CHAPTER 17

HTTP

“The dream behind the Web is of a common information space in which we communicate by sharing information. Its universality is essential: the fact that a hypertext link can point to anything, be it personal, local or global, be it draft or highly polished.”

—Tim Berners-Lee, *The World Wide Web: A very short personal history*

The *Hypertext Transfer Protocol*, already mentioned in Chapter 12, is the mechanism through which data is requested and provided on the World Wide Web. This chapter describes the protocol in more detail and explains the way browser JavaScript has access to it.

The protocol

If you type *eloquentjavascript.net/17_http.html* into your browser’s address bar, the browser first looks up the address of the server associated with *eloquentjavascript.net* and tries to open a TCP connection to it on port 80, the default port for HTTP traffic. If the server exists and accepts the connection, the browser sends something like this:

```
GET /17_http.html HTTP/1.1
Host: eloquentjavascript.net
User-Agent: Your browser's name
```

Then the server responds, through that same connection.

```
HTTP/1.1 200 OK
Content-Length: 65585
Content-Type: text/html
Last-Modified: Wed, 09 Apr 2014 10:48:09 GMT

<!doctype html>
... the rest of the document
```

The browser then takes the part of the response after the blank line and displays it as an HTML document.

The information sent by the client is called the *request*. It starts with this line:

```
GET /17_http.html HTTP/1.1
```

The first word is the *method* of the request. `GET` means that we want to *get* the specified resource. Other common methods are `DELETE` to delete a resource, `PUT` to replace it, and `POST` to send information to it. Note that the server is not obliged to carry out every request it gets. If you walk up to a random website and tell it to `DELETE` its main page, it'll probably refuse.

The part after the method name is the path of the resource the request applies to. In the simplest case, a resource is simply a file on the server, but the protocol doesn't require it to be. A resource may be anything that can be transferred *as if* it is a file. Many servers generate the responses they produce on the fly. For example, if you open *twitter.com/marijnjh*, the server looks in its database for a user named *marijnjh*, and if it finds one, it will generate a profile page for that user.

After the resource path, the first line of the request mentions `HTTP/1.1` to indicate the version of the HTTP protocol it is using.

The server's response will start with a version as well, followed by the status of the response, first as a three-digit status code and then as a human-readable string.

```
HTTP/1.1 200 OK
```

Status codes starting with a 2 indicate that the request succeeded. Codes starting with 4 mean there was something wrong with the request. 404 is probably the most famous HTTP status code—it means that the resource that was requested could not be found. Codes that start with 5 mean an error happened on the server and the request is not to blame.

The first line of a request or response may be followed by any number of *headers*. These are lines in the form “name: value” that specify extra information about the request or response. These headers were part of the example response:

```
Content-Length: 65585
Content-Type: text/html
Last-Modified: Wed, 09 Apr 2014 10:48:09 GMT
```

This tells us the size and type of the response document. In this case, it is an HTML document of 65,585 bytes. It also tells us when that document was last modified.

For the most part, a client or server decides which headers to include in a request or response, though a few headers are required. For example, the `Host` header, which specifies the hostname, should be included in a request because a server might be serving multiple hostnames on a single IP address, and without that header, the server won’t know which host the client is trying to talk to.

After the headers, both requests and responses may include a blank line followed by a *body*, which contains the data being sent. `GET` and `DELETE` requests don’t send along any data, but `PUT` and `POST` requests do. Similarly, some response types, such as error responses, do not require a body.

Browsers and HTTP

As we saw in the example, a browser will make a request when we enter a URL in its address bar. When the resulting HTML page references other files, such as images and JavaScript files, those are also fetched.

A moderately complicated website can easily include anywhere from 10 to 200 resources. To be able to fetch those quickly, browsers will make several requests simultaneously, rather than waiting for the responses one at a time. Such documents are always fetched using `GET` requests.

HTML pages may include *forms*, which allow the user to fill out information and send it to the server. This is an example of a form:

```
<form method="GET" action="example/message.html">
  <p>Name: <input type="text" name="name"></p>
  <p>Message:<br><textarea name="message"></textarea></p>
  <p><button type="submit">Send</button></p>
</form>
```

This code describes a form with two fields: a small one asking for a name and a larger one to write a message in. When you click the Send button, the information in those fields will be encoded into a *query string*. When the `<form>` element's `method` attribute is `GET` (or is omitted), that query string is tacked onto the `action` URL, and the browser makes a `GET` request to that URL.

```
GET /example/message.html?name=Jean&message=Yes%3F HTTP/1.1
```

The start of a query string is indicated by a question mark. After that follow pairs of names and values, corresponding to the `name` attribute on the form field elements and the content of those elements, respectively. An ampersand character (`&`) is used to separate the pairs.

The actual message encoded in the previous URL is "Yes?", even though the question mark is replaced by a strange code. Some characters in query strings must be escaped. The question mark, represented as `%3F`, is one of those. There seems to be an unwritten rule that every format needs its own way of escaping characters. This one, called *URL encoding*, uses a percent sign followed by two hexadecimal digits that encode the character code. In this case, 3F, which is 63 in decimal notation, is the code of a question mark character. JavaScript provides the `encodeURIComponent` and `decodeURIComponent` functions to encode and decode this format.

```
console.log(encodeURIComponent("Hello & goodbye"));
// → Hello%20%26%20goodbye
```

```
console.log(decodeURIComponent("Hello%20%26%20goodbye"));
// → Hello & goodbye
```

If we change the `method` attribute of the HTML form in the example we saw earlier to `POST`, the HTTP request made to submit the form will use the `POST` method and put the query string in body of the request, rather than adding it to the URL.

```
POST /example/message.html HTTP/1.1
Content-length: 24
Content-type: application/x-www-form-urlencoded

name=Jean&message=Yes%3F
```

By convention, the `GET` method is used for requests that do not have side effects, such as doing a search. Requests that change something on the server, such as creating a new account or posting a message, should be expressed with other methods, such as `POST`. Client-side software, such as a browser, knows that it shouldn't blindly make `POST` requests but will often implicitly make `GET` requests—for example, to prefetch a resource it believes the user will soon need.

The next chapter will return to forms and talk about how we can script them with JavaScript.

XMLHttpRequest

The interface through which browser JavaScript can make HTTP requests is called `XMLHttpRequest` (note the inconsistent capitalization). It was designed by Microsoft, for its Internet Explorer browser, in the late 1990s. During this time, the XML file format was *very* popular in the world of business software—a world where Microsoft has always been at home. In fact, it was so popular that the acronym XML was tacked onto the front of the name of an interface for HTTP, which is in no way tied to XML.

The name isn't completely nonsensical, though. The interface allows you to parse response documents as XML if you want. Conflating two distinct concepts (making a request and parsing the response) into a single thing is terrible design, of course, but so it goes.

When the `XMLHttpRequest` interface was added to Internet Explorer, it allowed people to do things with JavaScript that had been very hard before. For example, websites started showing lists of suggestions when the user was typing something into a text field. The script would send the text to the server over HTTP as the user typed. The server, which had some database of possible inputs, would match the database entries against the partial input and send back possible completions to show the user. This was considered spectacular—people were used to waiting for a full page reload for every interaction with a website.

The other significant browser at that time, Mozilla (later Firefox), did not want to be left behind. To allow people to do similarly neat things in *its* browser, Mozilla copied the interface, including the bogus name. The next generation of browsers followed this example, and today `XMLHttpRequest` is a de facto standard interface.

Sending a request

To make a simple request, we create a request object with the `XMLHttpRequest` constructor and call its `open` and `send` methods.

```
var req = new XMLHttpRequest();
req.open("GET", "example/data.txt", false);
req.send(null);
console.log(req.responseText);
// → This is the content of data.txt
```

The `open` method configures the request. In this case, we choose to make a `GET` request for the *example/data.txt* file. URLs that don't start with a protocol name (such as *http:*) are relative, which means that they are interpreted relative to the current document. When they start with a slash (/), they replace the current path, which is the part after the server name. When they do not, the part of the current path up to and including its last slash character is put in front of the relative URL.

After opening the request, we can send it with the `send` method. The argument to send is the request body. For `GET` requests, we can pass `null`. If the third argument to `open` was `false`, `send` will return only after the response to our request was received. We can read the request object's `responseText` property to get the response body.

The other information included in the response can also be extracted from this object. The status code is accessible through the `status` property, and the human-readable status text is accessible through `statusText`. Headers can be read with `getResponseHeader`.

```
var req = new XMLHttpRequest();
req.open("GET", "example/data.txt", false);
req.send(null);
console.log(req.status, req.statusText);
// → 200 OK
console.log(req.getResponseHeader("content-type"));
// → text/plain
```

Header names are case-insensitive. They are usually written with a capital letter at the start of each word, such as "Content-Type", but "content-type" and "cOnTeNt-TyPe" refer to the same header.

The browser will automatically add some request headers, such as "Host" and those needed for the server to figure out the size of the body. But you can add your own headers with the `setRequestHeader` method. This is needed only for advanced uses and requires the cooperation of the server you are talking to—a server is free to ignore headers it does not know how to handle.

Asynchronous Requests

In the examples we saw, the request has finished when the call to `send` returns. This is convenient because it means properties such as `responseText` are available immediately. But it also means that our program is suspended as long as the browser and server are communicating. When the connection is bad, the server is slow, or the file is big, that might take quite a while. Worse, because no event handlers can fire while our program is suspended, the whole document will become unresponsive.

If we pass `true` as the third argument to `open`, the request is *asynchronous*. This means that when we call `send`, the only thing that happens right away is that the request is scheduled to be sent. Our program can continue, and the browser will take care of the sending and receiving of data in the background.

But as long as the request is running, we won't be able to access the response. We need a mechanism that will notify us when the data is available.

For this, we must listen for the `"load"` event on the request object.

```
var req = new XMLHttpRequest();
req.open("GET", "example/data.txt", true);
req.addEventListener("load", function() {
  console.log("Done:", req.status);
});
req.send(null);
```

Just like the use of `requestAnimationFrame` in Chapter 15, this forces us to use an asynchronous style of programming, wrapping the things that have to be done after the request in a function and arranging for that to be called at the appropriate time. We will come back to this later.

Fetching XML Data

When the resource retrieved by an `XMLHttpRequest` object is an XML document, the object's `responseXML` property will hold a parsed representation of this document. This representation works much like the DOM discussed in Chapter 13, except that it doesn't have HTML-specific functionality like the `style` property. The object that `responseXML` holds corresponds to the `document` object. Its `documentElement` property refers to the outer tag of the XML document. In the following document (*example/fruit.xml*), that would be the `<fruits>` tag:

```
<fruits>
  <fruit name="banana" color="yellow"/>
  <fruit name="lemon" color="yellow"/>
  <fruit name="cherry" color="red"/>
</fruits>
```

We can retrieve such a file like this:

```
var req = new XMLHttpRequest();
req.open("GET", "example/fruit.xml", false);
req.send(null);
console.log(req.responseXML.querySelectorAll("fruit").length);
// → 3
```

XML documents can be used to exchange structured information with the server. Their form—tags nested inside other tags—lends itself well to storing most types of data, or at least better than flat text files. The DOM interface is rather clumsy for extracting information, though, and XML documents tend to be verbose. It is often a better idea to communicate using JSON data, which is easier to read and write, both for programs and for humans.

```
var req = new XMLHttpRequest();
req.open("GET", "example/fruit.json", false);
req.send(null);
console.log(JSON.parse(req.responseText));
// → {banana: "yellow", lemon: "yellow", cherry: "red"}
```

HTTP SANDBOXING

Making HTTP requests in web page scripts once again raises concerns about security. The person who controls the script might not have the same interests as the person on whose computer it is running. More specifically, if I visit *themafia.org*, I do not want its scripts to be able to make a request to *mybank.com*, using identifying information from my browser, with instructions to transfer all my money to some random mafia account.

It is possible for websites to protect themselves against such attacks, but that requires effort, and many websites fail to do it. For this reason, browsers protect us by disallowing scripts to make HTTP requests to other *domains* (names such as *themafia.org* and *mybank.com*).

This can be an annoying problem when building systems that want to access several domains for legitimate reasons. Fortunately, servers can include a header like this in their response to explicitly indicate to browsers that it is okay for the request to come from other domains:

```
Access-Control-Allow-Origin: *
```

Abstracting requests

In Chapter 10, in our implementation of the AMD module system, we used a hypothetical function called `backgroundReadFile`. It took a filename and a function and called that function with the contents of the file when it had finished fetching it. Here's a simple implementation of that function:

```
function backgroundReadFile(url, callback) {
  var req = new XMLHttpRequest();
  req.open("GET", url, true);
  req.addEventListener("load", function() {
    if (req.status < 400)
      callback(req.responseText);
  });
  req.send(null);
}
```

This simple abstraction makes it easier to use `XMLHttpRequest` for simple `GET` requests. If you are writing a program that has to make HTTP requests, it is a good idea to use a helper function so that you don't end up repeating the ugly `XMLHttpRequest` pattern all through your code.

The function argument's name, `callback`, is a term that is often used to describe functions like this. A callback function is given to other code to provide that code with a way to "call us back" later.

It is not hard to write an HTTP utility function, tailored to what your application is doing. The previous one does only `GET` requests and doesn't give us control over the headers or the request body. You could write another variant for `POST` requests or a more generic one that supports various kinds of requests. Many JavaScript libraries also provide wrappers for `XMLHttpRequest`.

The main problem with the previous wrapper is its handling of failure. When the request returns a status code that indicates an error (400 and up), it does nothing. This might be okay, in some circumstances, but imagine we put a "loading" indicator on the page to indicate that we are fetching information. If the request fails because the server crashed or the connection is briefly interrupted, the page will just sit there, misleadingly looking like it is doing something. The user will wait for a while, get impatient, and consider the site uselessly flaky.

We should also have an option to be notified when the request fails so that we can take appropriate action. For example, we could remove the "loading" message and inform the user that something went wrong.

Error handling in asynchronous code is even trickier than error handling in synchronous code. Because we often need to defer part of our work, putting it in a callback function, the scope of a `try` block becomes meaningless. In the following code, the exception will *not* be caught because the call to `backgroundReadFile` returns immediately. Control then leaves the `try` block, and the function it was given won't be called until later.

```
try {
  backgroundReadFile("example/data.txt", function(text) {
    if (text != "expected")
      throw new Error("That was unexpected");
  });
} catch (e) {
  console.log("Hello from the catch block");
}
```

To handle failing requests, we have to allow an additional function to be passed to our wrapper and call that when a request goes wrong. Alternatively, we can use the convention that if the request fails, an additional argument describing the problem is passed to the regular callback function. Here's an example:

```
function getURL(url, callback) {
  var req = new XMLHttpRequest();
  req.open("GET", url, true);
  req.addEventListener("load", function() {
    if (req.status < 400)
      callback(req.responseText);
    else
      callback(null, new Error("Request failed: " +
                               req.statusText));
  });
  req.addEventListener("error", function() {
    callback(null, new Error("Network error"));
  });
  req.send(null);
}
```

We have added a handler for the `"error"` event, which will be signaled when the request fails entirely. We also call the callback function with an error argument when the request completes with a status code that indicates an error.

Code using `getURL` must then check whether an error was given and, if it finds one, handle it.

```
getURL("data/nonsense.txt", function(content, error) {
  if (error != null)
    console.log("Failed to fetch nonsense.txt: " + error);
  else
    console.log("nonsense.txt: " + content);
});
```

This does not help when it comes to exceptions. When chaining several asynchronous actions together, an exception at any point of the chain will still (unless you wrap each handling function in its own `try/catch` block) land at the top level and abort your chain of actions.

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*.

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 `then` itself 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 `catch` clause 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.

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* 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.
```

» Display hints...

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);
});
```

» Display hints...