.

## Challenge - Promises A+ and ES6 Spec Tests

It's now time to wrap up our course. If you've stuck with it, you know that this was a long course, but I hope you got a lot out of this course. I had an ah-ha moment when I saw all these pieces working together; generators, promises, async await. When I saw all of this come together and culminate in a solution that helps tackle a problem that we all face and that we all hate, dealing with asynchronicity in JavaScript, I was excited to start sharing this with other people and I thought I would retrace the steps for you that led me to my ah-ha moment, hoping that you would have the same one. If you'd like to take this further and learn more and get your mind even more around how Promises work, then you'll want to take a look at a couple of test suites that can help you plug in the Promise framework that we started in this course and continue to develop the features of it to match that of the Promises/A+ specification, which you can get the test for here on this GitHub repository and then also there's another set of tests that help you make sure that the framework you're building is compliant with the ES6 spec. If you take these two sets of tests and put them together and get them working with your Promise framework, you'll find some of the Edge cases that we didn't get time to cover in this course and you'll learn a little more about the protections and functionality that's built into the ES6 Promises that you'll be using. Both of these basically require the use of an adaptor where you plug in certain calls from your Promise framework into the test suite and then the test suite just runs and tells you the things your framework doesn't provide or provides inaccurately and I think there are several hundred tests in total. Anyways, this is a fun exercise if you really want to make sure you understand Promises.

## The End - Value Proposition Revisited

Back toward the beginning of the course we had this example here of getting the current city and we even have an error check if that fails. We then make a call to get the weather and forecast in parallel, using the city that came back. We capture those responses in local global variables and we do that so that we can then synchronize these responses and handle them together and that's what our finishIfReady function did. We also have, of course, error handling in here and in this case we just bail out of the test if anything goes wrong and then in finishIfReady we have this check to make sure that we get both responses and once we do then we know we're done. We can then perform our final computation on both the results and print out whatever we'd like. Now when I went over this, I also pointed out that we're missing something here. Any of these function calls we make to kick off a callback could throw errors so we should have try/catches around each and every one of these. If we don't, then we could be missing thrown exceptions and that could bring our application down. This code is just simply unruly. You don't want to write this stuff, but this is what you're faced with with asynchronous operations, especially on the server side when you're interacting with the database or web services and you're making all kinds of async calls to fulfill an operation, your server side handling logic is insane. It's just not even something you would do. I told you at the beginning this is what I think this should look like. We should be able to get the current city and then we should be able to do something like grab two requests and we just saw this with async await where we can get promises that represent these async ops so that we kick these off in parallel and then we should just be able to wait for the results just like that and as we get them, print them out in whatever order we'd like. Well, I've shown you how we can do that now. We can use these new async functions to pull this off. So we have this weatherProcess. We can make an async function for that and we can return the weatherProcess which means that our test framework will catch any errors and show them to us, which means we don't need any error handling logic inside this weatherProcess function unless we're going to try and recover from something. I can literally copy this example from here and paste this in here now and we can pull this off and however we'd like, we can move results around. If we want to synchronize the requests, well, that's just as simple as putting this right down here. Boom! We have access to both of the results at the same time; we could combine them or do whatever we'd like. Transparent synchronization of parallelism. Amazing. So I can ask you, which would you rather maintain? This? Or This? And that's the value proposition of this course. I hope you have a really good understanding of Promises now and likewise understanding a good use case of generators and more importantly, understanding these new construct called async await that's coming in a future version of JavaScript. I hope you're excited enough about this that you're going to try to use some of the libraries that can allow you to leverage this stuff in your apps right now because this right here, this makes writing code in JavaScript much more worthwhile, especially on the server side of things. Now we have the non-blocking I/O nature of JavaScript which makes it extremely performant. We now have that and we have code that's highly readable. We get the best of both worlds, which is something that's hard to pull off in other platforms that are synchronous and have blocking I/O. If you have questions, don't forget to throw those into the discussion board or just reach out to me, and if you'd like to know more about what I work on and what I do, check out my newsletter. You can sign up for this and if you want, you can check out past issues to see what I write about