

Extracted from:

Mastering Dojo

JavaScript and Ajax Tools for Great Web Experiences

This PDF file contains pages extracted from Mastering Dojo, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2008The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The logo features a large, stylized green letter 'B' on the left. To its right, the words 'Beta' and 'Book' are stacked vertically in a bold, green, sans-serif font.

Beta Book

Agile publishing for agile developers

The book you're reading is still under development. As part of our Beta book program, we're releasing this copy well before we normally would. That way you'll be able to get this content a couple of months before it's available in finished form, and we'll get feedback to make the book even better. The idea is that everyone wins!

Be warned. The book has not had a full technical edit, so it will contain errors. It has not been copyedited, so it will be full of typos and other weirdness. And there's been no effort spent doing layout, so you'll find bad page breaks, over-long lines with little black rectangles, incorrect hyphenations, and all the other ugly things that you wouldn't expect to see in a finished book. We can't be held liable if you use this book to try to create a spiffy application and you somehow end up with a strangely shaped farm implement instead. Despite all this, we think you'll enjoy it!

Throughout this process you'll be able to download updated PDFs from your account on <http://pragprog.com>. When the book is finally ready, you'll get the final version (and subsequent updates) from the same address. In the meantime, we'd appreciate you sending us your feedback on this book at <http://books.pragprog.com/titles/rgdojo/errata>, or by using the links at the bottom of each page.

Thank you for being part of the Pragmatic community!

► **Andy Hunt**

1. The total is 1
2. The total is 3

`dojo.publish("numbers", [1])` causes a call to `numberAccumulator.add(1)` followed by a call to `showTotal(1)`. This demonstrates how `dojo.publish` *must* be provided an array for its second argument, but deals the contents of the array out as individual arguments to all subscribed functions. Since `showTotal` doesn't care about the arguments, it just ignores them.

Also notice that the example demonstrates how to connect a handler that is a method on an object (⑩) and just a simple function (⑩).

Publish subscribe is a very useful design pattern. The key to its power lies in its ability to separate concerns. The “publishing” process need not concern itself with the consequences of publishing, but rather is *only* concerned with knowing when and what to publish. On the other hand, subscribers need only concern themselves with taking actions consequent to some publication. Indeed, publishers and subscribers don't even need to know about each other. Since each process (that is, each publisher and each subscriber) is independent, the complexity of each process is consequently decreased, which, in turn, increases the chances that each process will function as desired.

All things considered, both DOM and user-defined events are fairly easy to understand and use, particularly since Dojo works out browser incompatibilities for us. When we start working with more complicated asynchronous processes this won't be true and we'll want and need more capabilities. So, let's turn our attention to these more advanced models starting with Dojo's Deferred class.

6.4 Managing Callbacks with Dojo Deferred

Event programming is an example of a *simple, loosely coupled* asynchronous programming system. Communications between triggering functions and handlers is conceptually limited to a binary signal that says an event occurred. Yes, there is the event object, but the information contained within the object is very generic in nature. There is no provision for communicating more-elaborate results (for example, error conditions) nor is there any process control (for example, canceling or chaining the handlers). Sometimes, more is needed.

In this section we're going to explore `dojo.Deferred`—a Dojo class that manages more advanced interactions between an asynchronous func-

tion and one or more callback functions. Since some of the conceptual framework can be a bit abstract and theoretical, we'll work through a real-world example that illustrates how to use Deferreds effectively. Along the way, we'll see that building systems with Deferreds results in powerful, simple, and elegant solutions to otherwise-complicated asynchronous programming problems.

The Example: Building a High-Performance Display Engine

Let's consider the example of displaying a panel of data from a database. The process is governed by a display engine that makes a request to the server for both the metadata (background decoration, layout information, and user interaction logic) and the data (ultimately defined by a SQL query). When the metadata arrives, the panel is created and initialized with various HTML controls, decorations, and event logic. Similarly, the data is stored in a cache when it arrives. Finally, after *both* the metadata and the data arrive, the data is pushed into the HTML controls on the panel and the panel is released to the user.

A naive implementation might use the algorithm given in Figure 6.4, on the next page. The good thing about this design is that it is very simple and would surely work. Unfortunately, the browser may freeze for several seconds while the data and metadata are being retrieved. Further, the design retrieves and processes the metadata and data serially rather than concurrently as expressed by the process description. Our goal is to come up with an implementation that solves these problems while keeping the design simple.

To achieve our goal, the metadata and data service requests are executed independently and asynchronously. This will keep the browser from freezing. Here are a few other reasons for this design:

- It is possible—even likely—that the metadata is already in a local browser cache. However, if we bundle the metadata with the data, then there is no chance that the metadata would ever be cached since the data is always changing.
- It's also possible that the metadata may be cached at an edge server, a physically different server than where the data resides. Clearly, two independent queries ought to be used in this scenario.
- After either the metadata or the data is received, there is some processing that takes place before both items are required to complete

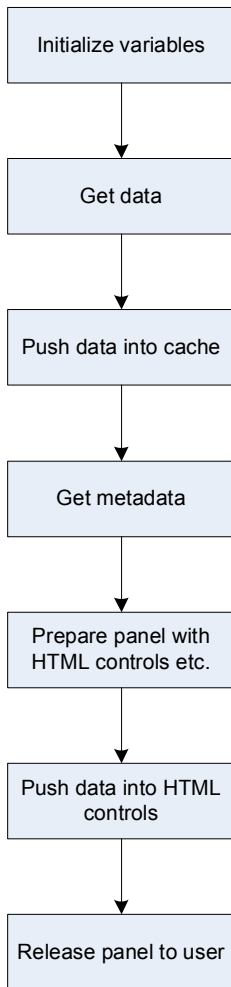


Figure 6.4: Naive Display Engine Design

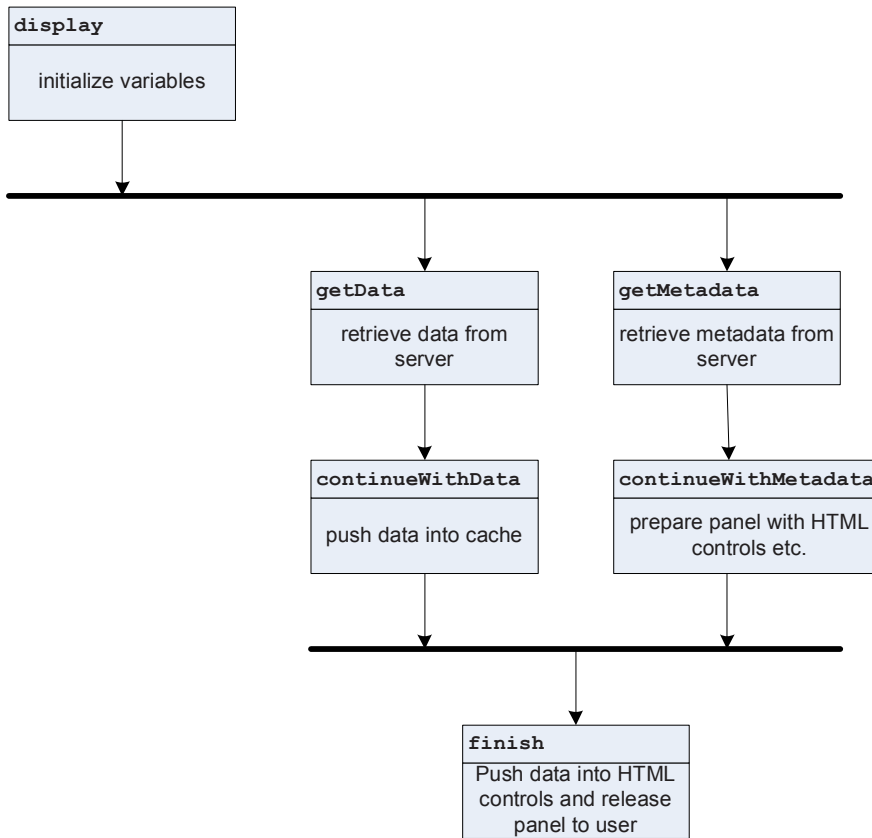


Figure 6.5: Asynchronous Display Engine Design

the overall task. Why not get this work done with the first arriving component while the other component is still transmitting?

Figure 6.5 shows a revised design that uses two asynchronous processes. The algorithm gets started by calling `display` which sets up some infrastructure to coordinate the rest of the work, fires off two asynchronous functions (`getData` and `getMetadata`), and then returns. The asynchronous functions signal two callback functions (`continueWithData` and `continueWithMetadata`) when they complete. The callbacks take processing as far as possible along each fork and then call `finish` which joins the two asynchronous processes into a single thread of execution that finishes up by displaying the data as given by the metadata.

If we implemented this system in terms of user-defined events we'd have to build a fair amount of scaffolding that has nothing to do with the problem at hand. Fortunately, `dojo.deferred` already contains this scaffolding. So we'll use it to build the solution. Let's start by implementing `display`.

Implementing the Controlling Process

Notice that the various functions depicted in Figure 6.5, on the previous page are tightly coupled. Each function takes some information from the function above it, does something, and passes information to the function below. By defining all of the functions within the `display` function, we can implement this coupling by sharing a local variable. Here is our first attempt:

[Download](#) `asynchronous-programming/asynchronous-programming-deferred.js`

```
//implements a display engine:
//display the data given by query in the panel given by panelId
function display(panelId, query) {

    //displayInfo is a shared bookkeeping area
    var displayInfo= {
        panelId: panelID,
        query: query,
        processCompleteCount: 0};

    //get an HTML div in which to place the panel
    //allocatePanel is provided by the pane management logic of our application
    displayInfo.div= allocatePanel(displayInfo);

    //TODO #1
    //make an asynchronous call to get the data
    //after the asynchronous call finishes,
    //continue by calling continueWithData

    //TODO #2
    //make an asynchronous call to get the metadata
    //after the asynchronous call finishes,
    //continue by calling continueWithMetadata

    //process the data as far as we can without the metadata
    function continueWithData(data){
        //put the data in an application-wide cache
        //no need to wait for the metadata to do this!
        cacheData(data);

        //save quick access to the data
        displayInfo.data= data;
```

```

    finishDisplay();
}

//process the metadata as far as we can without the data
function continueWithMetadata(metadata) {
    //set up the HTML and other control structures
    //given by the metadata...
    //no need to wait for the data to do this!
    preparePanel(metadata);

    //save quick access to the metadata
    displayInfo.metadata= metadata;

    finishDisplay();
}

//when continueWithData and continueWithMetadata have completed,
//finish with the display process
function finishDisplay(){
    //TODO #3
}

//note: the functions allocatePanel, cacheData, preparePanel,
//populatePanelWithData, and releasePanelToUser are provided
//by the application elsewhere (they are not Dojo functions!).
//Their construction is not important to this discussion.
}

```

Try reading the code like a novel. Read each line, including comments, sequentially. We think this code is really elegant, and it's all due to the powerful programming paradigm that `dojo.Deferred` provides.

As for the TODOs, somehow we must query the server for the data/metadata asynchronously and then, upon completion of each of these functions, call `continueWithData/continueWithMetaData`, passing the received data/metadata. So we need machinery to coordinate an asynchronous function call with a callback function. This is exactly what `dojo.Deferred` does.

Registering Callbacks with `dojo.Deferred`

`dojo.Deferred` is a constructor function that creates objects that manage the interaction between a single asynchronous function call, a chain of one or more callback functions, and other clients that are interested in the asynchronous operation.⁴ The Deferred interface includes methods

4. Remember, any Dojo function that starts with a capital letter is a constructor function and consequently defines a class.

for the asynchronous function to signal normal or abnormal completion and return results as well as methods that let other clients:

- Specify a chain of functions to execute upon normal and/or abnormal completion of the asynchronous function (that is, a chain of callbacks).
- Check to see if the asynchronous function completed, and, if so, access any results returned.
- Cancel waiting for the asynchronous function.

Typically, the process that makes the asynchronous call creates a Deferred instance, shares it with the asynchronous function, adds one or more callbacks, and then forgets about it.

Getting back to our example, we have two independent callback functions, namely `continueWithData` and `continueWithMetadata`, which continue after the asynchronous functions that get the data and metadata complete. We can use Deferreds to set this up:

[Download](#) asynchronous-programming/asynchronous-programming-deferred.js

```
//TODO #1 partial implementation...
displayInfo.dataDeferred= new dojo.Deferred();
dataDeferred.addCallback(continueWithData);
//still TODO:
//somehow, get the data asynchronously and signal
//displayInfo.dataDeferred when this completes

//TODO #2 partial implementation...
displayInfo.metadataDeferred= new dojo.Deferred();
dataDeferred.addCallback(continueWithMetadata);
//still TODO:
//somehow, get the metadata asynchronously and signal
//displayInfo.metadataDeferred when this completes
```

The example shows that callback functions are registered by calling the Deferred method `addCallback`. We will discuss registering callback functions in detail in Section 6.4, *Specifying Callbacks and Errbacks*, on page 132.

We'll implement the asynchronous functions by using XHR objects. We've already seen these back in Chapter 3, *Connecting to Outside Services*, on page 36 and we're going to explore them in great detail later in Chapter 8, *Remote Scripting with XHR, script, and iFrame*, on page 176 so we won't say too much more about them now. The key to making the Deferred objects we created come to life is having the XHRs

call the Deferreds' callback methods with their result. When this happens, Deferred orchestrates passing the results to the callback functions. Here are the completed TODOs.

Download [asynchronous-programming/asynchronous-programming-deferred.js](https://github.com/jashkenas/dojo/blob/master/async/async-programming-deferred.js)

```
//TODO #1 implementation...
displayInfo.dataDeferred= new dojo.Deferred();
dataDeferred.addCallback(continueWithData);
dojo.xhrGet({
  url: "data",
  content: query,
  handleAs: "json-comment-filtered",
  load: function(data) {
    deferred.callback(data);
  }
});

//TODO #2 implementation...
displayInfo.metadataDeferred= new dojo.Deferred();
dataDeferred.addCallback(continueWithMetadata);
dojo.xhrGet({
  url: "metadata",
  content: {id:panelId},
  handleAs: "json-comment-filtered",
  load: function(data) {
    deferred.callback(data);
  }
});
```

According to the semantics of XHR, the load argument (a function) is called with the result of evaluating the response text with the content handler given by the handleAs argument. In our case, we are expecting the response to be sent as comment-filtered JSON (we'll discuss this in the (as yet) unwritten *xhr-load-functions*). So, if everything goes as planned, continueWithData will be called after the data has been received and wrapped up in a JavaScript object; the same goes for continueWithMetadata and the metadata. This is exactly the effect we are looking for.

Deferred.callback is always passed a single argument that holds the result of the asynchronous function computation. Since JavaScript is not capable of returning multiple results like some other languages, exactly one argument is expected. In the event you need to pass multiple results around, simply wrap the results in either an array or an object.

Let's now turn to TODO #3 which joins the two asynchronous callback functions once they've both completed. The idea is to keep track of how many asynchronous processes have completed and when they've all completed, continue on with the process. Here is the JavaScript:

```
Download asynchronous-programming/asynchronous-programming-deferred.js
//when continueWithData and continueWithMetadata have completed,
//finish with the display process
function finishDisplay() {
//TODO #3 implementation...
  displayInfo.processCompleteCount++;
  if (displayInfo.processCompleteCount==2) {
    //both have completed...
    populatePanelWithData(displayInfo);
    releasePanelToUser(displayInfo);
  }
}
```

The example also nicely demonstrates how to fork a single process into several processes (implemented as asynchronous function calls) and later join the processes back into a single process.

At this point, the basic display engine is complete and works perfectly so long as no abnormal conditions are encountered. If you step back for a moment, you'll notice a fairly complex algorithm was expressed in very clear and concise code. This was one of the promises we made at the beginning. Next we'll expand on the solution to include error handling.

Handling Errors

What should we do if either or both asynchronous calls fail? Perhaps the panel div was initialized with a message saying "Retrieving Data". If either asynchronous function fails, this message will stay forever, without any further notification to the user about the true status of the request. We must do better.

`dojo.Deferred` can register callback functions to execute if/when an error occurs. To avoid confusing "error path" callbacks with normal path callbacks, we'll call them "errbacks". Naturally, they are registered with the `dojo.Deferred` method `addErrback`.

For our display engine, upon failure of either asynchronous function, we can use `addErrback` to hook up an errback that asks the user if he would like to retry the operation. If the user says "yes", then a new

asynchronous function call is made; if the user says “no”, then the div is destroyed.

Since the new design sets up the asynchronous function calls twice—the initial call and the call made by the retry—lets encapsulate this work in a couple of functions. Here’s the code for getting the data (the code for getting the metadata looks similar):

[Download](#) asynchronous-programming/asynchronous-programming-deferred.js

```
//important: this is put inside the function display()
function getData(){
  displayInfo.dataDeferred = new dojo.Deferred();
  dataDeferred.addCallback(continueWithData);
  dataDeferred.addErrback(handleDataError);
  ❶ dojo.xhrGet({
    url: "data",
    content: query,
    handleAs: "json-comment-filtered",
    load: function(data){
      deferred.callback(data);
    },
    ❷ error: function() {
      deferred.errback();
    }
  });
}
```

We’ve done three things here. First, we encapsulated TODO #1 in the function `getData`. `getData` must be defined within the body of the function `display` so that all of the referenced functions and data are visible. Second, in addition to registering a callback function, we registered the errback function `handleDataError` ❶ (we’ll write `handleDataError` in a moment). Finally, we set up the XHR call to signal the `Deferred` through errback when an error occurs ❷.

Next, let’s write `handleDataError`; it’s trivial:

[Download](#) asynchronous-programming/asynchronous-programming-deferred.js

```
//important: this is put inside the function display()

//informs the user that an error occurred retrieving the data
//gives the user the option to retry or cancel
function handleDataError(){
  giveRetryCancelMessage(
    "The server failed to deliver the data.", //message to give to the user
    getData, //retry function
    function() { //cancel function
      //destroyPanel is provided elsewhere by our application
      destroyPanel(displayInfo);
    }
  );
}
```

```

    }
  )
}

```

The implementation of `giveRetryCancelMessage` isn't important to the discussion. It simply displays a dialog that presents a message, asks the user if he'd like to retry or cancel, and provides retry/cancel push-buttons. When the user pushes one of the buttons, `giveRetryCancelMessage` calls the appropriate function. Since `getData` is given as the retry function, another asynchronous call will be attempted if the user selects retry. Similarly, if the user presses cancel, `destroy panel` will be called.

Our display engine is nearly complete. But before we do more, let's take a closer look at registering callbacks and errbacks.

Specifying Callbacks and Errbacks

The callback processing in the example is simple: if the asynchronous call succeeds, call one function (for example, `continueWithData`); otherwise call another function (for example, `handleDataError`). Most of the time, this is all you'll need. For more-complex scenarios, `Deferreds` are capable of registering a list of callback and errback functions, and the functions will be executed in the order they were registered.

`Deferreds` maintain a queue that holds the functions waiting to execute. Callback/errbacks are always added to the queue in pairs—a callback paired with an errback. As the queue is traversed, one or the other functions is called depending upon whether the `Deferred` instance is in a normal or error state (we'll discuss these states in a moment). There are four methods for appending continuation functions to the end of the queue. CHANGE TO DDL LIST

```
addCallback(context, f)
```

Add the (callback, errback) pair given by `(dojo.hitch(context, f), null)`. When a null is encountered while traversing the queue, the entry is just ignored and processing proceeds to the next entry.

```
addErrorback(context, f)
```

Add the (callback, errback) pair given by `(null, dojo.hitch(context, f))`; null as above.

```
addBoth(context, f)
```

Add the (callback, errback) pair given by `(dojo.hitch(context, f), dojo.hitch(context, f))`.

```
addCallbacks(callback, errback)
```

Add the (callback, errback) pair given by (callback, errback).

Any of these methods can be called during the lifetime of the Deferred object. If the asynchronous call has already been completed and the queue is empty, then the newly registered callback/errback is executed immediately; otherwise, it goes to the back of the queue.

Each successive callback/errback is passed the result of the previous function in the queue. So, the first callback/errback is passed the result of the asynchronous function, the second callback/errback is passed the result of the first, and so on.

A Deferred enters an error state under any of the following conditions:

1. The asynchronous function returns a result by calling the Deferred method `errback`.
2. The asynchronous function or any subsequent callback/errback returns a result that is an instance of `Error`.
3. Any callback/errback throws an exception. If this occurs, then the Deferred instance will catch the exception and the caught exception object becomes the next result.

On the other hand, a Deferred enters a non-error state when original asynchronous function or any subsequent callback/errback returns a result that is not an instance of `Error`. This means that a Deferred object can enter into and out of an error state, and as the queue is traversed, either the callback or errback function is executed depending on the error state (that is, processing can switch back and forth between callback and errback functions).

You can always query the current state of a Deferred object through its `fired` property:

- `-1`: indicates the Deferred object is still waiting for the asynchronous function to finish computing.
- `0`: indicates the Deferred object has received results and is in a non-error state.
- `1`: indicates the Deferred object has received results and is in a error state.

Given a Deferred instance `d`, the last results computed are also always available through `d.results[d.fired]`.

Of course, there is no limitation on the type of single argument that is returned. This brings up the mind-boggling possibility that a callback/errback might return a Deferred. When this happens, any subsequent callback/errback function is not called until the returned Deferred has results.

Canceling Callback Processing

Let's get back to our example. So far, the panel starts life by telling the user that it is working on retrieving the data. If everything goes as planned, the data is displayed after both the data and the metadata arrive. On the other hand, if an error occurs, the user is given the option to try again or abort the operation. So at this point, the user can cancel the request only if an error occurs.

There are other situations where the request should be cancelable. For example, the user may wish to cancel the operation before it either completes or returns an error. Other processes may like to cancel asynchronous processing as well. Deferred provides a solution to this problem.

When the Deferred method `cancel` is called on a Deferred instance that is still waiting on the asynchronous function, it stops waiting, sets its internal state to indicate an error (sets its `fired` property to 1) and begins traversing the callback queue. On the other hand, if the asynchronous function has already returned results and the queue has already been traversed, `cancel` has no affect except for one very odd circumstance: if the last callback/errback executed returned yet another Deferred instance as *its* result, then this other Deferred is canceled.

The default cancel processing can be a bit harsh since it just creates a JavaScript Error instance and starts down the error callback queue. Fortunately, there is a gentler way. The Deferred constructor takes an optional single argument—a function termed the “canceler”. If `cancel` is invoked on a Deferred instance that was created with a canceler, then the canceler is called with that Deferred instance passed as an argument. Armed with this information, we can make the final adjustments to the function `display`.

First we need to write a canceler function. It should cancel the other Deferred and clean up the panel. Here it is:

```
Download asynchronous-programming/asynchronous-programming-deferred.js
```

```
//Important: this function is put inside the function display().
```

```

function doCancel(theDeferred) {

    //execute this routine for the first Deferred canceled only...
    if (!displayInfo.dataDeferred) {
        return;
    }

    //figure out which deferred was canceled and take
    //a reference to the other deferred
    var temp= displayInfo.dataDeferred==theDeferred ?
        displayInfo.metadataDeferred : displayInfo.dataDeferred;

    //ensure this routine is executed only once
    displayInfo.dataDeferred= null;
    displayInfo.metadataDeferred= null;

    //cancel the other Deferred
    temp.cancel();

    //clean up the display
    destroyPanel(displayInfo);
}

```

Since the Deferred instance that's canceled first must cancel the other Deferred instance, a little care is required to avoid infinite recursion. The first Deferred instance canceled clobbers both `displayInfo.metadataDeferred` and `displayInfo.dataDeferred` so that the canceler will return immediately when the other Deferred instance is canceled.

The canceler should be hooked up to both Deferred instances when they are created so the line:

```
Download asynchronous-programming/asynchronous-programming-deferred.js
```

```
displayInfo.dataDeferred= new dojo.Deferred();
```

Should be replaced with the line:

```
Download asynchronous-programming/asynchronous-programming-deferred.js
```

```
displayInfo.dataDeferred= new dojo.Deferred(doCancel);
```

And similarly for `displayInfo.metadataDeferred`.

It is important to remember that when a Deferred instance is canceled, the error functions in the callback queue will be invoked. Since we don't want to give the user a "retry/abort" message after he cancels, we'll modify our errorback to give the message box only if the process has *not* been canceled. Here is the revised errback for `getData` (the errback for `getMetadata` looks similar):

[Download](#) asynchronous-programming/asynchronous-programming-deferred.js

```
//informs the user that an error occurred retrieving the metadata
//gives the user the option to retry or cancel
function handleDataError(){
  if (displayInfo.dataDeferred) {
    giveRetryCancelMessage(
      "The server failed to deliver the metadata.",
      function() { //retry function
        getData();
      },
      function() { //cancel function
        //destroyPanel is provided elsewhere by our application
        destroyPanel(displayInfo);
      }
    )
  } else {
    //the process was canceled
  }
}
```

Finally, a cancel button could be placed on the initial panel that currently says “Retrieving Data” and a click handler connected that executes `displayInfo.dataDeferred.cancel()`.

That’s it for our display example. It’s fairly powerful. It has normal processing, full error recovery, and is user cancelable. The code is easy to read and understand while also being very concise. We could have wired all of this functionality together with events, but Dojo Deferred did most of the work for us so this was unnecessary. Here is the complete final code for you enjoyment!

[Download](#) asynchronous-programming/asynchronous-programming-deferred.js

```
//implements a display engine:
//display the data given by query in the panel given by panelId
function display(panelId, query) {

  //displayInfo is a shared bookkeeping area
  var displayInfo= {
    panelId: panelID,
    query: query,
    processCompleteCount: 0};

  //get an HTML div in which to place the panel
  displayInfo.div= allocatePanel(displayInfo);

  getData();
  getMetadata();

  function getData() {
```

```

displayInfo.dataDeferred= new dojo.Deferred(doCancel);
dataDeferred.addCallback(continueWithData);
dataDeferred.addErrback(handleDataError);
dojo.xhrGet({
  url: "data",
  content: query,
  handleAs: "json-comment-filtered",
  load: function(data) {
    deferred.callback(data);
  },
  error: function() {
    deferred.errback();
  }
});
}

function getMetadata() {
  displayInfo.metadataDeferred= new dojo.Deferred(doCancel);
  dataDeferred.addCallback(continueWithMetadata);
  dataDeferred.addErrback(handleMetadataError);
  dojo.xhrGet({
    url: "metadata",
    content: id,
    handleAs: "json-comment-filtered",
    load: function(data) {
      deferred.callback(data);
    },
    error: function() {
      deferred.errback();
    }
  });
}

//process the data as far as we can without the metadata
function continueWithData(data){
  //put the data in an application-wide cache
  //for use with other requests...
  cacheData(data);

  //save quick access to the data
  displayInfo.data= data;

  finishDisplay();
}

//process the metadata as far as we can without the data
function continueWithMetadata(metadata) {
  //start to set up the HTML and other control structures
  //given by the metadata...
  preparePanel(metadata);
}

```

```

//save quick access to the metadata
displayInfo.metadata= metadata;

finishDisplay();
}

//when continueWithData and continueWithMetadata have completed,
//finish with the display process
function finishDisplay() {
  displayInfo.processCompleteCount++;
  if (displayInfo.processCompleteCount==2) {
    //both have completed...
    populatePanelWithData(displayInfo);
    releasePanelToUser(displayInfo);
  }
}

function handleDataError(){
  if (displayInfo.dataDeferred) {
    giveRetryCancelMessage(
      "The server failed to deliver the data.",
      function() { //retry function
        getData();
      },
      function() { //cancel function
        doCancel(displayInfo.dataDeferred);
      }
    )
  } else {
    //the process was canceled
  }
}

function handleMetadataError(){
  if (displayInfo.metaDataDeferred) {
    giveRetryCancelMessage(
      "The server failed to deliver the metadata.",
      function() { //retry function
        getMetadata();
      },
      function() { //cancel function
        doCancel(displayInfo.metaDataDeferred);
      }
    )
  } else {
    //the process was canceled
  }
}

function doCancel(theDeferred) {

```

```

//execute this routine for the first Deferred canceled only...
if (!displayInfo.dataDeferred) {
    return;
}

//figure out which deferred was canceled and take
//a reference to the other deferred
var temp= displayInfo.dataDeferred==theDeferred ?
    displayInfo.metadataDeferred : displayInfo.dataDeferred;

//ensure this routine is executed only once
displayInfo.dataDeferred= null;
displayInfo.metadataDeferred= null;

//cancel the other Deferred
temp.cancel();

//clean up the display
destroyPanel(displayInfo);
}
}

```

If you find yourself having trouble with Deferreds, try not to worry too much about *how* they do things, but instead concentrate on what they do. Also, there's no need to bother with Deferreds *unless* the design includes problems like chaining callbacks and/or canceling. Deferreds shine in these situations, turning an unmanageable mess of spaghetti into clean code.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style, and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Mastering Dojo's Home Page

<http://pragprog.com/titles/rgdojo>

Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates

<http://pragprog.com/updates>

Be notified when updates and new books become available.

Join the Community

<http://pragprog.com/community>

Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy

<http://pragprog.com/news>

Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/rgdojo.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragprog.com/catalog
Customer Service:	orders@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com