**Promises**

For complicated projects, writing asynchronous code in plain callback style is hard to do correctly. It is easy to forget to check for an error or to allow an unexpected exception to cut the program short in a crude way. Additionally, arranging for correct error handling when the error has to flow through multiple callback functions and catch blocks is tedious.

There have been a lot of attempts to solve this with extra abstractions. One of the more successful ones is called *promises*. Promises wrap an asynchronous action in an object, which can be passed around and told to do certain things when the action finishes or fails. This interface is set to become part of the next version of the JavaScript language but can already be used as a library.

The interface for promises isn’t entirely intuitive, but it is powerful. This chapter will only roughly describe it. You can find a more thorough treatment at [*www.promisejs.org*](https://www.promisejs.org/).

To create a promise object, we call the Promise constructor, giving it a function that initializes the asynchronous action. The constructor calls that function, passing it two arguments, which are themselves functions. The first should be called when the action finishes successfully, and the second should be called when it fails.

Once again, here is our wrapper for GET requests, this time returning a promise. We’ll simply call it get this time.

function get(url) {

return new Promise(function(succeed, fail) {

var req = new XMLHttpRequest();

req.open("GET", url, true);

req.addEventListener("load", function() {

if (req.status < 400)

succeed(req.responseText);

else

fail(new Error("Request failed: " + req.statusText));

});

req.addEventListener("error", function() {

fail(new Error("Network error"));

});

req.send(null);

});

}

Note that the interface to the function itself is now a lot simpler. You give it a URL, and it returns a promise. That promise acts as a *handle* to the request’s outcome. It has a then method that you can call with two functions: one to handle success and one to handle failure.

get("example/data.txt").then(function(text) {

console.log("data.txt: " + text);

}, function(error) {

console.log("Failed to fetch data.txt: " + error);

});

So far, this is just another way to express the same thing we already expressed. It is only when you need to chain actions together that promises make a significant difference.

Calling then produces a new promise, whose result (the value passed to success handlers) depends on the return value of the first function we passed to then. This function may return another promise to indicate that more asynchronous work is being done. In this case, the promise returned by thenitself will wait for the promise returned by the handler function, succeeding or failing with the same value when it is resolved. When the handler function returns a nonpromise value, the promise returned by then immediately succeeds with that value as its result.

This means you can use then to transform the result of a promise. For example, this returns a promise whose result is the content of the given URL, parsed as JSON:

function getJSON(url) {

return get(url).then(JSON.parse);

}

That last call to then did not specify a failure handler. This is allowed. The error will be passed to the promise returned by then, which is exactly what we want—getJSON does not know what to do when something goes wrong, but hopefully its caller does.

As an example that shows the use of promises, we will build a program that fetches a number of JSON files from the server and, while it is doing that, shows the word *loading*. The JSON files contain information about people, with links to files that represent other people in properties such as father, mother, or spouse.

We want to get the name of the mother of the spouse of *example/bert.json*. And if something goes wrong, we want to remove the *loading* text and show an error message instead. Here is how that might be done with promises:

<script>

function showMessage(msg) {

var elt = document.createElement("div");

elt.textContent = msg;

return document.body.appendChild(elt);

}

var loading = showMessage("Loading...");

getJSON("example/bert.json").then(function(bert) {

return getJSON(bert.spouse);

}).then(function(spouse) {

return getJSON(spouse.mother);

}).then(function(mother) {

showMessage("The name is " + mother.name);

}).catch(function(error) {

showMessage(String(error));

}).then(function() {

document.body.removeChild(loading);

});

</script>

The resulting program is relatively compact and readable. The catch method is similar to then, except that it only expects a failure handler and will pass through the result unmodified in case of success. Much like with the catchclause for the try statement, control will continue as normal after the failure is caught. That way, the final then, which removes the loading message, is always executed, even if something went wrong.

You can think of the promise interface as implementing its own language for asynchronous control flow. The extra method calls and function expressions needed to achieve this make the code look somewhat awkward but not remotely as awkward as it would look if we took care of all the error handling ourselves.

**Appreciating HTTP**

When building a system that requires communication between a JavaScript program running in the browser (client-side) and a program on a server (server-side), there are several different ways to model this communication.

A commonly used model is that of *remote procedure calls*. In this model, communication follows the patterns of normal function calls, except that the function is actually running on another machine. Calling it involves making a request to the server that includes the function’s name and arguments. The response to that request contains the returned value.

When thinking in terms of remote procedure calls, HTTP is just a vehicle for communication, and you will most likely write an abstraction layer that hides it entirely.

Another approach is to build your communication around the concept of resources and HTTP methods. Instead of a remote procedure called addUser, you use a PUT request to /users/larry. Instead of encoding that user’s properties in function arguments, you define a document format or use an existing format that represents a user. The body of the PUT request to create a new resource is then simply such a document. A resource is fetched by making a GET request to the resource’s URL (for example, /user/larry), which returns the document representing the resource.

This second approach makes it easier to use some of the features that HTTP provides, such as support for caching resources (keeping a copy on the client side). It can also help the coherence of your interface since resources are easier to reason about than a jumble of functions.

**Security and HTTPS**

Data traveling over the Internet tends to follow a long, dangerous road. To get to its destination, it must hop through anything from coffee-shop Wi-Fi networks to networks controlled by various companies and states. At any point along its route it may be inspected or even modified.

If it is important that something remain secret, such as the password to your email account, or that it arrive at its destination unmodified, such as the account number you transfer money to from your bank’s website, plain HTTP is not good enough.

The secure HTTP protocol, whose URLs start with *https://*, wraps HTTP traffic in a way that makes it harder to read and tamper with. First, the client verifies that the server is who it claims to be by requiring that server to prove that it has a cryptographic certificate issued by a certificate authority that the browser recognizes. Next, all data going over the connection is encrypted in a way that should prevent eavesdropping and tampering.

Thus, when it works right, HTTPS prevents both the someone impersonating the website you were trying to talk to and the someone snooping on your communication. It is not perfect, and there have been various incidents where HTTPS failed because of forged or stolen certificates and broken software. Still, plain HTTP is trivial to mess with, whereas breaking HTTPS requires the kind of effort that only states or sophisticated criminal organizations can hope to make.

**Summary**

In this chapter, we saw that HTTP is a protocol for accessing resources over the Internet. A *client* sends a request, which contains a method (usually GET) and a path that identifies a resource. The *server* then decides what to do with the request and responds with a status code and a response body. Both requests and responses may contain headers that provide additional information.

Browsers make GET requests to fetch the resources needed to display a web page. A web page may also contain forms, which allow information entered by the user to be sent along in the request made when the form is submitted. You will learn more about that in the [next chapter](http://eloquentjavascript.net/18_forms.html#forms).

The interface through which browser JavaScript can make HTTP requests is called XMLHttpRequest. You can usually ignore the “XML” part of that name (but you still have to type it). There are two ways in which it can be used—synchronous, which blocks everything until the request finishes, and asynchronous, which requires an event handler to notice that the response came in. In almost all cases, asynchronous is preferable. Making a request looks like this:

var req = new XMLHttpRequest();

req.open("GET", "example/data.txt", true);

req.addEventListener("load", function() {

console.log(req.status);

});

req.send(null);

Asynchronous programming is tricky. *Promises* are an interface that makes it slightly easier by helping route error conditions and exceptions to the right handler and by abstracting away some of the more repetitive and error-prone elements in this style of programming.

**Exercises**

**Content negotiation**

One of the things that HTTP can do, but that we have not discussed in this chapter, is called *content negotiation*. The Accept header for a request can be used to tell the server what type of document the client would like to get. Many servers ignore this header, but when a server knows of various ways to encode a resource, it can look at this header and send the one that the client prefers.

The URL [*eloquentjavascript.net/author*](http://eloquentjavascript.net/author) is configured to respond with either plaintext, HTML, or JSON, depending on what the client asks for. These formats are identified by the standardized *media types* text/plain, text/html, and application/json.

Send requests to fetch all three formats of this resource. Use the setRequestHeader method of your XMLHttpRequest object to set the header named Accept to one of the media types given earlier. Make sure you set the header *after* calling open but before calling send.

Finally, try asking for the media type application/rainbows+unicorns and see what happens.

// Your code here.

**Waiting for multiple promises**

The Promise constructor has an all method that, given an array of promises, returns a promise that waits for all of the promises in the array to finish. It then succeeds, yielding an array of result values. If any of the promises in the array fail, the promise returned by all fails too (with the failure value from the failing promise).

Try to implement something like this yourself as a regular function called all.

Note that after a promise is resolved (has succeeded or failed), it can’t succeed or fail again, and further calls to the functions that resolve it are ignored. This can simplify the way you handle failure of your promise.

function all(promises) {

return new Promise(function(success, fail) {

// Your code here.

});

}

// Test code.

all([]).then(function(array) {

console.log("This should be []:", array);

});

function soon(val) {

return new Promise(function(success) {

setTimeout(function() { success(val); },

Math.random() \* 500);

});

}

all([soon(1), soon(2), soon(3)]).then(function(array) {

console.log("This should be [1, 2, 3]:", array);

});

function fail() {

return new Promise(function(success, fail) {

fail(new Error("boom"));

});

}

all([soon(1), fail(), soon(3)]).then(function(array) {

console.log("We should not get here");

}, function(error) {

if (error.message != "boom")

console.log("Unexpected failure:", error);

});