

# Get Programming with JavaScript

John R. Larsen



 manning



***Get Programming with JavaScript***

by John R. Larsen

**Chapter 23**

Copyright 2016 Manning Publications

# *brief contents*

---

## **PART 1 CORE CONCEPTS ON THE CONSOLE .....1**

- 1 ■ Programming, JavaScript, and JS Bin 3
- 2 ■ Variables: storing data in your program 16
- 3 ■ Objects: grouping your data 27
- 4 ■ Functions: code on demand 40
- 5 ■ Arguments: passing data to functions 57
- 6 ■ Return values: getting data from functions 70
- 7 ■ Object arguments: functions working with objects 83
- 8 ■ Arrays: putting data into lists 104
- 9 ■ Constructors: building objects with functions 122
- 10 ■ Bracket notation: flexible property names 147

## **PART 2 ORGANIZING YOUR PROGRAMS .....169**

- 11 ■ Scope: hiding information 171
- 12 ■ Conditions: choosing code to run 198
- 13 ■ Modules: breaking a program into pieces 221
- 14 ■ Models: working with data 248

- 15 ■ Views: displaying data 264
- 16 ■ Controllers: linking models and views 280

### **PART 3    JAVASCRIPT IN THE BROWSER.....299**

- 17 ■ HTML: building web pages 301
- 18 ■ Controls: getting user input 323
- 19 ■ Templates: filling placeholders with data 343
- 20 ■ XHR: loading data 367
- 21 ■ Conclusion: get programming with JavaScript 387
  
- 22 ■ Node: running JavaScript outside the browser online
- 23 ■ Express: building an API online
- 24 ■ Polling: repeating requests with XHR online
- 25 ■ Socket.IO: real-time messaging online

# Express: building an API

## ***This chapter covers***

- Installing packages with npm
- Creating a web server with Express.js
- Building a central game server for *The Crypt*

Node.js has a rich ecosystem of packages, where a *package* is a collection of modules that work together to complete a particular job. Express is one of those packages. It provides objects and properties to make creating versatile web servers easy, hiding away a lot of the boilerplate code you'd have to write if you implemented all of your websites and APIs using only Node.

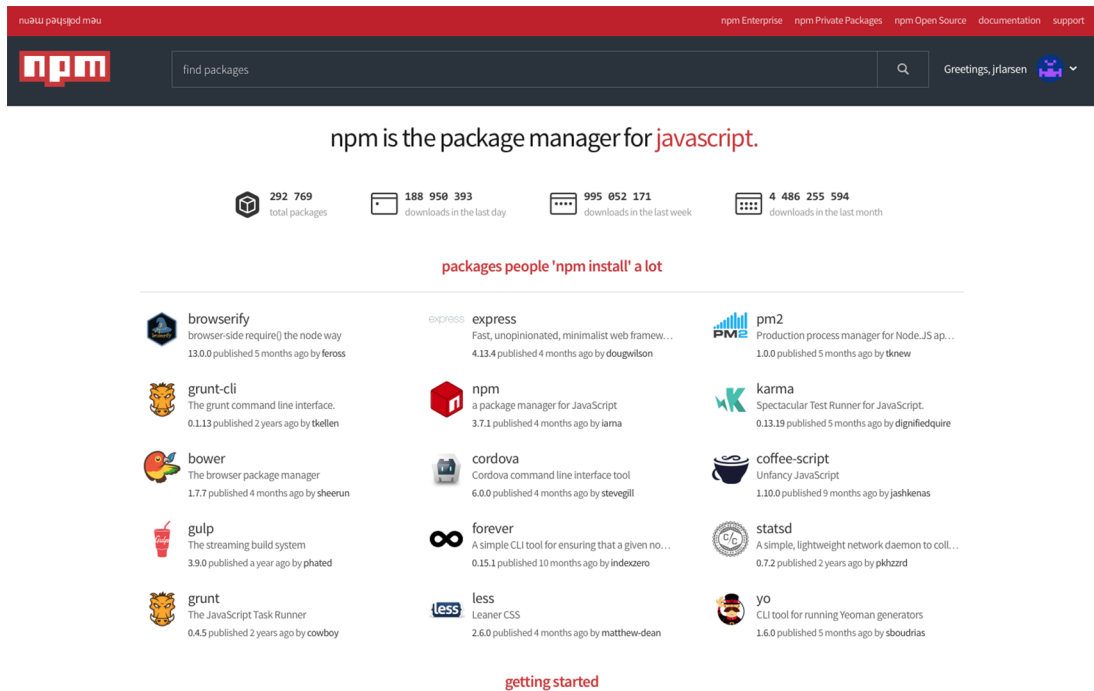
In this chapter you'll learn how to use Express to build servers that respond to requests from browsers (and any other programs that send HTTP requests). The server may send assets like web pages, JavaScript and CSS files, and images and videos, and it may also send JSON data. You'll use what you learn to split your code for *The Crypt* into server-side and client-side modules.

But how do you get hold of Express?

## 23.1 Installing packages from npm

Chapter 22 explained how to export code with `module.exports` and import it with the `require` statement. As well as being able to export and import modules that *you* create, you can install and import modules written by *other* developers. Welcome to Nerdy Programmers Mingling, or National Public Mania, or Naughty Push Message. Welcome to npm.

npm is a program installed along with Node that you use to install and manage packages in your Node projects. Lots of developers are creating lots of packages for Node; the place to find those packages is [www.npmjs.com](http://www.npmjs.com) (figure 23.1).



**Figure 23.1** Browse or search for packages at [npmjs.com](http://npmjs.com).

You can search for packages related to a job you need to do in your program. At the top of each page on [npmjs.com](http://npmjs.com) is a search box. Type “express” in the search box and navigate to the Express package page, as shown in figure 23.2.

A package’s npm page should explain how to install and use the package.

For the first part of this chapter, you need a blank folder called `webserver`. In that folder save the `package.json` file shown here in listing 23.1.

Narcissistic Piano Mover

npm Enterprise npm Private Packages npm Open Source documentation support

npm find packages

Greetings, jrlarsen

Hooks are here. Get notifications of registry and package changes as they happen...

★ **express** pub  
Fast, unopinionated, minimalist web framework

express

npm v4.13.4 downloads 6M/month linux passing windows passing coverage 100%

```
var express = require('express')
var app = express()

app.get('/', function (req, res) {
  res.send('Hello World')
})

app.listen(3000)
```

**Installation**

```
$ npm install express
```

npm camp

The first-ever npm community conference comes to Oakland this July.  
Tickets now on sale »

npm install express  
how? learn more

dougwilson published 4 months ago

4.13.4 is the latest of 273 releases

github.com/expressjs/express

MIT

Collaborators

Figure 23.2 The express package page on npmjs.com

**Listing 23.1 Information about your package (package.json)**

```
{
  "name": "hello-server",
  "version": "1.0.0"
}
```

The file includes information about your project. You could include more properties, and you can find out more in the npm documentation at <https://docs.npmjs.com/getting-started/using-a-package.json>.

To install Express for a project, navigate into your project folder at the command line and type

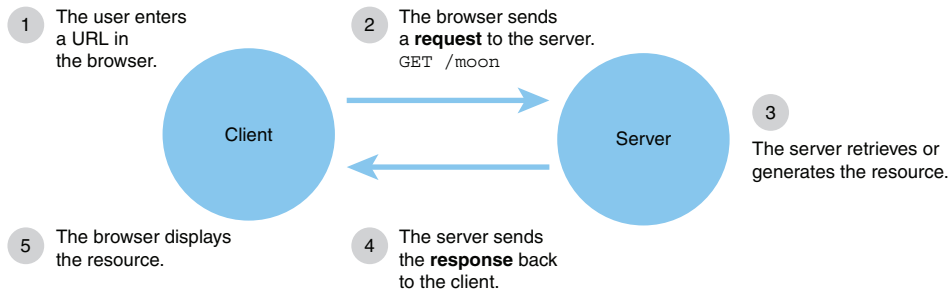
```
npm install express --save
```

You may need administrator privileges to perform the install (on OS X use `sudo npm install express --save`, if needed, and on Windows open the command prompt as an administrator).

npm will create a `node_modules` folder inside your project folder and download the Express code files into that folder.

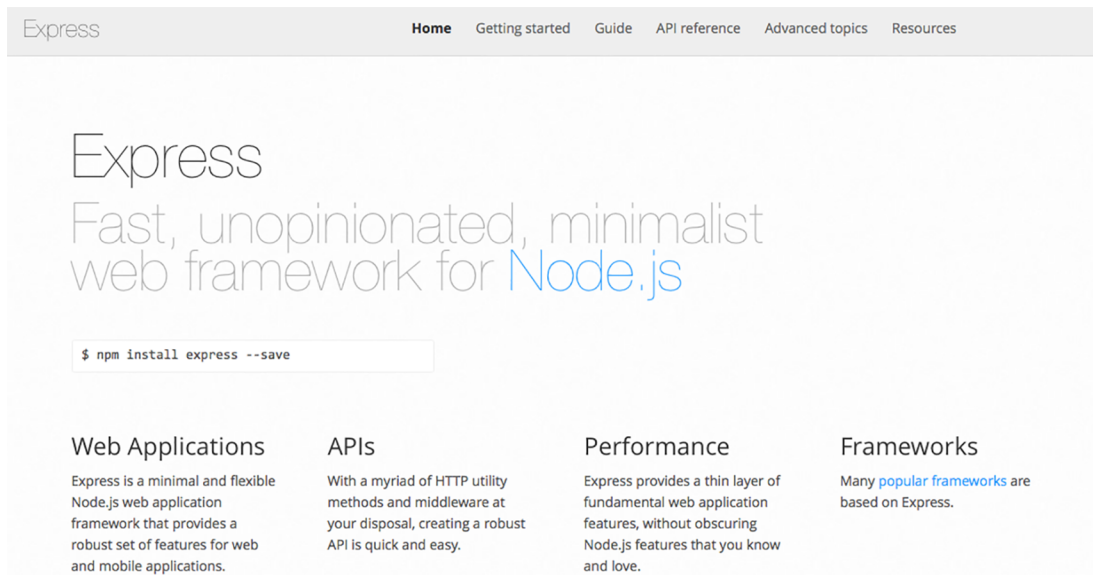
## 23.2 Creating a web server with Express

Express is a web framework for Node. It makes it easy to create servers that listen for requests from clients over the internet and send responses back. Figure 23.3 shows a client (your browser) making a request and the server sending a response.



**Figure 23.3** Client/server interaction with requests and responses

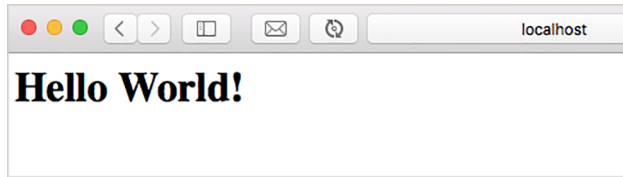
The responses are usually web pages, assets like images and other files, or JSON data. Express makes it easy to set up the functions that respond to the different requests a client might make. You can find out more about Express at <http://expressjs.com> (figure 23.4).



**Figure 23.4** Find out more about Express at [expressjs.com](http://expressjs.com).



You're going to start using Express by creating a simple web server that responds with the page shown in figure 23.5.



**Figure 23.5** Displaying “Hello World!” in a web page

This next listing shows the code for the server, in a file saved as `hello.js`.

#### Listing 23.2 A simple webserver (`hello.js`)

```
var express = require('express');  
var app = express();  
app.get('/', sayHello);  
function sayHello (req, res) {  
  res.send('<h1>Hello World!</h1>');  
}  
app.listen(1337);
```

Import the express module

Create a server and assign it to the app variable

Assign the sayHello function to the root route

Declare the sayHello function

Start the webserver on port 1337

Send a message to the client

First, you import Express into your program by using the `require` statement and assign the function it exports to the `express` variable. Calling the `express` function returns a web server object, and you assign that to the `app` variable. The third statement in the program assigns the `sayHello` function as the route handler for the `/` route. When the web server receives an HTTP request to the `/` route (in other words, to the root of the website), it will execute the `sayHello` function. The `sayHello` function sends a snippet of HTML back to the client. The last line of the program starts the web server. It listens on port 1337. The port allows you to run several servers at once, each with its own number.

Run the program at the command line with `node hello`, and then visit `localhost:1337` in a web browser. `localhost` refers to the computer on which the browser is running. Because the server is on the same machine as the browser, your computer can pass your request straight to the server, without the need to venture across the internet. To stop the web server, exit the Node program with `Ctrl-C` on the command line.

### 23.2.1 Adding more routes

You want your website to have more than one page; you need to add more routes. The next listing adds `/moon` and `/adventure` routes. Figure 23.6 shows the web pages the server returns when you visit the routes in a browser.

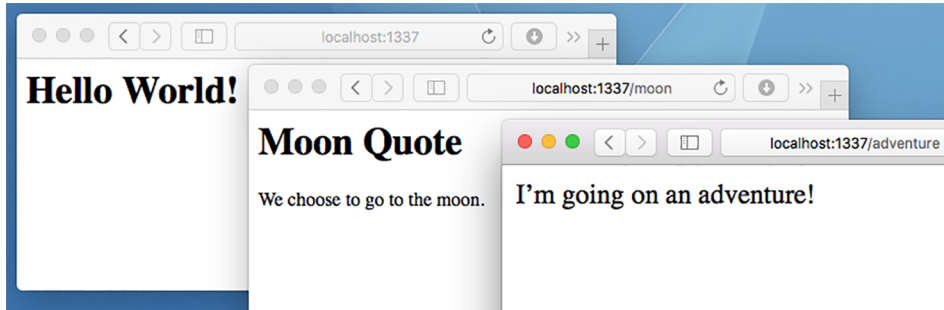


Figure 23.6 Visiting the three routes in a browser

The following listing shows the code for the new route handler functions, `goMoon` and `setOff`, and for assigning the functions to the routes with `app.get`.

#### Listing 23.3 A webserver with three routes (`threeRoutes.js`)

```
var express = require('express');
var app = express();

app.get('/', sayHello);
app.get('/moon', goMoon);
app.get('/adventure', setOff);

function sayHello (req, res) {
  res.send('<h1>Hello World!</h1>');
}

function goMoon (req, res) {
  var html = '<h1>Moon Quote</h1>';
  html += '<p>We choose to go to the moon.</p>';
  res.send(html);
}

function setOff (req, res) {
  res.send('<p>I'm going on an adventure!</p>');
}

app.listen(1337);
```

Register two more  
route handlers

Define the  
handler for the  
/moon route

Define the  
handler for the  
/adventure route

You now have three routes set up for your server: `/`, `/moon`, and `/adventure`. When a browser sends a request to the server, the server tries to match the request to a route and call the corresponding route handler function. The route handler might retrieve

a file like a web page or image and send it as the response, or it might generate the response, maybe using templates and database data.

When the server calls a route handler for a matched route, it passes the handler function a request object and a response object as arguments. When defining the handler functions, it's common to use a `req` parameter for the request object and a `res` parameter for the response object. In listings 23.2 and 23.3, you used the `send` method of the response object to send your response back to the client:

```
function sayHello (req, res) {
    res.send('<h1>Hello World!</h1>');
}
```

Express passes the `sayHello` function request and response objects

Use the `res.send` method to send a response to the browser

If you haven't done so already, run the webserver from listing 23.3 by typing `node threeRoutes` on the command line. Visit the three routes in a browser: `localhost:1337`, `localhost:1337/moon`, and `localhost:1337/adventure`.

### 23.2.2 Sending static files

You don't really want to be mixing snippets of HTML in with the JavaScript for your route handlers. Your system can store files like static web pages, images, and style sheets and send them to a client when requested. Express makes sending static files easy with its `express.static` function. The following listing shows how to serve files from a folder called `public`.

#### Listing 23.4 Serving static files (`static.js`)

```
var express = require('express');
var app = express();
app.use(express.static('public'));
app.listen(1337);
```

Look for requested files in the public folder

When a user visits `localhost:1337/moon.html` in their browser, the server will send the `moon.html` file from the `public` folder. Figure 23.7 shows files in the `public` folder that could be accessed in the same way.

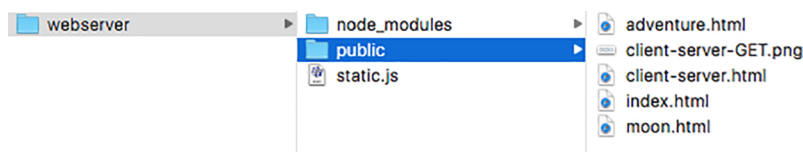


Figure 23.7 Static files in the `public` folder will be served when requested.

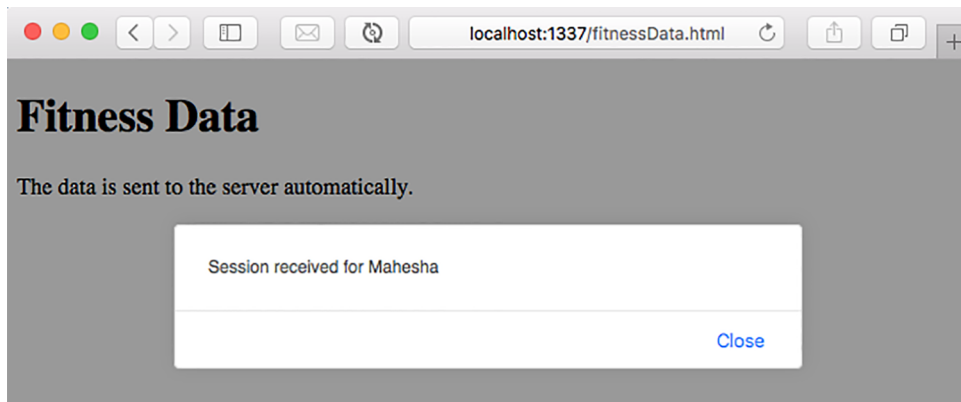
Most websites involve a mix of static files and routes with associated route handler functions. You may also want your server to send and receive data for your apps.

### 23.3 Sending data to and from the server

In parts 2 and 3 of *Get Programming with JavaScript*, you worked on a fitness app that let users log exercise sessions. In chapter 20, you used the `XMLHttpRequest` object to retrieve user data for the app. You now want to send fitness session data back to the server. There are five steps to implement:

- 1 Prepare the data in the browser.
- 2 Send the data from the browser to the server.
- 3 Process the data on the server.
- 4 Send a response from the server to the browser.
- 5 Process the response in the browser.

To test the process, you'll write code, to be executed in the browser, that sends the data to the server and displays the response that the server sends back, as shown in figure 23.8.



**Figure 23.8** The page displays a message received from the server.

You implement the steps for the browser and server in the following five sections.

#### 23.3.1 Prepare the data in the browser

Fitness app users log the data and duration of their exercise sessions. The data, in code, is held as a JavaScript object, like this:

```
var data = {  
  name: "Mahesha",  
  sessionDate: "2017-02-07",  
  duration: 60  
};
```

You send the data from the browser to the server as a JSON string. JavaScript provides a `JSON.stringify` method for converting JavaScript objects into JSON strings:

```
var json = JSON.stringify(data);
```

You need to send the JSON data to the server. The next section describes how.

### 23.3.2 Send the data from the browser to the server

XHR objects can also be used to send, or *post*, data to the server. In chapter 20, you retrieved data by specifying "GET" as the method:

```
xhr.open("GET", url);
```

When sending data that will create a new record of some kind on the server, you can specify "POST" as the method instead:

```
xhr.open("POST", url);
```

*Headers* are extra pieces of information sent with requests and responses that help browsers and servers process communications. Let the server know that you're sending the data as JSON by setting an appropriate header for the request:

```
xhr.setRequestHeader('Content-Type', 'application/json');
```

Finally, pass the data to send as an argument to the `send` method:

```
xhr.send(json);
```

The following listing shows the pieces put together. Create a `js` folder in the public folder, and save this file as `xhrPost.js` in the `js` folder.

#### Listing 23.5 Posting data with XHR (xhrPost.js partial)

```
var data = {
  name: "Mahesha",
  sessionDate: "2017-02-07",
  duration: 60
};
var json = JSON.stringify(data);
var url = "/fitness";
var xhr = new XMLHttpRequest();
xhr.open("POST", url);
xhr.setRequestHeader('Content-Type', 'application/json');
xhr.send(json);
```

**Convert the JavaScript data object into a JSON string**

**Specify a route that will handle the request**

**Use the POST method to send data**

**Let the server know you're sending JSON data**

**Pass the data as an argument to the send method**

You need a route on the server to handle the data.

### 23.3.3 Process the data on the server

In order to get hold of the data sent to the server by the browser, you need to access the *body* of the request. A very popular npm module that does the job you need is `body-parser`. In the root of the project, install `body-parser` at the command prompt:

```
npm install body-parser --save
```

The `body-parser` module grabs the JSON data from the body of the request and makes it available as a JavaScript object assigned to the `body` property of the request object. The next listing shows how to instruct an Express application to use the `body-parser` module when expecting JSON data.

#### Listing 23.6 A route for handling fitness data (`fitnessServer.js` partial)

```
var express = require('express');
var app = express();
var bodyParser = require('body-parser');

app.use(express.static('public'));
app.use(bodyParser.json());

app.post('/fitness', processSession);

function processSession (req, res) {
  console.log(req.body);
};

app.listen(1337);
```

Import the body-parser module

Tell Express to use body-parser, looking for JSON data

Assign a handler to the /fitness route

Access the data via req.body

The `body-parser` module is capable of parsing data from requests formatted in a number of ways. You're sending JSON data from the browser, so you tell your Express app to be on the lookout for JSON, like this:

```
app.use(bodyParser.json());
```

To test your code, you need a web page in which to run the XHR code. The page in listing 23.7 should do the job. Save it in the public folder. Run the server at the command line with `node fitnessServer` and then visit `localhost:1337/fitnessData.html` in your browser. The data sent from the browser should be logged to the command line:

```
{ name: 'Mahesha', sessionId: '2017-02-07', duration: 60 }
```

#### Listing 23.7 A fitness app XHR test page (`fitnessData.html`)

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>Fitness Data</title>
</head>
```

```

<body>
  <h1>Fitness Data</h1>
  <p>The data is sent to the server automatically.</p>

  <script src="js/xhrPost.js"></script>
</body>

```

← Load the code that sends data to the server

The data from the browser has reached the server, but you also want to send a response from the server back to the browser.

### 23.3.4 Send a response from the server to the browser

Figure 23.8 showed the browser popping up a dialog box with a message from the server. The Express response object has a `json` method that lets you send JSON data as the response:

```
res.json(data);
```

Complete the fitness server by using `res.json` to send a response back to the browser, as shown in the following listing.

#### Listing 23.8 Send a response message (fitnessServer.js)

```

var express = require('express');
var app = express();
var bodyParser = require('body-parser');

app.use(express.static('public'));
app.use(bodyParser.json());

app.post('/fitness', processSession);

function processSession (req, res) {
  console.log(req.body);
  res.json({
    'status': 'OK',
    'message': 'Session received for ' + req.body.name
  });
};

app.listen(1337);

```

You don't need to call `stringify` or set any headers for the response; the `json` method sorts everything out automatically.

The last step is to update the browser code to handle the response from the server.

### 23.3.5 Process the response in the browser

You sent the fitness session data from the browser using an XHR object. You can tell the XHR object to listen for a response by assigning an event handler for the `load` event. Update `xhrPost.js` to include an event listener, as shown here.

**Listing 23.9** Displaying the message from the server (xhrPost.js)

```

var data = {
  name: "Mahesha",
  sessionDate: "2017-02-07",
  duration: 60
};

var json = JSON.stringify(data);

var url = "/fitness";

var xhr = new XMLHttpRequest();
xhr.open("POST", url);

xhr.setRequestHeader('Content-Type', 'application/json');

xhr.addEventListener("load", function () {
  var responseData = JSON.parse(xhr.responseText);
  alert(responseData.message);
});

xhr.send(json);

```

Assign a function to call when the data loads

Convert the JSON data string to a JavaScript object

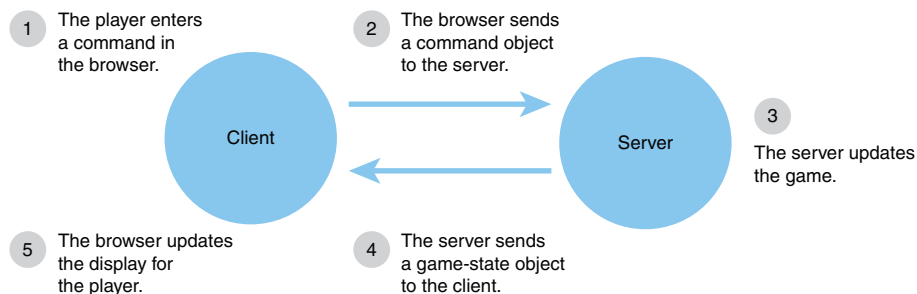
Display the message property of the data

Restart the server and visit the web page, `localhost:1337/fitnessData.html`, in your browser. The server code logs the data from the browser to the console (the command line), and the browser pops up the message returned from the server.

Now that you can send data from browser to server and back again, it's time to return to *The Crypt* and build a game server that can communicate with multiple players, each using its own browser.

### 23.4 The Crypt—server and client code

In your new version of *The Crypt*, the server runs the game. The browser displays the current state of the game and relays player actions to the server. Players type commands (get, go, and use) into the browser interface, and the game code on the browser communicates their actions to the server. Figure 23.9 shows the steps taken for each player action in the game.



**Figure 23.9** The steps involved when a player issues a command



The server receives the player's command, updates the game, and sends data that represents the new state of the game back to the browser. Say the player enters a command to move to the east, as shown in figure 23.10. When the player clicks the button, the code running in the browser will parse the command in the text box and then send the command data to the server.



**Figure 23.10** A player enters a command to go east.

The command data sent from the browser to the server looks like this:

```
{
  "gameID": 7773,
  "playerName": "Jahver",
  "command": {
    "type": "go",
    "direction": "east"
  }
}
```

The data includes a `gameID` property. Whereas before, each player had their own version of the game loaded in their browser, in the new version of *The Crypt* the server will manage all of the games for all players. The browser will need to send the server an ID to identify which game a command is for. There's also a `playerName` property that will identify individual players in a game.

The server acts on the command and responds with an object representing the current state of the game.

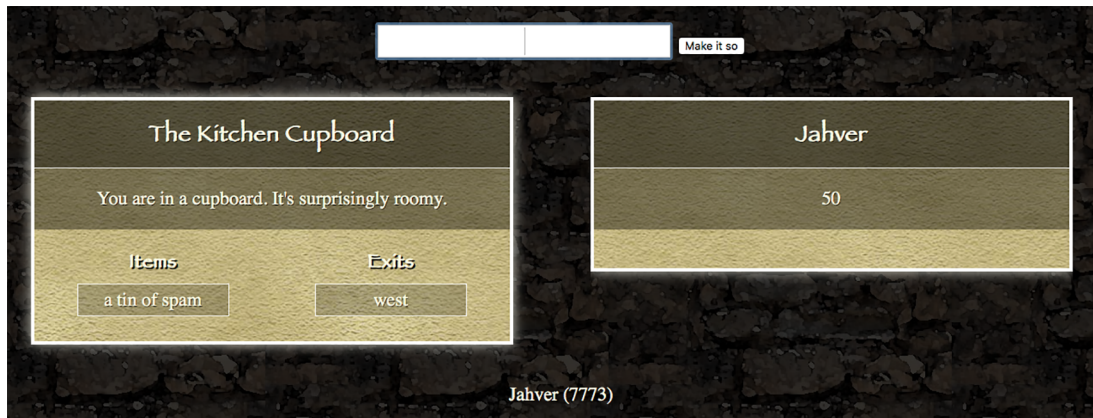
```
{
  "gameID": 7773,
  "players": [{
```

```

    "name": "Jahver",
    "health": 50,
    "items": [],
    "place": "The Kitchen Cupboard"
  }],
  "place": {
    "title": "The Kitchen Cupboard",
    "description": "You are in a cupboard. It's surprisingly roomy.",
    "items": ["a tin of spam"],
    "exits": ["west"]
  },
  "inPlay": true,
  "messages": []
}

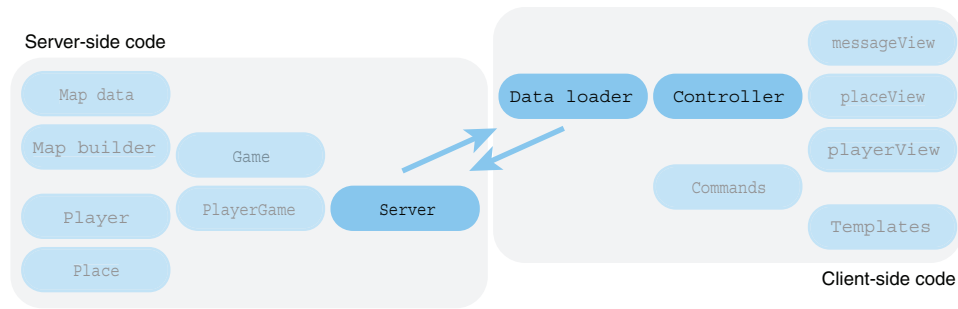
```

The browser passes the game-state data it receives from the server to the appropriate views, and they use templates to update the display, as shown in figure 23.11.



**Figure 23.11** The new state of the game is displayed by the views.

The code modules are now split between the server and the browser. Most of the modules are unchanged from previous chapters. In this chapter, you focus on the game-server code that responds to requests and the client-side code that makes those requests as a result of player commands. Figure 23.12 shows all of the modules, server-side and client-side, highlights the three modules you build in the sections that follow, and uses arrows to represent the client/server interactions at the heart of *The Crypt*.



**Figure 23.12** Server-side and client-side code modules

## 23.5 The Crypt—building the game server

The game server you build needs to start the game and handle requests to get, go, and use within the game. Listing 23.10 includes all the code for the server; it will perform six main functions, discussed after the listing:

- 1 Create the server app.
- 2 Start the game.
- 3 Register route handlers for routes.
- 4 Define route handler functions.
- 5 Serve static files.
- 6 Start listening for requests.

### Listing 23.10 The game-server (gameServer.js)

```

var express = require('express');
var app = express();

var bodyParser = require('body-parser');
app.use(bodyParser.json());

var mapData = require('./maps/theDarkHouse');
var Game = require('./lib/game');
var PlayerGame = require('./lib/playerGame');

var game = new Game(1, mapData);
var playerGame = new PlayerGame("Kandra", game);

app.post('/api/get', get);
app.post('/api/go', go);
app.post('/api/use', use);
app.get('/api/start', start);

function get (req, res) {
  playerGame.clearMessages();
  res.json(playerGame.get());
}
  
```

← Create the server

← Start a game

Define the API

← Use res.json to send JSON data

```

function go (req, res) {
  playerGame.clearMessages();
  var command = req.body.command;
  res.json(playerGame.go(command.direction));
}

function use (req, res) {
  playerGame.clearMessages();
  var command = req.body.command;
  res.json(playerGame.use(command.item, command.direction));
}

function start (req, res) {
  res.json(playerGame.getData());
}

app.use(express.static("public"));

app.listen(1337);

```

Access browser data on req.body

Serve static files from the public folder

Start the server

### 23.5.1 Create the server app

The server's an Express app. You import Express and call the `express` function to create the app object.

```

var express = require('express');
var app = express();

```

The server will work with JSON data sent as the body of requests, so you tell it to use the `body-parser` module:

```

var bodyParser = require('body-parser');
app.use(bodyParser.json());

```

### 23.5.2 Start the game

In chapter 22, you created the game and player game modules to manage games in *The Crypt*. You tested them out on the Node REPL. It's time to put them to use in the game proper. The `Game` constructor function requires map data and the `PlayerGame` constructor function requires a game object. The following snippet shows the steps you use to start a game:

```

var mapData = require('./maps/theDarkHouse');
var Game = require('./lib/game');
var PlayerGame = require('./lib/playerGame');

var game = new Game(1, mapData);
var playerGame = new PlayerGame('Kandra', game);

```

For this chapter, you stick with a single player called Kandra. Chapter 24 has the juicy, multiplayer version, with extra code to manage multiple games and players.

Now that you have a game object, `playerGame`, you can execute commands to update the state of the game: `playerGame.get()`, `playerGame.go('south')`, and

`playerGame.use('holy water', 'south')`, for example. Each command returns the state of the game as a JavaScript object.

### 23.5.3 Register route handlers for routes

When a player gets, goes, or uses in the game, the state of the game changes: an item is added, a place is set, a challenge is overcome, or a message is created. When requests will change the state of objects on the server, your game server expects the browser to send those requests using the `POST` method rather than the `GET` method. You register the routes in your Express app like this:

```
app.post('/api/get', get);
app.post('/api/go', go);
app.post('/api/use', use);
app.get('/api/start', start);
```

Requests to these routes  
change the state of the game

← Requests to this route  
make no changes

For example, the third statement tells your app to call the `use` function when a request using the `POST` method matches the route `/api/use`. In other words, the app will call the `use` function when a `POST` request has the URL `localhost:1337/api/use`.

You start each route with `/api` because the routes are all part of your game-server API; they make changes to the game state and return data in response to behind-the-scenes requests from the browser. Rather than serving full web pages and associated assets for users navigating in the browser, the routes work with pure JSON data for programmatic requests from the code in an application. The routes form your application programming interface rather than your UI.

### 23.5.4 Define route handler functions

The functions that your routes call all work with the `playerGame` object to update the state of the game and return it. The simplest, for now, is the `start` function. It returns the current state of the game. The browser will call it once, to set up the initial game display.

```
function start (req, res) {
  res.json(playerGame.getData());
}
```

In its current form, the `start` function makes no changes to the state of the game; you register the route with `app.get` instead of `app.post`. But you'll be making changes to it in the next chapter, and it will be responsible for creating new `playerGame` and game objects, so `app.post` may be more appropriate then.

The remaining handlers all clear the player's array of messages before performing their actions. The first, `get`, doesn't require any data to do its job.

```
function get (req, res) {
  playerGame.clearMessages();
  res.json(playerGame.get());
}
```

The last two route-handler functions use the data, sent by the browser, that body-parser assigns to the `body` property of the request object. In particular, they use the `command` property to specify items and directions.

```
function go (req, res) {
  playerGame.clearMessages();
  var command = req.body.command;
  res.json(playerGame.go(command.direction));
}

function use (req, res) {
  playerGame.clearMessages();
  var command = req.body.command;
  res.json(playerGame.use(command.item, command.direction));
}
```

### 23.5.5 Serve static files

The API works with data to update the game. But when a user first visits the game's web page, the server needs to send the static HTML, CSS, JavaScript, and image files that form the application. All those files will be in the `public` folder on the server; it requires only a single line of code to make them available:

```
app.use(express.static('public'));
```

### 23.5.6 Start listening for requests

Finally, you start the server:

```
app.listen(1337);
```

## 23.6 The Crypt—loading data in the browser

The server is ready. It patiently awaits requests from the browser. It doesn't judge; it simply responds. (*Deep down, it may frown on zombie hugging and leopard licking, but on the surface it presents a neutral face and does its job.*) It'll be waiting forever if you don't send it any requests.

You need to send a new request from the browser to the server for every action a player takes. The following listing wraps generic XHR request code into a single `loadData` function. The function includes a callback parameter so you can pass it a function to execute when any response data is received.

#### Listing 23.11 Posting data using XHR (in `dataLoader-xhr.js`)

```
function loadData (url, method, data, callback) {
  var xhr = new XMLHttpRequest();
  var requestData = null;
  xhr.open(method, url);

  if (data) {
    requestData = JSON.stringify(data);
```

Convert any request  
data to a JSON string

```

    xhr.setRequestHeader('Content-Type', 'application/json');
  }

  xhr.addEventListener("load", function () {
    var responseData = JSON.parse(xhr.responseText);
    callback(responseData);
  });

  xhr.send(requestData);
}

```

Set the appropriate request header

Pass response data to the callback function for processing and display

Any project that requires you to send and receive JSON data can use the `loadData` function. For *The Crypt*, you need to send command objects the API expects, so you also use the `postAction` and `getStartData` functions shown in the next listing.

#### Listing 23.12 Posting a command object (dataLoader-xhr.js)

```

(function () {
  "use strict";

  function loadData (url, method, postData, callback) {
    /* listing 23.11 */
  }

  function postAction (command, callback) {
    var url = "/api/" + command.type;

    var data = {
      gameId: theCrypt.gameID,
      playerName: theCrypt.playerName,
      command: command
    };

    loadData(url, "POST", data, callback);
  }

  function getStartData (callback) {
    var url = '/api/start';

    loadData(url, "GET", {}, callback);
  }

  if (window.theCrypt === undefined) {
    window.theCrypt = {};
  }

  theCrypt.data = {
    postAction: postAction,
    getStartData: getStartData
  };

})();

```

Define a function to send command data to the server

Define a function to retrieve the initial game state

The two interface methods, `postAction` and `getStartData`, are called by the game controller code, shown next.

**Listing 23.13 The game controller (gameController.js)**

```

(function () {
  "use strict";

  function render (data) {
    if (data.place) {
      theCrypt.placeView.render(data.place);
    }

    if (data.players) {
      theCrypt.playerView.render(data.players);
    }

    if (data.messages && data.messages.length) {
      theCrypt.messageView.render(data.messages.join('<br />'));
    }
  }

  function updateState (data) {
    theCrypt.gameID = data.gameID;
    render(data);
  }

  function init (playerName, gameID) {
    theCrypt.playerName = playerName;
    theCrypt.gameID = gameID;

    theCrypt.data.getStartData(updateState);
  }

  function doAction (command) {
    theCrypt.data.postAction(command, updateState);
  }

  window.game = {
    do: doAction,
    init: init
  };
})();

```

Define a function to pass data to views

Define a function to call whenever data is loaded

Define a function to start the game

Define a function to send commands to the server

You call the `init` method to start the game in the browser and the `do` method after a user enters a command in the UI and it is parsed. You pass the `updateState` function as a callback to `getStartData` and `postAction`. The data loader code will call `updateState` when the server returns game-state data, and `updateState` will call `render` to update the display in the browser.

All of the code is available in the chapter 23 folder on GitHub: [https://github.com/jrlarsen/GetProgramming/tree/master/Ch23\\_Express/TheCrypt](https://github.com/jrlarsen/GetProgramming/tree/master/Ch23_Express/TheCrypt). To run the game, make sure Express and body-parser are installed (see the following sidebar); then execute `node gameServer` on the command line in the root of the project. Visit `localhost:1337/jahvers-crypt.html` in your browser to play.



### Installing dependencies with npm

If you download the code for the chapter 23 version of *The Crypt* from GitHub, you can install the packages it needs, `express` and `body-parser`, with a single command:

```
npm install
```

npm looks inside the `package.json` file to find which packages to install. During the evolution of the project, each time you've installed a package, you've included the `--save` flag in the command:

```
npm install express --save
```

The `--save` flag automatically adds the package as a dependency in the `package.json` file:

```
{
  "name": "jahvers-crypt-express",
  "version": "1.0.0",
  "description": "A text-based adventure game",
  "main": "gameServer.js",
  "dependencies": {
    "body-parser": "^1.15.1",
    "express": "^4.13.4"
  }
}
```

## 23.7 Summary

- Search on [npmjs.com](http://npmjs.com) for Node packages that perform tasks your programs use.
- Install packages at the command line:

```
npm install express --save
npm install body-parser --save
```

- Make sure you have a `package.json` file that describes your package:

```
{
  "name": "jahvers-crypt",
  "version": "1.0.0"
}
```

- Use Express to build servers that respond to HTTP requests:

```
var express = require("express");
var app = express();
```



- Register handler functions for routes in Express:

```
app.get("/info", getInfo);  
app.post("/update", makeChange);
```

Use get to retrieve  
information

Use post to make  
changes on the server

- Serve static files from a folder of your choice:

```
app.use(express.static("public"));
```

- Access JSON data included in the body of a request with the body-parser package:

```
var bodyParser = require("body-parser");  
app.use(bodyParser.json());
```

- Use methods of the response object to send information back to the client:

```
function getPara (req, res) {  
    res.send("<p>A paragraph for the browser.</p>");  
}  
  
function getData (req, res) {  
    res.json({  
        message: "Data for the browser."  
    });  
}
```

- Set headers when sending JSON data from the browser with XHR:

```
xhr.open("POST", url);  
xhr.setRequestHeader("Content-Type", "application/json");  
xhr.send(data);
```

# Get Programming with JavaScript

John R. Larsen    Foreword by Remy Sharp

**A**re you ready to start writing your own web apps, games, and programs? You're in the right place! **Get Programming with JavaScript** is a hands-on introduction to programming for readers who have never written a line of code.

Since you're just getting started, this friendly book offers you lots of examples backed by careful explanations. As you go along, you'll find exercises to check your understanding and plenty of opportunities to practice your new skills. You don't need anything special to follow the examples—just the text editor and web browser already installed on your computer. We even give you links to working online code so you can see how everything should look live on your screen.

## WHAT'S INSIDE

- All the basics—objects, functions, responding to users, and more
- Think like a coder and design your own programs
- Create a text-based adventure game
- Enhance web pages with JavaScript
- Run your programs in a web browser

No experience required! All you need is a web browser and an internet connection.

**John Larsen** is a web developer and professional teacher in the UK who has many years of experience working with students of all levels, helping them to successfully write their first lines of code.



*"Provides the guidance you need to get started ..., the support to keep practicing, and the encouragement to enjoy the adventure."*

—From the Foreword by Remy Sharp  
Founder of JS Bin

*"A great book for the new programmer who wants to learn JavaScript."*

—Alvin Raj, Oracle

*"An approachable and interactive way of learning JavaScript."*

—Giselle Stidston, Breville Pty Ltd

*"Great interactive code examples! Building a computer game was my favorite part of the book."*

—Ivan Rubelj, Vipnet

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit [manning.com/books/get-programming-with-javascript](http://manning.com/books/get-programming-with-javascript)

ISBN-13: 978-1-61729-310-8  
ISBN-10: 1-61729-310-5



9 781617 293108