

# Get Programming with JavaScript

John R. Larsen



 **manning**



# *Get Programming with JavaScript*

by John R. Larsen

## **Chapter 25**

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

# 25

## *Socket.IO: real-time messaging*

---

### ***This chapter covers***

- Using Socket.IO on the server and in the browser
- Sending and receiving messages
- Targeting messages with rooms

You can use XHR for synchronizing multiple browsers accessing the same data on a server. But the method you used in chapter 24, polling, isn't efficient; it sends repeated requests, even when no updates are needed. It's possible to send a request and leave it open until the server is ready to send a response, a technique called *long-polling*, but it would be better to have proper two-way communication between the server and the client.

The WebSocket protocol lets servers send messages to clients without having to wait for a request first. In this chapter you use Socket.IO, a package that provides real-time communication functionality on the server and in the browser. It uses WebSocket where possible, falling back on other techniques in older browsers. With it, you update your auction app to keep bidders in the loop, and you finish your multiplayer version of *The Crypt*, enabling *CryptFlash*<sup>™</sup> technology for the latest monster-news headlines.

## 25.1 High Fives—sending messages with Socket.IO

In chapter 24 you created a simple auction app for High Fives Auctions, to test out your ideas for a multiuser system. You used XHR to send requests from each browser to the server for the latest bids, sending the requests once a second. Figure 25.1 shows the interface for displaying bids and making a bid.

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

**Figure 25.1** Your initial High Fives Auctions interface

Rather than the browser requesting information from the server again and again and again, it would be better if the server sends messages to any interested parties as soon as the bid value changes. Socket.IO gives you the tools to do just that.

### 25.1.1 Using Socket.IO on the server

Navigate to your project root and install Socket.IO using npm. (If you're starting a new project, don't forget a package.json file.)

```
npm install socket.io --save
```

The following listing shows the new server code.

**Listing 25.1** The auction server with socket.io (auctionApp.js)

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

var server = require('http').Server(app);
var io = require('socket.io')(server);

var bid = 5;

io.on('connection', socketsRouter);

function socketsRouter (socket) {

    socket.on('bid', makeBid);
    socket.on('state', getState);

    function makeBid () {
        bid = bid + 5;
        io.emit('update', { bid: bid });
    }
}
```

← Create an HTTP server

← Tell Socket.IO to use the server

← Register a handler to run whenever a client connects using Socket.IO

Register handlers for routes the connected client might use

← Send bid data to all connected clients

```

function getState () {
  socket.emit('update', { bid: bid });
}
}

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

server.listen(1337);

```

← Send bid data to the client that sent the state message

Socket.IO doesn't use the Express app as a server; it uses the underlying Node HTTP server. You import the `http` module, create a server, and tell your Express app to use that server. You also set up Socket.IO to use the same server.

```

var server = require('http').Server(app);
var io = require('socket.io')(server);

```

### CREATING SOCKETS FOR NEW CONNECTIONS

Socket.IO is event driven; you register handler functions to run when certain events occur. Whenever a client connects, you want to set up event handlers for that connection.

```

io.on('connection', socketsRouter);

```

You tell Socket.IO to call the `socketsRouter` function whenever a new client connects to the server. Socket.IO automatically passes the `socketsRouter` function an object, a *socket*, representing the connection. You use the object to send and receive messages from the client that made the connection.

```

function socketsRouter (socket) {
  // Send and receive messages
  // using socket
}

```

← Socket.IO passes the function an object representing the connection to a client

### SETTING HANDLERS FOR SOCKET EVENTS

Once the client has connected to the server, the client can send messages to the server and the server can send messages to the client. On the server, you assign handlers to events, the Socket.IO version of routes.

```

socket.on('bid', makeBid);
socket.on('state', getState);

```

When the client emits a 'bid' event, call the `makeBid` function. When the client emits a 'state' event, call the `getState` function.

**SENDING MESSAGES TO THE CLIENTS**

You want to keep all bidders at the auction in sync. You need to send messages to all of the connected clients whenever a bid comes from one. To send messages across all the sockets, call `io.emit`.

```
function makeBid () {
  bid = bid + 5;
  io.emit('update', { bid: bid });
}
```

← **Send the latest bid info to all connected clients**

To send a message to a single client, use their socket object and call `socket.emit`. For example, when a bidder first loads the auction website, they want the latest information, but they aren't making a bid.

```
function getState () {
  socket.emit('update', { bid: bid });
}
```

← **Send the latest bid info to a single client**

In both the `makeBid` and the `getState` functions, you emit an 'update' event. The browser will have to listen for that event, so that it can update its display. But how's the client sending and receiving these messages? You need to use Socket.IO in the browser code as well.

**25.1.2 Using Socket.IO on the browser**

You've installed Socket.IO on the server using npm. Importing and using Socket.IO in your server code also makes it available to load on the browser. The listing here, from `auction.html` in the public folder, shows the script tags you use to load Socket.IO on the browser.

**Listing 25.2 The auction web page (in `auction.html`)**

```
/* rest of page */

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

The browser loads the code and Socket.IO creates a global `io` variable you can use to create a socket object. The socket represents the connection to the server. You use it to emit events and listen for events, as shown in the next listing, saved as `auction.js` in the public folder.

**Listing 25.3 The client-side code (`auction.js`)**

```
var bid = document.getElementById("bid");
var asking = document.getElementById("asking");
var bidButton = document.getElementById("btnBid");
var socket = io('http://localhost:1337');
```

← **Use io to create a socket**



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

function sendBid () {
  socket.emit('bid');
}

bidButton.addEventListener("click", sendBid);
socket.on('update', updateDisplay);
socket.emit('state');
```

Use the socket to emit an event to the server

Listen for update events from the server

Ask the server for the latest bid info

Give it a whirl! Start the server with `node auctionApp` and visit the auction page at `localhost:1337/auction.html` in a couple of browser windows. A bid from one browser window should automatically update the other.

That's pretty nifty stuff. But sometimes you want to send messages from the server to a selected group of clients, not to all of them. Socket.IO makes that possible with *rooms*.

### 25.1.3 Working with multiple rooms

Okay, you have the real-time messaging up and running but only for a single item. The Fruitinator! is an insanely popular app, but buyers at the auction probably want a little more variety. You pluck up the courage and update the bidding app to include three items. (Steady—you don't want a bidding frenzy!) Figure 25.2 shows

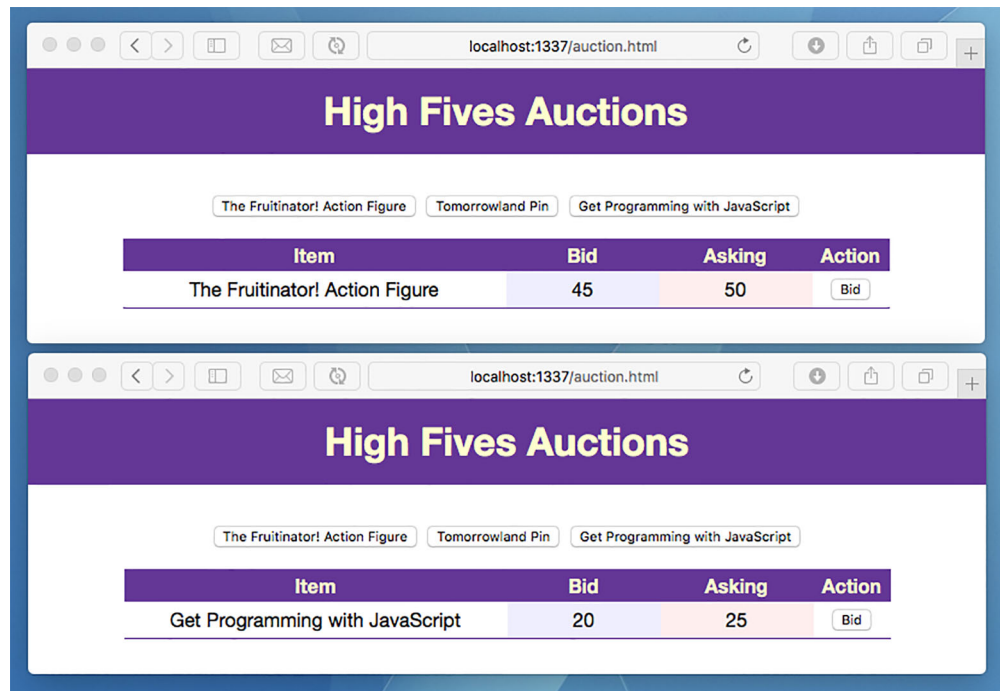


Figure 25.2 You can now bid on different items in the auction.

two visitors to the auction page. The page now has buttons to switch among the three items up for sale.

Using XHR to repeatedly and continually poll the server was inefficient, so you switched to Socket.IO for your messaging needs. In a similar bid for efficiency, you want to send bid updates for an item only to those bidding on *that* item. Those bidding on the Tomorrowland pin don't need updates about action figures and amazing programming books.

Socket.IO lets you send messages targeted at sockets that belong to a room, with each room given a unique name. Sockets can join and leave a room by using the room's name:

```
socket.leave('Tomorrowland Pin');
socket.join('The Fruitinator! Action Figure');
```

You can then emit events just to sockets in a room:

```
io.to('The Fruitinator! Action Figure').emit('update', bidData);
```

The following listing shows the server code updated to include three auction items and a room for each item.

#### Listing 25.4 The auction server with rooms (auctionApp.js)

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

var server = require('http').Server(app);
var io = require('socket.io')(server);

var bids = {
  "The Fruitinator! Action Figure": 5,
  "Tomorrowland Pin": 5,
  "Get Programming with JavaScript": 5
};

io.on('connection', socketsRouter);

function socketsRouter (socket) {

  socket.on('bid', makeBid);
  socket.on('state', getState);

  function makeBid (data) {
    var item = data.item;
    bids[item] += 5;

    var bidData = {
      item: item,
      bid: bids[item]
    };

    io.to(item).emit('update', bidData);
  }
}
```

Store bid info  
for three  
items

Send the latest  
info to sockets in  
the item's room

```

function getState (data) {
  var item = data.item;

  var bidData = {
    item: item,
    bid: bids[item]
  };

  socket.leave(socket.currentRoom);
  socket.join(item);
  socket.currentRoom = item;

  socket.emit('update', bidData);
}

}

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

server.listen(1337);

```

Leave the previous room

Join the room for the requested item

Keep a record of the room joined

For the auction app, you want sockets to belong to only one room at a time. You assign the current room's name to `socket.currentRoom`, a property you add to the socket object. You store the name so the socket can leave the current room before joining a new one.

The auction page shown in figure 24.6 included three buttons. The HTML for the buttons is shown here.

#### Listing 25.5 The item buttons (in auction.html)

```

<p id="itemButtons">
  <button>The Fruitinator! Action Figure</button>
  <button>Tomorrowland Pin</button>
  <button>Get Programming with JavaScript</button>
</p>

```

The buttons are children of a paragraph element with an id of "itemButtons". You'll use the paragraph to get at the buttons rather than giving each of them their own id attribute.

The following listing shows the extra code needed on the browser to cope with the multiple items available in the auction.

#### Listing 25.6 Client-side code for multiple rooms (auction.js)

```

function getById (id) {
  return document.getElementById(id);
}

var item = getById("item");
var bid = getById("bid");
var asking = getById("asking");

var socket = io('http://localhost:1337');

```

```

var currentItem;

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

function sendBid() {
  socket.emit('bid', {item: currentItem});
}

function watchItem (e) {
  var button = e.target;
  currentItem = button.innerHTML;
  socket.emit('state', {item: currentItem});
}

function addListeners () {
  var itemButtons = getById("itemButtons");
  var bidButton = getById("btnBid");

  var buttons = itemButtons.getElementsByTagName("button");
  var i, len;

  for (i = 0, len = buttons.length; i < len; i++) {
    buttons[i].addEventListener("click", watchItem);
  }

  bidButton.addEventListener("click", sendBid);

  socket.on('update', updateDisplay);
}

addListeners();

```

← **Declare a variable to hold the name of the current item**

← **Include the name of the item on which to bid**

← **Include a parameter for the event object**

← **Use the event object to obtain a reference to the button clicked**

← **Get a list of the item buttons**

← **Tell each button to call the watchItem function when clicked**

You use the `getElementsByTagName` method, available in the browser, to obtain a list of the three buttons. You then use a `for` loop to assign an event handler to each button. When any one of the buttons is clicked, the code will call the `watchItem` function, passing it an event object, `e`. You haven't used event objects before; they include a number of properties related to the event that caused the handler to run. Here, you use the event object to obtain a reference to the button that was clicked, `e.target`. The button includes the name of the item, and that name is sent to the server when the socket emits its event.

Test out the new auction app. Multiple clients should be able to bid on the three items with only relevant clients receiving updates for the items they're watching.

You upgraded the auction app from one that continually polls the server with XHR requests to one that emits and receives events with Socket.IO. You follow that same upgrade path as you finally take *The Crypt* into the land of *CryptFlash*<sup>TM</sup> technology.

## 25.2 The Crypt—spreading the love

Players of *The Crypt* let you know that waiting a couple of seconds for their browsers to synchronize with the game server engenders a frisson of uncertainty that they find uncomfortable and unwelcome. The thought of another player getting a head start and preemptively pocketing the Spam or slipping through a newly opened door fills them with distress. You need to send them a news flash about the latest activity in their location the moment it happens.

You decide to use Socket.IO to broadcast immediate zombie, vampire, and leopard updates to those with a stake in proceedings. But, because you're managing multiple players in multiple rooms in multiple games, you need more than a couple of sockets and an event or two: in a burst of humble activity, you create *CryptFlash™* technology!

### ***CryptFlash™* technology—a disclaimer**

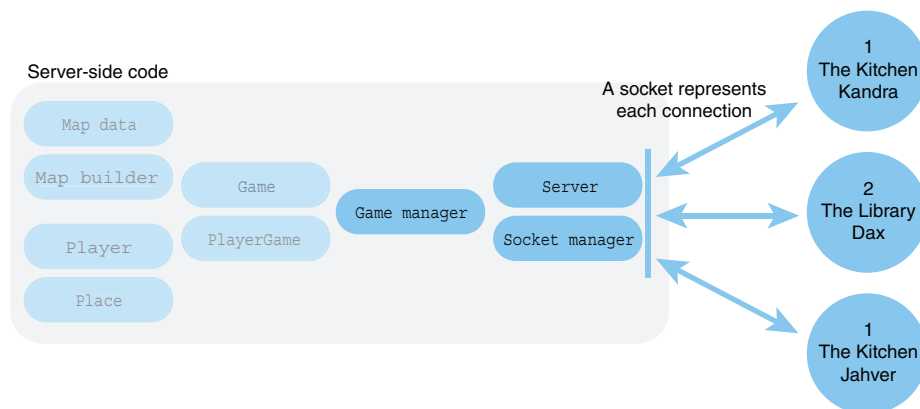
*CryptFlash™* is an entirely fictitious technology that in no way exists. We take no responsibility for any consternation or confusion caused by the pretend use of what would be an exciting and groundbreaking innovation.

### 25.2.1 Managing sockets and rooms

You want to target messages from the server to players in the same room of the same game. To identify the game-room combination, you use a key of this form:

"game\_1\_The Kitchen"

Figure 25.3 shows Kandra and Jahver both in The Kitchen while playing game 1.



**Figure 25.3** The socket manager lets sockets join and leave rooms and send messages to players in a room.

If Kandra picks up an item, you need to send an update (a CryptFlash) to Jahver (and anyone else in The Kitchen for that game) but not to Dax. You want to be able to use code like this:

```
kandraSocket.to("game_1_The Kitchen").emit("state", gameData);
```

Using `to` as a method of a socket object (rather than `io.to`) sends the message to others in the room but doesn't send it across the socket itself. For that message to reach Jahver, the socket for his client-server connection needs to have joined the "game\_1\_The Kitchen" room:

```
jahverSocket.join("game_1_The Kitchen");
```

The following listing shows the constructor function for a socket manager, to simplify all the joining and leaving of rooms and the sending of targeted messages.

#### Listing 25.7 The socket manager (socketManager.js)

```
function SocketManager (socket, playerGame) {
  this._socket = socket;
  this._playerGame = playerGame;
  this._keyStub = "game_" + playerGame.gameID + "_";
}

SocketManager.prototype = {
  _getChannel: function () {
    return this._keyStub + this._playerGame.getPlaceTitle();
  },

  _send: function (data, onlyToMe) {
    var socket = this._socket;
    var channel = this._getChannel();

    if (onlyToMe) {
      socket.emit('state', data);
    } else {
      delete data.messages;
      socket.to(channel).emit('state', data);
    }
  },

  sendAll: function () {
    var data = this._playerGame.getData();

    this._send(data, true);
    this._send(data);
  },

  sendOthers: function () {
    this._send(this._playerGame.getData(true));
  },
}
```

Pass the constructor a socket and a player game

Get the name of the player's current room

Use the socket to emit a message

Send game data to all players in the player's room

Send game data to other players in the player's room

```

join: function () {
    this._socket.join(this._getChannel());
},
leave: function (playerGame) {
    this._socket.leave(this._getChannel());
}
};

module.exports = SocketManager;

```

Join the current room

Leave the current room

Each player will get their own socket manager, so you pass the constructor their socket and their player game.

### INCLUDING AND EXCLUDING MESSAGES

When a player takes an action in a game, the game might generate a message for them. It will add it to the messages for their player game. But when the latest game data is sent out to all of the players in a room, the other players shouldn't receive the messages for the player who took the action; Jahver doesn't need to know that Kandra couldn't go north. The `_send` method in listing 25.7 includes code to remove the messages when sending data to the other players:

```

_send: function (data, onlyToMe) {
    var socket = this._socket;
    var channel = this._getChannel();

    if (onlyToMe) {
        socket.emit('state', data);
    } else {
        delete data.messages;
        socket.to(channel).emit('state', data);
    }
}

```

Set `onlyToMe` to true to send data only to the socket manager's player

Send the game data to the socket manager's player

Remove the messages because they're not for other players

Send the game data, without the messages, to the other players in the room

The `sendAll` and `sendOthers` methods then make use of the `onlyToMe` argument when calling `_send`:

```

sendAll: function () {
    var data = this._playerGame.getData();

    this._send(data, true);
    this._send(data);
},

sendOthers: function () {
    this._send(this._playerGame.getData(true));
}

```

Send data, including messages, to the current player

Send data, without messages, to the other players in the room

Send data, without the current player or messages, to the other players in the room

“CryptFlash! CryptFlash! Programmer completes JavaScript book online extras!” With the socket manager in hand, you can now update the game server and browser data loader to use Socket.IO. Then *Get Programming with JavaScript* will ride into the sunset, its work done and quest over, ready for the next adventure.

### 25.2.2 Using the socket manager

Say Jahver is playing a game and is in The Kitchen when Kandra joins the same game. Table 25.1 shows the actions Kandra might take, along with how you use her socket manager to update her browser, Jahver’s browser, and the browser of anyone else in her room. To create the socket manager for Kandra, you would pass the constructor her socket and player game:

```
var socketManager = new SocketManager(socket, playerGame);
```

**Table 25.1** How Kandra’s socket manager is used to update her browser and Jahver’s browser when she takes different actions in the game

Kandra’s action	Code	Comments
Start the game	<code>socketManager.join();</code> <code>socketManager.sendAll();</code>	Kandra needs to join the current room in order to get updates. Jahver needs to know that Kandra is now in the room. Kandra needs initial game data.
Pick up an item	<code>socketManager.sendAll();</code>	Kandra now has the item. It’s no longer in the room. Kandra and Jahver both need their browsers updated.
Use an item	<code>socketManager.sendAll();</code>	The item used may have been consumed and so may no longer appear in Kandra’s list. Kandra will have a new message. Kandra and Jahver both need their browsers updated.
Move to a new room	<code>socketManager.leave();</code> <code>socketManager.sendOthers();</code> <code>// move</code> <code>socketManager.join();</code> <code>socketManager.sendAll();</code>	Kandra leaves the current room. Jahver needs to know Kandra is no longer in the room. Kandra joins the new room. Everyone in the new room needs to know she is there.

The following listing puts the ideas from the table into action. It assigns the socket manager created for each player who connects to the server to the `cryptFlash` variable. The socket managers are being used to send a news flash, or *CryptFlash*<sup>TM</sup>, to players in a room.

#### Listing 25.8 The game server using sockets (gameServer.js)

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



```

var server = require('http').Server(app);
var io = require('socket.io')(server);

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

io.on('connection', socketsRouter);

function socketConnection (socket) {
  var cryptFlash;

  socket.on('start', start);
  socket.on('get', get);
  socket.on('go', go);
  socket.on('use', use);

  function start (data) {
    var playerGame = games.join(data.playerName, data.gameID);
    cryptFlash = new SocketManager(socket, playerGame);

    cryptFlash.join();
    cryptFlash.sendAll();
  }

  function get (data) {
    var playerGame = games.getPlayerGame(data.playerName, data.gameID);

    playerGame.clearMessages();
    playerGame.get();

    cryptFlash.sendAll();
  }

  function go (data) {
    var playerGame = games.getPlayerGame(data.playerName, data.gameID);

    playerGame.clearMessages();

    cryptFlash.leave();
    cryptFlash.sendOthers();

    playerGame.go(data.command.direction);

    cryptFlash.join();
    cryptFlash.sendAll();
  }

  function use (data) {
    var playerGame = games.getPlayerGame(data.playerName, data.gameID);

    playerGame.clearMessages();
    playerGame.use(data.command.item, data.command.direction);

    cryptFlash.sendAll();
  }
}

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

server.listen(1337);

```

Import the socket manager module

Declare a cryptFlash variable to hold the socket manager for the connection

Create a new socket manager and assign it to the cryptFlash variable

The server is performing the same tasks as it did in chapter 24, but rather than using Express routes to match and respond to requests, you're using Socket.IO events to receive and respond to socket messages.

### 25.2.3 Loading data on the browser

And so to the browser, for the last piece of the puzzle. You need to change only one module: the data loader. This listing shows the new code.

**Listing 25.9 Loading data in the browser (dataLoader-socketio.js)**

```
(function () {
  "use strict";

  var socket = io('http://localhost:1337');

  function postAction (command) {
    var data = {
      gameID: theCrypt.gameID,
      playerName: theCrypt.playerName,
      command: command
    };

    socket.emit(command.type, data);
  }

  function getStartData (callback) {
    var gameID = theCrypt.gameID;

    var data = {
      playerName: theCrypt.playerName
    };

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

    socket.on('state', callback);

    socket.emit('start', data);
  }

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

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

})();
```

Send a player action to the server

Start listening for updates (CryptFlash messages) from the server

Join the game

And finally, make sure the Socket.IO JavaScript code is loaded, by adding a script tag before the other modules at the bottom of the HTML file, as shown in this, the last listing.

**Listing 25.10 The web page (jahvers-crypt-socketio.html)**

```

/* rest of page */

<script src="/socket.io/socket.io.js"></script>

<!-- Modules -->
<script src="js/commands.js"></script>
<script src="js/dataLoader-socketio.js"></script>
<script src="js/gameController.js"></script>
<script src="js/mainView.js"></script>
<script src="js/messageView.js"></script>
<script src="js/placeView.js"></script>
<script src="js/playerView.js"></script>
<script src="js/templates.js"></script>

<script src="js/startGame.js"></script>

</body>
</html>

```

Start the server with node gameServer and visit the page at localhost:1337/jahvers-crypt-socketio.html.

Good luck and best wishes. Your adventure has only just begun.

## 25.3 Summary

- Use Socket.IO to send real-time messages between clients and servers. It utilizes the WebSocket protocol where possible.
- Install Socket.IO on the server by using npm:

```
npm install socket.io --save
```

- Import Socket.IO into your server-side project and pass it a Node HTTP server:

```
var server = require('http').Server(app);
var io = require('socket.io')(server);
```

- Run code whenever a client uses Socket.IO to connect to the HTTP server, by assigning a handler function to the connection event:

```
io.on('connection', socketsRouter);
```

- Use the socket object to send and receive messages; Socket.IO passes it to the handler function when a client connection triggers the event.

```
function socketsRouter (socket) {
    // Send and receive messages
    // using socket
}
```

- Assign handlers to events that the client might send:

```
socket.on('bid', makeBid);
socket.on('state', getState);
```

- Send messages to the client by emitting your own events:

```
socket.emit('update', { msg: 'Hello' });
```

- Send messages to all connected sockets by calling `io.emit`:

```
io.emit('update', { msg: 'Hello Everyone' });
```

- Use *rooms* to send messages to groups of sockets:

```
socket1.join('Fan Club');
socket2.join('Fan Club');
io.to('Fan Club')
  .emit('update', { msg: 'Hello Members' });
socket1.leave('Fan Club');
```

Join a room

Specify a room for your message

Send the message

Leave a room

- Send messages to groups of sockets, but not to the sender, by using the `socket` object to send the message:

```
socket.to('Fan Club').emit('update', { msg: 'Hello Members' });
```

- In the HTML for the client, include a script tag to load Socket.IO:

```
<script src="/socket.io/socket.io.js"></script>
<script src="auction.js"></script>
```

Load the Socket.IO code

Load your code that uses Socket.IO

- Use the `io` variable, made available when you loaded the Socket.IO code, to create a socket object:

```
var socket = io('http://localhost:1337');
```

- Use the client-side socket to send and receive messages:

```
socket.on('update', updateDisplay);
socket.emit('state');
```

Listen for the update event from the server

Emit a state event to the server

# 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