

Get Programming with JavaScript

John R. Larsen



 manning



Get Programming with JavaScript

by John R. Larsen

Chapter 24

Copyright 2016 Manning Publications

brief contents

PART 1 CORE CONCEPTS ON THE CONSOLE1

- 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 PROGRAMS169

- 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

24

Polling: repeating requests with XHR

This chapter covers

- Sending repeated, regular requests to the server
- Using `setInterval` to call functions
- Registering middleware for routes in Express
- Keeping players' browsers synchronized in *The Crypt*

The promise of multiuser apps is very appealing; users can chat, collaborate, compete, and improve their efficiency as a team. But with users all applying changes to the same data in the app, how do you ensure they have the latest information at all times? You've seen how XHR works with request-response pairs, where a client sends a request to the server and the server sends back a response to that client. In this chapter, you use XHR to poll the server repeatedly, at regular intervals; you make the browser send requests every five seconds, say, just to ask the server, "Has anything happened?"

You start by writing an auction app that keeps all bidders updated with the latest price for items under the hammer. Then you develop *The Crypt* to manage multiple

games and multiple players, and you use polling to keep the adventurers in the loop with the latest zombie, goblin, and cheese news from their rooms in the tombs.

24.1 High Fives—building an auction app

A local auction house, High Fives, has asked you to write a bidding app. Specializing in low-value items, its simplified auction model of accepting bids only in increments of \$5 has proven to be hugely popular. To test out your bidding code, you quickly create a test page, shown in figure 24.1.

High Fives Auctions			
Item	Bid	Asking	Action
The Fruitinator! Action Figure	5	10	<input type="button" value="Bid"/>

Figure 24.1 An item up for auction on the High Fives site

The page lists items in the auction, shows the current accepted bid value and asking price, and includes a button to place a bid. Clicking the Bid button sends an instruction to the server to add \$5 to the bid value. The server sends back the new bid value to the browser and it updates its display (figure 24.2).

High Fives Auctions			
Item	Bid	Asking	Action
The Fruitinator! Action Figure	10	15	<input type="button" value="Bid"/>

Figure 24.2 Clicking the Bid button has increased the bid and asking amounts by 5.

An auction needs multiple bidders to be exciting (and profitable), and figure 24.3 shows a second bidder visiting the site along with the first. The second bidder's browser loads the latest auction information from the server, and the figure shows the bid and asking values matching for both browsers.

You need to write code on the server and browser that synchronizes the display for all bidders. It's essential that you avoid what's shown in figure 24.4, where the second bidder has submitted a new bid but the first bidder's browser hasn't updated its information.

You can't have a visitor thinking they're slipping in a cheeky \$15 bid when the asking price is really at \$150! The seller's The Fruitinator! action figure may be mint and

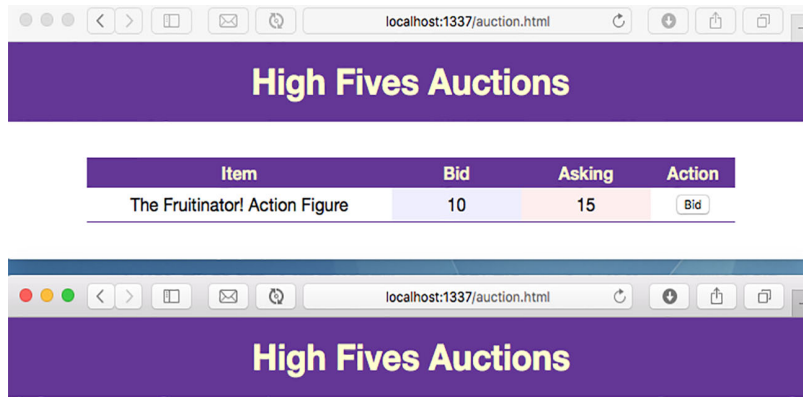


Figure 24.3 When a second bidder visits the site, they are shown the latest bid and asking values.

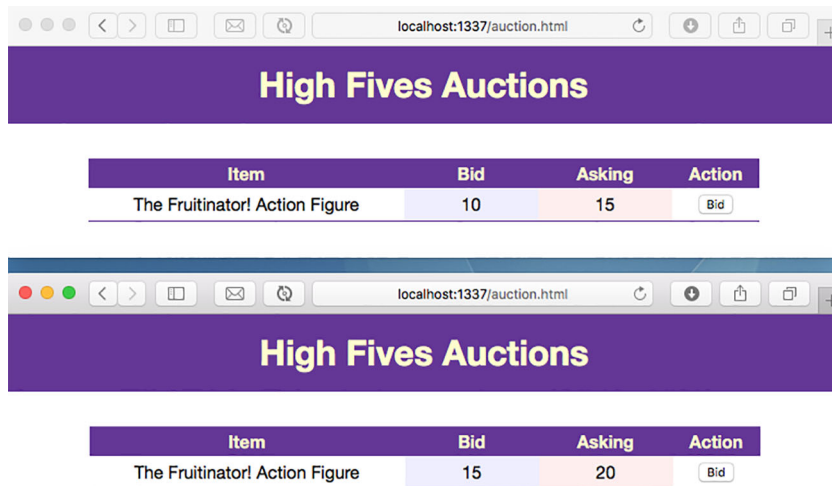


Figure 24.4 When the second bidder makes a bid, the site hasn't updated the first bidder's display. You need to add code to keep all bidders in sync.

still in its blister pack, but you don't want out-of-date asking prices leading to any legal splatage, kumquat accessory or no kumquat accessory.

In this chapter, you look at one way of tackling the problem: using XMLHttpRequest.

24.2 High Fives—polling the server for updates

Make a new folder and add the package.json file listed here, to describe the project for npm.

Listing 24.1 Project info (package.json)

```
{
  "name": "high-fives",
  "version": "1.0.0"
}
```

Your bid program uses Express to simplify development, so navigate to your project folder on the command line, and use npm to install the Express package:

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

The auction server itself is very simple and is shown in the next listing. It has only two routes: /state and /bid. Save it as auctionApp.js.

Listing 24.2 The High Fives server (auctionApp.js)

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

var bid = 5;

app.get('/state', getState);
app.post('/bid', makeBid);

function makeBid (req, res) {
  bid = bid + 5;
  res.json({ bid: bid });
}

function getState (req, res) {
  res.json({ bid: bid });
}

app.use(express.static("public"));
app.listen(1337);
```

**Declare a variable
to keep track of
the bid value**

**Register route handlers
for the two routes**

**Define a handler to
update the bid and send
the updated value**

**Define a handler to send
the current bid value**

**Serve static files from
the public directory**

The /state route simply returns information; it doesn't make any changes to the bid. You use app.get to register its handler. The /bid route, on the other hand, does change the bid. You use app.post to register its handler. There are other possible methods you might use when setting up your routes: put, patch, and delete, for example. You stick with get and post; further discussions of which methods to use when are for another book.

The High Fives Auctions web page uses a table to list the auction items (although you include only one item for your current test). Four of the elements in the table are

given IDs: three table cells and a button. The following listing shows the full HTML file, saved in the public folder within your project folder.

Listing 24.3 The auction web page (auction.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>High Fives Auctions</title>
  <link rel="stylesheet" href="auction.css" />
</head>
<body>
<h1>High Fives Auctions</h1>

<table>
  <thead>
    <tr>
      <th>Item</th>
      <th>Bid</th>
      <th>Asking</th>
      <th>Action</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td id="item">The Fruitinator! Action Figure</td>
      <td id="bid">...loading...</td>
      <td id="asking">...loading...</td>
      <td><button id="btnBid">Bid</button></td>
    </tr>
  </tbody>
</table>

<script src="auction.js"></script>
</body>
</html>
```

Use a table to list the auction items

Assign IDs to elements you want to access in JavaScript

Load the JavaScript file that drives the page's behavior

The JavaScript for the auction web page is shown in listing 24.4. It uses XHR to send and receive auction updates and a method, `setInterval`, to repeatedly poll the server for the latest information. The code will perform these tasks:

- Get references to elements on the page
- Define a function for loading data via XHR
- Send POST requests to bid in the auction
- Send GET requests to retrieve the latest information
- Define a function to update the display when data is loaded
- Retrieve initial information when the page is first loaded
- Repeatedly poll the server to make sure the auction information is current

Save the `auction.js` file in the public folder; it'll be loaded by the `auction.html` web page.

Listing 24.4 Client-side code (auction.js)

```

var bid = document.getElementById("bid");
var asking = document.getElementById("asking");
var bidButton = document.getElementById("btnBid");

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

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

    xhr.send();
}

function updateDisplay (data) {
    bid.innerHTML = data.bid;
    asking.innerHTML = data.bid + 5;
}

function sendBid () {
    loadData("/bid", "POST", updateDisplay);
}

function getState () {
    loadData("/state", "GET", updateDisplay);
}

bidButton.addEventListener("click", sendBid);

getState();

setInterval(getState, 1000);

```

Define a function to load data using XHR

Define a function to update the display with the latest data

Set updateDisplay as a callback function to be called when data is loaded

Call sendBid whenever the button is clicked

Retrieve the current information for initial display

Call getState every 1000 milliseconds

The browser provides the `setInterval` method that executes a function repeatedly, with each request separated from the previous one by an interval of time specified in milliseconds. The auction page will poll the server once a second.

To test your High Fives Auction site, start the server with `node auctionApp` and load the `localhost:1337/auction.html` page in a browser. Try submitting a few bids. Open a second browser window for the auction. Make bids in either browser and watch the other update.

It should work well enough. Unfortunately, polling the server repeatedly is inefficient. You're sending requests from the browser whether or not any bids have been made. It would be great if the server could send messages to the browser without first having to receive a request. That's what the WebSocket protocol provides: a way of initiating messages from the browser and the server. You won't look at WebSocket directly; you'll use the `Socket.IO` package, in chapter 25, to provide a simple way of sending and receiving messages.

Before moving on to creating a multiplayer version of *The Crypt*, have a play with `setInterval` on JS Bin.

24.2.1 The `setInterval` method

Figure 24.5 shows `setInterval` being used on JS Bin to repeatedly log a message to the console.



Figure 24.5 Repeating a message with `setInterval`

The following listing shows the code used to generate the repeating message. It passes two arguments to `setInterval`: the `sayHello` function and a duration in milliseconds.



Listing 24.5 Repeating a message with `setInterval` (<http://jsbin.com/yulili/edit?js,console>)

```
function sayHello () {
  console.log("Hello World!");
}
```

Declare a function

```
setInterval(sayHello, 2000);
```

← **Pass the function and a duration to `setInterval`**

Explore `setInterval` by changing the interval duration and calling other functions. Can you use `setInterval` to call multiple functions?

24.3 The Crypt—running a central server

Up until chapter 23, users had been loading all of the game code for *The Crypt*, a large collection of JavaScript files, into their browsers. Each player had their own version of the game. Once loaded, the game had no need to talk to the server. Figure 24.6 shows the clients (each player's browser) with their own games and no server interaction.

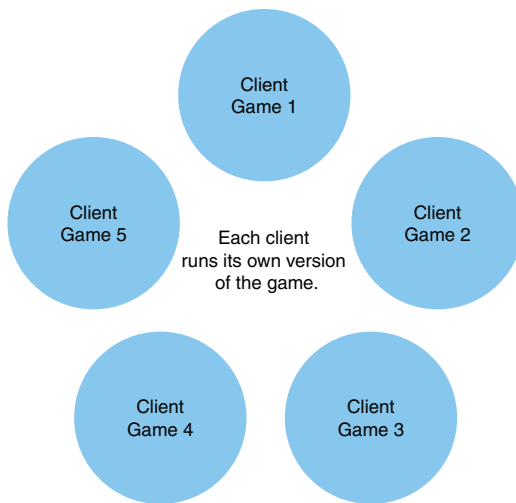


Figure 24.6 Each client runs its own version of the game, without server interaction.

In chapter 23 you set up a server to manage a single game. The browser would relay user actions to the server, and the server would update the game and send the new game state back to the browser.

You now want to let multiple players enjoy *The Crypt* simultaneously. Their browsers will continually interact with the server as they send commands and receive game-state updates. Figure 24.7 shows the games being managed on the server. Each client sends

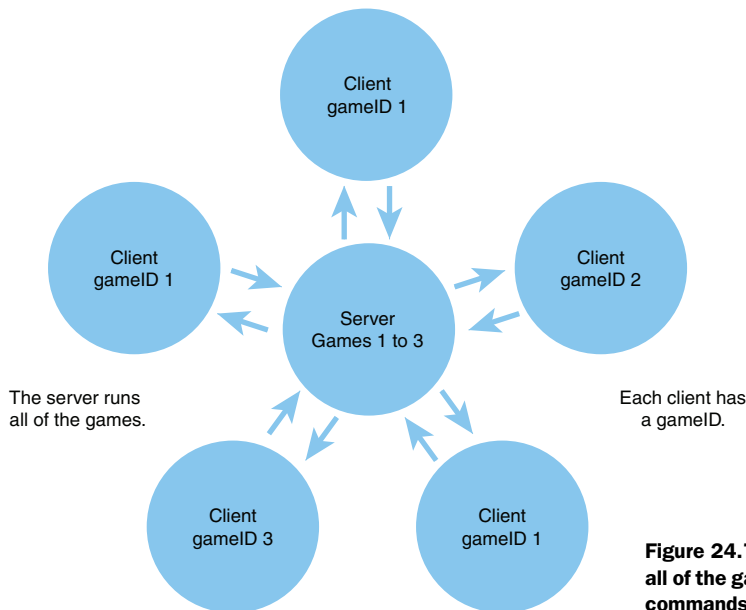


Figure 24.7 A central server runs all of the games. Clients send game commands to the server.

requests with commands to update *its* game and the server sends responses back with the latest state of *that* game.

The clients include a `gameID` property with every request so that the server knows which game to update. Multiple players can play in each game, so each browser will also have to let the server know its player's name, and the server will have to manage multiple player game objects.

Let's start by updating the browser to allow users to enter their names and, if they want to join an ongoing adventure, a game ID.

24.4 The Crypt—managing multiple games on the browser

Although some players will want to plow a lonely furrow, others are more gregarious and prefer to work as part of a team—in a friendly furrow. For players to work the fields together, you need to provide a game ID for their common game. The first player will start the game and will be supplied an ID to share. If subsequent players use the same ID, then they'll join the same game.

Figure 24.8 shows a new start screen for *The Crypt*. The first player of a new game enters their name but leaves the ID box blank.



The Crypt

Name
Jahver

Game ID (to join an existing game)

Start Your Adventure

Figure 24.8 Players enter their name and, optionally, the ID of a game to join.

Once the first player has started the game, the browser shows their name and the ID at the bottom of the screen (figure 24.9).

Subsequent players who wish to join the game load the web page in their browsers and enter their name and the shared game ID, as shown in figure 24.10.



Figure 24.9 The initial game screen with player name and game ID at the bottom

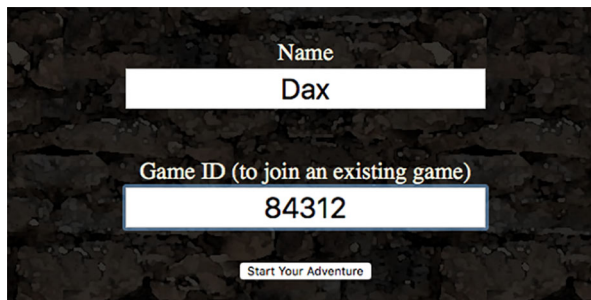


Figure 24.10 A new player can join an existing game by entering its ID on the start screen.

If the first player hasn't moved to a different location in the game, then both players will be shown in the second player's browser (figure 24.11).

Unfortunately, the first player's display won't automatically update to show the second player until the first player takes some kind of action in the game. You'll look at ways of updating all players' displays in response to one player's actions later in the chapter. Here, you implement the browser code for the start screen. All the client-side code is in the project's public folder.



Figure 24.11 Two players are playing the same game.

24.4.1 Adding the start screen to the HTML

When a player first visits the game's web page, it presents them with the text boxes for their name and the game ID; it hides the usual game views. To hide the view elements, you place them in a div element with a class of `hidden`, as shown here.

Listing 24.6 The game views are hidden initially (in `jahvers-crypt-xhr.html`)

```
<div id="main" class="hidden">
  <div id="messages"></div>
  <div id="controls">
    <input type="text" id="txtCommand" />
    <input type="button" id="btnCommand" value="Make it so" />
  </div>
  <div id="views">
    <div id="place"></div>
    <div id="players"></div>
  </div>
</div>
```

← Assign the div a class of "hidden"

The page uses CSS to hide any elements with a class of `"hidden"` (see sidebar). The next listing shows the markup for the start screen name and ID text boxes. The page displays them when a user loads the web page.

Listing 24.7 The start screen (in `jahvers-crypt-xhr.html`)

```
<div id="startControls">
  <label for="txtPlayerName">Name</label>
  <input id="txtPlayerName" />
```

```

<label for="txtGameID">Game ID (to join an existing game)</label>
<input id="txtGameID" />

<p><button id="btnStart">Start Your Adventure</button></p>
</div>

```

When the game begins, you replace the contents of the `startControls` div with the player's name and the game's ID.

Showing and hiding elements by setting classes

You use CSS to modify the presentation of elements on a page. You can hide elements by setting their `display` property to `none`. To hide all elements with a class of `hidden`, use this rule:

```

.hidden {
    display: none;
}

```

The browser would not show the following paragraph:

```
<p id="example" class="hidden">Boo!</p>
```

To show the paragraph, you can remove the `hidden` class with JavaScript:

```

var para = document.getElementById("example");
para.className = "";

```

Ta-da!

24.4.2 Hiding the start screen and showing the game views

To manage switching from the initial text boxes to the standard message, player, and place views, you create a new main view. As usual for views, it has a `render` method that updates the browser display. When you call `render`, it shows the div with an `id` of `main` by removing the `hidden` class and replaces the name and ID text boxes with the player's name and the game ID. The code is shown here.

Listing 24.8 The main view (`mainView.js`)

```

(function () {
    "use strict";

    var mainDiv = document.getElementById("main");
    var startFormDiv = document.getElementById("startControls");

    var userDetailsTemplate =
        document.getElementById("userDetailsTemplate")
            .innerHTML;

```

**Get hold of the
new user details
template**


```

function showMain () {
  mainDiv.className = "";
}

function renderUser () {
  startFormDiv.innerHTML = gpwj.templates.fill(
    userDetailsTemplate,
    {
      playerName: theCrypt.playerName,
      gameID: theCrypt.gameID
    }
  );
}

function render () {
  showMain();
  renderUser();
}

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

theCrypt.mainView = {
  render: render
};

})();

```

← Show the main views by removing the class of hidden

← Replace the player name and game ID text boxes with user details

The client-side game controller code calls `mainView.render` when the browser loads data from the server, data that includes the `playerName` and `gameID` properties. The following listing shows the updated `render` function for the game controller.

Listing 24.9 The controller's render function (in `gameController.js`)

```

function render (data) {
  theCrypt.mainView.render();
  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 />'));
  }
}

```

← Make the main view visible and replace the initial text boxes with user details

24.4.3 Listening for the start of the game

To start the game, a player enters their name and, optionally, a game ID and then clicks the Start Your Adventure button. You need code to listen for that button click. The following listing shows the code used to kick off proceedings.

Listing 24.10 Starting the game (startGame.js)

```

var btnStart = document.getElementById('btnStart');

btnStart.addEventListener('click', function () {
    var playerName = document.getElementById('txtPlayerName').value;
    var gameID = document.getElementById('txtGameID').value;

    game.init(playerName, gameID);
});

```

Listen for the click event

Start the game

24.5 The Crypt—managing multiple games on the server

In chapter 23, you created a game server that managed a single game with a single player. The server now needs to manage multiple games and multiple players. A game object, an object created with the Game constructor, represents each game. The game maintains a list of all its players. A player game object, an object created with the PlayerGame constructor, represents each player in a game. The player game object provides the game methods like get, go, and use.

The game server must be able to retrieve the correct object when a command request arrives from the browser. The server must handle three types of requests:

- Start a new game
- Join a game
- Take an action in an existing game

You create a game manager module to start, join, and retrieve games. The new module works with the game server to manage the multiplayer version of *The Crypt*. Figure 24.12 shows the game manager's place among the server-side modules.

The code for the game manager module is shown here and will be discussed next.

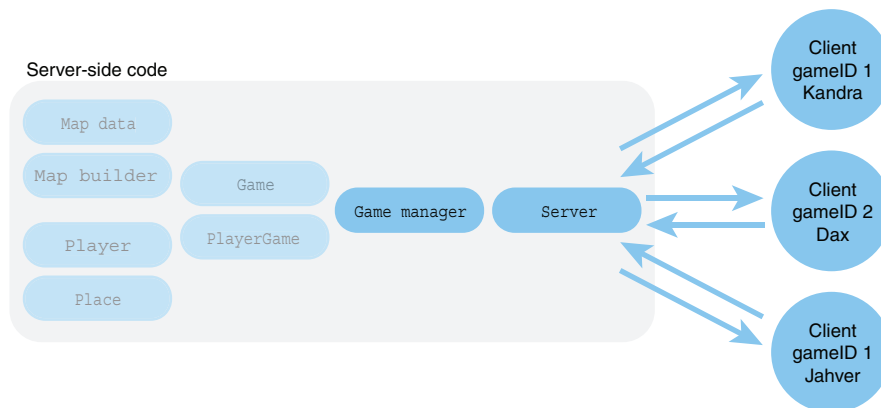


Figure 24.12 The game manager helps the server handle multiple games for multiple players. Kandra and Jahver are playing the same game, while Dax is playing separately.

Listing 24.11 The game manager (gameManager.js)

```

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

var games = {};
var playerGames = {};

function getID () {
    var id;
    do {
        id = Math.floor(Math.random() * 100000 + 1);
    } while (games[id]);
    return id;
}

function getNewGame () {
    var game = new Game(getID(), mapData);
    games[game.id] = game;
    return game;
}

function joinGame (playerName, id) {
    var game = games[id] || getNewGame();

    var playerGame = new PlayerGame(playerName, game);
    playerGames[game.id + "_" + playerName] = playerGame;

    return playerGame;
}

function getPlayerGame (playerName, id) {
    return playerGames[id + "_" + playerName];
}

module.exports = {
    join: joinGame,
    getPlayerGame: getPlayerGame
};

```

Use objects to act as caches

Define a function to generate random, unique IDs

Create a new game and add it to the cache

Join a game, creating it first if necessary

Retrieve a player game from the cache

24.5.1 Caching games

The manager puts games and player games into caches, ready for retrieval. For games, it uses the game's ID as the key. For player games, it uses a combination of game ID and player name.

```

games[game.id] = game;
playerGames[game.id + "_" + playerName] = playerGame;

```

Say Kandra is a player in a game with ID 34567. You could retrieve the game and player game from the caches like this:

```

game = games[34567];
playerGame = playerGames["34567_Kandra"];

```

24.5.2 Generating unique game IDs

Furrow goblins are a menace and you may need to work with other players to overcome the gamboling goblins' violent pranks. But if you were expecting to soak in the gentle pleasures of a team retreat at Unicorn Lake, finding yourself in a goblin-infested furrow may leave you mildly dischuffed and short on rainbows. You need to join the right game.

You use game IDs to join games in progress. To separate one game from another, the IDs must be unique, and to make it tricky for rogue players to gatecrash your game, the IDs should be hard to guess. The `getID` function generates random numbers until it finds one that no game is using as an ID:

```
function getID () {  
  var id;  
  
  do {  
    id = Math.floor(Math.random() * 100000 + 1);  
  } while (games[id]);  
  
  return id;  
}
```

The do-while loop executes its code block, generating a random number. If `games[id]` exists, the while condition will evaluate to `true` and the do-while loop will execute its code block again, creating another random ID. If `games[id]` is undefined, it'll evaluate to `false` and `getID` will return the generated ID.

The `getID` function is not a robust solution; it'll slow down as IDs are used up and get stuck in an infinite loop once all the IDs are taken. As a separate function, you can easily improve the implementation at a later date. In production, you might choose to have user accounts and sessions as well.

24.5.3 Joining games

Use the `joinGame` function to join an existing game or to start and join a new game. Include an ID as an argument if you want to join an existing game. The function first tries to retrieve a game from the cache:

```
var game = games[id] || getNewGame();
```

If `games[id]` is undefined, in other words there's no game with that ID, then the statement calls `getNewGame` to create a new game instead. The statement uses the OR operator, `||`, which returns the first operand, `games[id]`, if the first operand evaluates to `true` and the second operand, the game that `getNewGame` returns, otherwise.

24.5.4 Updating the game-server with middleware

Ready with your new game manager module, you update the server code to use the module's functionality. The following listing shows the latest code. It includes a call to `app.use` that makes use of a *middleware* function, discussed after the listing.

Listing 24.12 Updating the game-server (gameServer.js)

```

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

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

var games = require('./lib/gameManager');

app.use('/api/*', function (req, res, next) {
    req.playerGame = games.getPlayerGame(
        req.body.playerName,
        req.body.gameID);

    next();
});

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

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

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

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

function start (req, res) {
    var playerGame = games.join(
        req.body.playerName,
        req.body.gameID);

    res.json(playerGame.getData());
}

app.use(express.static("public"));
app.listen(1337);

```

Import the game manager

Use a middleware function to retrieve games for /api routes

Use the player game retrieved by the middleware

Start or join a game

Don't forget to serve static files from the public folder

The `/get`, `/go`, and `/use` routes all need access to an existing player game object. Rather than repeating code in the routes, you make use of a middleware function. Express calls middleware functions in a sequence, with each function performing a specific task. When a middleware function has completed its task, it calls `next` to pass execution to the next function in the chain. Express manages the chain of middleware functions.

In the previous listing, you register a middleware function before the existing routes. You use a wildcard character, `*`, to match all routes that start with `/api/`.

```
app.use('/api/*', function (req, res, next) {
    req.playerGame = games.getPlayerGame(
        req.body.playerName,
        req.body.gameID);
    next();
});
```

Express passes the next function as an argument

Retrieve the player game from the game manager and assign it to a property of the request object

Call the next function to let Express know the function is finished

The middleware function uses data sent by the browser to retrieve a player game from the game manager and assigns the player game to a property of the request object, `req.playerGame`. It then calls `next`, to let Express know that its work is done. Express will call the subsequent function that matches a route. But that subsequent route handler will now have access to `req.playerGame`.

And with that sprinkle of middleware magic, your game server is ready for multiple players in multiple games. (Okay, you could add various checks for errors: what if the player game is undefined, for example?)

24.6 The Crypt—keeping players updated

Dax and Kandra are playing the same game and they're both in The Kitchen. That cheese sure is tempting! Dax gets the cheese, and his browser display updates to show the cheese among his items and no longer in the room's items. But Kandra's display doesn't update; she has no idea Dax has grabbed the *fromage*. Kandra's display will update only when her browser receives a response to a request that *it* sends the server.

In fact, while Dax and Kandra have been pondering dairy produce, Jahver moved into The Kitchen, dissolved the zombie, and moved south to The Old Library. Jahver saw Dax and Kandra on his display, but Dax and Kandra's displays did not show Jahver. Ninja skills!

Although you can successfully play a mysterious multiplayer game with such partial information, it'll be better if all players are updated whenever a game event takes place. Sticking with XHR for now, you need each player's browser to repeatedly ask the server for updates at regular intervals.

24.6.1 Adding a route for state requests

The game manager and middleware code you’ve already put in place makes it easy to add a new route to your game API. In `gameserver.js`, add one more route:

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

← Add a route that will send back the state of a specified player game

Notice that the new route uses `app.post`. Even though the handler will make no changes to the game—the request is just retrieving information—you use `app.post` so that your middleware function can use the body parser to retrieve information from the body of the request. The browser includes the game ID and player name as data with its request. Add the `getState` route handler as well:

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

← Use the `playerGame` property added to the request object by your middleware function

Your middleware function retrieves the appropriate player game for each request, so you can access it at `req.playerGame`.

The server is ready to handle requests for the latest state info. Next, you update the browser’s data-loading code to send those requests.

24.6.2 Polling the server from the browser

The act of sending requests for information is called *polling*. Each player’s browser will poll the server every five seconds. The browser is essentially asking, “Has anything happened?” ... “What about now?” ... “And now?” ... “What about now?” ... and so on, and on, and on. It’s not very efficient, but it’ll do the job.

This last listing uses `setInterval` to call the `pollState` function repeatedly.

Listing 24.13 Polling the server (in `dataLoader-xhr.js`)

```
function loadData (url, method, postData, callback) {
  /* unchanged */
}

function postAction (command, callback) {
  /* unchanged */
}

function pollState (playerName, gameID, callback) {
  var url = "/api/state";

  var data = {
    gameID: gameID,
    playerName: playerName
  };
```

← Define a function that requests the latest data from the server

```

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

function getStartData (callback) {
    var url = '/api/start';
    var gameId = theCrypt.gameID;

    var data = {
        playerName: theCrypt.playerName
    };

    if (gameID !== undefined && gameId !== "") {
        data.gameID = gameId;
    }

    loadData(url, "POST", data, function (resData) {

        setInterval(function () {
            pollState(
                theCrypt.playerName,
                resData.gameID,
                callback);
        }, 5000);

        callback(resData);
    });
}

```

Once the initial data loads, start polling the server every 5 seconds

Start the game server by running `node gameServer` on the command line. Visit `localhost:1337/jahvers-crypt-xhr.html` in a number of browser windows. Start new games and join existing ones.

Polling does the job of keeping players' browsers synchronized, but it's very inefficient. Players may take a while to ponder their next move; is it cheese, Spam, or holy water for zombie removal? While no actions are being taken, all the players' browsers are busy requesting the latest information, again and again. Chapter 25 describes working with `Socket.IO`, a JavaScript library that does away with repeated polling wherever possible and lets you send messages only when there's some information worth shouting about.

24.7 Summary

- Keep multiple browsers synchronized by *polling* the server from each browser. Send a request for the latest information at regular intervals. The length of the interval depends on how often the information is likely to change and how important it is that users see the latest data immediately.
- Use the `setInterval` method to call a function repeatedly, with a specified duration, in milliseconds, between calls.
- Call `setInterval` with a previously defined function:

```

function sayHello () {
    console.log("Hello World!");
}

setInterval(sayHello, 3000);

```


- Call `setInterval` with a function expression:

```
setInterval(function () {  
    console.log("Hello World!");  
}, 3000);
```

- Use middleware with Express to call a chain of functions that each perform a specific task:

```
app.use(middleware1);  
app.use(middleware2);  
app.use(middleware3);
```

The middleware functions will be executed in the order in which they are registered

- Don't forget to call `next` in your middleware functions, to let Express know that each function has finished its task. When you call `next`, Express will automatically call the next middleware function in the chain:

```
function middleware1 (req, res, next) {  
    // Do some great stuff  
    next();  
}
```

← **You could add properties to the request object, for use by subsequent functions in the middleware chain**

- Register middleware for specific routes or families of routes:

```
app.use("/api/*", middleware1);
```

Get Programming with JavaScript

John R. Larsen Foreword by Remy Sharp

Are 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

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



9 781617 293108