```
    }
    gamestate.render();
};

// If the user goes back or forward in history, we'll get a popstate event
// on the window object with a copy of the state object we saved with
// pushState. When that happens, render the new state.
window.onpopstate = (event) => {
    gamestate = GameState.fromStateObject(event.state); // Restore the state
    gamestate.render();                                 // and display it
};
</script>
</body></html>
```

## 15.11 Networking

Every time you load a web page, the browser makes network requests—using the HTTP and HTTPS protocols—for an HTML file as well as the images, fonts, scripts, and stylesheets that the file depends on. But in addition to being able to make network requests in response to user actions, web browsers also expose JavaScript APIs for networking as well.

This section covers three network APIs:

- The `fetch()` method defines a Promise-based API for making HTTP and HTTPS requests. The `fetch()` API makes basic GET requests simple but has a comprehensive feature set that also supports just about any possible HTTP use case.
- The Server-Sent Events (or SSE) API is a convenient, event-based interface to HTTP "long polling" techniques where the web server holds the network connection open so that it can send data to the client whenever it wants.
- WebSockets is a networking protocol that is not HTTP but is designed to interoperate with HTTP. It defines an asynchronous message-passing API where clients and servers can send and receive messages from each other in a way that is similar to TCP network sockets.

### 15.11.1 fetch()

For basic HTTP requests, using `fetch()` is a three-step process:

1. Call `fetch()`, passing the URL whose content you want to retrieve.
2. Get the response object that is asynchronously returned by step 1 when the HTTP response begins to arrive and call a method of this response object to ask for the body of the response.

3. Get the body object that is asynchronously returned by step 2 and process it however you want.

The `fetch()` API is completely Promise-based, and there are two asynchronous steps here, so you typically expect two `then()` calls or two `await` expressions when using `fetch()`. (And if you've forgotten what those are, you may want to reread Chapter 13 before continuing with this section.)

Here's what a `fetch()` request looks like if you are using `then()` and expect the server's response to your request to be JSON-formatted:

```
fetch("/api/users/current")             // Make an HTTP (or HTTPS) GET request
    .then(response => response.json()) // Parse its body as a JSON object
    .then(currentUser => {             // Then process that parsed object
        displayUserInfo(currentUser);
    });
```

Here's a similar request made using the `async` and `await` keywords to an API that returns a plain string rather than a JSON object:

```
async function isServiceReady() {
    let response = await fetch("/api/service/status");
    let body = await response.text();
    return body === "ready";
}
```

If you understand these two code examples, then you know 80% of what you need to know to use the `fetch()` API. The subsections that follow will demonstrate how to make requests and receive responses that are somewhat more complicated than those shown here.

## Goodbye XMLHttpRequest

The `fetch()` API replaces the baroque and misleadingly named XMLHttpRequest API (which has nothing to do with XML). You may still see XHR (as it is often abbreviated) in existing code, but there is no reason today to use it in new code, and it is not documented in this chapter. There is one example of XMLHttpRequest in this book, however, and you can refer to §13.1.3 if you'd like to see an example of old-style JavaScript networking.

### HTTP status codes, response headers, and network errors

The three-step `fetch()` process shown in §15.11.1 elides all error-handling code. Here's a more realistic version:

```
fetch("/api/users/current")   // Make an HTTP (or HTTPS) GET request.
    .then(response => {       // When we get a response, first check it
        if (response.ok &&    // for a success code and the expected type.
```

```
        response.headers.get("Content-Type") === "application/json") {
        return response.json(); // Return a Promise for the body.
    } else {
        throw new Error(        // Or throw an error.
            `Unexpected response status ${response.status} or content type`
        );
    }
})
.then(currentUser => {    // When the response.json() Promise resolves
    displayUserInfo(currentUser); // do something with the parsed body.
})
.catch(error => {         // Or if anything went wrong, just log the error.
    // If the user's browser is offline, fetch() itself will reject.
    // If the server returns a bad response then we throw an error above.
    console.log("Error while fetching current user:", error);
});
```

The Promise returned by `fetch()` resolves to a Response object. The `status` property of this object is the HTTP status code, such as 200 for successful requests or 404 for "Not Found" responses. (`statusText` gives the standard English text that goes along with the numeric status code.) Conveniently, the `ok` property of a Response is `true` if `status` is 200 or any code between 200 and 299 and is `false` for any other code.

`fetch()` resolves its Promise when the server's response starts to arrive, as soon as the HTTP status and response headers are available, but typically before the full response body has arrived. Even though the body is not available yet, you can examine the headers in this second step of the fetch process. The `headers` property of a Response object is a Headers object. Use its `has()` method to test for the presence of a header, or use its `get()` method to get the value of a header. HTTP header names are case-insensitive, so you can pass lowercase or mixed-case header names to these functions.

The Headers object is also iterable if you ever need to do that:

```
fetch(url).then(response => {
    for(let [name,value] of response.headers) {
        console.log(`${name}: ${value}`);
    }
});
```

If a web server responds to your `fetch()` request, then the Promise that was returned will be fulfilled with a Response object, even if the server's response was a 404 Not Found error or a 500 Internal Server Error. `fetch()` only rejects the Promise it returns if it cannot contact the web server at all. This can happen if the user's computer is offline, the server is unresponsive, or the URL specifies a hostname that does not exist. Because these things can happen on any network request, it is always a good idea to include a `.catch()` clause any time you make a `fetch()` call.

### Setting request parameters

Sometimes you want to pass extra parameters along with the URL when you make a request. This can be done by adding name/value pairs at the end of a URL after a ?. The URL and URLSearchParams classes (which were covered in §11.9) make it easy to construct URLs in this form, and the fetch() function accepts URL objects as its first argument, so you can include request parameters in a fetch() request like this:

```
async function search(term) {
    let url = new URL("/api/search");
    url.searchParams.set("q", term);
    let response = await fetch(url);
    if (!response.ok) throw new Error(response.statusText);
    let resultsArray = await response.json();
    return resultsArray;
}
```

### Setting request headers

Sometimes you need to set headers in your fetch() requests. If you're making web API requests that require credentials, for example, then you may need to include an Authorization header that contains those credentials. In order to do this, you can use the two-argument version of fetch(). As before, the first argument is a string or URL object that specifies the URL to fetch. The second argument is an object that can provide additional options, including request headers:

```
let authHeaders = new Headers();
// Don't use Basic auth unless it is over an HTTPS connection.
authHeaders.set("Authorization",
                `Basic ${btoa(`${username}:${password}`)}`);
fetch("/api/users/", { headers: authHeaders })
    .then(response => response.json())          // Error handling omitted...
    .then(usersList => displayAllUsers(usersList));
```

There are a number of other options that can be specified in the second argument to fetch(), and we'll see it again later. An alternative to passing two arguments to fetch() is to instead pass the same two arguments to the Request() constructor and then pass the resulting Request object to fetch():

```
let request = new Request(url, { headers });
fetch(request).then(response => ...);
```

### Parsing response bodies

In the three-step fetch() process that we've demonstrated, the second step ends by calling the json() or text() methods of the Response object and returning the Promise object that those methods return. Then, the third step begins when that Promise resolves with the body of the response parsed as a JSON object or simply as a string of text.

These are probably the two most common scenarios, but they are not the only ways to obtain the body of a web server's response. In addition to `json()` and `text()`, the Response object also has these methods:

`arrayBuffer()`
: This method returns a Promise that resolves to an ArrayBuffer. This is useful when the response contains binary data. You can use the ArrayBuffer to create a typed array (§11.2) or a DataView object (§11.2.5) from which you can read the binary data.

`blob()`
: This method returns a Promise that resolves to a Blob object. Blobs are not covered in any detail in this book, but the name stands for "Binary Large Object," and they are useful when you expect large amounts of binary data. If you ask for the body of the response as a Blob, the browser implementation may stream the response data to a temporary file and then return a Blob object that represents that temporary file. Blob objects, therefore, do not allow random access to the response body the way that an ArrayBuffer does. Once you have a Blob, you can create a URL that refers to it with `URL.createObjectURL()`, or you can use the event-based FileReader API to asynchronously obtain the content of the Blob as a string or an ArrayBuffer. At the time of this writing, some browsers also define Promise-based `text()` and `arrayBuffer()` methods that give a more direct route for obtaining the content of a Blob.

`formData()`
: This method returns a Promise that resolves to a FormData object. You should use this method if you expect the body of the Response to be encoded in "multipart/form-data" format. This format is common in POST requests made to a server, but uncommon in server responses, so this method is not frequently used.

### Streaming response bodies

In addition to the five response methods that asynchronously return some form of the complete response body to you, there is also an option to stream the response body, which is useful if there is some kind of processing you can do on the chunks of the response body as they arrive over the network. But streaming the response is also useful if you want to display a progress bar so that the user can see the progress of the download.

The `body` property of a Response object is a ReadableStream object. If you have already called a response method like `text()` or `json()` that reads, parses, and returns the body, then `bodyUsed` will be `true` to indicate that the `body` stream has already been read. If `bodyUsed` is `false`, however, then the stream has not yet been read. In this case, you can call `getReader()` on `response.body` to obtain a stream