*"If you have knowledge, let others light their candles at it."*

—Margaret Fuller

CHAPTER 21

# Project: Skill-Sharing Website

A *skill-sharing* meeting is an event where people with a shared interest come together and give small, informal presentations about things they know. At a gardening skill-sharing meeting, someone might explain how to cultivate celery. Or in a programming skill-sharing group, you could drop by and tell people about Node.js.

Such meetups—also often called *users' groups* when they are about computers—are a great way to broaden your horizon, learn about new developments, or simply meet people with similar interests. Many larger cities have JavaScript meetups. They are typically free to attend, and I've found the ones I've visited to be friendly and welcoming.

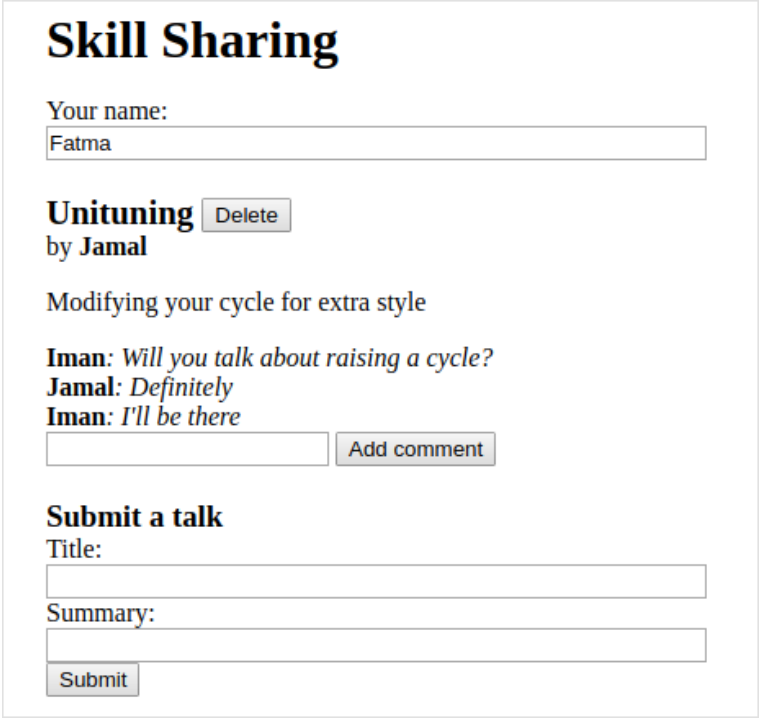
In this final project chapter, our goal is to set up a website for managing talks given at a skill-sharing meeting. Imagine a small group of people meeting up regularly in the office of one of the members to talk about unicycling. The previous organizer of the meetings moved to another town, and nobody stepped forward to take over this task. We want a system that will let the participants propose and discuss talks among themselves, without a central organizer.

The full code for the project can be downloaded from *https://eloquentjavascript.net/code/skillsharing.zip*.

## Design

There is a *server* part to this project, written for Node.js, and a *client* part, written for the browser. The server stores the system's data and provides it to the client. It also serves the files that implement the client-side system.

The server keeps the list of talks proposed for the next meeting, and the client shows this list. Each talk has a presenter name, a title, a summary, and an array of comments associated with it. The client allows users to propose new talks (adding them to the list), delete talks, and comment on existing talks. Whenever the user makes such a change, the client makes an HTTP request to tell the server about it.




The application will be set up to show a *live* view of the current proposed talks and their comments. Whenever someone, somewhere, submits a new talk or adds a comment, all people who have the page open in their browsers should immediately see the change. This poses a bit of a challenge—there is no way for a web server to open a connection to a client, nor is there a good way to know which clients are currently looking at a given website.

A common solution to this problem is called *long polling*, which happens to be one of the motivations for Node's design.

## Long polling

To be able to immediately notify a client that something changed, we need a connection to that client. Since web browsers do not traditionally accept connections and clients are often behind routers that would block such connections anyway, having the server initiate this connection is not practical.

We can arrange for the client to open the connection and keep it around so that the server can use it to send information when it needs to do so.

But an HTTP request allows only a simple flow of information: the client sends a request, the server comes back with a single response, and that is it. There is a technology called *WebSockets*, supported by modern browsers, that makes it possible to open connections for arbitrary data exchange. But using

them properly is somewhat tricky.

In this chapter, we use a simpler technique—long polling—where clients continuously ask the server for new information using regular HTTP requests, and the server stalls its answer when it has nothing new to report.

As long as the client makes sure it constantly has a polling request open, it will receive information from the server quickly after it becomes available. For example, if Fatma has our skill-sharing application open in her browser, that browser will have made a request for updates and will be waiting for a response to that request. When Iman submits a talk on Extreme Downhill Unicycling, the server will notice that Fatma is waiting for updates and send a response containing the new talk to her pending request. Fatma's browser will receive the data and update the screen to show the talk.

To prevent connections from timing out (being aborted because of a lack of activity), long polling techniques usually set a maximum time for each request, after which the server will respond anyway, even though it has nothing to report, after which the client will start a new request. Periodically restarting the request also makes the technique more robust, allowing clients to recover from temporary connection failures or server problems.

A busy server that is using long polling may have thousands of waiting requests, and thus TCP connections, open. Node, which makes it easy to manage many connections without creating a separate thread of control for each one, is a good fit for such a system.

## HTTP interface

Before we start designing either the server or the client, let's think about the point where they touch: the HTTP interface over which they communicate.

We will use JSON as the format of our request and response body. Like in the file server from Chapter 20, we'll try to make good use of HTTP methods and headers. The interface is centered around the `/talks` path. Paths that do not start with `/talks` will be used for serving static files—the HTML and JavaScript code for the client-side system.

A `GET` request to `/talks` returns a JSON document like this:

```
[{"title": "Unituning",
  "presenter": "Jamal",
  "summary": "Modifying your cycle for extra style",
  "comments": []}]
```

Creating a new talk is done by making a `PUT` request to a URL like `/talks/Unituning`, where the part after the second slash is the title of the talk. The `PUT` request's body should contain a JSON object that has `presenter` and `summary` properties.

Since talk titles may contain spaces and other characters that may not appear normally in a URL, title strings must be encoded with the `encodeURIComponent` function when building up such a URL.

```
console.log("/talks/" + encodeURIComponent("How to Idle"));
// → /talks/How%20to%20Idle
```

A request to create a talk about idling might look something like this:

```
PUT /talks/How%20to%20Idle HTTP/1.1
Content-Type: application/json
Content-Length: 92

{"presenter": "Maureen",
 "summary": "Standing still on a unicycle"}
```

Such URLs also support `GET` requests to retrieve the JSON representation of a talk and `DELETE` requests to delete a talk.

Adding a comment to a talk is done with a `POST` request to a URL like `/talks/Unituning/comments`, with a JSON body that has `author` and `message` properties.

```
POST /talks/Unituning/comments HTTP/1.1
Content-Type: application/json
Content-Length: 72

{"author": "Iman",
 "message": "Will you talk about raising a cycle?"}
```

To support long polling, `GET` requests to `/talks` may include extra headers that inform the server to delay the response if no new information is available. We'll use a pair of headers normally intended to manage caching: `ETag` and `If-None-Match`.

Servers may include an `ETag` ("entity tag") header in a response. Its value is a string that identifies the current version of the resource. Clients, when they later request that resource again, may make a *conditional request* by including an `If-None-Match` header whose value holds that same string. If the resource

hasn't changed, the server will respond with status code 304, which means "not modified", telling the client that its cached version is still current. When the tag does not match, the server responds as normal.

We need something like this, where the client can tell the server which version of the list of talks it has, and the server responds only when that list has changed. But instead of immediately returning a 304 response, the server should stall the response and return only when something new is available or a given amount of time has elapsed. To distinguish long polling requests from normal conditional requests, we give them another header, `Prefer: wait=90`, which tells the server that the client is willing to wait up to 90 seconds for the response.

The server will keep a version number that it updates every time the talks change and will use that as the `ETag` value. Clients can make requests like this to be notified when the talks change:

```
GET /talks HTTP/1.1
If-None-Match: "4"
Prefer: wait=90

(time passes)

HTTP/1.1 200 OK
Content-Type: application/json
ETag: "5"
Content-Length: 295

[....]
```

The protocol described here does not do any access control. Everybody can comment, modify talks, and even delete them. (Since the Internet is full of hooligans, putting such a system online without further protection probably wouldn't end well.)

## THE SERVER

Let's start by building the server-side part of the program. The code in this section runs on Node.js.

### Routing

Our server will use `createServer` to start an HTTP server. In the function that handles a new request, we must distinguish between the various kinds of requests (as determined by the method and the path) that we support. This can be done with a long chain of `if` statements, but there is a nicer way.

A *router* is a component that helps dispatch a request to the function that can handle it. You can tell the router, for example, that `PUT` requests with a path that matches the regular expression `/^\/talks\/([^\/]+)$/` (`/talks/` followed by a talk title) can be handled by a given function. In addition, it can help extract the meaningful parts of the path (in this case the talk title), wrapped in parentheses in the regular expression, and pass them to the handler function.

There are a number of good router packages on NPM, but here we'll write one ourselves to illustrate the principle.

This is `router.js`, which we will later `require` from our server module:

```
const {parse} = require("url");

module.exports = class Router {
  constructor() {
    this.routes = [];
  }
  add(method, url, handler) {
    this.routes.push({method, url, handler});
  }
  resolve(context, request) {
    let path = parse(request.url).pathname;

    for (let {method, url, handler} of this.routes) {
      let match = url.exec(path);
      if (!match || request.method != method) continue;
      let urlParts = match.slice(1).map(decodeURIComponent);
      return handler(context, ...urlParts, request);
    }
    return null;
  }
};
```

The module exports the `Router` class. A router object allows new handlers to be registered with the `add` method and can resolve requests with its `resolve` method.

The latter will return a response when a handler was found, and `null` other-

wise. It tries the routes one at a time (in the order in which they were defined) until a matching one is found.

The handler functions are called with the `context` value (which will be the server instance in our case), match strings for any groups they defined in their regular expression, and the request object. The strings have to be URL-decoded since the raw URL may contain `%20`-style codes.

### Serving files

When a request matches none of the request types defined in our router, the server must interpret it as a request for a file in the `public` directory. It would be possible to use the file server defined in Chapter 20 to serve such files, but we neither need nor want to support `PUT` and `DELETE` requests on files, and we would like to have advanced features such as support for caching. So let's use a solid, well-tested static file server from NPM instead.

I opted for `ecstatic`. This isn't the only such server on NPM, but it works well and fits our purposes. The `ecstatic` package exports a function that can be called with a configuration object to produce a request handler function. We use the `root` option to tell the server where it should look for files. The handler function accepts `request` and `response` parameters and can be passed directly to `createServer` to create a server that serves *only* files. We want to first check for requests that we should handle specially, though, so we wrap it in another function.

```
const {createServer} = require("http");
const Router = require("./router");
const ecstatic = require("ecstatic");

const router = new Router();
const defaultHeaders = {"Content-Type": "text/plain"};

class SkillShareServer {
  constructor(talks) {
    this.talks = talks;
    this.version = 0;
    this.waiting = [];

    let fileServer = ecstatic({root: "./public"});
    this.server = createServer((request, response) => {
      let resolved = router.resolve(this, request);
      if (resolved) {
        resolved.catch(error => {
          if (error.status != null) return error;
```

```
          return {body: String(error), status: 500};
        }).then(({body,
                  status = 200,
                  headers = defaultHeaders}) => {
          response.writeHead(status, headers);
          response.end(body);
        });
      } else {
        fileServer(request, response);
      }
    });
  }
  start(port) {
    this.server.listen(port);
  }
  stop() {
    this.server.close();
  }
}
```

This uses a similar convention as the file server from the previous chapter for responses—handlers return promises that resolve to objects describing the response. It wraps the server in an object that also holds its state.

### Talks as resources

The talks that have been proposed are stored in the `talks` property of the server, an object whose property names are the talk titles. These will be exposed as HTTP resources under `/talks/[title]`, so we need to add handlers to our router that implement the various methods that clients can use to work with them.

The handler for requests that `GET` a single talk must look up the talk and respond either with the talk's JSON data or with a 404 error response.

```
const talkPath = /^\/talks\/([^\/]+)$/;

router.add("GET", talkPath, async (server, title) => {
  if (title in server.talks) {
    return {body: JSON.stringify(server.talks[title]),
            headers: {"Content-Type": "application/json"}};
  } else {
    return {status: 404, body: `No talk '${title}' found`};
  }
```

```
});
```

Deleting a talk is done by removing it from the `talks` object.

```
router.add("DELETE", talkPath, async (server, title) => {
  if (title in server.talks) {
    delete server.talks[title];
    server.updated();
  }
  return {status: 204};
});
```

The `updated` method, which we will define later, notifies waiting long polling requests about the change.

To retrieve the content of a request body, we define a function called `readStream`, which reads all content from a readable stream and returns a promise that resolves to a string.

```
function readStream(stream) {
  return new Promise((resolve, reject) => {
    let data = "";
    stream.on("error", reject);
    stream.on("data", chunk => data += chunk.toString());
    stream.on("end", () => resolve(data));
  });
}
```

One handler that needs to read request bodies is the `PUT` handler, which is used to create new talks. It has to check whether the data it was given has `presenter` and `summary` properties, which are strings. Any data coming from outside the system might be nonsense, and we don't want to corrupt our internal data model or crash when bad requests come in.

If the data looks valid, the handler stores an object that represents the new talk in the `talks` object, possibly overwriting an existing talk with this title, and again calls `updated`.

```
router.add("PUT", talkPath,
           async (server, title, request) => {
  let requestBody = await readStream(request);
  let talk;
  try { talk = JSON.parse(requestBody); }
  catch (_) { return {status: 400, body: "Invalid JSON"}; }
```

```
  if (!talk ||
      typeof talk.presenter != "string" ||
      typeof talk.summary != "string") {
    return {status: 400, body: "Bad talk data"};
  }
  server.talks[title] = {title,
                         presenter: talk.presenter,
                         summary: talk.summary,
                         comments: []};
  server.updated();
  return {status: 204};
});
```

Adding a comment to a talk works similarly. We use `readStream` to get the content of the request, validate the resulting data, and store it as a comment when it looks valid.

```
router.add("POST", /^\/talks\/([^\/]+)\/comments$/,
           async (server, title, request) => {
  let requestBody = await readStream(request);
  let comment;
  try { comment = JSON.parse(requestBody); }
  catch (_) { return {status: 400, body: "Invalid JSON"}; }

  if (!comment ||
      typeof comment.author != "string" ||
      typeof comment.message != "string") {
    return {status: 400, body: "Bad comment data"};
  } else if (title in server.talks) {
    server.talks[title].comments.push(comment);
    server.updated();
    return {status: 204};
  } else {
    return {status: 404, body: `No talk '${title}' found`};
  }
});
```

Trying to add a comment to a nonexistent talk returns a 404 error.

### Long polling support

The most interesting aspect of the server is the part that handles long polling. When a `GET` request comes in for `/talks`, it may be either a regular request or a long polling request.

There will be multiple places in which we have to send an array of talks to the client, so we first define a helper method that builds up such an array and includes an `ETag` header in the response.

```
SkillShareServer.prototype.talkResponse = function() {
  let talks = [];
  for (let title of Object.keys(this.talks)) {
    talks.push(this.talks[title]);
  }
  return {
    body: JSON.stringify(talks),
    headers: {"Content-Type": "application/json",
              "ETag": `"${this.version}"`,
              "Cache-Control": "no-store"}
  };
};
```

The handler itself needs to look at the request headers to see whether `If-None-Match` and `Prefer` headers are present. Node stores headers, whose names are specified to be case insensitive, under their lowercase names.

```
router.add("GET", /^\/talks$/, async (server, request) => {
  let tag = /"(.*)"/.exec(request.headers["if-none-match"]);
  let wait = /\bwait=(\d+)/.exec(request.headers["prefer"]);
  if (!tag || tag[1] != server.version) {
    return server.talkResponse();
  } else if (!wait) {
    return {status: 304};
  } else {
    return server.waitForChanges(Number(wait[1]));
  }
});
```

If no tag was given or a tag was given that doesn't match the server's current version, the handler responds with the list of talks. If the request is conditional and the talks did not change, we consult the `Prefer` header to see whether we should delay the response or respond right away.

Callback functions for delayed requests are stored in the server's `waiting` ar-

ray so that they can be notified when something happens. The `waitForChanges` method also immediately sets a timer to respond with a 304 status when the request has waited long enough.

```
SkillShareServer.prototype.waitForChanges = function(time) {
  return new Promise(resolve => {
    this.waiting.push(resolve);
    setTimeout(() => {
      if (!this.waiting.includes(resolve)) return;
      this.waiting = this.waiting.filter(r => r != resolve);
      resolve({status: 304});
    }, time * 1000);
  });
};
```

Registering a change with `updated` increases the `version` property and wakes up all waiting requests.

```
SkillShareServer.prototype.updated = function() {
  this.version++;
  let response = this.talkResponse();
  this.waiting.forEach(resolve => resolve(response));
  this.waiting = [];
};
```

That concludes the server code. If we create an instance of `SkillShareServer` and start it on port 8000, the resulting HTTP server serves files from the `public` subdirectory alongside a talk-managing interface under the `/talks` URL.

```
new SkillShareServer(Object.create(null)).start(8000);
```

## The client

The client-side part of the skill-sharing website consists of three files: a tiny HTML page, a style sheet, and a JavaScript file.

### HTML

It is a widely used convention for web servers to try to serve a file named `index.html` when a request is made directly to a path that corresponds to a

directory. The file server module we use, `ecstatic`, supports this convention. When a request is made to the path `/`, the server looks for the file `./public/index.html` (`./public` being the root we gave it) and returns that file if found.

Thus, if we want a page to show up when a browser is pointed at our server, we should put it in `public/index.html`. This is our index file:

```
<!doctype html>
<meta charset="utf-8">
<title>Skill Sharing</title>
<link rel="stylesheet" href="skillsharing.css">

<h1>Skill Sharing</h1>

<script src="skillsharing_client.js"></script>
```

It defines the document title and includes a style sheet, which defines a few styles to, among other things, make sure there is some space between talks.

At the bottom, it adds a heading at the top of the page and loads the script that contains the client-side application.

### Actions

The application state consists of the list of talks and the name of the user, and we'll store it in a `{talks, user}` object. We don't allow the user interface to directly manipulate the state or send off HTTP requests. Rather, it may emit *actions* that describe what the user is trying to do.

The `handleAction` function takes such an action and makes it happen. Because our state updates are so simple, state changes are handled in the same function.

```
function handleAction(state, action) {
  if (action.type == "setUser") {
    localStorage.setItem("userName", action.user);
    return Object.assign({}, state, {user: action.user});
  } else if (action.type == "setTalks") {
    return Object.assign({}, state, {talks: action.talks});
  } else if (action.type == "newTalk") {
    fetchOK(talkURL(action.title), {
      method: "PUT",
      headers: {"Content-Type": "application/json"},
      body: JSON.stringify({
        presenter: state.user,
        summary: action.summary
```

```
      })
    }).catch(reportError);
  } else if (action.type == "deleteTalk") {
    fetchOK(talkURL(action.talk), {method: "DELETE"})
      .catch(reportError);
  } else if (action.type == "newComment") {
    fetchOK(talkURL(action.talk) + "/comments", {
      method: "POST",
      headers: {"Content-Type": "application/json"},
      body: JSON.stringify({
        author: state.user,
        message: action.message
      })
    }).catch(reportError);
  }
  return state;
}
```

We'll store the user's name in `localStorage` so that it can be restored when the page is loaded.

The actions that need to involve the server make network requests, using `fetch`, to the HTTP interface described earlier. We use a wrapper function, `fetchOK`, which makes sure the returned promise is rejected when the server returns an error code.

```
function fetchOK(url, options) {
  return fetch(url, options).then(response => {
    if (response.status < 400) return response;
    else throw new Error(response.statusText);
  });
}
```

This helper function is used to build up a URL for a talk with a given title.

```
function talkURL(title) {
  return "talks/" + encodeURIComponent(title);
}
```

When the request fails, we don't want to have our page just sit there, doing nothing without explanation. So we define a function called `reportError`, which at least shows the user a dialog that tells them something went wrong.

```
function reportError(error) {
  alert(String(error));
}
```

### Rendering components

We'll use an approach similar to the one we saw in Chapter 19, splitting the application into components. But since some of the components either never need to update or are always fully redrawn when updated, we'll define those not as classes but as functions that directly return a DOM node. For example, here is a component that shows the field where the user can enter their name:

```
function renderUserField(name, dispatch) {
  return elt("label", {}, "Your name: ", elt("input", {
    type: "text",
    value: name,
    onchange(event) {
      dispatch({type: "setUser", user: event.target.value});
    }
  }));
}
```

The `elt` function used to construct DOM elements is the one we used in Chapter 19.

A similar function is used to render talks, which include a list of comments and a form for adding a new comment.

```
function renderTalk(talk, dispatch) {
  return elt(
    "section", {className: "talk"},
    elt("h2", null, talk.title, " ", elt("button", {
      type: "button",
      onclick() {
        dispatch({type: "deleteTalk", talk: talk.title});
      }
    }, "Delete")),
    elt("div", null, "by ",
        elt("strong", null, talk.presenter)),
    elt("p", null, talk.summary),
    ...talk.comments.map(renderComment),
    elt("form", {
      onsubmit(event) {
```

```
        event.preventDefault();
        let form = event.target;
        dispatch({type: "newComment",
                  talk: talk.title,
                  message: form.elements.comment.value});
        form.reset();
      }
    }, elt("input", {type: "text", name: "comment"}), " ",
       elt("button", {type: "submit"}, "Add comment")));
}
```

The `"submit"` event handler calls `form.reset` to clear the form's content after creating a `"newComment"` action.

When creating moderately complex pieces of DOM, this style of programming starts to look rather messy. There's a widely used (non-standard) JavaScript extension called *JSX* that lets you write HTML directly in your scripts, which can make such code prettier (depending on what you consider pretty). Before you can actually run such code, you have to run a program on your script to convert the pseudo-HTML into JavaScript function calls much like the ones we use here.

Comments are simpler to render.

```
function renderComment(comment) {
  return elt("p", {className: "comment"},
             elt("strong", null, comment.author),
             ": ", comment.message);
}
```

Finally, the form that the user can use to create a new talk is rendered like this:

```
function renderTalkForm(dispatch) {
  let title = elt("input", {type: "text"});
  let summary = elt("input", {type: "text"});
  return elt("form", {
    onsubmit(event) {
      event.preventDefault();
      dispatch({type: "newTalk",
                title: title.value,
                summary: summary.value});
      event.target.reset();
    }
  }, elt("h3", null, "Submit a Talk"),
```

```
      elt("label", null, "Title: ", title),
      elt("label", null, "Summary: ", summary),
      elt("button", {type: "submit"}, "Submit"));
}
```

### Polling

To start the app we need the current list of talks. Since the initial load is closely related to the long polling process—the `ETag` from the load must be used when polling—we'll write a function that keeps polling the server for `/talks` and calls a callback function when a new set of talks is available.

```
async function pollTalks(update) {
  let tag = undefined;
  for (;;) {
    let response;
    try {
      response = await fetchOK("/talks", {
        headers: tag && {"If-None-Match": tag,
                         "Prefer": "wait=90"}
      });
    } catch (e) {
      console.log("Request failed: " + e);
      await new Promise(resolve => setTimeout(resolve, 500));
      continue;
    }
    if (response.status == 304) continue;
    tag = response.headers.get("ETag");
    update(await response.json());
  }
}
```

This is an `async` function so that looping and waiting for the request is easier. It runs an infinite loop that, on each iteration, retrieves the list of talks—either normally or, if this isn't the first request, with the headers included that make it a long polling request.

When a request fails, the function waits a moment and then tries again. This way, if your network connection goes away for a while and then comes back, the application can recover and continue updating. The promise resolved via `setTimeout` is a way to force the `async` function to wait.

When the server gives back a 304 response, that means a long polling request

timed out, so the function should just immediately start the next request. If the response is a normal 200 response, its body is read as JSON and passed to the callback, and its `ETag` header value is stored for the next iteration.

### The application

The following component ties the whole user interface together:

```
class SkillShareApp {
  constructor(state, dispatch) {
    this.dispatch = dispatch;
    this.talkDOM = elt("div", {className: "talks"});
    this.dom = elt("div", null,
                   renderUserField(state.user, dispatch),
                   this.talkDOM,
                   renderTalkForm(dispatch));
    this.syncState(state);
  }

  syncState(state) {
    if (state.talks != this.talks) {
      this.talkDOM.textContent = "";
      for (let talk of state.talks) {
        this.talkDOM.appendChild(
          renderTalk(talk, this.dispatch));
      }
      this.talks = state.talks;
    }
  }
}
```

When the talks change, this component redraws all of them. This is simple but also wasteful. We'll get back to that in the exercises.

We can start the application like this:

```
function runApp() {
  let user = localStorage.getItem("userName") || "Anon";
  let state, app;
  function dispatch(action) {
    state = handleAction(state, action);
    app.syncState(state);
  }

  pollTalks(talks => {
```

```
    if (!app) {
      state = {user, talks};
      app = new SkillShareApp(state, dispatch);
      document.body.appendChild(app.dom);
    } else {
      dispatch({type: "setTalks", talks});
    }
  }).catch(reportError);
}

runApp();
```

If you run the server and open two browser windows for *http://localhost:8000* next to each other, you can see that the actions you perform in one window are immediately visible in the other.

## Exercises

The following exercises will involve modifying the system defined in this chapter. To work on them, make sure you download the code first (*https://eloquentjavascript.net/code/skillsharing.zip*), have Node installed *https://nodejs.org*, and have installed the project's dependency with `npm install`.

### Disk persistence

The skill-sharing server keeps its data purely in memory. This means that when it crashes or is restarted for any reason, all talks and comments are lost.

Extend the server so that it stores the talk data to disk and automatically reloads the data when it is restarted. Do not worry about efficiency—do the simplest thing that works.

### Comment field resets

The wholesale redrawing of talks works pretty well because you usually can't tell the difference between a DOM node and its identical replacement. But there are exceptions. If you start typing something in the comment field for a talk in one browser window and then, in another, add a comment to that talk, the field in the first window will be redrawn, removing both its content and its focus.

In a heated discussion, where multiple people are adding comments at the same time, this would be annoying. Can you come up with a way to solve it?

# Exercise Hints

The hints below might help when you are stuck with one of the exercises in this book. They don't give away the entire solution, but rather try to help you find it yourself.

## Program Structure

### Looping a triangle

You can start with a program that prints out the numbers 1 to 7, which you can derive by making a few modifications to the even number printing example given earlier in the chapter, where the `for` loop was introduced.

Now consider the equivalence between numbers and strings of hash characters. You can go from 1 to 2 by adding 1 (`+= 1`). You can go from `"#"` to `"##"` by adding a character (`+= "#"`). Thus, your solution can closely follow the number-printing program.

### FizzBuzz

Going over the numbers is clearly a looping job, and selecting what to print is a matter of conditional execution. Remember the trick of using the remainder (`%`) operator for checking whether a number is divisible by another number (has a remainder of zero).

In the first version, there are three possible outcomes for every number, so you'll have to create an `if`/`else if`/`else` chain.

The second version of the program has a straightforward solution and a clever one. The simple solution is to add another conditional "branch" to precisely test the given condition. For the clever solution, build up a string containing the word or words to output and print either this word or the number if there is no word, potentially by making good use of the `||` operator.

### Chessboard

You can build the string by starting with an empty one (`""`) and repeatedly adding characters. A newline character is written `"\n"`.

To work with two dimensions, you will need a loop inside of a loop. Put braces around the bodies of both loops to make it easy to see where they start and end. Try to properly indent these bodies. The order of the loops must follow the order in which we build up the string (line by line, left to right, top to bottom). So the outer loop handles the lines, and the inner loop handles the characters on a line.

You'll need two bindings to track your progress. To know whether to put a space or a hash sign at a given position, you could test whether the sum of the two counters is even (`% 2`).

Terminating a line by adding a newline character must happen after the line has been built up, so do this after the inner loop but inside the outer loop.

## Functions

### Minimum

If you have trouble putting braces and parentheses in the right place to get a valid function definition, start by copying one of the examples in this chapter and modifying it.

A function may contain multiple `return` statements.

### Recursion

Your function will likely look somewhat similar to the inner `find` function in the recursive `findSolution` example in this chapter, with an `if`/`else if`/`else` chain that tests which of the three cases applies. The final `else`, corresponding to the third case, makes the recursive call. Each of the branches should contain a `return` statement or in some other way arrange for a specific value to be returned.

When given a negative number, the function will recurse again and again, passing itself an ever more negative number, thus getting further and further away from returning a result. It will eventually run out of stack space and abort.

### Bean counting

Your function will need a loop that looks at every character in the string. It can run an index from zero to one below its length (`< string.length`). If the character at the current position is the same as the one the function is looking for, it adds 1 to a counter variable. Once the loop has finished, the counter can be returned.

Take care to make all the bindings used in the function *local* to the function by properly declaring them with the `let` or `const` keyword.

## Data Structures: Objects and Arrays

### The sum of a range

Building up an array is most easily done by first initializing a binding to `[]` (a fresh, empty array) and repeatedly calling its `push` method to add a value. Don't forget to return the array at the end of the function.

Since the end boundary is inclusive, you'll need to use the `<=` operator rather than `<` to check for the end of your loop.

The step parameter can be an optional parameter that defaults (using the `=` operator) to 1.

Having `range` understand negative step values is probably best done by writing two separate loops—one for counting up and one for counting down—because the comparison that checks whether the loop is finished needs to be `>=` rather than `<=` when counting downward.

It might also be worthwhile to use a different default step, namely, `-1`, when the end of the range is smaller than the start. That way, `range(5, 2)` returns something meaningful, rather than getting stuck in an infinite loop. It is possible to refer to previous parameters in the default value of a parameter.

### Reversing an array

There are two obvious ways to implement `reverseArray`. The first is to simply go over the input array from front to back and use the `unshift` method on the new array to insert each element at its start. The second is to loop over the input array backwards and use the `push` method. Iterating over an array backwards requires a (somewhat awkward) `for` specification, like `(let i = array.length - 1; i >= 0; i--)`.

Reversing the array in place is harder. You have to be careful not to overwrite elements that you will later need. Using `reverseArray` or otherwise copying

the whole array (`array.slice(0)` is a good way to copy an array) works but is cheating.

The trick is to *swap* the first and last elements, then the second and second-to-last, and so on. You can do this by looping over half the length of the array (use `Math.floor` to round down—you don't need to touch the middle element in an array with an odd number of elements) and swapping the element at position `i` with the one at position `array.length - 1 - i`. You can use a local binding to briefly hold on to one of the elements, overwrite that one with its mirror image, and then put the value from the local binding in the place where the mirror image used to be.

### A list

Building up a list is easier when done back to front. So `arrayToList` could iterate over the array backwards (see the previous exercise) and, for each element, add an object to the list. You can use a local binding to hold the part of the list that was built so far and use an assignment like `list = {value: X, rest: list}` to add an element.

To run over a list (in `listToArray` and `nth`), a `for` loop specification like this can be used:

```
for (let node = list; node; node = node.rest) {}
```

Can you see how that works? Every iteration of the loop, `node` points to the current sublist, and the body can read its `value` property to get the current element. At the end of an iteration, `node` moves to the next sublist. When that is null, we have reached the end of the list, and the loop is finished.

The recursive version of `nth` will, similarly, look at an ever smaller part of the "tail" of the list and at the same time count down the index until it reaches zero, at which point it can return the `value` property of the node it is looking at. To get the zeroth element of a list, you simply take the `value` property of its head node. To get element $N + 1$, you take the $N$th element of the list that's in this list's `rest` property.

### Deep comparison

Your test for whether you are dealing with a real object will look something like `typeof x == "object" && x != null`. Be careful to compare properties only when *both* arguments are objects. In all other cases you can just immediately return the result of applying `===`.

Use `Object.keys` to go over the properties. You need to test whether both objects have the same set of property names and whether those properties have identical values. One way to do that is to ensure that both objects have the same number of properties (the lengths of the property lists are the same). And then, when looping over one of the object's properties to compare them, always first make sure the other actually has a property by that name. If they have the same number of properties and all properties in one also exist in the other, they have the same set of property names.

Returning the correct value from the function is best done by immediately returning false when a mismatch is found and returning true at the end of the function.

## Higher-Order Functions

### Everything

Like the `&&` operator, the `every` method can stop evaluating further elements as soon as it has found one that doesn't match. So the loop-based version can jump out of the loop—with `break` or `return`—as soon as it runs into an element for which the predicate function returns false. If the loop runs to its end without finding such an element, we know that all elements matched and we should return true.

To build `every` on top of `some`, we can apply *De Morgan's laws*, which state that `a && b` equals `!(!a || !b)`. This can be generalized to arrays, where all elements in the array match if there is no element in the array that does not match.

### Dominant writing direction

Your solution might look a lot like the first half of the `textScripts` example. You again have to count characters by a criterion based on `characterScript` and then filter out the part of the result that refers to uninteresting (script-less) characters.

Finding the direction with the highest character count can be done with `reduce`. If it's not clear how, refer to the example earlier in the chapter, where `reduce` was used to find the script with the most characters.

# The Secret Life of Objects

## A vector type

Look back to the `Rabbit` class example if you're unsure how `class` declarations look.

Adding a getter property to the constructor can be done by putting the word `get` before the method name. To compute the distance from (0, 0) to (x, y), you can use the Pythagorean theorem, which says that the square of the distance we are looking for is equal to the square of the x-coordinate plus the square of the y-coordinate. Thus, $\sqrt{x^2 + y^2}$ is the number you want, and `Math.sqrt` is the way you compute a square root in JavaScript.

## Groups

The easiest way to do this is to store an array of group members in an instance property. The `includes` or `indexOf` methods can be used to check whether a given value is in the array.

Your class's constructor can set the member collection to an empty array. When `add` is called, it must check whether the given value is in the array or add it, for example with `push`, otherwise.

Deleting an element from an array, in `delete`, is less straightforward, but you can use `filter` to create a new array without the value. Don't forget to overwrite the property holding the members with the newly filtered version of the array.

The `from` method can use a `for/of` loop to get the values out of the iterable object and call `add` to put them into a newly created group.

## Iterable groups

It is probably worthwhile to define a new class `GroupIterator`. Iterator instances should have a property that tracks the current position in the group. Every time `next` is called, it checks whether it is done and, if not, moves past the current value and returns it.

The `Group` class itself gets a method named by `Symbol.iterator` that, when called, returns a new instance of the iterator class for that group.

## Borrowing a method

Remember that methods that exist on plain objects come from `Object.prototype` .

Also remember that you can call a function with a specific `this` binding by using its `call` method.

## Project: A Robot

### Measuring a robot

You'll have to write a variant of the `runRobot` function that, instead of logging the events to the console, returns the number of steps the robot took to complete the task.

Your measurement function can then, in a loop, generate new states and count the steps each of the robots takes. When it has generated enough measurements, it can use `console.log` to output the average for each robot, which is the total number of steps taken divided by the number of measurements.

### Robot efficiency

The main limitation of `goalOrientedRobot` is that it considers only one parcel at a time. It will often walk back and forth across the village because the parcel it happens to be looking at happens to be at the other side of the map, even if there are others much closer.

One possible solution would be to compute routes for all packages and then take the shortest one. Even better results can be obtained, if there are multiple shortest routes, by preferring the ones that go to pick up a package instead of delivering a package.

### Persistent group

The most convenient way to represent the set of member values is still as an array since arrays are easy to copy.

When a value is added to the group, you can create a new group with a copy of the original array that has the value added (for example, using `concat`). When a value is deleted, you filter it from the array.

The class's constructor can take such an array as argument and store it as the instance's (only) property. This array is never updated.

To add a property (`empty`) to a constructor that is not a method, you have to add it to the constructor after the class definition, as a regular property.

You need only one `empty` instance because all empty groups are the same and instances of the class don't change. You can create many different groups from that single empty group without affecting it.

# Bugs and Errors

## Retry

The call to `primitiveMultiply` should definitely happen in a `try` block. The corresponding `catch` block should rethrow the exception when it is not an instance of `MultiplicatorUnitFailure` and ensure the call is retried when it is.

To do the retrying, you can either use a loop that stops only when a call succeeds—as in the `look` example earlier in this chapter—or use recursion and hope you don't get a string of failures so long that it overflows the stack (which is a pretty safe bet).

## The locked box

This exercise calls for a `finally` block. Your function should first unlock the box and then call the argument function from inside a `try` body. The `finally` block after it should lock the box again.

To make sure we don't lock the box when it wasn't already locked, check its lock at the start of the function and unlock and lock it only when it started out locked.

# Regular Expressions

## Quoting style

The most obvious solution is to replace only quotes with a nonword character on at least one side—something like `/\W'|'\W/`. But you also have to take the start and end of the line into account.

In addition, you must ensure that the replacement also includes the characters that were matched by the `\W` pattern so that those are not dropped. This can be done by wrapping them in parentheses and including their groups in the replacement string (`$1`, `$2`). Groups that are not matched will be replaced by nothing.

## Numbers again

First, do not forget the backslash in front of the period.

Matching the optional sign in front of the number, as well as in front of the exponent, can be done with `[+\-]?` or `(\+|-|)` (plus, minus, or nothing).

The more complicated part of the exercise is the problem of matching both `"5."` and `".5"` without also matching `"."`. For this, a good solution is to use

the | operator to separate the two cases—either one or more digits optionally followed by a dot and zero or more digits *or* a dot followed by one or more digits.

Finally, to make the *e* case insensitive, either add an `i` option to the regular expression or use `[eE]`.

# MODULES

## A MODULAR ROBOT

Here's what I would have done (but again, there is no single *right* way to design a given module):

The code used to build the road graph lives in the `graph` module. Because I'd rather use `dijkstrajs` from NPM than our own pathfinding code, we'll make this build the kind of graph data that `dijkstrajs` expects. This module exports a single function, `buildGraph`. I'd have `buildGraph` accept an array of two-element arrays, rather than strings containing hyphens, to make the module less dependent on the input format.

The `roads` module contains the raw road data (the `roads` array) and the `roadGraph` binding. This module depends on `./graph` and exports the road graph.

The `VillageState` class lives in the `state` module. It depends on the `./roads` module because it needs to be able to verify that a given road exists. It also needs `randomPick`. Since that is a three-line function, we could just put it into the `state` module as an internal helper function. But `randomRobot` needs it too. So we'd have to either duplicate it or put it into its own module. Since this function happens to exist on NPM in the `random-item` package, a good solution is to just make both modules depend on that. We can add the `runRobot` function to this module as well, since it's small and closely related to state management. The module exports both the `VillageState` class and the `runRobot` function.

Finally, the robots, along with the values they depend on such as `mailRoute`, could go into an `example-robots` module, which depends on `./roads` and exports the robot functions. To make it possible for `goalOrientedRobot` to do route-finding, this module also depends on `dijkstrajs`.

By offloading some work to NPM modules, the code became a little smaller. Each individual module does something rather simple and can be read on its own. Dividing code into modules also often suggests further improvements to the program's design. In this case, it seems a little odd that the `VillageState` and the robots depend on a specific road graph. It might be a better idea to

make the graph an argument to the state's constructor and make the robots read it from the state object—this reduces dependencies (which is always good) and makes it possible to run simulations on different maps (which is even better).

Is it a good idea to use NPM modules for things that we could have written ourselves? In principle, yes—for nontrivial things like the pathfinding function you are likely to make mistakes and waste time writing them yourself. For tiny functions like `random-item`, writing them yourself is easy enough. But adding them wherever you need them does tend to clutter your modules.

However, you should also not underestimate the work involved in *finding* an appropriate NPM package. And even if you find one, it might not work well or may be missing some feature you need. On top of that, depending on NPM packages means you have to make sure they are installed, you have to distribute them with your program, and you might have to periodically upgrade them.

So again, this is a trade-off, and you can decide either way depending on how much the packages help you.

### Roads module

Since this is a CommonJS module, you have to use `require` to import the graph module. That was described as exporting a `buildGraph` function, which you can pick out of its interface object with a destructuring `const` declaration.

To export `roadGraph`, you add a property to the `exports` object. Because `buildGraph` takes a data structure that doesn't precisely match `roads`, the splitting of the road strings must happen in your module.

### Circular dependencies

The trick is that `require` adds modules to its cache *before* it starts loading the module. That way, if any `require` call made while it is running tries to load it, it is already known, and the current interface will be returned, rather than starting to load the module once more (which would eventually overflow the stack).

If a module overwrites its `module.exports` value, any other module that has received its interface value before it finished loading will have gotten hold of the default interface object (which is likely empty), rather than the intended interface value.

### Tracking the scalpel

This can be done with a single loop that searches through the nests, moving forward to the next when it finds a value that doesn't match the current nest's name and returning the name when it finds a matching value. In the `async` function, a regular `for` or `while` loop can be used.

To do the same in a plain function, you will have to build your loop using a recursive function. The easiest way to do this is to have that function return a promise by calling `then` on the promise that retrieves the storage value. Depending on whether that value matches the name of the current nest, the handler returns that value or a further promise created by calling the loop function again.

Don't forget to start the loop by calling the recursive function once from the main function.

In the `async` function, rejected promises are converted to exceptions by `await`. When an `async` function throws an exception, its promise is rejected. So that works.

If you implemented the non-`async` function as outlined earlier, the way `then` works also automatically causes a failure to end up in the returned promise. If a request fails, the handler passed to `then` isn't called, and the promise it returns is rejected with the same reason.

### Building Promise.all

The function passed to the `Promise` constructor will have to call `then` on each of the promises in the given array. When one of them succeeds, two things need to happen. The resulting value needs to be stored in the correct position of a result array, and we must check whether this was the last pending promise and finish our own promise if it was.

The latter can be done with a counter that is initialized to the length of the input array and from which we subtract 1 every time a promise succeeds. When it reaches 0, we are done. Make sure you take into account the situation where the input array is empty (and thus no promise will ever resolve).

Handling failure requires some thought but turns out to be extremely simple. Just pass the `reject` function of the wrapping promise to each of the promises in the array as a `catch` handler or as a second argument to `then` so that a failure in one of them triggers the rejection of the whole wrapper promise.

# Project: A Programming Language

## Arrays

The easiest way to do this is to represent Egg arrays with JavaScript arrays.

The values added to the top scope must be functions. By using a rest argument (with triple-dot notation), the definition of `array` can be *very* simple.

## Closure

Again, we are riding along on a JavaScript mechanism to get the equivalent feature in Egg. Special forms are passed the local scope in which they are evaluated so that they can evaluate their subforms in that scope. The function returned by `fun` has access to the `scope` argument given to its enclosing function and uses that to create the function's local scope when it is called.

This means that the prototype of the local scope will be the scope in which the function was created, which makes it possible to access bindings in that scope from the function. This is all there is to implementing closure (though to compile it in a way that is actually efficient, you'd need to do some more work).

## Comments

Make sure your solution handles multiple comments in a row, with potentially whitespace between or after them.

A regular expression is probably the easiest way to solve this. Write something that matches "whitespace or a comment, zero or more times". Use the `exec` or `match` method and look at the length of the first element in the returned array (the whole match) to find out how many characters to slice off.

## Fixing scope

You will have to loop through one scope at a time, using `Object.getPrototypeOf` to go to the next outer scope. For each scope, use `hasOwnProperty` to find out whether the binding, indicated by the `name` property of the first argument to `set`, exists in that scope. If it does, set it to the result of evaluating the second argument to `set` and then return that value.

If the outermost scope is reached (`Object.getPrototypeOf` returns null) and we haven't found the binding yet, it doesn't exist, and an error should be thrown.

# The Document Object Model

## Build a table

You can use `document.createElement` to create new element nodes, `document.createTextNode` to create text nodes, and the `appendChild` method to put nodes into other nodes.

You'll want to loop over the key names once to fill in the top row and then again for each object in the array to construct the data rows. To get an array of key names from the first object, `Object.keys` will be useful.

To add the table to the correct parent node, you can use `document.getElementById` or `document.querySelector` to find the node with the proper `id` attribute.

## Elements by tag name

The solution is most easily expressed with a recursive function, similar to the `talksAbout` function defined earlier in this chapter.

You could call `byTagname` itself recursively, concatenating the resulting arrays to produce the output. Or you could create an inner function that calls itself recursively and that has access to an array binding defined in the outer function, to which it can add the matching elements it finds. Don't forget to call the inner function once from the outer function to start the process.

The recursive function must check the node type. Here we are interested only in node type 1 (`Node.ELEMENT_NODE`). For such nodes, we must loop over their children and, for each child, see whether the child matches the query while also doing a recursive call on it to inspect its own children.

## The cat's hat

`Math.cos` and `Math.sin` measure angles in radians, where a full circle is $2\pi$. For a given angle, you can get the opposite angle by adding half of this, which is `Math.PI`. This can be useful for putting the hat on the opposite side of the orbit.

# Handling Events

## Balloon

You'll want to register a handler for the `"keydown"` event and look at `event.key` to figure out whether the up or down arrow key was pressed.

The current size can be kept in a binding so that you can base the new size on it. It'll be helpful to define a function that updates the size—both the binding and the style of the balloon in the DOM—so that you can call it from your event handler, and possibly also once when starting, to set the initial size.

You can change the balloon to an explosion by replacing the text node with another one (using `replaceChild`) or by setting the `textContent` property of its parent node to a new string.

### Mouse trail

Creating the elements is best done with a loop. Append them to the document to make them show up. To be able to access them later to change their position, you'll want to store the elements in an array.

Cycling through them can be done by keeping a counter variable and adding 1 to it every time the `"mousemove"` event fires. The remainder operator (`% elements.length`) can then be used to get a valid array index to pick the element you want to position during a given event.

Another interesting effect can be achieved by modeling a simple physics system. Use the `"mousemove"` event only to update a pair of bindings that track the mouse position. Then use `requestAnimationFrame` to simulate the trailing elements being attracted to the position of the mouse pointer. At every animation step, update their position based on their position relative to the pointer (and, optionally, a speed that is stored for each element). Figuring out a good way to do this is up to you.

### Tabs

One pitfall you might run into is that you can't directly use the node's `childNodes` property as a collection of tab nodes. For one thing, when you add the buttons, they will also become child nodes and end up in this object because it is a live data structure. For another, the text nodes created for the whitespace between the nodes are also in `childNodes` but should not get their own tabs. You can use `children` instead of `childNodes` to ignore text nodes.

You could start by building up an array of tabs so that you have easy access to them. To implement the styling of the buttons, you could store objects that contain both the tab panel and its button.

I recommend writing a separate function for changing tabs. You can either store the previously selected tab and change only the styles needed to hide that and show the new one, or you can just update the style of all tabs every time a new tab is selected.

You might want to call this function immediately to make the interface start with the first tab visible.

## Project: A Platform Game

### Pausing the game

An animation can be interrupted by returning `false` from the function given to `runAnimation`. It can be continued by calling `runAnimation` again.

So we need to communicate the fact that we are pausing the game to the function given to `runAnimation`. For that, you can use a binding that both the event handler and that function have access to.

When finding a way to unregister the handlers registered by `trackKeys`, remember that the *exact* same function value that was passed to `addEventListener` must be passed to `removeEventListener` to successfully remove a handler. Thus, the `handler` function value created in `trackKeys` must be available to the code that unregisters the handlers.

You can add a property to the object returned by `trackKeys`, containing either that function value or a method that handles the unregistering directly.

### A monster

If you want to implement a type of motion that is stateful, such as bouncing, make sure you store the necessary state in the actor object—include it as constructor argument and add it as a property.

Remember that `update` returns a *new* object, rather than changing the old one.

When handling collision, find the player in `state.actors` and compare its position to the monster's position. To get the *bottom* of the player, you have to add its vertical size to its vertical position. The creation of an updated state will resemble either `Coin`'s `collide` method (removing the actor) or `Lava`'s (changing the status to `"lost"`), depending on the player position.

## Drawing on Canvas

### Shapes

The trapezoid (1) is easiest to draw using a path. Pick suitable center coordinates and add each of the four corners around the center.

The diamond (2) can be drawn the straightforward way, with a path, or the interesting way, with a `rotate` transformation. To use rotation, you will have to apply a trick similar to what we did in the `flipHorizontally` function. Because you want to rotate around the center of your rectangle and not around the point (0,0), you must first `translate` to there, then rotate, and then translate back.

Make sure you reset the transformation after drawing any shape that creates one.

For the zigzag (3) it becomes impractical to write a new call to `lineTo` for each line segment. Instead, you should use a loop. You can have each iteration draw either two line segments (right and then left again) or one, in which case you must use the evenness (`% 2`) of the loop index to determine whether to go left or right.

You'll also need a loop for the spiral (4). If you draw a series of points, with each point moving further along a circle around the spiral's center, you get a circle. If, during the loop, you vary the radius of the circle on which you are putting the current point and go around more than once, the result is a spiral.

The star (5) depicted is built out of `quadraticCurveTo` lines. You could also draw one with straight lines. Divide a circle into eight pieces for a star with eight points, or however many pieces you want. Draw lines between these points, making them curve toward the center of the star. With `quadraticCurveTo`, you can use the center as the control point.

### THE PIE CHART

You will need to call `fillText` and set the context's `textAlign` and `textBaseline` properties in such a way that the text ends up where you want it.

A sensible way to position the labels would be to put the text on the line going from the center of the pie through the middle of the slice. You don't want to put the text directly against the side of the pie but rather move the text out to the side of the pie by a given number of pixels.

The angle of this line is `currentAngle + 0.5 * sliceAngle`. The following code finds a position on this line 120 pixels from the center:

```
let middleAngle = currentAngle + 0.5 * sliceAngle;
let textX = Math.cos(middleAngle) * 120 + centerX;
let textY = Math.sin(middleAngle) * 120 + centerY;
```

For `textBaseline`, the value `"middle"` is probably appropriate when using this approach. What to use for `textAlign` depends on which side of the circle we are on. On the left, it should be `"right"`, and on the right, it should be

`"left"`, so that the text is positioned away from the pie.

If you are not sure how to find out which side of the circle a given angle is on, look to the explanation of `Math.cos` in Chapter 14. The cosine of an angle tells us which x-coordinate it corresponds to, which in turn tells us exactly which side of the circle we are on.

### A bouncing ball

A box is easy to draw with `strokeRect`. Define a binding that holds its size or define two bindings if your box's width and height differ. To create a round ball, start a path and call `arc(x, y, radius, 0, 7)`, which creates an arc going from zero to more than a whole circle. Then fill the path.

To model the ball's position and speed, you can use the `Vec` class from Chapter 16. Give it a starting speed, preferably one that is not purely vertical or horizontal, and for every frame multiply that speed by the amount of time that elapsed. When the ball gets too close to a vertical wall, invert the x component in its speed. Likewise, invert the y component when it hits a horizontal wall.

After finding the ball's new position and speed, use `clearRect` to delete the scene and redraw it using the new position.

### Precomputed mirroring

The key to the solution is the fact that we can use a canvas element as a source image when using `drawImage`. It is possible to create an extra `<canvas>` element, without adding it to the document, and draw our inverted sprites to it, once. When drawing an actual frame, we just copy the already inverted sprites to the main canvas.

Some care would be required because images do not load instantly. We do the inverted drawing only once, and if we do it before the image loads, it won't draw anything. A `"load"` handler on the image can be used to draw the inverted images to the extra canvas. This canvas can be used as a drawing source immediately (it'll simply be blank until we draw the character onto it).

## HTTP and Forms

### Content negotiation

Base your code on the `fetch` examples earlier in the chapter.

Asking for a bogus media type will return a response with code 406, "Not acceptable", which is the code a server should return when it can't fulfill the

`Accept` header.

### A JavaScript workbench

Use `document.querySelector` or `document.getElementById` to get access to the elements defined in your HTML. An event handler for `"click"` or `"mousedown"` events on the button can get the `value` property of the text field and call `Function` on it.

Make sure you wrap both the call to `Function` and the call to its result in a `try` block so you can catch the exceptions it produces. In this case, we really don't know what type of exception we are looking for, so catch everything.

The `textContent` property of the output element can be used to fill it with a string message. Or, if you want to keep the old content around, create a new text node using `document.createTextNode` and append it to the element. Remember to add a newline character to the end so that not all output appears on a single line.

### Conway's Game of Life

To solve the problem of having the changes conceptually happen at the same time, try to see the computation of a generation as a pure function, which takes one grid and produces a new grid that represents the next turn.

Representing the matrix can be done in the way shown in Chapter 6. You can count live neighbors with two nested loops, looping over adjacent coordinates in both dimensions. Take care not to count cells outside of the field and to ignore the cell in the center, whose neighbors we are counting.

Ensuring that changes to checkboxes take effect on the next generation can be done in two ways. An event handler could notice these changes and update the current grid to reflect them, or you could generate a fresh grid from the values in the checkboxes before computing the next turn.

If you choose to go with event handlers, you might want to attach attributes that identify the position that each checkbox corresponds to so that it is easy to find out which cell to change.

To draw the grid of checkboxes, you can either use a `<table>` element (see Chapter 14) or simply put them all in the same element and put `<br>` (line break) elements between the rows.

# Project: A Pixel Art Editor

## Keyboard bindings

The `key` property of events for letter keys will be the lowercase letter itself, if SHIFT isn't being held. We're not interested in key events with SHIFT here.

A `"keydown"` handler can inspect its event object to see whether it matches any of the shortcuts. You can automatically get the list of first letters from the `tools` object so that you don't have to write them out.

When the key event matches a shortcut, call `preventDefault` on it and dispatch the appropriate action.

## Efficient drawing

This exercise is a good example of how immutable data structures can make code *faster*. Because we have both the old and the new picture, we can compare them and redraw only the pixels that changed color, saving more than 99 percent of the drawing work in most cases.

You can either write a new function `updatePicture` or have `drawPicture` take an extra argument, which may be undefined or the previous picture. For each pixel, the function checks whether a previous picture was passed with the same color at this position and skips the pixel when that is the case.

Because the canvas gets cleared when we change its size, you should also avoid touching its `width` and `height` properties when the old picture and the new picture have the same size. If they are different, which will happen when a new picture has been loaded, you can set the binding holding the old picture to null after changing the canvas size because you shouldn't skip any pixels after you've changed the canvas size.

## Circles

You can take some inspiration from the `rectangle` tool. Like that tool, you'll want to keep drawing on the *starting* picture, rather than the current picture, when the pointer moves.

To figure out which pixels to color, you can use the Pythagorean theorem. First figure out the distance between the current pointer position and the start position by taking the square root (`Math.sqrt`) of the sum of the square (`Math.pow(x, 2)`) of the difference in x-coordinates and the square of the difference in y-coordinates. Then loop over a square of pixels around the start position, whose sides are at least twice the radius, and color those that are within the

circle's radius, again using the Pythagorean formula to figure out their distance from the center.

Make sure you don't try to color pixels that are outside of the picture's boundaries.

### Proper lines

The thing about the problem of drawing a pixelated line is that it is really four similar but slightly different problems. Drawing a horizontal line from the left to the right is easy—you loop over the x-coordinates and color a pixel at every step. If the line has a slight slope (less than 45 degrees or $¼\pi$ radians), you can interpolate the y-coordinate along the slope. You still need one pixel per $x$ position, with the $y$ position of those pixels determined by the slope.

But as soon as your slope goes across 45 degrees, you need to switch the way you treat the coordinates. You now need one pixel per $y$ position since the line goes up more than it goes left. And then, when you cross 135 degrees, you have to go back to looping over the x-coordinates, but from right to left.

You don't actually have to write four loops. Since drawing a line from $A$ to $B$ is the same as drawing a line from $B$ to $A$, you can swap the start and end positions for lines going from right to left and treat them as going left to right.

So you need two different loops. The first thing your line drawing function should do is check whether the difference between the x-coordinates is larger than the difference between the y-coordinates. If it is, this is a horizontal-ish line, and if not, a vertical-ish one.

Make sure you compare the *absolute* values of the $x$ and $y$ difference, which you can get with `Math.abs`.

Once you know along which axis you will be looping, you can check whether the start point has a higher coordinate along that axis than the endpoint and swap them if necessary. A succinct way to swap the values of two bindings in JavaScript uses destructuring assignment like this:

```
[start, end] = [end, start];
```

Then you can compute the slope of the line, which determines the amount the coordinate on the other axis changes for each step you take along your main axis. With that, you can run a loop along the main axis while also tracking the corresponding position on the other axis, and you can draw pixels on every iteration. Make sure you round the non-main axis coordinates since they are likely to be fractional and the `draw` method doesn't respond well to fractional coordinates.

# Node.js

## Search tool

Your first command line argument, the regular expression, can be found in `process.argv[2]`. The input files come after that. You can use the `RegExp` constructor to go from a string to a regular expression object.

Doing this synchronously, with `readFileSync`, is more straightforward, but if you use `fs.promises` again to get promise-returning functions and write an `async` function, the code looks similar.

To figure out whether something is a directory, you can again use `stat` (or `statSync`) and the stats object's `isDirectory` method.

Exploring a directory is a branching process. You can do it either by using a recursive function or by keeping an array of work (files that still need to be explored). To find the files in a directory, you can call `readdir` or `readdirSync`. The strange capitalization—Node's file system function naming is loosely based on standard Unix functions, such as `readdir`, that are all lowercase, but then it adds `Sync` with a capital letter.

To go from a filename read with `readdir` to a full path name, you have to combine it with the name of the directory, putting a slash character (`/`) between them.

## Directory creation

You can use the function that implements the `DELETE` method as a blueprint for the `MKCOL` method. When no file is found, try to create a directory with `mkdir`. When a directory exists at that path, you can return a 204 response so that directory creation requests are idempotent. If a nondirectory file exists here, return an error code. Code 400 ("bad request") would be appropriate.

## A public space on the web

You can create a `<textarea>` element to hold the content of the file that is being edited. A `GET` request, using `fetch`, can retrieve the current content of the file. You can use relative URLs like *index.html*, instead of *http://localhost:8000/index.html*, to refer to files on the same server as the running script.

Then, when the user clicks a button (you can use a `<form>` element and `"submit"` event), make a `PUT` request to the same URL, with the content of the `<textarea>` as request body, to save the file.

You can then add a `<select>` element that contains all the files in the server's top directory by adding `<option>` elements containing the lines returned by a

`GET` request to the URL `/`. When the user selects another file (a `"change"` event on the field), the script must fetch and display that file. When saving a file, use the currently selected filename.

## Project: Skill-Sharing Website

### Disk persistence

The simplest solution I can come up with is to encode the whole `talks` object as JSON and dump it to a file with `writeFile`. There is already a method (`updated` ) that is called every time the server's data changes. It can be extended to write the new data to disk.

Pick a filename, for example `./talks.json`. When the server starts, it can try to read that file with `readFile`, and if that succeeds, the server can use the file's contents as its starting data.

Beware, though. The `talks` object started as a prototype-less object so that the `in` operator could reliably be used. `JSON.parse` will return regular objects with `Object.prototype` as their prototype. If you use JSON as your file format, you'll have to copy the properties of the object returned by `JSON.parse` into a new, prototype-less object.

### Comment field resets

The best way to do this is probably to make talks component objects, with a `syncState` method, so that they can be updated to show a modified version of the talk. During normal operation, the only way a talk can be changed is by adding more comments, so the `syncState` method can be relatively simple.

The difficult part is that, when a changed list of talks comes in, we have to reconcile the existing list of DOM components with the talks on the new list—deleting components whose talk was deleted and updating components whose talk changed.

To do this, it might be helpful to keep a data structure that stores the talk components under the talk titles so that you can easily figure out whether a component exists for a given talk. You can then loop over the new array of talks, and for each of them, either synchronize an existing component or create a new one. To delete components for deleted talks, you'll have to also loop over the components and check whether the corresponding talks still exist.

# Index