YOUR FIRST METEOR APPLICATION

Your First Meteor Application

A Complete Beginner's Guide to Meteor.js

David Turnbull

©2014 - 2015 David Turnbull

Tweet This Book!

Please help David Turnbull by spreading the word about this book on Twitter!

The suggested hashtag for this book is #meteorjs.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#meteorjs

Contents

Introduction **1**
- Screencasts 3
- Prerequisites 4
- What You'll Need 5
- Summary 6

Getting Started **7**
- Command Line 8
- Installing Meteor 10
- Summary 12

Projects **13**
- Create a Project 15
- Local Servers 19
- Default Application 23
- Summary 25

Databases, Part 1 . . . . **26**
- MongoDB vs. SQL 27
- Create a Collection 28
- Inserting Data 30
- Finding Data 33
- Summary 36

Templates **37**
- Create a Template 38
- Client vs. Server 40
- Create a Helper 45
- Each Blocks 49
- Summary 52

Events **53**
- Create an Event 54
- Event Selectors 56
- Summary 58

Sessions **59**
- Create a Session 60

The Player’s ID 62
Selected Effect, Part 1 64
Selected Effect, Part 2 66
Summary 69

Databases, Part 2 . . . . . **70**
Give 5 Points 71
Advanced Operators, Part 1 73
Advanced Operators, Part 2 77
Sorting Documents 79
Individual Documents 82
Summary 84

Forms **85**
Create a Form 86
The “submit” Event 88
The Event Object, Part 1 90
The Event Object, Part 2 92
Removing Players 95
Summary 96

Accounts **97**
Login Provider 98
Meteor.users 100
Login Interface 101
Logged-in Status 104
One Leaderboard Per User 105
Project Reset 108
Summary 109

Publish & Subscribe **110**
Data Security 111
autopublish 113
isServer 114
Publications, Part 1 115
Subscriptions 116
Publications, Part 2 117
Summary 119

Methods **120**
Create a Method 121
Inserting Data (Again) 123
Passing Arguments 125
Removing Players (Again) 127
Modifying Scores 130
Summary 133

Structure **134**

Your Project, Your Choice . . . 135
Thin Files, Fat Files . . . 136
Folder Conventions, Part 1 . . . 137
Folder Conventions, Part 2 . . . 138
Boilerplate Structures . . . 139
Summary . . . 141

Deployment . . . **142**

Conclusion . . . **143**

Introduction

There's a lot to love about the Meteor JavaScript framework:

- With nothing but JavaScript, you're able to build modern, real-time web applications for both desktop and mobile platforms.
- Beginning developers can quickly build something impressive and useful, while more advanced developers can appreciate Meteor's flexibility.
- There's an active community that organizes meetups, publishes free training material (like this book), and offers help wherever possible.

But when I first started working with Meteor, there was a gap. Because while there was plenty of training material available, the majority of it was aimed at experienced web developers.

There was nothing written for absolute beginners.

As a means of filling this gap (and furthering my own understanding of the framework), I launched meteortips.com and published a handful of tutorials that gently introduced readers to Meteor's fundamentals concepts. People responded well, but the only way to write for beginners was to assume as little as possible, meaning I soon found myself repeating the fundamentals again and again, which limited the scope of what I could write about.

To solve this problem, I started working on *Your First Meteor Application* — the book that you're reading right at this very moment.

The plan was simple enough:

Write the book that I wish had existed when I started using Meteor. Then, when someone came to my site, I could tell them: "Read the book before working through my tutorials."

But what begun as a side-project while house-sitting in New Zealand soon ballooned into something much bigger.

After releasing the book, experienced developers started recommending it, readers started pouring in from a number of sources, and the developers of Meteor itself even started to recommend it on their official website:

> David Turnbull's *Your First Meteor Application* is another excellent place to start learning, particularly for beginning developers. It's written in a practical, easy-to-follow conversational style, yet covers all the essential parts of getting an application off the ground.

At this stage:

- Tens of thousands of people have downloaded the book.

- The book has been rewritten three times, from scratch.
- Hundreds of people have said very nice things about the book, including dozens of people who have rated the book 5-stars on Amazon.

The core ambition of the book has remained consistent though, because ultimately, the coming pages are like a bridge. They don't contain every last detail there is to know about Meteor, but they do contain everything you need to know to make your first application with the framework, while also turning you into a confident, independent problem solver. So by the time you reach the final page, you won't be wondering: "Can I make this thing I want to make?" You'll be confident that you can. It'll just be a matter of tapping the keys on your keyboard to make it happen.

You've probably heard all of this before though. Everyone author thinks their book is going to be *the* book to make all the difference.

If you're feeling skeptical, here's what I'd suggest:

1. Brew yourself a cup of tea.
2. Settle down in a quiet space.
3. In a single sitting, work your way through the book until you reach the end of Chapter 6 (the "Events" chapter).

The chapters are short and it's by the end of Chapter 6 that you'll start to feel like a wizard who's been infused with the power of Meteor.

Let's begin.

Screencasts

If you prefer watching over reading, you might like to check out the video training series that I've put together. The series covers the same topics as the book, but many people find the screencasts even easier to follow.

To check out the video training series, visit:

meteortips.com/screencasts

The course includes:

- Over 2 hours of video training material.
- Detailed instruction for beginning developers.
- Free updates over the coming weeks and months.

All purchases are protected by a 30-day money-back guarantee, with no questions asked. (Just send me an email if you're not satisfied.)

Prerequisites

I've read a lot of technical books over the years and I've noticed that authors tend to use the words "for beginners" a little too liberally. With this in mind, I want to clearly explain what you *won't* need to make the most of the content inside this book:

1. **You won't need prior experience with Meteor**. I won't dwell on what Meteor is — it's a framework for building real-time web applications with JavaScript — but I will show you how to install it and how to start writing code as quickly and simply as possible.
2. **You won't need to have made a web application before**. There are some theoretical aspects of web development that we won't talk about, but if you're primarily (or entirely) a front-end developer, that's fine. You won't have any trouble grasping the details of Meteor development.
3. **You won't need to consult other sources along the way**. You're free to consult other books and tutorials, of course, but this book is designed to be an all-encompassing introduction to the basics.

You will, however, need a little background knowledge:

1. **You will need a basic understanding of JavaScript**. This means being familiar with variables, loops, conditionals, and functions. You don't need to be a JavaScript ninja. Just be comfortable with the fundamentals.
2. **You will need a basic understanding of databases**. This means being familiar with tables, rows, columns, and a primary keys. (It'll help if you've come into contact with something like a MySQL database.)

To acquire either of these skills, I'd suggest the following books:

- JavaScript and JQuery: Interactive Front-End Web Development
- MongoDB: The Definitive Guide

What You'll Need

You don't need much "stuff" to develop with Meteor. This might seem like a minor detail, but a common roadblock with other frameworks is a frustrating setup process before getting the chance to write some code.

With Meteor though, there's only a few things you'll need:

First, **you'll need a computer with a major operating system**. This could be Mac OS X, Windows, or Linux. All major operating systems are equally supported. (And if you're using an unsupported operating system, you can use a web-based service like nitrous.io.)

Second, **you'll need a text editor**. But there's no precise requirements. I'm quite fond of Sublime Text 3 — a cross-platform editor with plenty of plugins and productivity features — but we won't be using any power-user features, so just stick with whatever you prefer.

Third, **you'll need a relatively modern web browser**. Google Chrome is my browser of choice, and since we'll be using it throughout this book, I'd suggest installing a copy if you haven't already. (If you're familiar with the development features of Safari or Firefox though, feel free to use those.)

You'll also need Meteor itself, and we'll install that in the next chapter.

Summary

Each chapter of this book ends with a summary of what we've covered in that chapter. Over the last few pages, for instance, we've learned that:

- Meteor is a JavaScript framework that makes it easy for beginning web developers to build impressive, real-time web applications.
- You won't need a lot of background knowledge to get started with Meteor, but the more you know about JavaScript development and database theory, the better.
- There's not much "stuff" or setup to get started with Meteor, so there's a very small gap between learning about Meteor and actually writing code.

Along the way, I'll also share a number of exercises. You don't have to complete these exercises right away — in most cases, I'd suggest doing them *after* you've worked your way through the rest of the book — but I would suggest tackling them at some point. You'll be more than capable of solving the problems they present and each one will deepen your understanding of the framework.

Getting Started

Since we're not sitting in the same room, I can't be sure of what you do or don't know about building web applications. Maybe you've built entire application before. Maybe you just started learning JavaScript a week ago. Maybe you're not even sure why you're reading these words.

Either way, I'll assume two things in the coming chapter:

1. You've never operated a computer with the command line.
2. You haven't installed Meteor.

If these assumptions are wrong, feel free to skip this chapter, but if these assumptions are not wrong, then continue reading to get a handle on the basics that will allow you to start writing code as soon as possible.

Command Line

Once upon a time, computers didn't have graphical interfaces with buttons, windows, and menus. Instead, users controlled their computers by typing out commands and tapping the "Return" key.

These days, *command line interfaces* as they're known — or "CLI" for short — are still used by software developers.

Why?

Well, although graphical interfaces are more welcoming to beginners:

1. They're time-consuming to create.
2. They're ultimately slower to use.

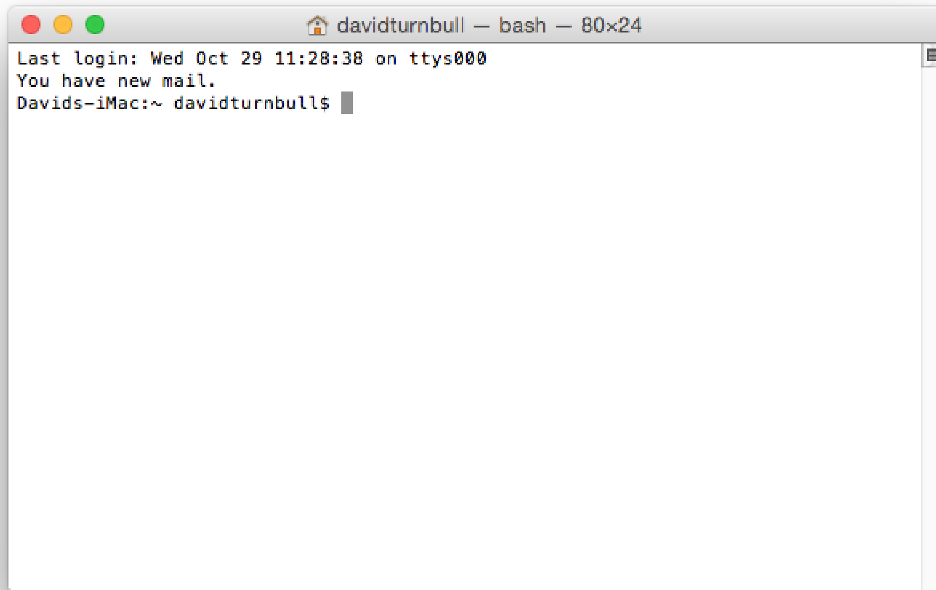
As such, there's no Meteor GUI. To install the Meteor framework on our computers, and then interact with that software, we have to work with the command line. This isn't as scary as it sounds though.

To get started, find the command line application on your computer.

All major operating systems have a command line application, but the name of the application will depend on the system:

- On Mac OS X, the command line application is *Terminal*.
- On Windows, the command line application is *Command Prompt*.
- On Linux, the command line application will depend on the distribution, but if you're using Linux, you probably know what you're doing.

After finding the command line application, open it up so we can continue.



The command line application on Mac OS X.

Installing Meteor

At the moment, Meteor is officially supported on:

- Mac: OS X 10.7 and above
- Windows:
  - Windows 7
  - Windows 8.1
  - Windows Server 2008
  - Windows Server 2012
- Linux: x86 and x86_64 systems

Installing Meteor on Windows is straight-forward enough. Just download and run the official Meteor installer. You only need to follow the prompts and you'll be good to go.

If you're using Mac OS X or Linux though, this is where we'll need to use the command line for the first time.

To begin, copy the following command to the clipboard:

```
curl https://install.meteor.com/ | sh
```

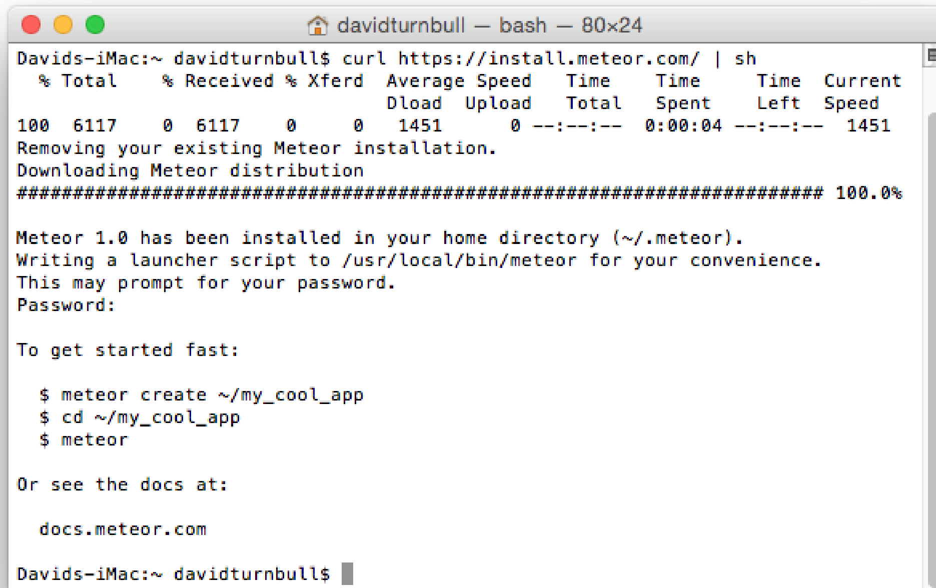
...and paste the command into the command line.

Then tap the "Return" key.

You don't need to understand exactly what this command is doing, but as a quick run-down, this command will:

1. Connect to "install.meteor.com".
2. Download the latest version of Meteor.
3. Install that version of Meteor.

If you're asked to enter your computer's password, fill it out and tap the "Return" key again. This is just to confirm that you have the appropriate privileges for installing Meteor on your computer.



Meteor is now installed.

Summary

In this chapter, we've learned that:

- Before computers had graphical interfaces, they were controlled through the use of commands. These days, command line interfaces are still heavily used by developers.

To gain a deeper understanding of what we've covered:

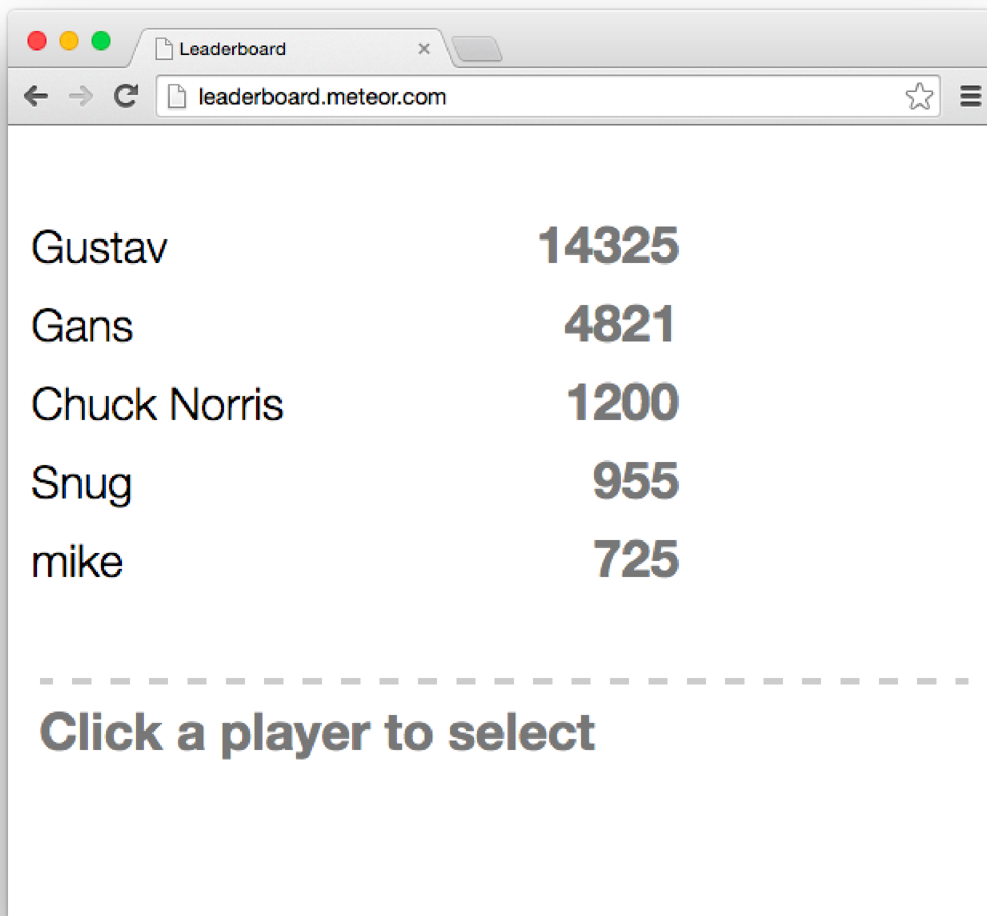
- Read "The Command Line Crash Course". This isn't necessary reading, but the more you know about working with the command line, the more productive you can be with it.

Projects

A big mistake that beginning web developers make when learning how to make web applications is trying to make progress without having a clear idea of what they're trying to build. But this is like driving to a new destination without a map. You might make a little progress in the right direction but you probably won't get where you need to go. You don't need to have everything figured out from the beginning, but you do at least need a direction.

With this in mind, we're going to build Leaderboard — an example application that was designed by the Meteor Development Group to show off what Meteor could do with very few lines of code.

Here's what it looks like:



It's not the prettiest thing in the world.

Leaderboard has since been replaced by more advanced examples on the official website, but it'll be our example project for two main reasons:

First, **the application already exists**. It's something we can play with, and that means we can get a good idea of what we're trying to build before we write a single line of code.

Second, **the application is simple**. This means we don't have to worry about the conceptual aspect of building the software (which is usually the most difficult part). Instead, we can focus on learning Meteor itself.

To get hands-on time with Leaderboard, visit leaderboard2.meteor.com and, while clicking around, take note of its core features:

- There's a list of players.
- Each player has a score.
- Players are ranked by their score.
- You can select players by clicking on them.
- You can increment a selected player's score.

We'll create additional features in later chapters, but even this relatively short list will allow us to cover a lot of Meteor's core functionality.

Create a Project

To create our first Meteor application, we'll need to create our first *project*, and a project is the self-contained set of files that form the foundation of an application. You can use the words "project" and "application" interchangeably, but "project" is better-suited when talking about the application as it's being developed.

Every project is different, but will generally contain:

- HTML files, to create the interface.
- CSS files, to assign styles to the interface.
- JavaScript files, to define application logic.
- Folders, to keep everything organized.

A project can contain other types of files, like images and CoffeeScript files, but we'll keep things as simple as possible throughout this book and only work with what we need.

Before we create a project for the Leaderboard application though, let's create a folder to store our Meteor projects. We don't *have* to do this, but it's a good idea for the sake of keeping things organized.

We could, of course, select the "New Folder" option from the "File" menu, but where's the fun in that? Instead, enter the following into the command line:

```
mkdir Meteor
```

Then tap the "Return" key.

Creating a folder with the `mkdir` command.

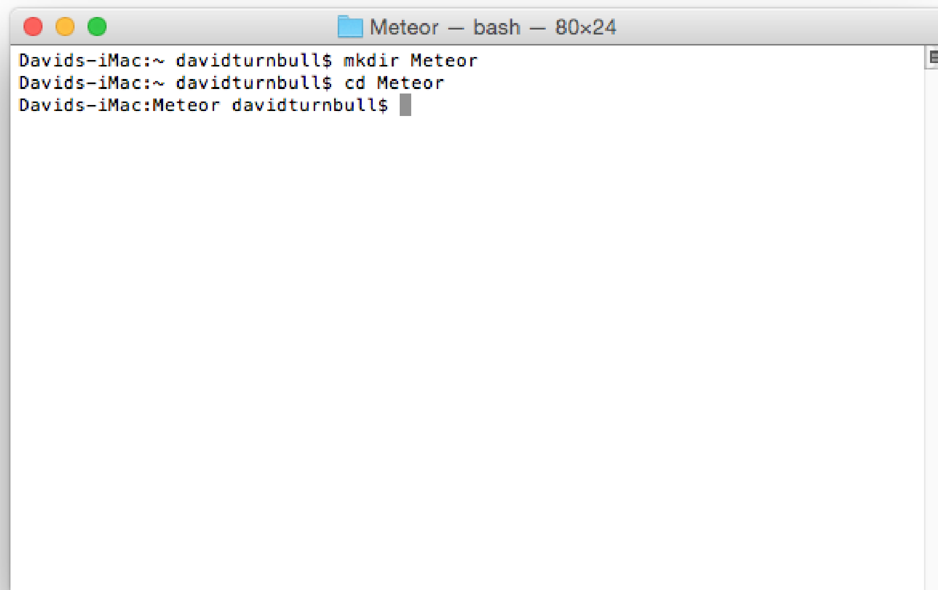
This `mkdir` command stands for "make directory" and, as you can probably guess from the name, it allows us to make a directory.

In this particular instance, we're creating a directory named "Meteor", but you can call the folder whatever you want. The precise location where the folder will appear will depend on your operating system, but at least on Mac OS X, the folder will appear inside the "Home" directory by default. (And if you can't find the created folder, simply search your computer.)

Once the directory is ready, navigate into it with the following command:

```
cd Meteor
```

This `cd` command stands for "change directory" and it's the command line equivalent of double-clicking on a directory from within the graphical interface, so after tapping the "Return" key, we'll be inside the "Meteor" directory.



Navigating into the "Meteor" folder.

To then create a Meteor project inside this directory, run the following:

```
meteor create leaderboard
```

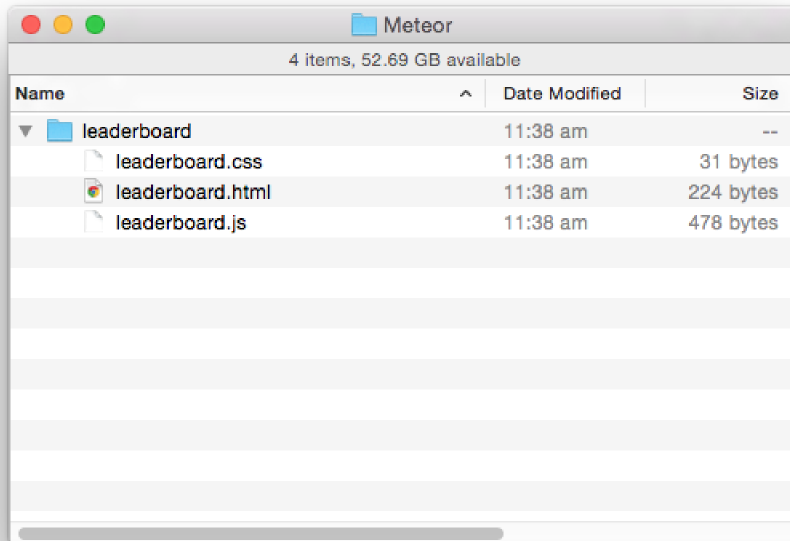
This command has three parts:

- The `meteor` part defines this as a Meteor command.
- The `create` part clarifies that we want to create a Meteor project.
- The `leaderboard` part is the name we're assigning to the project.

After running this command, a "leaderboard" directory will appear inside the "Meteor" folder, and by default this folder will contain three files:

- leaderboard.html
- leaderboard.css
- leaderboard.js

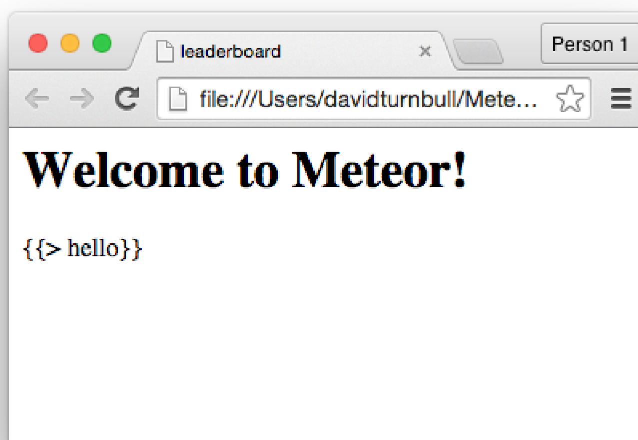
It will also contain a hidden folder — `.meteor` — but if your operating system hides this folder from view, that's fine. We won't be touching it.



Inside our project's folder

Local Servers

Web applications are not like static websites. We can't just open the `leaderboard.html` file and see the dynamic marvels of a Meteor application. In fact, if we open that file in Chrome, all we'll see is some static text:



There's nothing dynamic about this.

To get the web application working as planned, we need to launch what's known as a *local server*. This is a web server that runs on our local machine. It's included with Meteor itself and allows us to:

1. See the processed results of our JavaScript code.
2. Run a database on our local machine.

If you've used an application like MAMP for deployment with PHP and MySQL, this will be familiar, but if all of this sounds new and scary, fear not. In practice, it's quite simple.

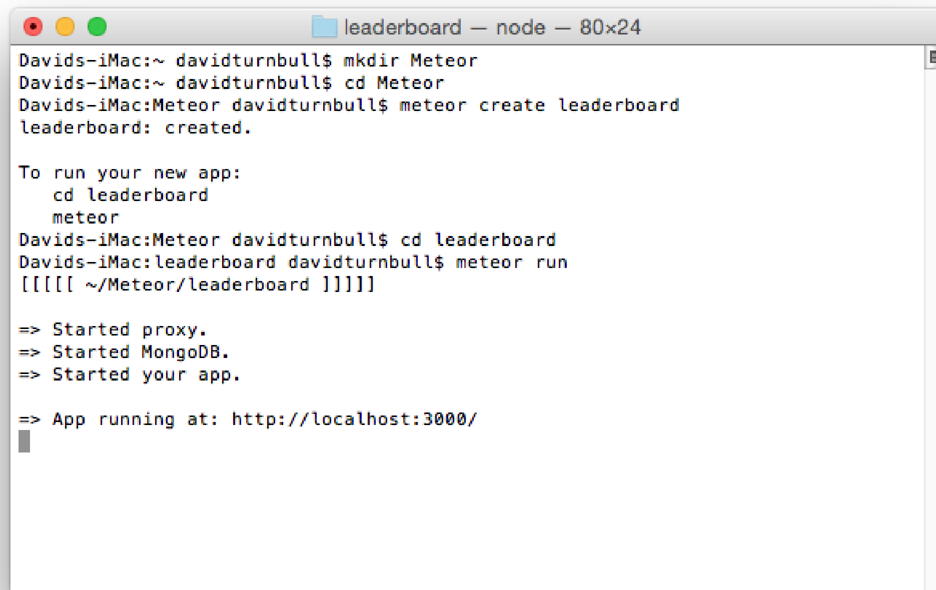
Through the command line, navigate into the "leaderboard" directory:

```
cd leaderboard
```

Then enter the following command:

```
meteor run
```

Here, the `meteor` part defines this as a Meteor command and the `run` part clarifies the precise action we want to take. In this context, we're wanting to `run` the local server.



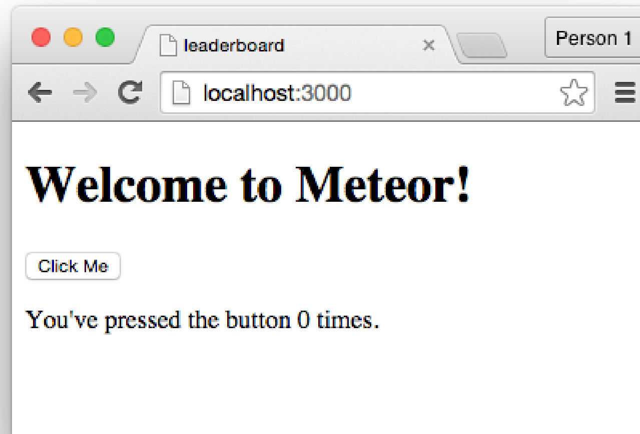
Starting the local server.

After tapping the "Return" key, the following should appear:

```
=> Started proxy.
=> Started MongoDB.
=> Started your app.
=> App running at: http://localhost:3000/
```

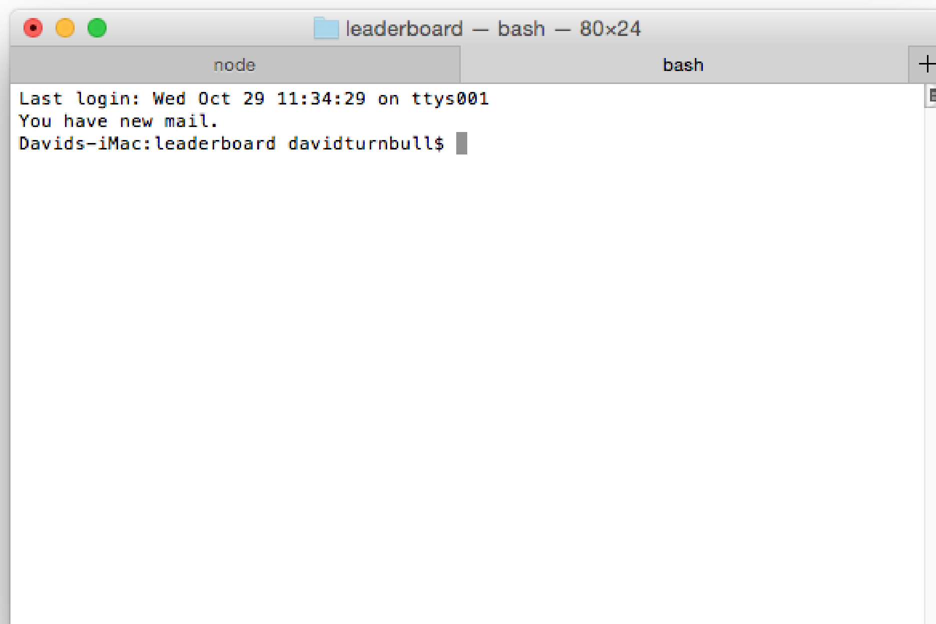
These lines confirm that the local server is starting and the URL on the last line — http://localhost:3000 — is what we can now use to view our Meteor project in a web browser.

Navigate to this URL from inside Chrome and notice that we're no longer seeing static text. Instead, we're seeing a functional web application. The application itself is the result of the code that's included with every Meteor project by default, and it's not the most interesting creation in the world, but we've nevertheless taken a step in the right direction.



This is the default Meteor application.

To continually see the results of our code, we'll need to keep the local server running. This simply means leaving the command line open from this point onward. You will, however, need to open a separate tab or window to write further commands:



A separate tab for running commands.

To stop the local server, either quit out of the command line, or with the command line in focus,

press `CTRL + C` on your keyboard. To then start the local server again, use the same command as before:

```
meteor run
```

Just make sure you're inside a project's folder before running the command.

Default Application

The default application is nothing special, but if we click the "Click Me" button, the number on the screen will increment. This provides a fairly vanilla demonstration of Meteor's real-time features. The code behind this application, however, isn't precisely important since we'll cover a far greater scope throughout the coming chapters.

For the moment, open the project's files and delete all of the default code. Don't even look at the code. Just get rid of it. We want to start from a completely blank slate.

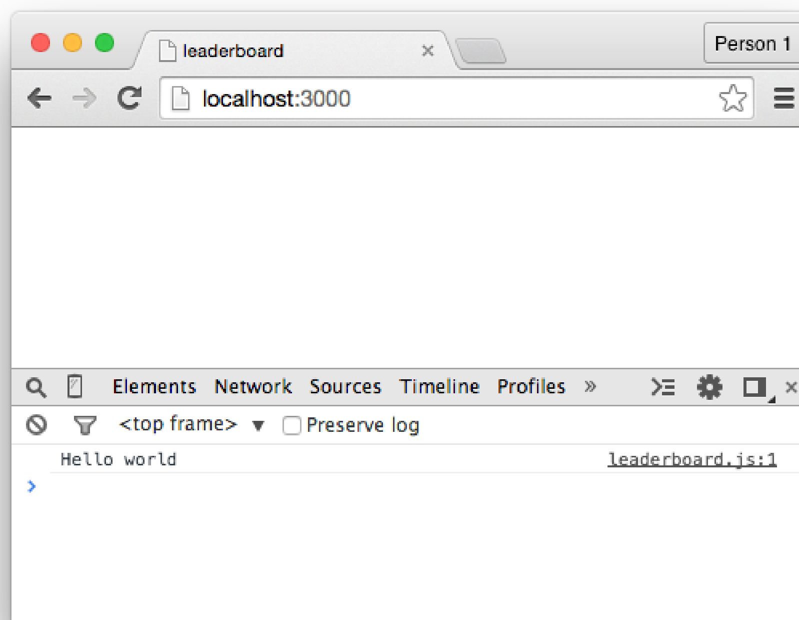
Once that's done, type the following into the JavaScript file:

```
console.log("Hello world");
```

Then save the file and open the JavaScript Console from inside Chrome:

1. Click on the `View` menu.
2. Hover over the `Developer` option.
3. Select the `JavaScript Console` option.

A pane will open at the bottom of the browser and display the "Hello world" text that we just passed through the `console.log` statement.



The "Hello World" text appears inside the JavaScript Console.

If this is familiar, great. If not, then know that `console.log` statements are used to see the output of code without creating an interface to display that output. This means that, before we invest time into creating an interface, we can:

1. Confirm that our code is working as expected.
2. Fix any bugs as soon as they appear.

We can also use the Console to manipulate our application's database, which is what we'll do in the next chapter.

Leave the Console open from this point onward, but feel free to delete the `console.log` statement from inside the JavaScript file.

Summary

In this chapter, we've learned that:

- When learning how to build a web application, it's important to have a clear idea of what you're trying to build.
- The command line can be used to quickly achieve familiar tasks, like creating folders and navigating between them.
- When developing a Meteor application, we refer to it as a "project", and we can create a project with the `meteor create` command.
- To view our web applications on our local machines, we can use the `meteor run` command to launch a local web server.
- When combined with `console.log` statements, the JavaScript Console is a very handy tool for Meteor development.

To gain a deeper understanding of Meteor:

- If you haven't already, play around with the original Leaderboard application. It doesn't have a lot of features, but that just means there's no reason to not have a strong grasp of its functionality.
- Close the command line application, then open it again and get back to where you were. You should be able to navigate into your project's folder with the `cd` command start the local server with `meteor run`.
- Create a second Meteor project and use this project to experiment whenever you learn something new. Messing around for the sake of it is a fantastic way to drill the details deeper into your brain.

To see the code in its current state, check out the GitHub commit.

Databases, Part 1

One of the difficult parts of writing a technical book is deciding when to introduce certain ideas. The topics that need to be taught are consistent between books, but the order in which you talk about them can drastically affect the reader's ability to grasp certain content.

More often than not, for instance, technical authors try to talk about creating an interface as soon as possible This is because it's fun to see the visual results of your code, and it's nice to feel like you're making quick progress.

But this approach does introduce a couple of problems:

1. It's harder to grasp anything relating to the front-end (the interface) when you're not familiar with the back-end (the database, etc).
2. If we talk about the front-end first, we'll have to back-track in the next chapter, so any sensation of quick progress will be short-lived.

As such, we'll start by talking about creating and managing a database within our project. This isn't a "sexy" topic, but if we spend a few minutes covering the basics, we'll have a strong foundation for the rest of the book.

MongoDB vs. SQL

If you've built something on the web before, you've probably come into contact with some kind of database. Maybe you've installed a copy of WordPress, or used phpMyAdmin, or even built some software with a language like PHP. In these cases, you would have come into contact with an SQL database.

By default, every Meteor project comes with its own database. There's no setup or configuration required. Whenever you create a project, a database is automatically created for that project, and whenever the local server is running, so is that database. This database, however, is *not* an SQL database. Instead, it's what's known as a MongoDB database.

If you've never come across MongoDB before, you might be a little worried, but fear not. Mongo databases are different from SQL databases, but as far as beginner's are concerned, the differences are small.

For the moment, you only need to know two things:

First, **there's no other type of database available for Meteor**. If you'd like to use an SQL database, for instance, it's just not possible yet. Other options will be available in the future, but the time-line isn't clear.

Second, **Mongo uses different words to describe familiar concepts**. We won't, for instance, use words like "tables" and "rows", but the concepts are basically the same. You can see the differences in this table:

SQL	MongoDB
database	database
table	collection
row	document
column	field
primary key	primary key

MongoDB vs. SQL

It can be tricky to think of familiar concepts with new words, but I'll offer plenty of reminders as we progress through the book.

Create a Collection

The central feature of the Leaderboard application is the list of players. Without the list of players appearing inside the interface, we can't build anything else of value. This is therefore a good place to start — from the "middle" of the application, working outward toward the finer details.

Here's two questions to consider:

- Where do we store the data associated with each player?
- How do we display this data from within the interface?

We'll answer the second question in the next chapter, but what about the first question: "Where do we store the data associated with each player?"

The facetious answer would be "in a database", but the more useful answer would be "in a *collection*", and as shown in the previous section, a collection is equivalent to an SQL table.

To illustrate the purpose of a collection, imagine we're creating our own version of WordPress with Meteor. If that were the case, we'd create a collection for the posts, a collection for the comments, and a collection for the pages. We'd create a collection for each *type* of data. Since we're creating this Leaderboard application though, we'll create a collection for the players.

To do this, open the JavaScript file and write the following statement:

```
new Mongo.Collection('players');
```

Here, we're creating a collection named "players" inside our project's Mongo database. You can name the collection whatever you want, but it must be unique. If the name is not unique, Meteor will return an error.

Despite this line of code though, we haven't defined a way for us to reference this collection, and therefore we have no way of manipulating it.

To fix this, place the collection inside a variable:

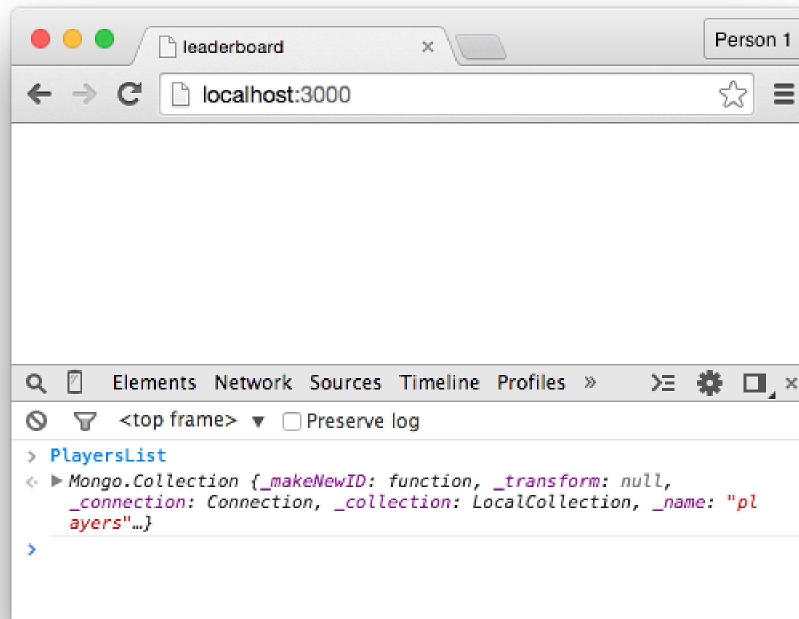
```
PlayersList = new Mongo.Collection('players');
```

But notice that we didn't use the `var` keyword, and that's because we want to create a `global` variable. This will allow us to reference and manipulate the collection throughout all of our project's files.

To confirm that the collection exists, save the file, switch to Chrome, and enter the name of the collection's variable into the Console:

```
PlayersList
```

You should see something like the following:



The collection exists.

This shows that the collection is working as expected.

If an error is returned, it’s probably because you mistyped the name of the variable in the Console, or made a syntactical mistake in the code.

Inserting Data

When we want to insert data into a collection, we have four options. We can insert data through the JavaScript Console, through the command line, through the JavaScript file, and through a form in the interface. We'll see how to use all of these options throughout this book, but the first option — through the JavaScript Console — is the simplest, so it's the best place to start.

Inside the Console, write the following:

```
PlayersList.insert();
```

This is the syntax we use to manipulate a collection.

We start with the variable assigned to the collection — the "PlayersList" variable — and then attach a function to it. In this case, we're using the `insert` function, but there's a range of other functions, like `find`, `update`, and `remove` (and we'll cover the details of these soon enough).

But if we tapped the "Return" key at this point, nothing would happen, and that's because we need to pass data between the brackets of the function for it to actually change the contents of the collection.

The data we pass through needs to be in the JSON format, and if you're not familiar with the JSON format, this is what it looks like:

```
{
    name: "David",
    score: 0
}
```

Here, there's a few things going on:

First, **the data is wrapped in a pair of curly braces**. This is how we distinguish our JSON data from the rest of the code.

Second, **we've defined a pair of keys**. These keys are known as `name` and `score`, and in Mongo terminology, these are the *fields* for our collection. So because every player in the collection will have a name and a score, we have the `name` and `score` fields to hold these values.

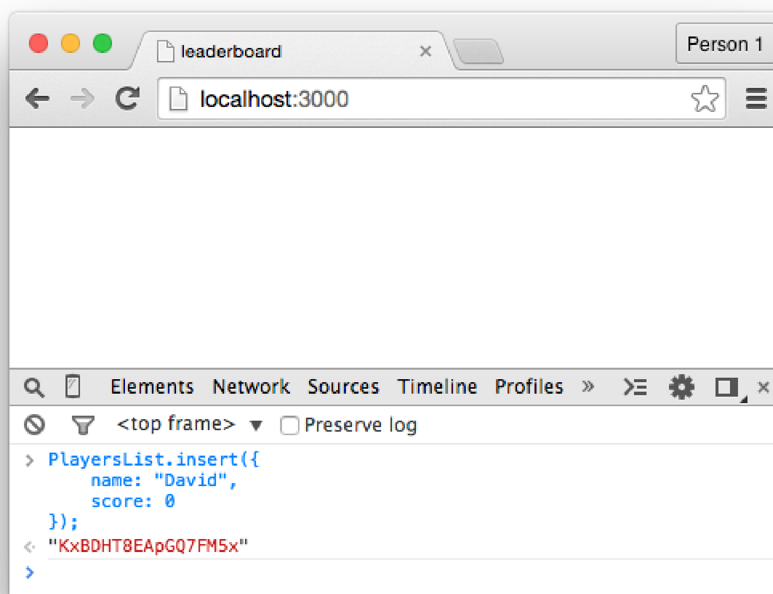
Third, **we've defined the values associated with our keys**. In this case, the value of the `name` field is "David" and the value of the `score` field is `0`.

Fourth, **the key-value pairs are separated with commas**. This is because the JSON format ignores white-space, so commas are required to provide structure.

We can pass this data through the brackets, like so:

```
PlayersList.insert({
    name: "David",
    score: 0
});
```

This is a complete `insert` function, and if we type this statement into the Console and tap the "Return" key, a *document* will be created inside the "PlayersList" collection. Documents are equivalent to SQL rows and, at this point, we want to create one document for every player we want in our collection. So if we want our leaderboard to contain six players, we'll need to use the `insert` function six times, thereby creating six documents.



Inserting data.

To achieve this, repeat this statement a few times, making sure to define unique values for the `name` field so we can distinguish the players:

```
PlayersList.insert({
    name: "Bob",
    score: 0
});
```

Since white-space is ignored though, the statement can be written on one line:

```
PlayersList.insert({ name: "Bob", score: 0 });
```

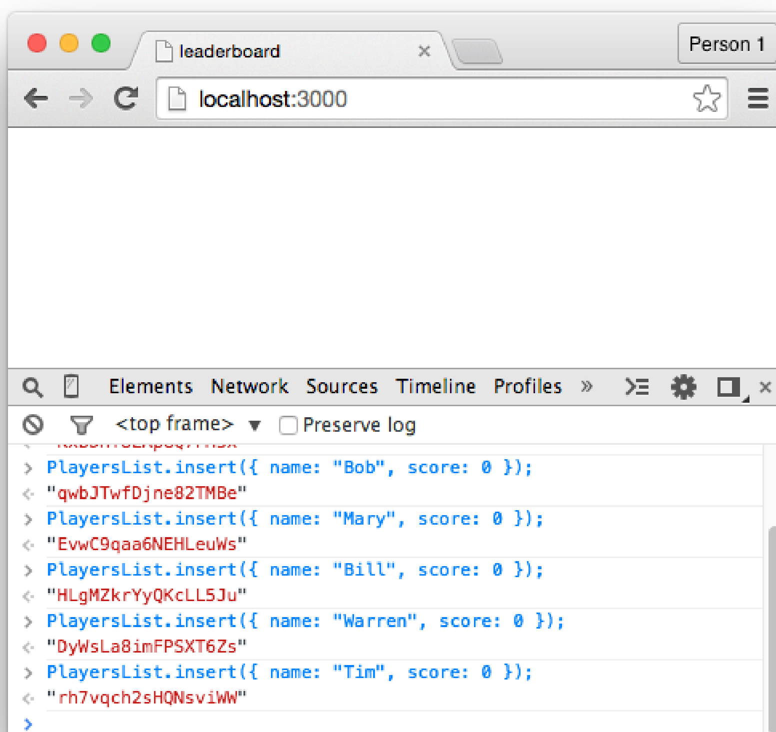
Also notice that, after creating each document, a random jumble of numbers and letters appears in the Console. This jumble is a unique ID that's automatically created by MongoDB and associated with each document. It's known as the *primary key*, and it'll be important later on. For the moment, just note that it exists so you're not surprised when we talk about it again.

Before we continue, insert a few more players into the collection. The example application has six players and that should be enough.

The players in my list are:

- David
- Bob
- Mary
- Bill
- Warren
- Tim

And they all have their score field set to `0`.



Inserting the remaining players.

Finding Data

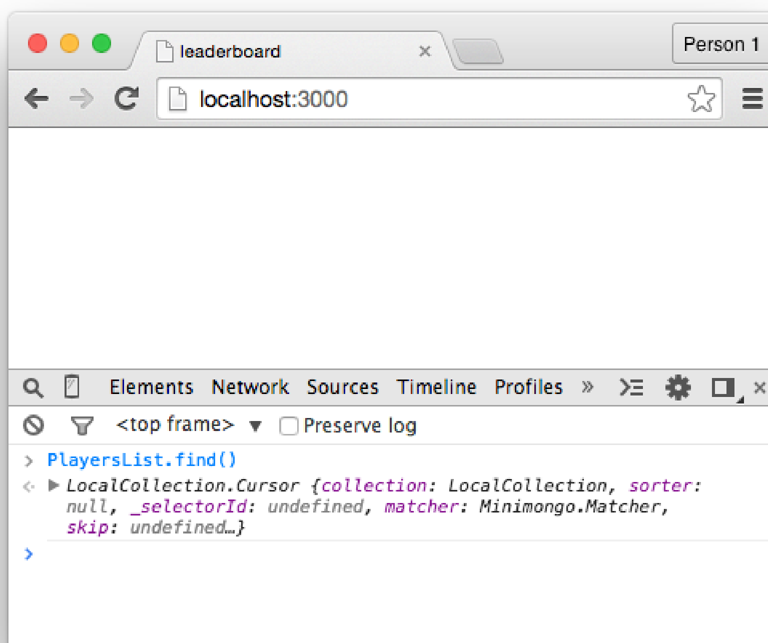
Now that there's some data in our collection, we're able to retrieve that data. We'll do this through the interface in the next chapter, but for the moment, let's simply do it through the Console.

Inside the Console, enter the following:

```
PlayersList.find();
```

Here, we're using this `find` function, which is used to retrieve data from the specified collection. Because we're not passing anything through the brackets, this statement will retrieve all of the data from the collection.

The following should appear in the Console:



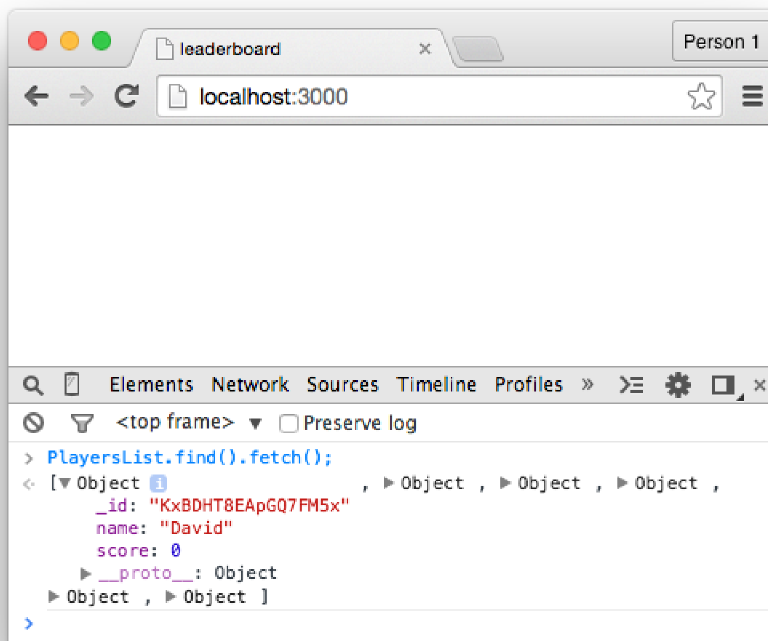
Using the `find` function.

But obviously, this isn't the most readable response. Our application can make sense of it, but we can't.

To retrieve data in a more readable format, attach a `fetch` function to end of it to convert the retrieved data into an array:

```
PlayersList.find().fetch();
```

You should see the following:



Using the `find` and `fetch` function.

We can also click on the downward-facing arrows to see the data associated with each document, including:

- The `_id` field, which stores the unique ID of that document (the "primary key", which we mentioned before).
- The `name` field, which stores the name of the player.
- The `score` field, which stores the score of the player.

But what if we wanted to retrieve a selection of data from the collection, rather than all of the data? To do this, we could pass JSON-formatted data between the brackets:

```
PlayersList.find({ name: "David" }).fetch();
```

Here, we're passing a field name and a value through the `find` function, and as a result, we're able to retrieve only the documents where the player's name field is equal to "David". In our case, this will only retrieve one document, but if our collection contained multiple players named "David", they would all be returned based on this query.

What's also useful is the ability to count the number of documents returned by a query by attaching the `count` function to the `find` function:

```
PlayersList.find().count();
```

Since this statement will count all of the documents in the collection, if there's six players (documents) in the collection, the number `6` will be returned.

Summary

In this chapter, we've learned that:

- When we create a Meteor project, a Mongo database is automatically created for us, and when the local server is running, so is that database.
- Mongo databases are different from SQL databases, but the differences are insignificant as far as beginner's are concerned.
- For each type of data we want to store in a Mongo database, we need to create a collection. Collections contain documents, and documents are made of fields and values.
- By using the `insert` function, we can insert data into a collection. This data is structured in the JSON format.
- By using the `find` function, we can retrieve data from a collection. This data can then be navigated through using the Console.

To gain a deeper understanding of Meteor:

- Notice that we haven't predefined a structure for the database. Instead, the structure of the database is defined on-the-fly, as we use the `insert` function.
- In a separate project, create a collection that stores a different type of data — such as blog posts, perhaps. What sort of fields would that collection have?

To see the code in its current state, check out the GitHub commit.

Templates

In this chapter, we're going to start building the user interface for the Leaderboard application. This involves creating our first *templates*.

To begin, place the following code inside the `leaderboard.html` file:

```
<head>
    <title>Leaderboard</title>
</head>
<body>
    <h1>Leaderboard</h1>
</body>
```

There's nothing special about this code — it's just standard HTML — but there does appear to be a few things missing:

- We haven't included the `html` tags.
- We haven't included any JavaScript files
- We haven't included any CSS files.

But we haven't included these things because we don't need to include them. Meteor takes care of these details for us. It adds the `html` tags to the beginning and end of the file, and it automatically includes any resources contained within the project's folder (like JavaScript and CSS files).

This isn't the most remarkable feature in the world, but one of Meteor's core philosophies is developer happiness, so there's plenty of time-saving features like this spread throughout the framework.

Create a Template

Templates are used to create a connection between our interface and our JavaScript code. When we place interface elements inside a template, we're able to reference and manipulate those elements with application logic.

To create a template, add the following code to the very bottom of the HTML file, beneath the closing `body` tag:

```
<template name="leaderboard">
    Hello World
</template>
```

Here, we're using this `template` tag, which uses a `name` attribute to distinguish between the different templates we create. In this case, the name of the template is "leaderboard", and we'll soon reference this name from inside the JavaScript file.

If you save the file in its current state though, the template doesn't appear inside the web browser. It's inside the HTML file, but nowhere else. This is because, by default, templates don't appear inside the interface. This might sound weird, but consider that, in some cases:

- You might want a template to appear at certain times.
- You might want a template to *dis*appear at certain times.
- You might want a template to appear in multiple locations.

To account for this possibility, we need to manually include templates inside the interface. This does require an extra step, but it'll become increasingly useful as we get deeper into development.

To make the "leaderboard" template appear inside the browser, place this tag between the `body` takes inside the HTML file:

```
{{> leaderboard}}
```

Obviously, this isn't HTML. Instead. the use of double-curly braces means this is the *Spacebars* syntax, and Spacebars is the syntax we use in our HTML when we want something dynamic to occur. It's the syntax that bridges the gap between the interface and the application logic.

We'll learn more about Spacebars throughout this book, but for now, know that:

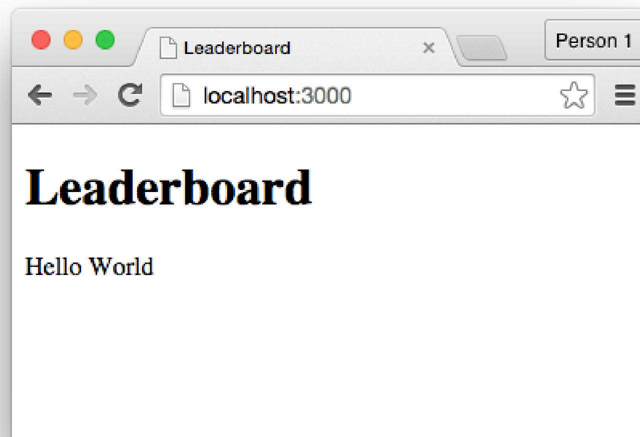
1. All Spacebars tags use double-curly braces to distinguish themselves.
2. We only use the greater-than symbol when we want to include a template.

Based on these changes, the HTML file should now resemble:

```
<head>
    <title>Leaderboard</title>
</head>
<body>
    <h1>Leaderboard</h1>
    {{> leaderboard}}
</body>

<template name="leaderboard">
    Hello World
</template>
```

Then, after saving the file, the “Hello World” from inside the “leaderboard” template should appear inside the browser:



The current interface.

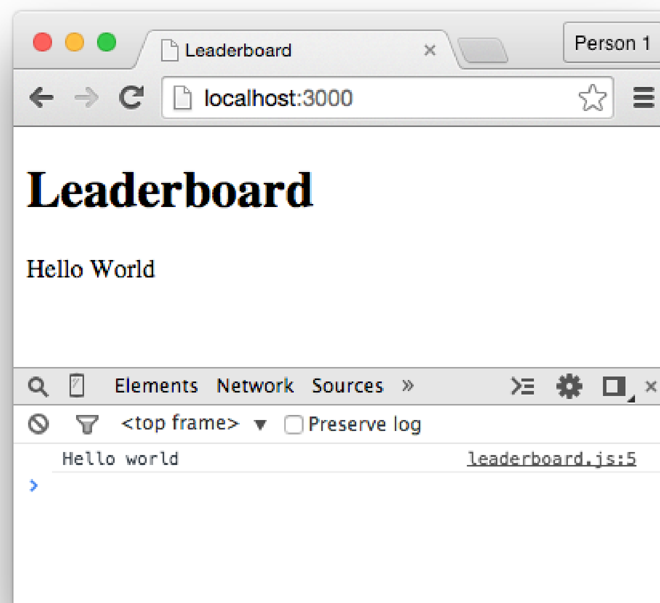
Client vs. Server

Before we continue, I want to demonstrate something. You don't have to fully grasp what we're about to cover, but do follow allow by writing out all of the code yourself.

Inside the JavaScript file, write the following `console.log` statement:

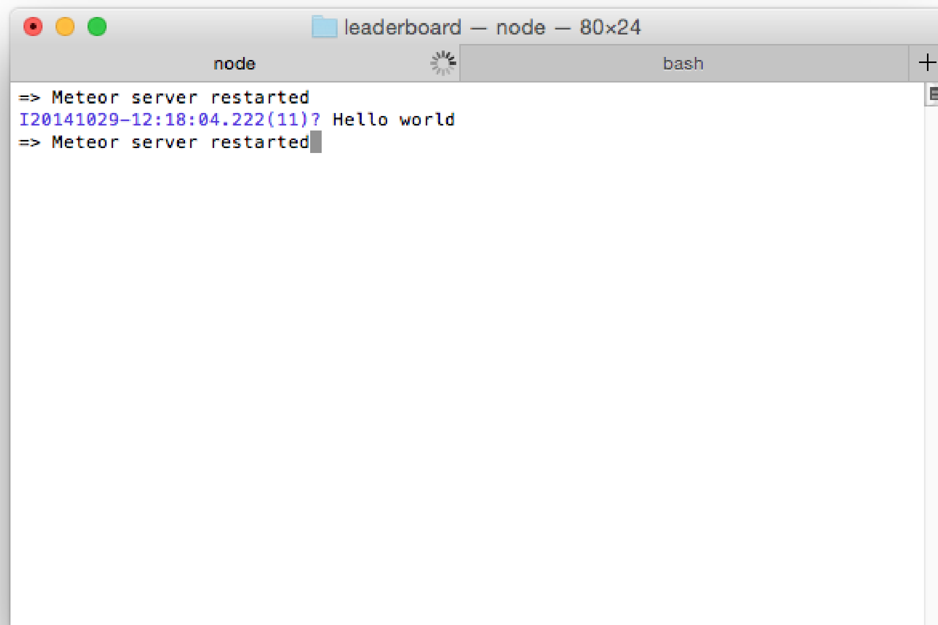
```
console.log("Hello world");
```

This is the statement statement we wrote in the "Projects" chapter, and after saving the file and switching back to Chrome, you should see the "Hello world" message appear in the Console:



"Hello World" appears in the Console.

What I didn't mention last time though is that, as a result of this statement, something else is also happening, and if we switch to the command line, we can see that the "Hello world" message also appears here:



"Hello World" appears on the command line.

This is significant because **we've written one line of code that's executed in two places**. The code is running on both the client (within the user's web browser) and on the server (where the application is hosted).

Why does this matter?

There's a few reasons, but here's one example:

Ever since we created the "PlayersList" collection, the following statement has been running on both the client and the server:

```
PlayersList = new Mongo.Collection('players');
```

But the code doesn't do the same thing in both places.

When the code is executed on the server, a collection is created inside the Mongo database. This is where our data is stored. When the code is executed from inside the user's web browser though — on the client-side — a local copy of that collection is created on that user's computer. As a result, when the user interacts with the database, they're actually interacting with their local copy. This is partly why Meteor applications are real-time by default. Data is manipulated on the user's local machine and then invisibly synced in the background with the server-side database.

But if all of this sounds a bit too conceptual, fear not. You don't have to understand the finer points of Meteor's "magic". You just need to grasp that one line of code can:

1. Run in two different environments (on the client and on the server).

2. Behave differently depending on the environment.

That said, **in some cases, we don't want our code to be running in two places at once**. If, for instance, we write code that only affects the interface of an application, it wouldn't make sense for that code to run on the server. We'd only want it to run on the client.

To accommodate for this, there's a pair of conditionals we can use to determine what code runs in which environment. You'll have a much better idea of when to use these conditionals as we progress through the book, but again, just follow along for the moment by writing out all of the code.

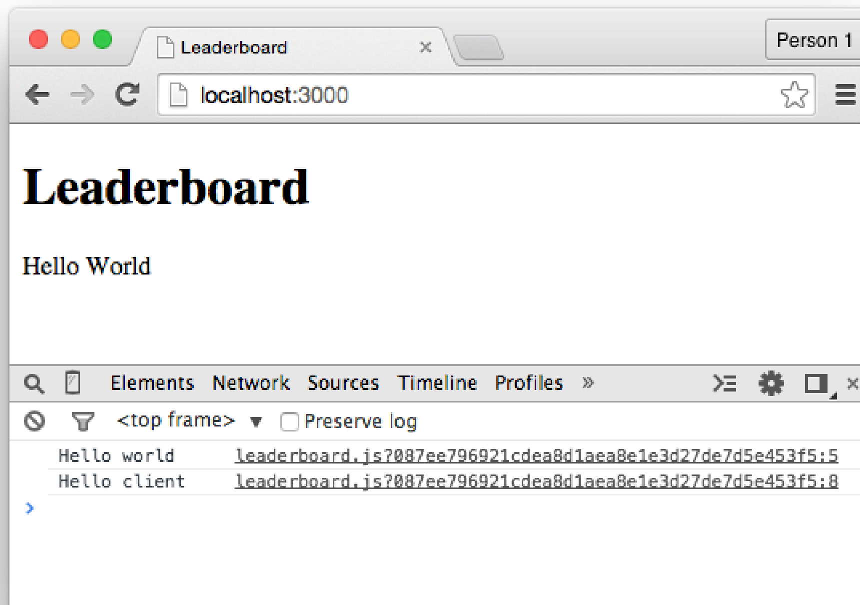
First, place a `Meteor.isClient` conditional beneath the `console.log` statement from before:

```
if(Meteor.isClient){
    // this code only runs on the client
}
```

This conditional allows us to specifically execute code on the client — from inside the user's web browser — and to demonstrate this, we can simply add a `console.log` statement inside the conditional:

```
if(Meteor.isClient){
    console.log("Hello client");
}
```

Save the file, switch to the browser, and notice that the "Hello client" message appears in the Console, but does *not* appear inside the command line. This is because the code is not being executed on the server.



"Hello client" only appears in the Console.

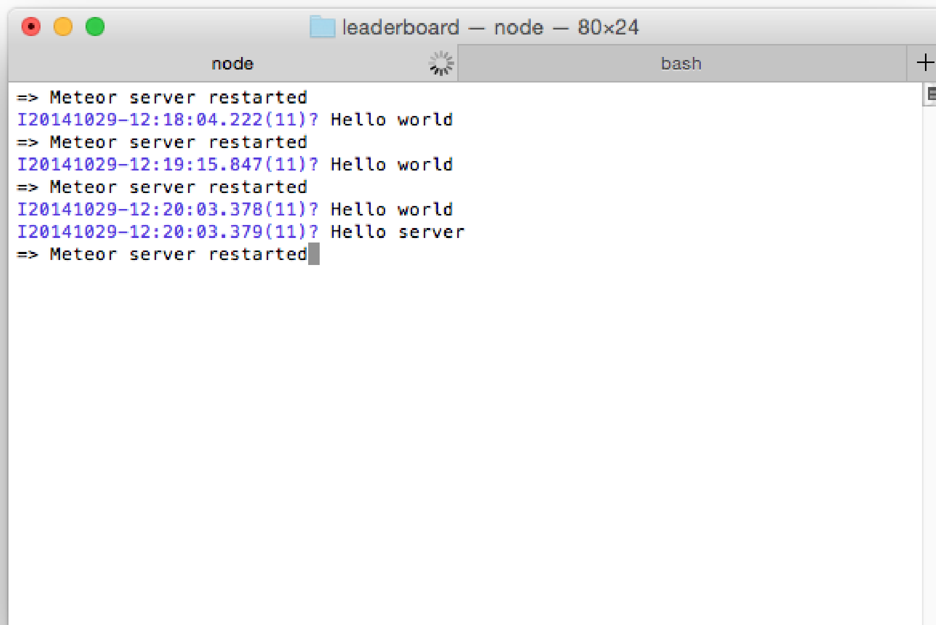
We can create the reverse effect with the `Meteor.isServer` conditional:

```
if(Meteor.isServer){
    // this code only runs on the server
}
```

Again, we'll place a `console.log` statement inside the conditional:

```
if(Meteor.isServer){
    console.log("Hello server");
}
```

Then, after saving the file, notice that the "Hello server" message appears inside the command line, but does not appear in the Console. This is because the code is only being executed on the server (where the application is hosted).



"Hello server" only appears on the command line.

But if none of this is really sinking in, just remember two things:

1. A single line of code can run on both the client and the server.
2. Sometimes we don't want our code to run in both places.

The precise moments where we need to consider these points will become clear soon enough. For now, just delete the `console.log` statements, but leave the conditionals where they are. We'll be using them soon.

Create a Helper

At this point, our “leaderboard” template only shows the static “Hello World” text. To fix this, we’re going to create a *helper function*, and a helper function is a regular JavaScript function that’s attached to a template and allows us to execute code from within an interface.

To begin, we’ll take an old approach for creating helper functions. This approach is deprecated, meaning it’s no longer officially supported, and by the time you’re reading these words, it may not work at all. But this older format is easier to teach and understand, and allows us to ease into the non-deprecated approach that we’ll talk about in a moment.

Inside the JavaScript file, write the following inside the `isClient` conditional:

```
Template.leaderboard.player
```

This is the deprecated syntax for creating a helper function, and it can be broken down into three parts:

First, **the `Template` keyword searches through the templates in our Meteor project**. We only have one template at the moment, but a complete project would have many more.

Second, **the `leaderboard` keyword is a reference to the name of the template we created earlier**. Every helper function must be attached to a template. In this case, the function is attached to the “leaderboard” template.

Third, **the `player` keyword is the name we’re giving to this function**. We will soon reference this name from inside the HTML file.

To attach code to this helper, associate it with a function:

```
Template.leaderboard.player = function(){
    // code goes here
}
```

The word “helper” might make this sound fancy, but we haven’t done anything special here. We’ve created a function, named it “player”, and connected it to the “leaderboard” template.

To add some functionality to the function, create a `return` statement that returns some static text:

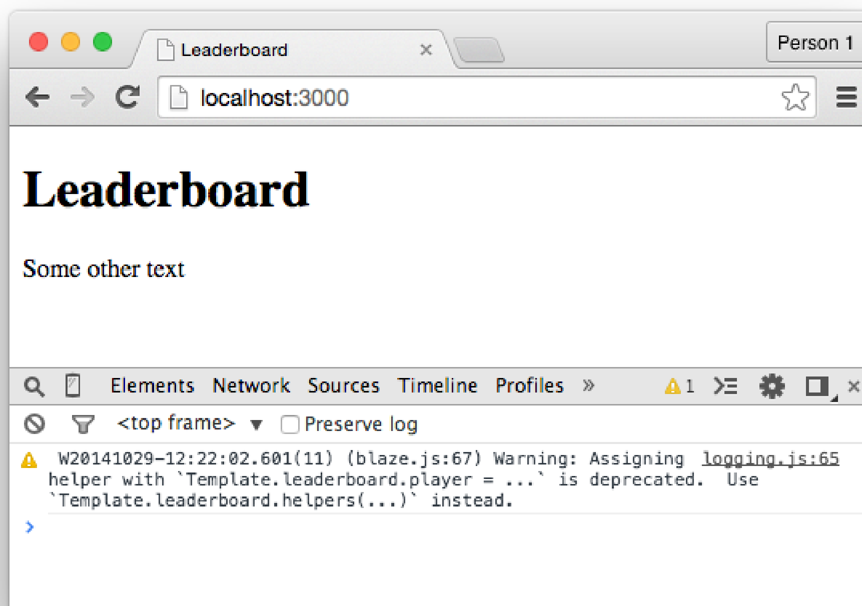
```
Template.leaderboard.player = function(){
    return "Some other text"
}
```

Then remove the “Hello World” text from the “leaderboard” template and replace it with the following tag:

```
{{player}}
```

Here, we're using another Spacebars tag, as evidenced by the use of double-curly braces. But notice that we're *not* using the greater-than symbol, and that's because we're not including a template. Instead, we're referencing the name of the `player` function.

After saving the file, the text from the `return` statement should appear inside the browser:



Using the deprecated approach to helper functions.

If the text doesn't appear, something is either wrong with the code, or this approach to creating helpers has been removed from Meteor. If you're unsure of which it is, check your code for bugs. If your code is exactly what I've told you to write and it's still not working, fear not. Now that you know the old way to create helper functions, we're ready to discuss the new way.

Delete all of the helper function we just created and replace it with:

```
Template.leaderboard.helpers
```

Here, we have this `Template` keyword, which searches through the templates inside our project, and this `leaderboard` keyword, which is a reference to the "leaderboard" template.

But what about this `helpers` keyword?

Are we creating a function named "helpers"?

Nope.

This `helpers` keyword is a special keyword that allows us to define multiple helper functions inside a single block of code.

Therefore, instead of creating a single helper function:

```
Template.leaderboard.player = function(){
    // code goes here
}
```

We create a block for all of a template's helper functions:

```
Template.leaderboard.helpers({
    // helper functions go here
});
```

These helpers are defined in the JSON format, with the name of the helper as the key and an associated function as the value:

```
Template.leaderboard.helpers({
    'player': function(){
        return "Some other text"
    }
});
```

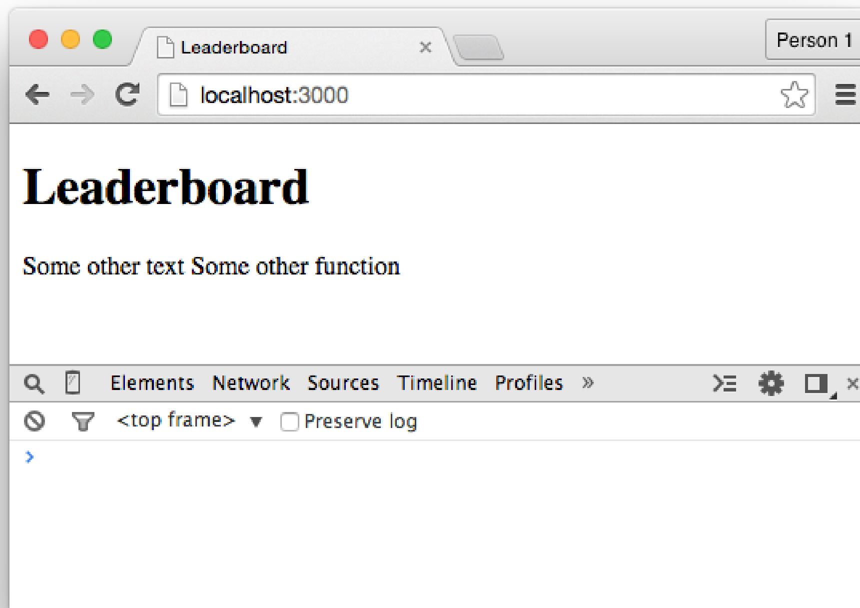
With the use of commas, we can create multiple helper functions:

```
Template.leaderboard.helpers({
    'player': function(){
        return "Some other text"
    },
    'otherHelperFunction': function(){
        return "Some other function"
    }
});
```

Both of these helper functions can then be used from inside the "leaderboard" template:

```
{{player}}
{{otherHelperFunction}}
```

This code might look a little busier than the deprecated approach, but that's only because we're working with a small amount of helpers. With a larger amount of helpers, organizing them like this is the far cleaner approach.



A pair of helper functions in a single template.

Each Blocks

We've made a helper function, but it just returns some static text, which isn't so interesting. What we really want is a helper function that retrieves the documents from the "PlayersList" collection. Then we'll be able to display that data from within the interface.

To achieve this, replace the `return` statement in the helper function with the following:

```
return PlayersList.find()
```

Here, we're using the `find` function function from the "Databases, Part 1" chapter. This function will retrieve all of the data from the "PlayersList" collection, and because we've placed it inside the helper function, this data is now accessible from inside the "leaderboard" template.

To see this in action, remove the following tag from the HTML file:

```
{{player}}
```

Before, our helper function returned a single piece of data — that string from before — and for that purpose, this tag was fine, but now the helper is returning an array of all the documents from inside the collection, and that means we need to loop through the data that's returned.

To achieve this, we can use the Spacebars syntax to create an `each` block:

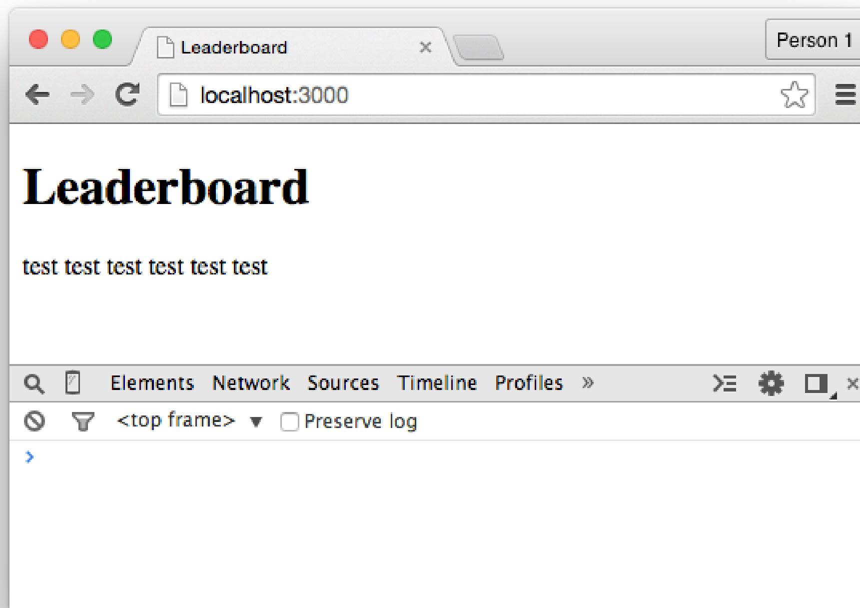
```
{{#each player}}
    test
{{/each}}
```

Here, there's a few things going on:

First, all of the documents from the "PlayersList" collection are retrieved based on the reference to the `player` function.

Second, we loop through the returned data with the `each` syntax.

Third, we output the word "test" for each document (player) that's retrieved. Because there are six players inside the collection, the word "test" will appear six times within the interface.



The word "test" appears for each player in the collection.

Conceptually, it's like we have an array:

```
var playersList = ['David', 'Bob', 'Mary', 'Bill', 'Warren', 'Tim'];
```

...and it's like we're using a `forEach` to loop through the values of this array:

```
var playersList = ['David', 'Bob', 'Mary', 'Bill', 'Warren', 'Tim'];
playersList.forEach(function(){
    console.log('test');
});
```

Within the `each` block, we can also retrieve the value of the fields from inside our documents. So because we're pulling data from the "PlayersList" collection, we can display the values of the `name` and `score` fields.

To display the player's names, for instance, we can write:

```
{{#each player}}
    {{name}}
{{/each}}
```

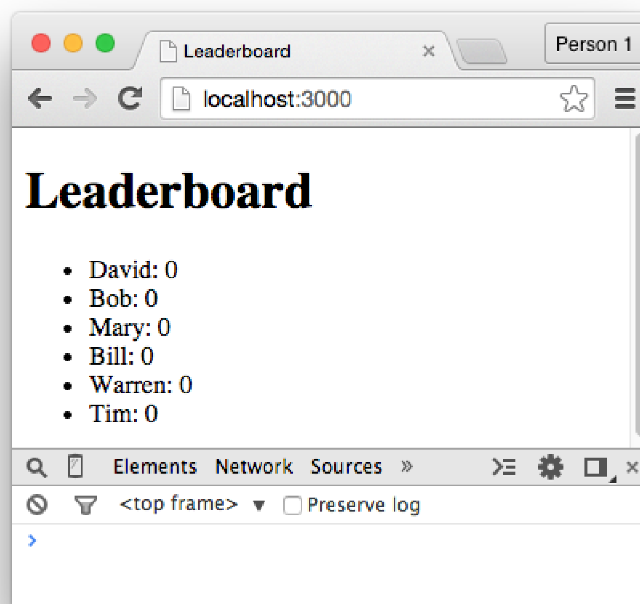
Then to display the player's scores, we can write:

```
{{#each player}}
    {{name}}: {{score}}
{{/each}}
```

But while we won't waste time making this application pretty, we will add some slight structure to the interface:

```
<ul>
    {{#each player}}
        <li>{{name}}: {{score}}</li>
    {{/each}}
</ul>
```

After saving the file, the player's names and scores will appear inside an unordered list. By default, the players will sorted by the time they were inserted into the collection — from the players added first to the players added more recently — but this is something we'll change in a later chapter.



An improved interface.

Summary

In this chapter, we've learned that:

- Meteor handles some of the boring details for us, like using the `html` tags and including JavaScript and CSS files.
- By creating templates, we're able to form a bridge between our application logic and our interface.
- Our project's code can run on both the client and the server, but we don't always want this to happen. We can use the `isClient` and `isServer` conditionals to control where the code is run.
- After creating a template, we need to manually include it within the interface. This gives us control over where and *when* it appears.
- By creating helper functions, we can execute code from within a template, thereby creating a dynamic interface.
- If a helper function returns an array of data, we can loop through that data from within a template using the `each` syntax.

To gain a deeper understanding of Meteor:

- Realize that templates can be placed anywhere within a project's folder. We could, for instance, place our "leaderboard" template in another HTML file and the reference to `{{> leaderboard}}` would continue to work.
- Break the application on purpose by moving the `each` block outside of the "leaderboard" template. Become familiar with errors that you'll inevitably encounter as a Meteor developer. Then you'll know how to deal with them when they're unexpected.
- Create a helper function that uses the `find` and `count` function to return the number of players in the "PlayersList" collection. Then display this data from inside the interface.

To see the code in its current state, check out the GitHub commit.

Events

At this point, we have a list of players that's appearing inside the interface, but there's no way for users to interact with this list. The data is dynamically retrieved from the "PlayersList" collection, but the user will still probably assume that the application is completely static.

We'll spend the rest of the book fixing this problem, but over the next couple of chapters in particular, we'll create the effect of being able to select players inside the list.

Specifically, when a user clicks on one of the players, the background color of that player's `li` element will change to yellow.

Create an Event

In this section, we'll create our first *event*, and events allow us to trigger the execution of code when a user clicks on a button, taps a key on their keyboard, or completes a range of other actions.

To demonstrate this, write the following inside the `isClient` conditional:

```
Template.leaderboard.events({
    // events go here
});
```

Here, there's a few things going on:

First, the `Template` part is used to search through all of the templates in the project.

Second, the `leaderboard` part is the name of the template we're about to attach an event to.

Third, the `events` part is special keyword that's used to specify that, within the coming block of code, we want to specify one or more events. (This code is very similar to how we create helper functions.)

Between the curly braces of the `events` block, create an event using the JSON format:

```
Template.leaderboard.events({
    'click': function(){
        // code goes here
    }
});
```

Here, there's two things going on:

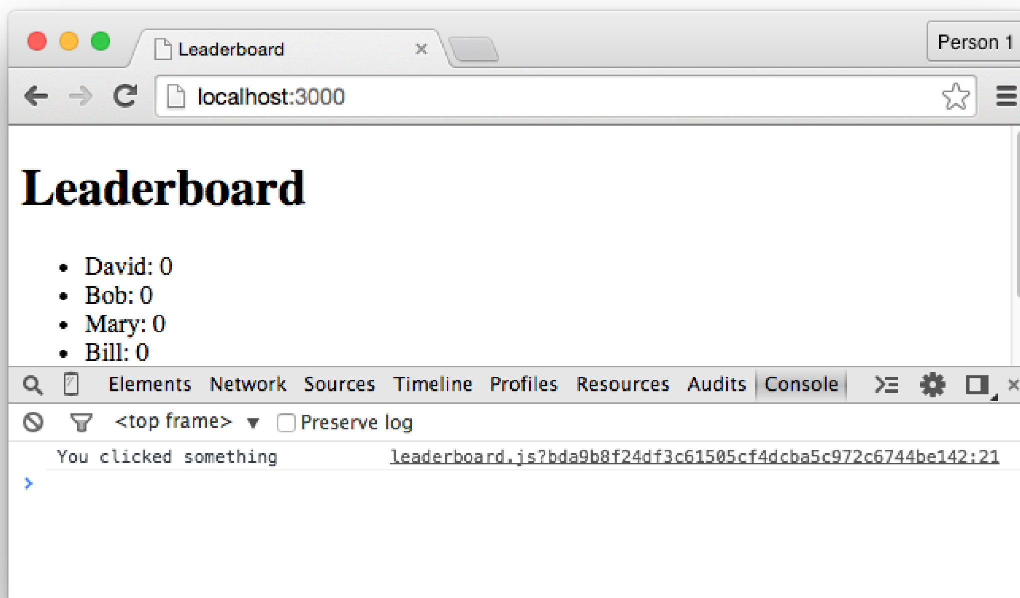
First, we've defined the event `type`. That's the `click` part. Because of this, the code inside the associated function will execute when a user clicks anywhere within the bounds of the "leaderboard" template.

Second, we've attached a function to this event, and inside this function, we can write whatever code we want to execute when that click occurs.

To demonstrate this, add a `console.log` statement inside the event:

```
Template.leaderboard.events({
    'click': function(){
        console.log("You clicked something");
    }
});
```

After saving the file, switch back to Chrome and click anywhere within the bounds of the "leaderboard" template. With each click, the "You clicked something" message will appear inside the Console.



Clicking something.

Event Selectors

The event we've created is too broad. It triggers when the user clicks anywhere within the bounds of the "leaderboard" template. That could be useful in certain circumstances, but generally we'll want an event to trigger when a user does something precise, like clicking a particular button.

To achieve this, we'll use event *selectors*, and selectors allow us to attach events to specific HTML elements. (If you've ever used jQuery, this process will be familiar, but if not, it'll still be quite easy to grasp.)

Earlier, we placed `li` tags inside the HTML file:

```
    <li>{{name}}: {{score}}</li>
```

The plan now is to make it so our event triggers when the user clicks on one of these `li` elements.

To do this, change the event to the following:

```
'click li': function(){
    console.log("You clicked an li element");
}
```

Here, we've made two changes:

First, we've added the `li` part after the `click` keyword. This means the event will now trigger when a user clicks any `li` element inside the "leaderboard" template.

Second, we've changed the output of the `console.log` statement.

There is, however, something that we haven't considered:

What would happen if we had other `li` elements inside the "leaderboard" template that aren't part of the player's list? That'll be the case later on and, in our code's current form, it'd cause a problem. The event would trigger when we didn't want it to trigger.

To fix this, add a `.player` class to the `li` elements:

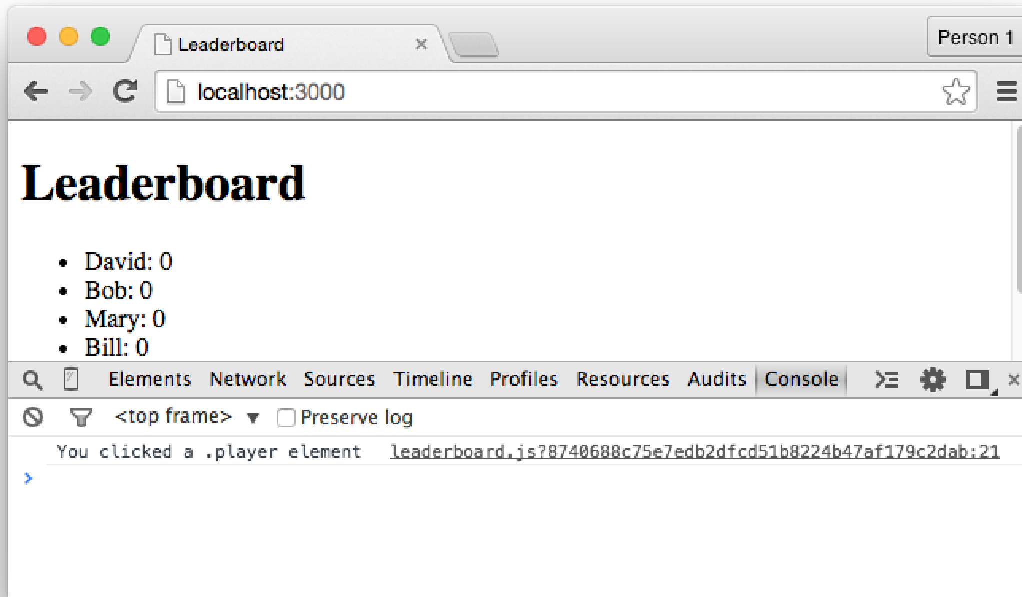
```
<li class="player">{{name}}: {{score}}</li>
```

Then use this class as the event selector:

```
'click .player': function(){
    console.log("You clicked a .player element");
}
```

Here, we've made is so the event will only trigger when the user clicks on an element that has the `.player` class attached to it.

After saving the file, the final result won't appear any different, but if we add other `li` elements to the template, the event won't trigger when it's not supposed to.



Clicking a player.

Summary

In this chapter, we've learned:

- When a user clicks a button, submits a form, or completes a range of other actions, we can trigger the execution of the code by using events.
- The most common event type is `click`, but there's a range of other options available to make our application interactive in a number of different ways.
- Through the use of event selectors, we're able to attach events to precise elements with a similar syntax to jQuery and CSS.

To gain a deeper understanding of Meteor:

- Experiment with different event types, including: `dblclick`, `focus`, `blur`, `mouseover`, and `change`. Figure out how these different types behave and try to integrate them with the Leaderboard application.

To see the code in its current state, check out the GitHub commit.

Sessions

When a user clicks one of the `.player` elements, a function executes. When this function is triggered, we want to change the background color of that element, thereby creating the effect of the player being selected.

To achieve this, we'll use *sessions*, and sessions allow us to store small pieces of data that is not saved to the database and and won't be remembered on return visits. This sort of data might not sound immediately useful, but it's a surprisingly versatile way to solve a lot of common problems.

Create a Session

To create a session, write the following statement inside the `click .player` event:

```
Session.set('selectedPlayer', 'session value test');
```

Here, we're using this `Session.set` function and passing through two arguments:

First, **we're passing through a name for the session**. This name is used as a reference. In this case, we're calling the session "selectedPlayer", but feel free to use whatever name you like.

Second, **we're passing through a value for the session**. This is the data we're storing inside the session. In this case, we're passing through the static value of "session value test", but we'll use a more interesting value in a moment.

To prove that our session is working as expected, retrieve the value of the sessions with the following statement:

```
Session.get('selectedPlayer');
```

The `events` block should then resemble:

```
Template.leaderboard.events({
    'click .player': function(){
        Session.set('selectedPlayer', 'session value test');
        Session.get('selectedPlayer');
    }
});
```

Here, we're using this `Session.get` function and passing through the name of the "selectedPlayer" session that we created a moment ago.

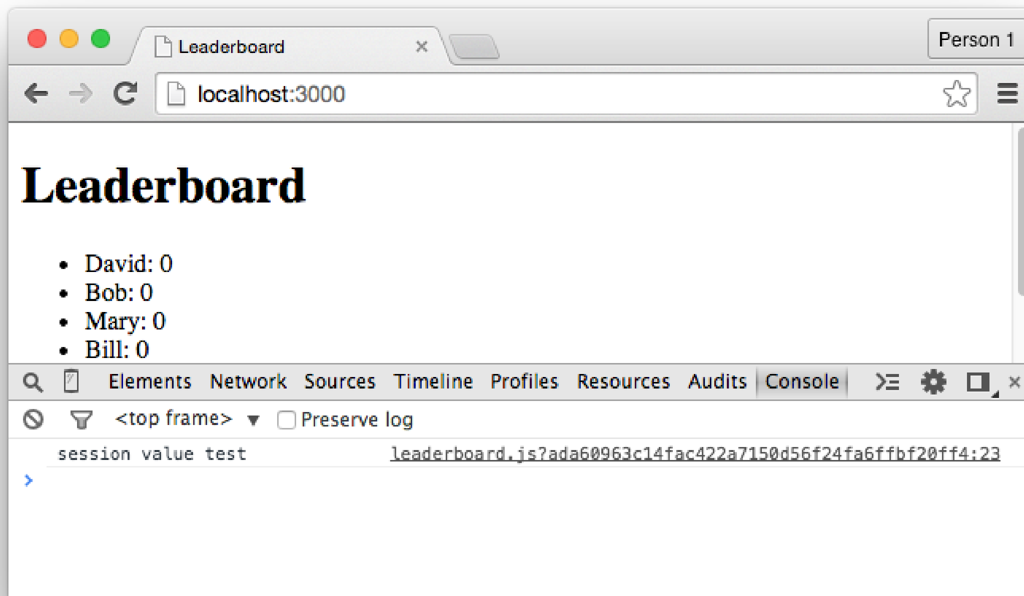
To output the value of this session to the Console, place it inside a variable:

```
var selectedPlayer = Session.get('selectedPlayer');
```

Then add a `console.log` statement beneath this line:

```
var selectedPlayer = Session.get('selectedPlayer');
console.log(selectedPlayer);
```

Now when a user clicks on one of the `player` elements, the "session value test" string will be stored inside a session and then immediately output to the Console. It's not the most useful code, but that'll change soon.



Creating and retrieving a session.

The Player's ID

When a user clicks one of the players in the list, we want to grab the unique ID of the player and store it inside the "selectedPlayer" session. This will then allow us to change the background color of that particular player's `li` element.

If you're not sure of what I mean when I say "the unique ID of the player" though, think back to when we inserted the players into the "PlayersList" collection. Each time we used the `insert` function, a random jumble of numbers and letters would appear. That jumble was the unique ID of that player.

To begin, create a "playerId" variable at the top of the `click .player` event, and make it equal to the "session value test" string from before:

```
var playerId = "session value test";
```

Then modify the `Session.set` function by passing through the "playerId" variable as the second argument. The event should then resemble:

```
'click .player': function(){
    var playerId = "session value test";
    Session.set('selectedPlayer', playerId);
    var selectedPlayer = Session.get('selectedPlayer');
}
```

At this point, the trick is to make the "playerId" variable equal to the unique ID of the player that's been clicked. This doesn't require a lot of code, but it does require some explanation.

For now, change the "playerId" variable to the following:

```
var playerId = this._id;
```

As for the explanation, there's two things going on:

First, we have a reference to `this`, and the value of `this` depends on the context. In this context, `this` refers to the document of the player that has just been clicked.

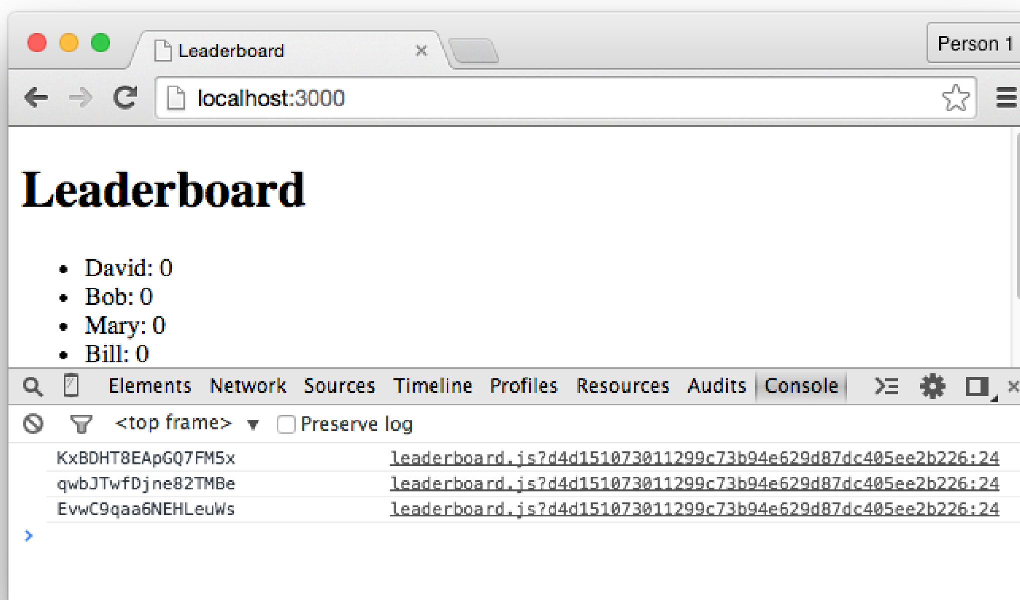
Second, the `_id` part is the name of the field that contains the unique ID of the player. So in the same way we created a `name` and `score` field, Mongo creates an `_id` field for each document. (The underscore itself doesn't have any special significance. It's just part of the field's name.)

Because of this change, the following is now possible:

1. A user clicks on one of the players.
2. The player's unique ID is stored inside the "playerId" variable.
3. The value of the "playerId" variable is stored inside the "selectedPlayer" session. (There can only be one value stored inside a session, so each time a new value is stored, the previous value is overwritten.)

4. The value of the “selectedPlayer” session is output to the Console.

To see this in action: save the file, switch back to Chrome, and click on any of the players in the list. Their unique ID will appear inside the Console.



After clicking on David, Bob, and Mary.

Since we don’t need to see the unique ID of the clicked player inside the Console though, we can simplify the event to the following:

```
'click .player': function(){
    var playerId = this._id;
    Session.set('selectedPlayer', playerId);
}
```

Here, we’re just setting the value of the “selectedPlayer” session to the unique ID of the clicked player.

Selected Effect, Part 1

When a user clicks one of the players inside our list, we want to change the `background-color` property of the `li` element that contains that player. This will create the effect of that player being selected.

To achieve this, open the project's CSS file and create a class named "selected". This class should have a `background-color` property, and for this example we'll pass through a value of "yellow":

```
.selected{
    background-color: yellow;
}
```

Then switch to the JavaScript file and create a "selectedClass" helper:

```
Template.leaderboard.helpers({
    'player': function(){
        return PlayersList.find()
    },
    'selectedClass': function(){
        // code goes here
    }
});
```

(You'll notice that both of the helpers are inside the same block of code, and as we talked about before, this is possible with the use of commas.)

As for the content of this function, we'll make it return the word "selected":

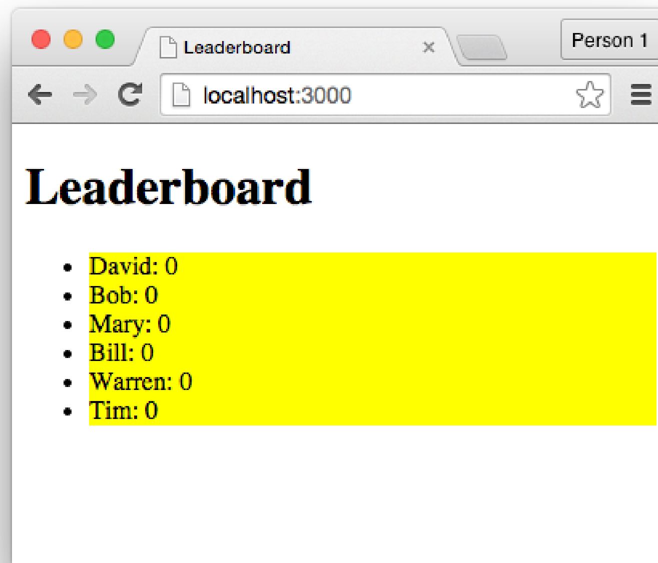
```
'selectedClass': function(){
    return "selected"
}
```

Note: We need the returned text to be equal to the name of the class in the CSS file, so because we named the class "selected" in the CSS file, we're returning this "selected" text from within the function.

Next, switch to the HTML file and place a reference to this "selectedClass" function inside the `li` element's `class` attribute:

```
<li class="player {{selectedClass}}">{{name}}: {{score}}</li>
```

The "selected" class will be applied to each `.player` element, thereby changing the background color of each element to yellow:



The "selected" class is applied to the `li` elements.

This not exactly what we want but it's an important step.

Selected Effect, Part 2

Before we continue, I want to demonstrate something.

Inside the `selectedClass` helper function, comment out the `return` statement:

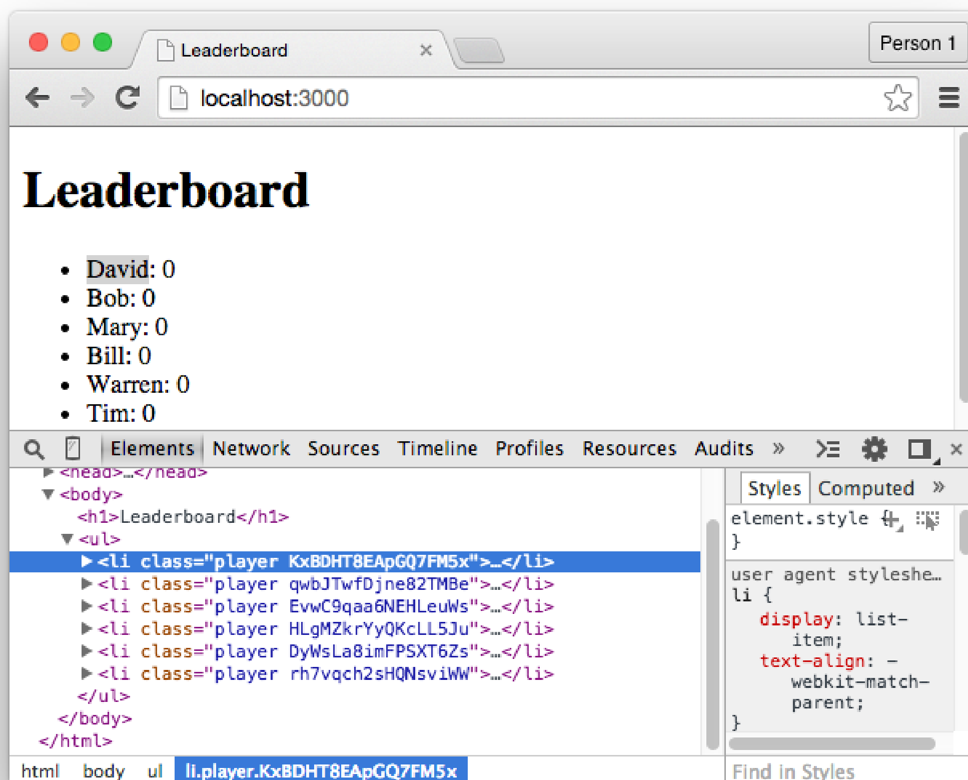
```
'selectedClass': function(){
    // return "selected"
}
```

Then write the following:

```
'selectedClass': function(){
    // return "selected"
    return this._id
}
```

Here, we're using `this._id` to retrieve the unique ID of the player. But instead of the ID being output to the Console, it'll appear inside the `class` attribute for each of the `li` elements. This is not exactly what we want but it's important to know that, because the `selectedClass` function is being executed from inside the `each` block, it has access to all of the data that is being iterated through (including the player's unique ID, name, and score).

For proof of this: save the file, switch to Chrome, right click on one of the `li` elements, and select the "Inspect Element" option. You'll notice that the unique ID of each player now appears inside the `class` attribute:



The unique ID of the players inside each `class` *attribute.*

Knowing this, we'll do a few things:

First, we'll delete the `return this._id` statement since it was only for demonstration purposes.

Second, we'll uncomment the `return` statement since we do want the `selectedClass` function to return the static text of "selected".

Third, we'll create a "playerId" variable at the top of the function that holds the value of `this._id`:

```
'selectedClass': function(){
    var playerId = this._id;
    return "selected"
}
```

Fourth, we'll create a "selectedPlayer" variable for the "selectedPlayer" session:

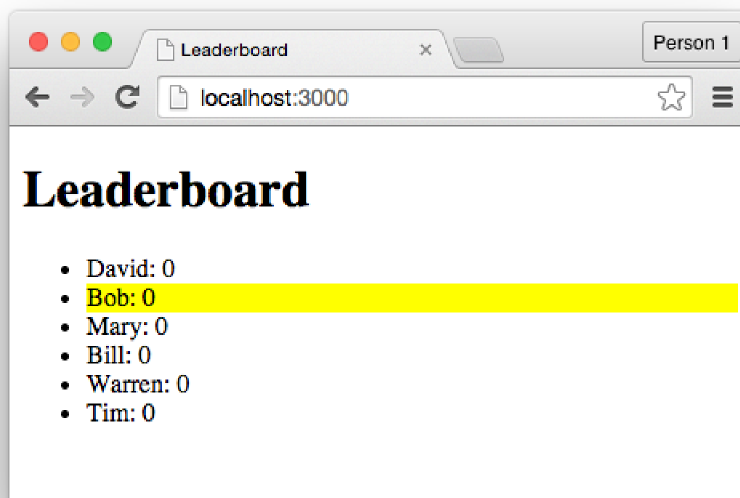
```
'selectedClass': function(){
    var playerId = this._id;
    var selectedPlayer = Session.get('selectedPlayer');
    return "selected"
}
```

Fifth, wrap the `return` statement in the following conditional:

```
'selectedClass': function(){
    var playerId = this._id;
    var selectedPlayer = Session.get('selectedPlayer');
    if(playerId == selectedPlayer){
        return "selected"
    }
}
```

If you're having trouble following the logic though, here's what's going on:

When a user clicks one of the players in the list, the unique ID of that player is stored inside the "selectedPlayer" session. The ID in that session is then matched against all of the IDs of the players in the list. Because the player's ID will always be unique, there can only ever be a single match, and when there is that match, the static text of "selected" will be returned by the `selectedClass` function and placed inside the `class` attribute for that players `li` element. Based on that class, the background color of the player's `li` element will be changed to yellow. (And because the session can only store a single value, only one player can be selected at a time.)



After clicking on Bob.

This is the most convoluted example in this book, but you only need a basic grasp of sessions to follow the remaining chapters. It doesn't really matter if you don't "get" everything right away.

Summary

In this chapter, we've learned that:

- Sessions are used to store small pieces of data that aren't saved to the database or remembered on return visits.
- To create a session, we use the `Session.set` function, while to retrieve the value of a session we use the `Session.get` function.
- Helper functions and events inside an `each` block have access to the data that's being iterated through by that block.

To gain a deeper understanding of Meteor:

- Consider how else we might use the "selectedPlayer" session. What else can we do with the unique ID of the selected player?

To see the code in its current state, check out the GitHub commit.

Databases, Part 2

There's a lot more to cover in the remaining pages of this book, but we have finished most of the features from the original Leaderboard application.

The remaining features we'll work on in this chapter include:

- The ability to increment the score of a selected player.
- Ranking players by their score (from highest to lowest).
- Displaying the selected player's name beneath the list.

We'll also make it possible to decrement the score of a selected player, which isn't a feature of the original application, but is simple enough to add.

Give 5 Points

Inside the "leaderboard" template, we'll create a "Give 5 Points" button that, when clicked, will increment the score of the selected player.

To begin, place the following button inside the "leaderboard" template:

```
<input type="button" class="increment" value="Give 5 Points">
```

The button should be located outside of the `each` block and have a `class` attribute that's set to "increment".

To make the button do something, add the following event to the `events` block that's inside the JavaScript file:

```
'click .increment': function(){
    // code goes here
}
```

The entire `events` block should resemble:

```
Template.leaderboard.events({
    'click .player': function(){
        var playerId = this._id;
        Session.set('selectedPlayer', playerId);
    },
    'click .increment': function(){
        // code goes here
    }
});
```

(Don't forget to separate the events with commas.)

Within the `click .increment` event, we'll use the unique ID of the selected player to find that player inside the "PlayersList" collection and increment the value of that player's `score` field by `5`.

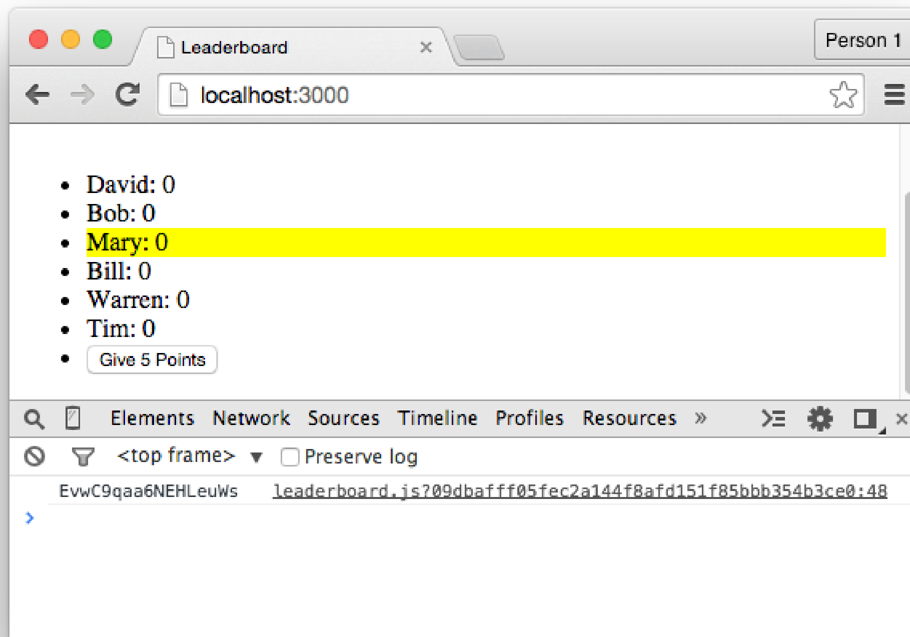
To access the unique ID of the selected player, use the `Session.get` function:

```
'click .increment': function(){
    var selectedPlayer = Session.get('selectedPlayer');
}
```

You can verify this functionality with a `console.log` statement:

```
'click .increment': function(){
    var selectedPlayer = Session.get('selectedPlayer');
    console.log(selectedPlayer);
}
```

After selecting a player, click the “Give 5 Points” button to display the unique ID of the selected player.



After clicking the “Give 5 Points” button, Mary’s ID appears in the Console.

Advanced Operators, Part 1

At this point, we want to make is so, when a user selects a player from the list and clicks the "Give 5 Points" button, the `score` field for that player is modified.

To do this, remove the `console.log` statement from the `click .increment` event and replace it with the following:

```
PlayersList.update();
```

This is the Mongo `update` function and, between the brackets, we can define:

1. What document (player) we want to modify.
2. How we want to modify that document.

To do this, we'll first retrieve the selected player's document. This can be done by passing through the unique ID of the selected player:

```
PlayersList.update(selectedPlayer);
```

The event should now resemble:

```
'click .increment': function(){
    var selectedPlayer = Session.get('selectedPlayer');
    PlayersList.update(selectedPlayer);
}
```

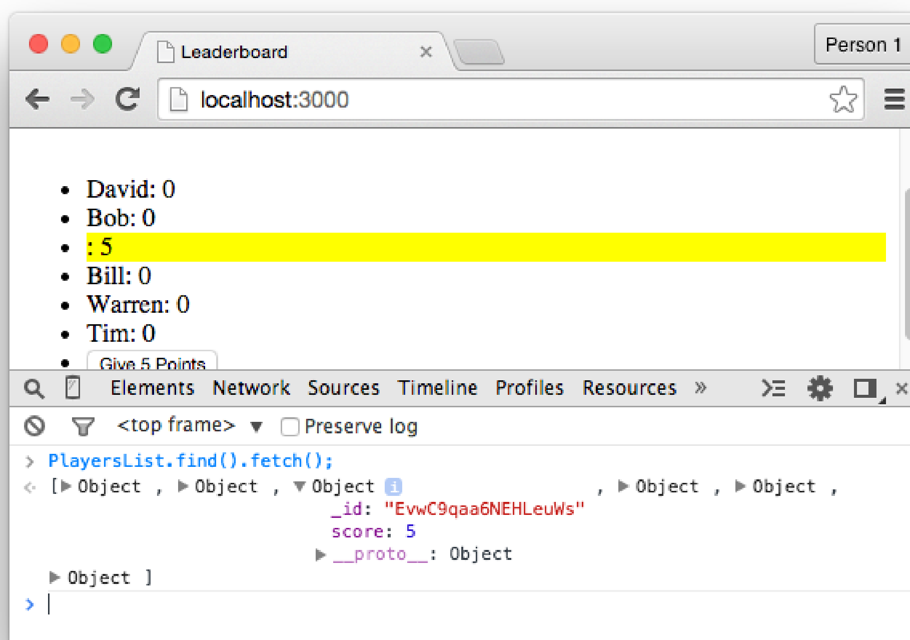
To modify the document, we pass a second argument into the `update` function to define what part of the document we want to change:

```
'click .increment': function(){
    var selectedPlayer = Session.get('selectedPlayer');
    PlayersList.update(selectedPlayer, {score: 5});
}
```

This statement will:

1. Find the document of the selected player, based on that player's ID.
2. Update that document by changing the value of the `score` field to `5`.

But if you test this feature, you'll notice that it's broken. If you select a player and click the "Give 5 Points" button, the name of that player will disappear. The value of the `score` field will change to `5`, as planned, but the `name` field will be completely removed from the document.



Where did Mary go?

This might seem like a bug, but by default, the `update` function works by deleting the original document and creating a new document with the data that we specify. The value of the `_id` field will remain the same, but since we've only specified the `score` field inside the `update` statement, that's the only other field that remains inside the modified document.

To account for this, we need to use a Mongo operator that allows us to set the value of the `score` field without deleting the original document.

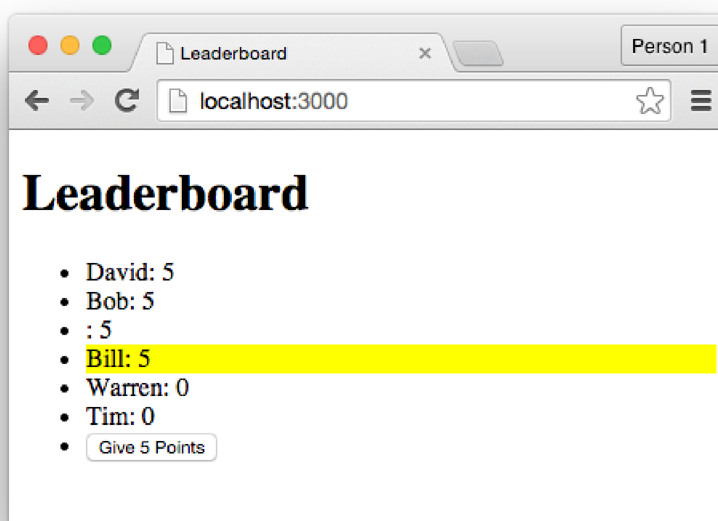
First, replace the `update` function's second argument with the following:

```
'click .increment': function(){
    var selectedPlayer = Session.get('selectedPlayer');
    PlayersList.update(selectedPlayer, {$set: });
}
```

Here, we're using this `$set` operator to modify the value of a field (or multiple fields) without deleting the original document. After the colon, we just have to pass through the fields we want to modify (and their new values):

```
'click .increment': function(){
    var selectedPlayer = Session.get('selectedPlayer');
    PlayersList.update(selectedPlayer, {$set: {score: 5} });
}
```

Because of this change, the `update` function won't be completely broken. If we save the file and switch back to Chrome, we can see that selecting a player and clicking the "Give 5 Points" button will modify the `score` field of that player without affecting the rest of the document.



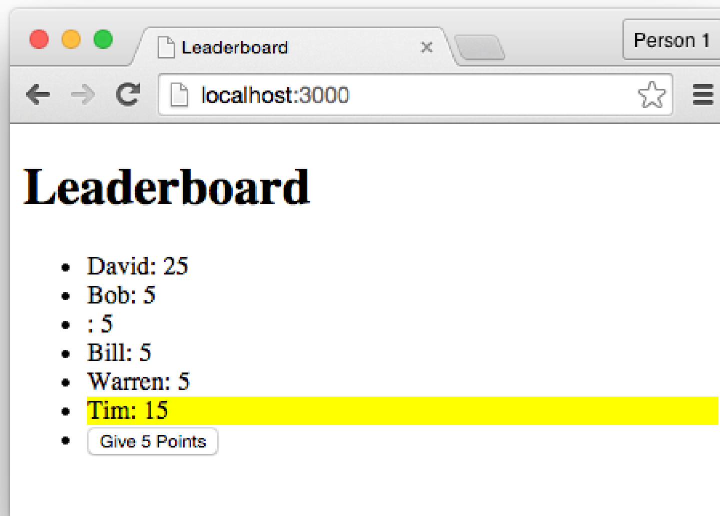
Setting the `score` field without breaking anything.

But despite this success, we still haven't created the feature that we aimed to create. Because while our button can set the selected player's `score` field to the value of `5`, that's all it can do. No matter how many times we click the button, the value of the field won't increase any further.

To fix this problem, replace the `$set` operator with the `$inc` operator:

```
'click .increment': function(){
    var selectedPlayer = Session.get('selectedPlayer');
    PlayersList.update(selectedPlayer, {$inc: {score: 5} });
}
```

Based on this change, whenever the `update` function is triggered, the value of the `score` field will be incremented by whatever value we specify (in this case, that's the value of `5`).



It's now possible to increment the value of the `score` *field.*

Advanced Operators, Part 2

A feature that's not present in the original Leaderboard application is the ability to decrement scores. Such a feature would be useful though since it means we could:

1. Penalize players for not following the rules.
2. Retract points that are mistakenly awarded.

It's also a very simple feature to throw together.

To begin, create a "Take 5 Points" button inside the "leaderboard" template:

```
<input type="button" class="decrement" value="Take 5 Points">
```

As with the "Give 5 Points" button, place it outside the `each` block and provide it with a unique `class` attribute (such as "decrement").

Next, switch to the JavaScript file, copy the `click .increment` event, and paste the code into the same events block.

The events block should resemble:

```
Template.leaderboard.events({
    'click .player': function(){
        var playerId = this._id;
        Session.set('selectedPlayer', playerId);
    },
    'click .increment': function(){
        var selectedPlayer = Session.get('selectedPlayer');
        PlayersList.update(selectedPlayer, {$inc: {score: 5} });
    },
    'click .increment': function(){
        var selectedPlayer = Session.get('selectedPlayer');
        PlayersList.update(selectedPlayer, {$inc: {score: 5} });
    }
});
```

At this point, we only need to make two changes:

First, change the selector for the newly created event from `.increment` to `.decrement`.

Second, pass a value of `-5` through the `inc` operator, rather than a value of `5`. The addition of the `-` reverses the functionality of the operator, so the `$inc` operator will now decrement the value of the `score` field.

The final code for the event should resemble:

```
'click .decrement': function(){
    var selectedPlayer = Session.get('selectedPlayer');
    PlayersList.update(selectedPlayer, {$inc: {score: -5} });
}
```

The code is somewhat redundant — we have two events that are almost identical — but that's something we'll fix in a later chapter.

Sorting Documents

At the moment, the players in the list are ranked by the time they were inserted into the collection, rather than being ranked by their scores.

To fix this, we'll modify the `return` statement that's inside the `player` helper function:

```
'player': function(){
    return PlayersList.find()
}
```

First, pass a pair of curly braces through the brackets of the `find` function:

```
'player': function(){
    return PlayersList.find({})
}
```

By using these curly braces, we're explicitly stating that we want to retrieve all of the data from the "PlayersList" collection. This is the default behavior, so both of these statements are technically the same:

```
return PlayersList.find()
return PlayersList.find({})
```

But by passing through the curly braces as the first argument, we can pass through a second argument, and it's within this second argument that we can define how we want to sort the data that's retrieved.

As the second argument, pass through the `sort` operator:

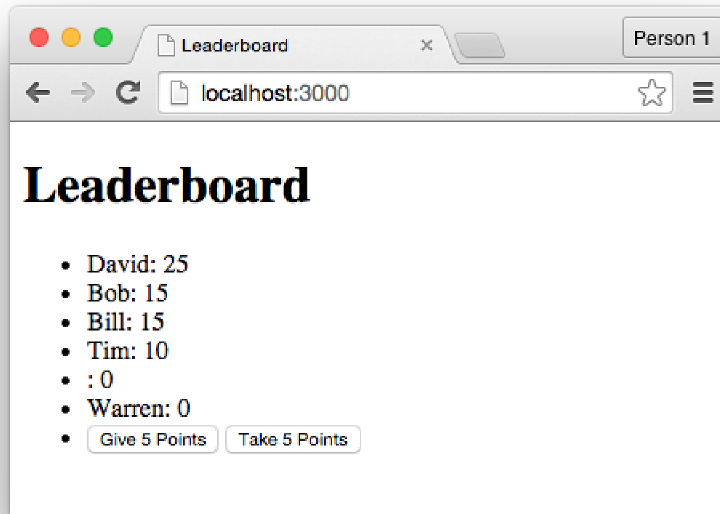
```
return PlayersList.find({}, {sort: })
```

(Unlike the `$set` and `$inc` operators, we don't use a dollar sign at the start of this operator's name.)

Then choose to sort by the value of the `score` field:

```
return PlayersList.find({}, {sort: {score: -1} })
```

By passing through a value of `-1`, we can sort in descending order. This means we're sorting the players from the highest score to the lowest score. If we passed through a value of `1`, the players would be sorted from the lowest score to the highest score.



Ranking players based on their scores.

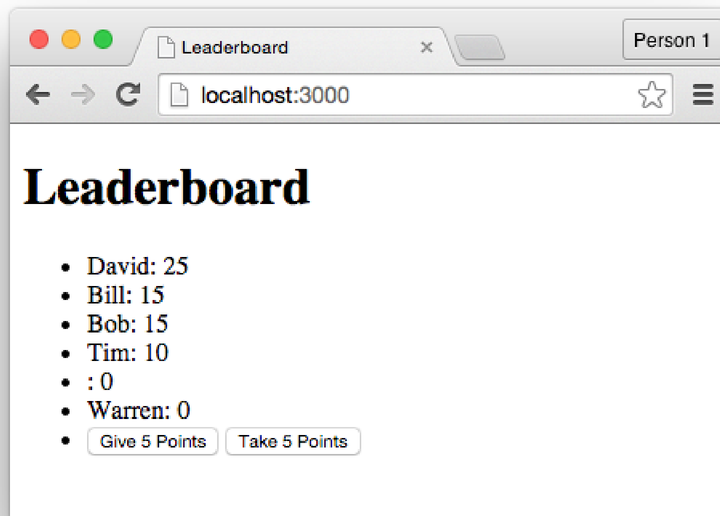
Based on this change, the players will be ranked based on their score, but what happens if two players have the same score?

Take "Bob" and "Bill", for instance. If they have the same score, Bill should be ranked above Bob because, alphabetically, his name comes first. But at the moment, that won't happen since Bob was added to the collection before Bill.

To fix this, pass the `name` field through the `sort` operator, but this time, pass through a value of `1` instead of `-1`:

```
return PlayersList.find({}, {sort: {score: -1, name: 1} })
```

The players will still be primarily ranked by their scores, but once that sorting has occurred, the players will also be ranked by their names. This secondary sorting will occur in ascending (alphabetical) order.



Ranking based on scores and names.

Based on this change, if Bob and Bill have the same scores, Bill will be ranked above Bob.

Individual Documents

When a user selects one of the players, that player's name should appear beneath the list of players. This isn't the most useful feature, but:

1. It's part of the original Leaderboard application.
2. It means we can talk about a couple of Meteor's features.

Inside the JavaScript file, create a "showSelectedPlayer" helper function that's attached to the "leaderboard" template:

```
'showSelectedPlayer': function(){
    // code goes here
}
```

Inside the function, retrieve the unique ID of the currently selected player:

```
'showSelectedPlayer': function(){
    var selectedPlayer = Session.get('selectedPlayer');
}
```

Then write a `return` statement that returns the data from a single document inside the "PlayersList" collection. We could use the `find` function, but the `findOne` function is the preferred alternative:

```
'showSelectedPlayer': function(){
    var selectedPlayer = Session.get('selectedPlayer');
    return PlayersList.findOne(selectedPlayer)
}
```

By using the `findOne` function, we can pass through the unique ID of a document as the only argument, and we're able to avoid unnecessary overhead since this function will only ever attempt to retrieve a single document. It won't look through the entire collection like the `find` function would.

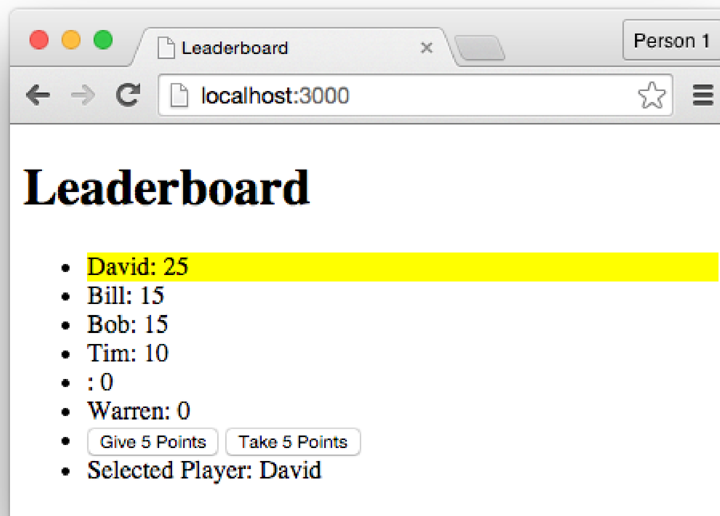
With this function in place, switch to the HTML file and place a reference to the function inside the "leaderboard" template. I've placed mine at the bottom of my list, between a pair of `li` tags:

```
<li>Selected Player: {{showSelectedPlayer}}</li>
```

But if we save the file, the output won't look quite right, and that's because the `findOne` function is retrieving the player's entire document. To fix this, we need to specify that we only want to show the value of the `name` field, which can be done with dot notation:

```
<li>Selected Player: {{showSelectedPlayer.name}}</li>
```

The interface will now resemble:



Showing the selected player's name.

We should also make it so the template doesn't attempt to display a player's name if a player isn't selected, which can be done with a simple conditional:

```
{{#if showSelectedPlayer}}
    <li>Selected Player: {{showSelectedPlayer.name}}</li>
{{/if}}
```

This list item will now only appear if a player is currently selected.

Summary

In this chapter, we've learned that:

- By default, the Mongo `update` function deletes the document that's being updated and recreates it with the specified fields (while retaining the same primary key).
- To change the values of a document without deleting it first, the `$set` operator needs to be used. This operator will only change the values of the specified documents without affecting the rest of the document.
- The `$inc` operator can be used to increment the value of a field within a particular document.
- The `$inc` operator can be used to *decrement* the value of a field by placing the minus symbol in front of the specified value.
- The `sort` operator can be used to sort the data that's returned by the `find` function. It allows for sorting by multiple fields at once.
- The `findOne` function will only ever retrieve a single document from a collection, which is the more efficient approach if you only need to retrieve a single document.

To gain a better understanding of Meteor:

- Make it so the "Give 5 Points" button only appears when the user has been selected. This is a feature of the original Leaderboard application.
- Browse through the "Operators" section of the Mongo documentation to see what's possible through pure database operations.

To see the code in its current state, check out the GitHub commit.

Forms

We've finished re-building the original Leaderboard application, but there's plenty of room to expand the application with new features. In this particular chapter, we're going to create a form that allows users to add players to the leaderboard, along with some other interface controls.

Create a Form

Inside the HTML file, create a second template named “addPlayerForm”:

```
<template name="addPlayerForm">
</template>
```

Then include this somewhere inside the “leaderboard” template:

```
{{> addPlayerForm}}
```

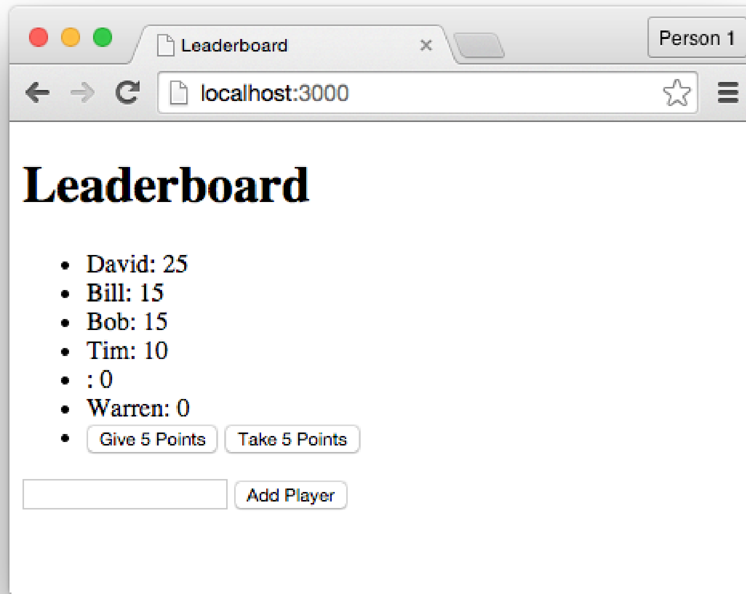
Within the “addPlayerForm” template, create the following two elements:

1. A text field with the `name` attribute set to “playerName”.
2. A submit button with the `value` attribute set to “Add Player”.

The template should then resemble:

```
<template name="addPlayerForm">
    <form>
        <input type="text" name="playerName">
        <input type="submit" value="Add Player">
    </form>
</template>
```

The resulting interface won’t be that pretty, but it’s all we need.



A form for adding players to the leaderboard.

The “submit” Event

We’ve already seen a couple examples of the `click` event, which allows us to trigger the execution of code when the user clicks on a particular element. In a similar vein, there’s also the `submit` event, which allows to trigger the execution of code when the user submits a form.

To do this, create another events block inside the `isClient` conditional:

```
Template.addPlayerForm.events({
    // events go here
});
```

(We need a new events block because this event will be attached to the new “addPlayerForm” template, rather than the “leaderboard” template.)

Within this events block, create an event with the event type set to “submit” and the selector set to “form”:

```
Template.addPlayerForm.events({
    'submit form': function(){
        // code goes here
    }
});
```

Based on this code, the event’s function will trigger when the form inside the “addPlayerForm” template is submitted.

But why don’t we just use the `click` event for the form? Won’t most users click the submit button anyway? That might be the case, but it’s important to remember that forms can be submitted in a number of ways. In some cases, the user will click the submit button, but at other times they’ll tap the “Return” key on their keyboard. By using the `submit` event type, we’re able to account for every possible way that form can be submitted.

To confirm the event is working as expected, place a `console.log` statement inside of it:

```
'submit form': function(){
    console.log("Form submitted");
}
```

But as it turns out, there is actually a problem with the event, because when we submit the form:

1. The web browser refreshes the page.
2. The “Form submitted” message doesn’t appear inside the Console.

Why does this happen?

When we place a form inside a web page, the browser assumes we want to take the data from that form and send it somewhere. The problem is, when working with Meteor, we don't want to send the data anywhere — we want it to remain within the current page — but since this isn't standard behavior as far as the browser is concerned, the web page simply refreshes.

Knowing this, **we must be disable the default behavior that web browsers attach to forms**.

This takes a couple of steps.

The Event Object, Part 1

When an event is triggered from within a Meteor application, we can access information about that event as it occurs. That might sound weird, but to show you what I mean, change the `submit form` event to the following:

```
'submit form': function(event){
    console.log("Form submitted");
    console.log(event.type);
}
```

Here, we're passing this "event" keyword through the brackets of the event's function, and then outputting the value of `event.type` to the Console.

This result of this is two-fold:

First, whatever keyword is passed through the brackets of event's function as the first parameter becomes a reference for that event. Because we've passed through the "event" keyword, we're able to reference the event inside the event's function using that keyword. You can, however, use whatever keyword you prefer. (A common convention is to use "evt" or "e" instead of "event".)

Second, the `event.type` part is a reference to the "type" property of the event object. As a result, this code should output the word "submit" to the Console since that's the type of event that's being triggered.

This doesn't solve the original problem though since our page still refreshes whenever the form is submitted and we can't see the `console.log` statements.

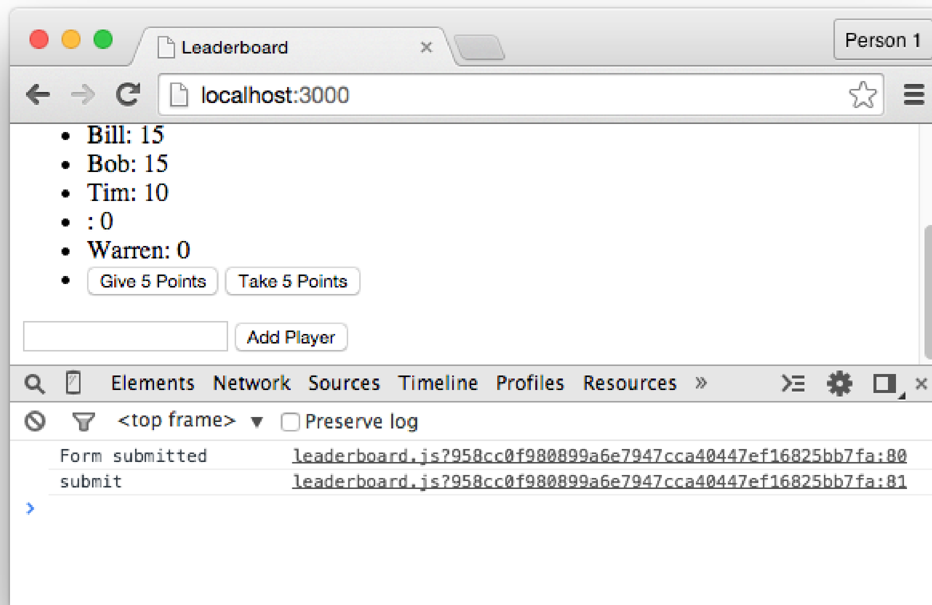
To fix this, use the `preventDefault` function:

```
'submit form': function(event){
    event.preventDefault();
    console.log("Form submitted");
    console.log(event.type);
}
```

When attached to the event object, this `preventDefault` function prevents the default behavior of the event from occurring. So because we've attached the function to the `submit form` event:

1. By default, submitting the form won't do anything.
2. We'll need to manually define the form's functionality.
3. The `console.log` statements will now work as expected.

Save the file, switch back to Chrome, and test the form to see that it's no longer refreshing the page.



Taking control of the form.

Note: The `preventDefault` function does not just apply to forms. You can, for instance, take complete control of the links within a template:

```
'click a': function(event){
    event.preventDefault();
}
```

With this code in place, any `a` elements within the template wouldn't behave as they usually would. You'd have to manually assign them functionality.

The Event Object, Part 2

Now that we have complete control over the form, we want the `submit form` event to grab the contents of the “playerName” text field when the form is submitted, and use that value when adding a player to the database.

To begin, create a variable named “playerNameVar”:

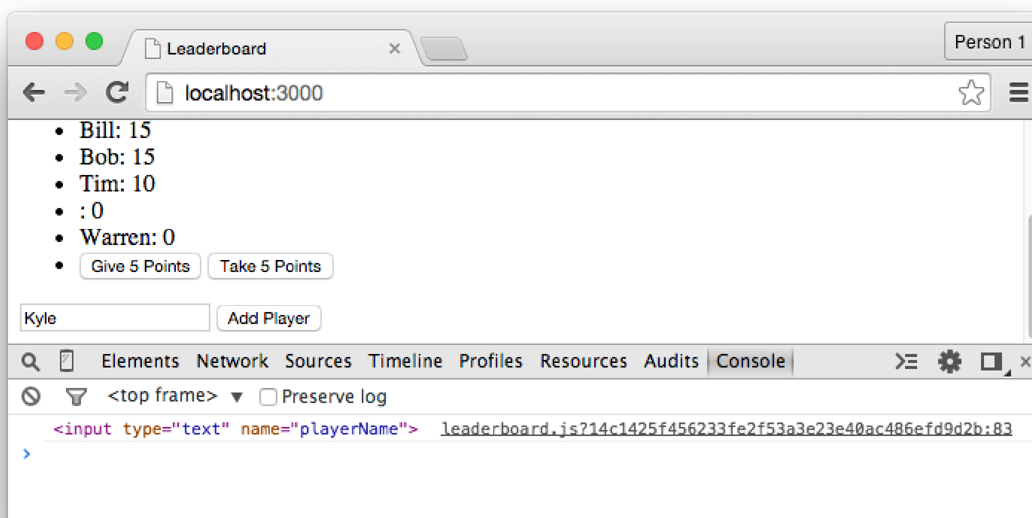
```
'submit form': function(event){
    event.preventDefault();
    var playerNameVar;
}
```

Then make this variable equal to “event.target.playerName” and output the value of the variable to the Console:

```
'submit form': function(event){
    event.preventDefault();
    var playerNameVar = event.target.playerName;
    console.log(playerNameVar);
}
```

Here, this statement uses the event object to grab whatever HTML element has the `name` attribute set to “playerName”.

But this code won’t work exactly as you might expect, since the `console.log` statement outputs the raw HTML for the text field, rather than its value:

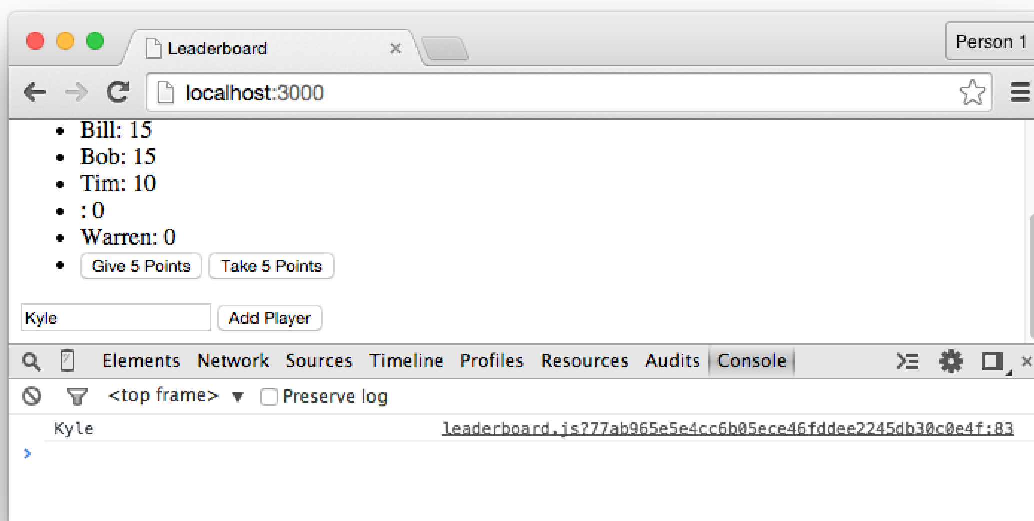


Grabbing the entire text field.

This is because we need to explicitly retrieve the `value` property:

```
'submit form': function(event){
    event.preventDefault();
    var playerNameVar = event.target.playerName.value;
    console.log(playerNameVar);
}
```

Based on this change, whatever the user types into the "playerName" text field will now be output to the Console when they submit the form.



Value of the text field appearing in the Console.

To insert the submitted player into the "PlayersList" collection, add the `insert` function inside the `submit form` event:

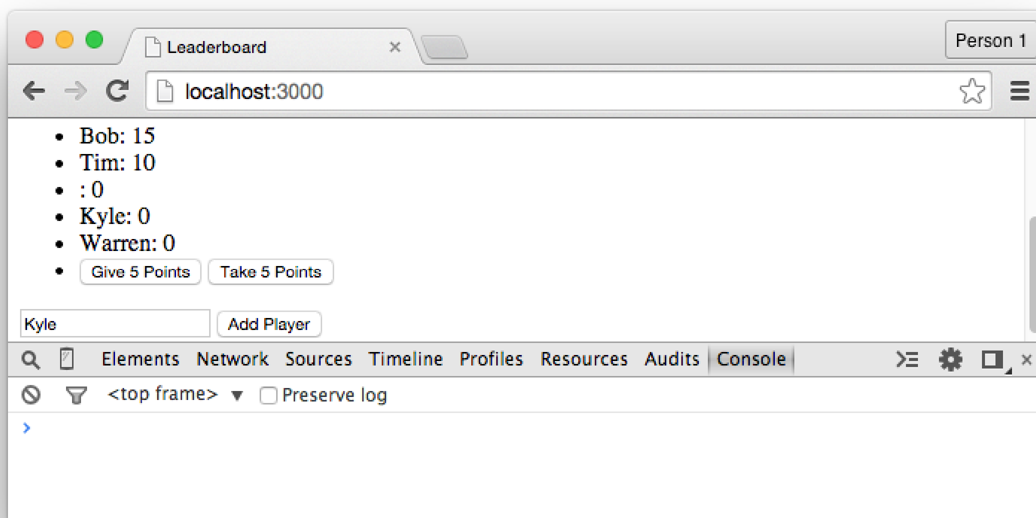
```
PlayersList.insert({
    name: playerNameVar,
    score: 0
});
```

But rather than passing through a hard-coded value to the `name` field, like "David" or "Bob", pass through a reference to the "playerNameVar" variable.

The code for the event should now resemble:

```
'submit form': function(event){
    event.preventDefault();
    var playerNameVar = event.target.playerName.value;
    PlayersList.insert({
        name: playerNameVar,
        score: 0
    });
}
```

And the form should now work as expected.



Kyle is added to the leaderboard.

Removing Players

Since we've made it possible to add players to the leaderboard, it's a good idea to make it possible to also remove players from the leaderboard.

To achieve this, first create a "Remove Player" button from inside the "leaderboard" template:

```
<input type="button" class="remove" value="Remove Player">
```

As with the other buttons in this project, attach a unique `class` attribute to it so we can reference the button from an events block.

Inside the JavaScript file, attach the following event to the "leaderboard" template:

```
'click .remove': function(){
    // code goes here
}
```

Retrieve the ID of the selected player from the "selectedPlayer" session:

```
'click .remove': function(){
    var selectedPlayer = Session.get('selectedPlayer');
}
```

Then use the `remove` function to remove the selected player from the collection:

```
'click .remove': function(){
    var selectedPlayer = Session.get('selectedPlayer');
    PlayersList.remove(selectedPlayer);
}
```

We haven't talked about the `remove` function yet, but there's nothing much to say about it. All we have to do is pass through the unique ID of a document as the only argument. That document will then be removed from the collection.

Users will now be able to delete players from the list.

Summary

In this chapter, we've learned that:

- By using the `submit` event type, we can trigger the execution of code when a form is submitted.
- The `submit` event is used instead of the `click` event since a form can be submitted in a number of different ways (like tapping the "Return" key.).
- We can access information about an event from inside that event's function, and also manipulate that event as it occurs.
- Browsers attach default behavior to forms, which interferes with our code, but this behavior can be disabled with the `preventDefault` function.
- When form fields have a `name` attribute, there's a easy syntax for grabbing the value of that form field.
- By passing the ID of a Mongo document through a `remove` function, we can remove that specific document from a collection.

To gain a deeper understanding of Meteor:

- Make it so, after submitting the "Add Player" form, the value of the "playerName" text field is reset to an empty value.
- Create an alert that asks users to confirm whether or not they *really* want to remove a player from the list after they click the "Remove Player" button.
- Add a "Score" field to the "Add Player" form, allowing users to define a score for a player when they're being submitted to the list.

To see the code in its current state, check out the GitHub commit.

Accounts

Our application has a number of useful features but still only supports a single list of players. This means there can only be one of users of the application at any particular time, which is just silly for an application on the web.

To fix this, we'll create a user accounts system, which just so happens to be one of the simplest things we can do with the framework.

With this system in place, we'll make it so:

- Users can register and login to the application.
- Logged-out users won't see the "Add Player" form.
- Every user will have their own, unique leaderboard.

It's a lot of functionality, but it won't take a lot of code.

Login Provider

To extend the functionality of our Meteor projects in a matter of seconds, we can install a range of *packages*, and packages are essentially plugins that:

1. Add important features to a project.
2. Reduce the amount of code we need to write.

By default, every Meteor project has local access to a number of official packages. These are packages that most developers will need to use at some point or another, but not necessarily inside every project. (There are also thousands of third-party packages, but they're beyond the scope of this book, so we'll just focus on the official packages.)

To add a user accounts system to our project, we'll first install a "login provider" package. These packages make it extremely easy to add a back-end for an accounts system to an application.

Usually, for instance, creating a user accounts system would involve creating a collection for the user's data:

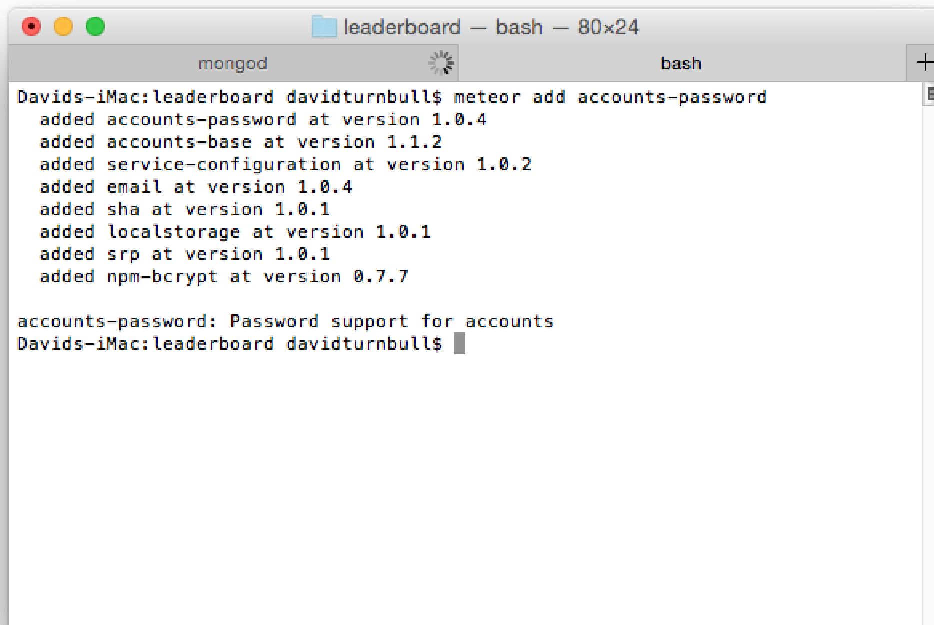
```
UserAccounts = new Mongo.Collection('users');
```

Then writing the application logic for registration and logging in, etc.

But when working with Meteor, all we have to do is switch to the command line and run the following command:

```
meteor add accounts-password
```

Here, we're adding this "accounts-password" package to the project. This package creates the back-end for an accounts system that relies on an email and password for registration and logging in.



Adding the `accounts-password` package to the project.

Specifically, this package:

1. Creates a collection for storing the data of registered users.
2. Provides us with a range of useful functions we'll soon cover.

There's other login provider packages available that allow users to login to our application through services like Google and Facebook, but since this adds an extra step to the process, we'll focus on an email and password system.

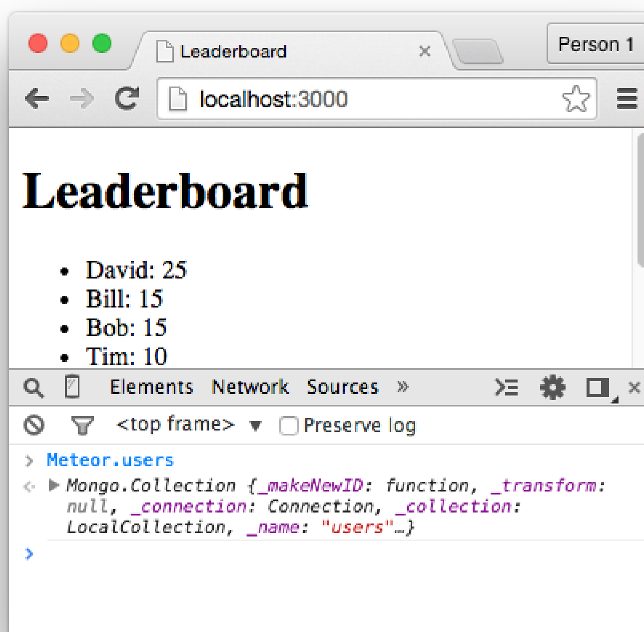
Meteor.users

Once the "accounts-password" package is added to the project, a collection is automatically created to store the data of registered users. This collection is known as `Meteor.users` and it works just like any collection that we might create ourselves.

To demonstrate this, enter the following command into the Console:

```
Meteor.users
```

The returned information confirms that this is just a regular collection:



Checking out the `Meteor.users` collection.

Knowing this, we can use the `find` and `fetch` functions on this collection:

```
Meteor.users.find().fetch();
```

But since there are no registered users yet, no data will be returned.

Login Interface

We've already setup the back-end for an accounts system, but what about the front-end? Are we expected to write the interface code that allows people to register and login and change their account details?

Nope.

We can create a custom interface, and it's actually a very simple thing to do, but there's an easier way to get up and running as soon as possible.

To instantly add the front-end of an accounts system to a project, we simply have to install the "accounts-ui" package:

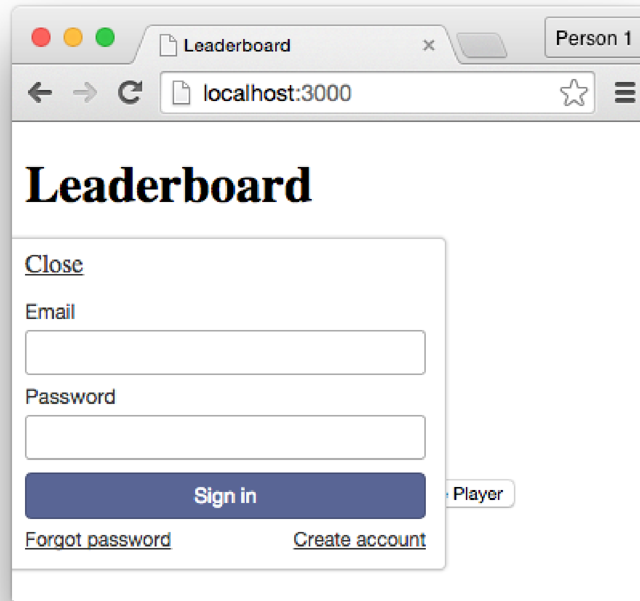
```
meteor add accounts-ui
```

Then, once installed, place the following between the `body` tags of the HTML file (or inside one of the templates):

```
{{> loginButtons}}
```

Here, we're including this "loginButtons" template, which is included with the "accounts-ui" package. So because that package has been added to this project, we can now include this template anywhere we want within the interface.

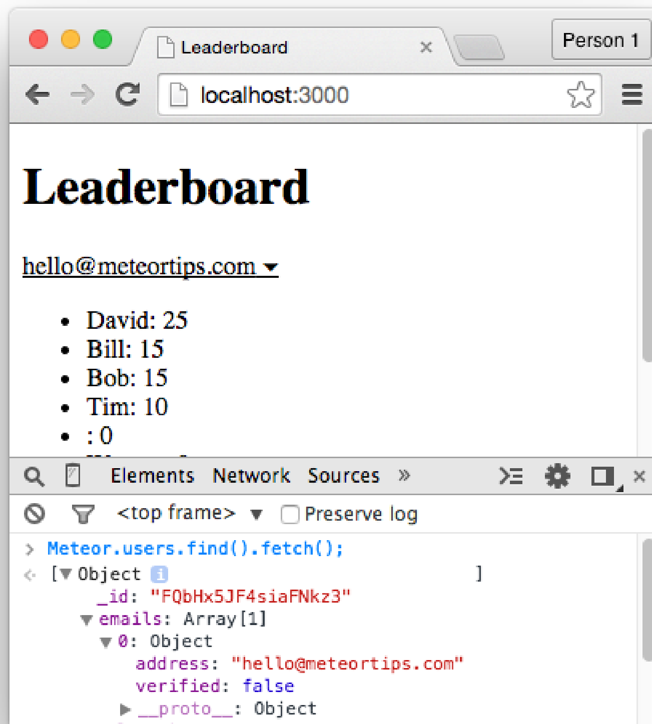
To see what this template contains, save the file and switch to the browser. You'll notice a "Sign In" button has appeared, and when clicked, a login form and a "Create Account" link will appear:



An instant interface.

This is not a mere dummy interface though. Already, without any configuration, it's possible for users to register, login, and logout. There's no reason to do any of these things — registered and non-registered users will see the same content — but that's something we'll fix in the next section.

For now, use the `find` and `fetch` function on the `Meteor.users` collection:



The first user's data.

You'll notice that a document is returned, and this document contains the data of the account that was just created. You can click the downward facing arrows to see the data associated with the account.

Logged-in Status

At the moment, unregistered users can see the "Add Player" form, which doesn't make a lot of sense. This form should only be accessible to registered users.

To achieve this, change the "addPlayerForm" template to the following:

```
<template name="addPlayerForm">
    {{#if currentUser}}
        <form>
            <input type="text" name="playerName">
            <input type="submit" value="Add Player">
        </form>
    {{/if}}
</template>
```

Here, we're referring to this `currentUser` object to check if the current user is logged-in or not. This object is provided by the "accounts-password" package, and the logic's fairly simple:

1. If the current user is logged-in, `currentUser` will return `true`.
2. If the current user is *not* logged-in, `currentUser` will return `false`.

As such, with just a couple lines of code, we've made it so only logged-in users will be able to see (and interact with) the form.

One Leaderboard Per User

To make our application somewhat useful to a wider audience, we need to make it so each registered user can make their own, independent list of players. It might not immediately obvious how we going about doing this — and the most difficult part about programming is figuring out how to approach such problems — but the process itself doesn't involve a lot of steps.

To begin, place the following statement:

```
var currentUserId = Meteor.userId();
```

...inside the `submit form` event:

```
'submit form': function(event){
    event.preventDefault();
    var playerNameVar = event.target.playerName.value;
    var currentUserId = Meteor.userId();
    PlayersList.insert({
        name: playerNameVar,
        score: 0
    });
}
```

Here, we're creating this "currentUserId" variable, which stores the value that's returned by the `Meteor.userId` function. We haven't talked about this function yet, but there's not much to explain. It simply returns the unique ID of the currently logged in user.

Then add a "createdBy" field inside the `insert` function, and pass through the "currentUserId" variable:

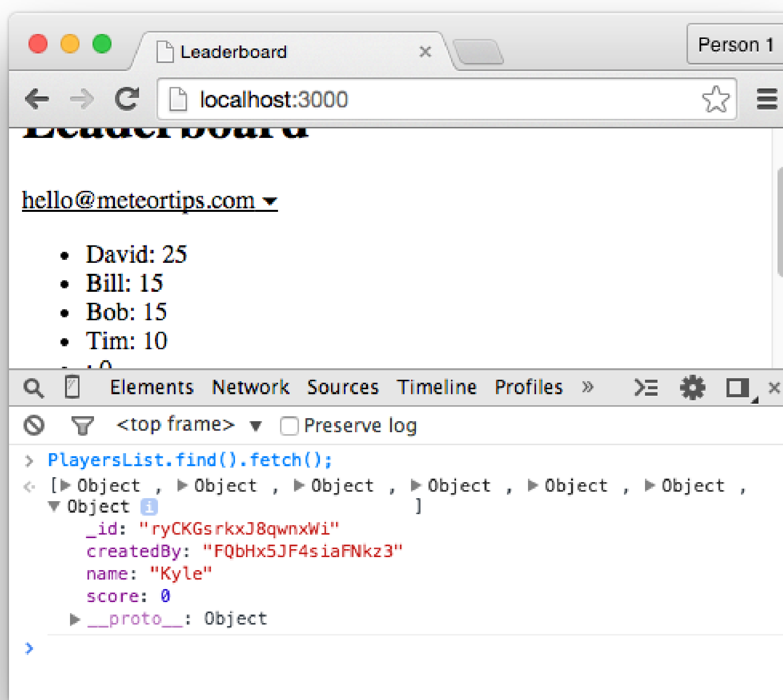
```
PlayersList.insert({
    name: playerNameVar,
    score: 0,
    createdBy: currentUserId
});
```

As a result, when a user adds a player to the leaderboard, the unique ID of that user will be associated with the player that's being added.

To demonstrate this:

1. Save the file.
2. Switch back to Chrome.
3. Add a player to the leaderboard.

Then use the find and fetch function on the “PlayersList” collection, and click the downward-facing arrow for the most recently created document. You’ll see how this document contains the ID of the user who added this player to the collection.



Associating a player with a user.

Next, we’re going to modify the player helper function:

```
'player': function(){
    return PlayersList.find({}, {sort: {score: -1, name: 1}});
}
```

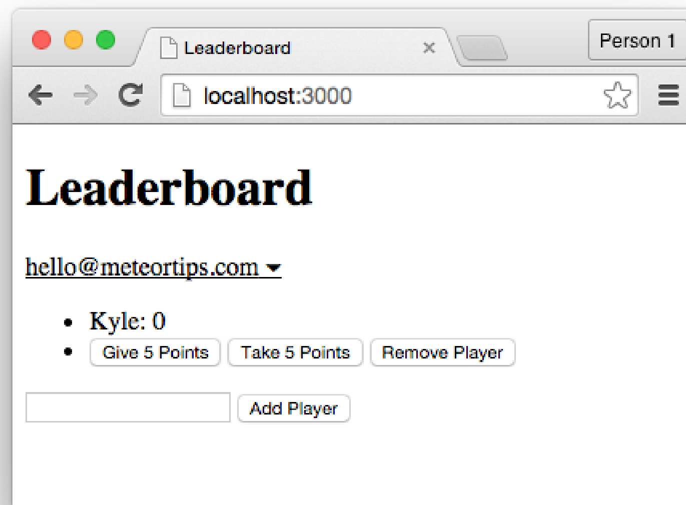
First, setup another “currentUserId” variable:

```
'player': function(){
    var currentUserId = Meteor.userId();
    return PlayersList.find({}, {sort: {score: -1, name: 1}});
}
```

Then change the return statement so it only returns players when their createdBy field is equal to the unique ID of the currently logged-in user:

```
'player': function(){
    var currentUserId = Meteor.userId();
    return PlayersList.find({createdBy: currentUserId},
                            {sort: {score: -1, name: 1}});
}
```

This ensures that users only see players they added to the leaderboard, thereby creating the effect that each user has their own, unique list of players.



Only see players that belong to the current user.

Project Reset

At the moment, there are some players in the database who aren't attached to any particular users — the players who were added to the database before the previous section — meaning we don't need them inside our collection.

To give ourselves a fresh start, then, switch to the command line, stop the local server with `CTRL + C`, and enter the following command:

```
meteor reset
```

This will wipe the database clean, and because of the code we wrote in the previous section, all players added to the collection at this point will be attached to the currently logged-in user.

You'll probably find yourself using this command semi-regularly, as lots of useless data can easily fill up a database during development.

Summary

In this chapter, we've learned that:

- Packages allow us to quickly add functionality to an application. There are some official packages, and also thousands of third-party packages.
- "Login Provider" packages are used to create the back-end of an accounts system. We can create a back-end that relies on an email and password, or on services like Twitter and Facebook (or a combination of services).
- After installing a login provider package, a `Meteor.users` collection is automatically created to store the data of registered users.
- The `accounts-ui` package allows us to quickly add a user interface for an accounts system to a project. You can take a custom approach, but taking this boilerplate approach is great for beginners.
- We can check whether or not the current user is logged-in by referencing the `curentUser` object from inside a template.
- To retrieve the unique ID of the currently logged-in user, we can use the `Meteor.userId()` function.

To gain a deeper understanding of Meteor:

- Install a different login provider package, like the `accounts-twitter` package (but make sure the `accounts-ui` package is also installed).
- Browse through atmospherejs.com to check out the many third-party packages that are available for Meteor.

To see the code in its current state, check out the GitHub commit.

Publish & Subscribe

So far, we've built a feature-rich application with Meteor, but we haven't said anything about security, which is a big part of developing software for the web. For the most part, I've wanted to show you how to build something as quickly and simply as possible, but there are a couple of security topics we should talk about before publishing the project to the web.

First up, let's talk about *publications and subscriptions.*

Data Security

To demonstrate one of our project's security flaws:

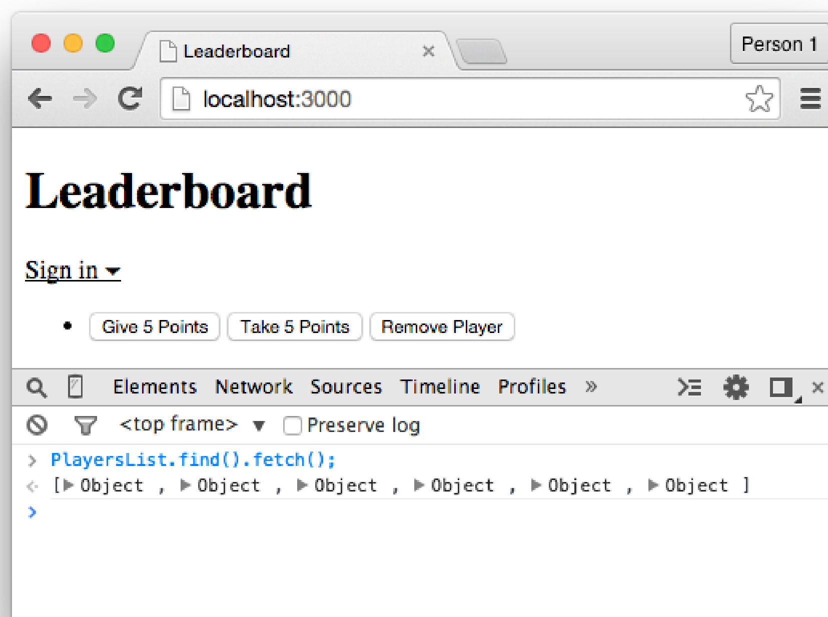
1. Register two separate user accounts.
2. Under each account, add three players.
3. Logout of both accounts.

Because of this, a total of six players should exist inside the database and they should "belong" to a total of two users.

Next, use the `find` and `fetch` function on the "PlayersList" collection:

```
PlayersList.find().fetch();
```

You'll notice that, as we've seen before, all of the data from the collection is returned. We can see all of the data that belongs to both of the users. But this is actually a problem. Because unless we turn off this feature, every user of this application will have this same, unbridled access to every bit of data inside the database. There's nothing stopping them from digging deep into the "PlayersList" collection with the `find` and `fetch` functions.



Accessing all of the data.

This project's data is not particularly sensitive — it's not like we're storing credit card numbers — but:

1. If we were storing sensitive data, this would be an unforgivable oversight.
2. It's bad practice to have data available to users when it's not necessary.

This does, however, beg the question:

Why does this feature exist inside of Meteor? If it's such a huge security risk to access data through the Console, why are allowed to do it?

And quite simply, *convenience*. Throughout this book, we've been using the `find` and `fetch` functions, and they've been great tools for managing and manipulating the contents of the database. It's just that, before we share the application with the world, we'll have to:

1. Disable this default behavior, limiting access to most of the data.
2. Precisely define what data should be available to specific users.

So this is what we'll discuss for the rest of the chapter.

autopublish

The functionality that allows us to navigate through a project's data using the Console is contained within an "autopublish" package that's included with every meteor project by default. If we remove this package, users won't be able to access any data through the Console, but it will also break the application, so we'll need to take a couple of extra steps again.

To remove the "autopublish" package from the project, run the following command:

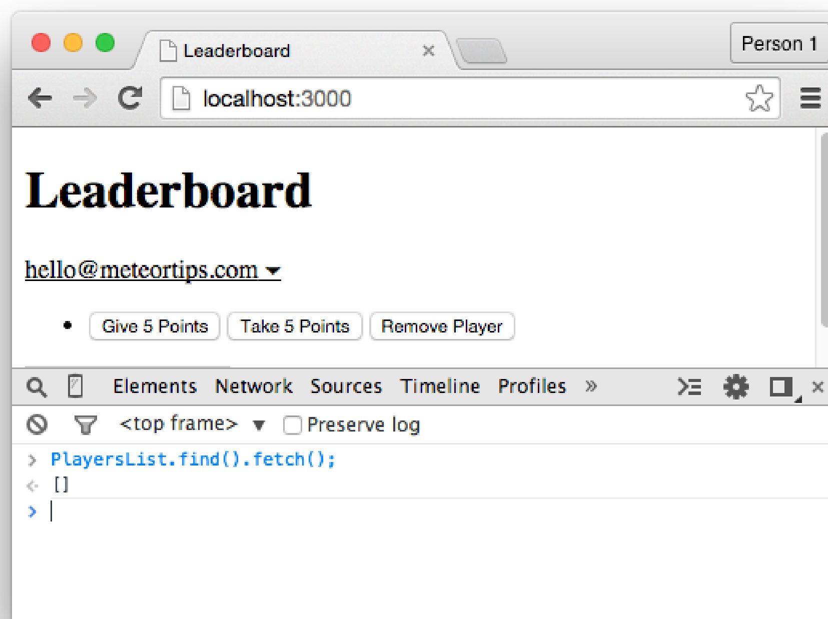
```
meteor remove autopublish
```

If you're logged out of both user accounts when the package is removed, you won't notice anything different, but try the `find` and `fetch` functions:

```
PlayersList.find().fetch();
```

You'll notice that we can no longer navigate through the data inside the collection. The only thing returned is an empty array. It looks like the data has been deleted, but that's not the case. It's just been secured.

The problem is, our data is now *too* secure, because if we login to either of the user accounts, the data is also inaccessible from the interface:



None of the data is available.

To fix this, we'll need to find some middle-ground between the two extremes we've faced — everything being accessible and nothing being accessible. This involves precisely defining what data should be available to our users.

isServer

Throughout this book, we've mostly been writing code inside the `isClient` conditional. This is because we've mostly been writing code that's meant to run inside the browser (such as code that affects the interface). There are, however, plenty of situations where we want code to run on the server.

To demonstrate one of these situations, place the following statement inside the `isServer` conditional that's inside the JavaScript file:

```
console.log(PlayersList.find().fetch());
```

Unsurprisingly, the output appears inside the command line (rather than the Console), but notice that we don't have any trouble retrieving the data from the "PlayersList" collection. Even after removing the "autopublish" package, we have free reign over the data while working directly with the server.

Why?

Well, **code that is executed on the server is inherently trusted**. So while we've stopped users of the application from accessing data on the front-end — on the client-side — we can continue to retrieve the data while on the server.

The usefulness of this detail will soon become clear.

Publications, Part 1

In this section, we're going to *publish* the data that's inside the "PlayersList" collection, and conceptually, you can think of publishing data as transmitting data from the server and into the ether. We're just specifying what data should be available to users. We don't care where that data ends up.

To achieve this, delete the `console.log` statement from the `isServer` conditional and replace it with a `Meteor.publish` function:

```
Meteor.publish();
```

Between the brackets of this function, pass through "thePlayers" as the first argument:

```
Meteor.publish('thePlayers');
```

This argument is a name that we'll reference in a moment.

Then, as the second argument, pass through a function:

```
Meteor.publish('thePlayers', function(){
    // inside the publish function
});
```

It's within this function that we specify what data should be available to users of the application. In this case, we'll return all of the data from the "PlayersList" collection:

```
Meteor.publish('thePlayers', function(){
    return PlayersList.find()
});
```

This code duplicates the functionality of the `autopublish`, meaning it's not exactly what we want, but it's a step in the right direction.

Subscriptions

Because of the `Meteor.publish` function that's executing on the server, we can now *subscribe* to this data from inside the `isClient` conditional, once again making the project's data accessible through the browser and Console.

If you imagine that the `publish` function is transmitting data into the ether, then the `subscribe` function is what we use to "catch" that data.

Inside the `isClient` conditional, write the following:

```
Meteor.subscribe();
```

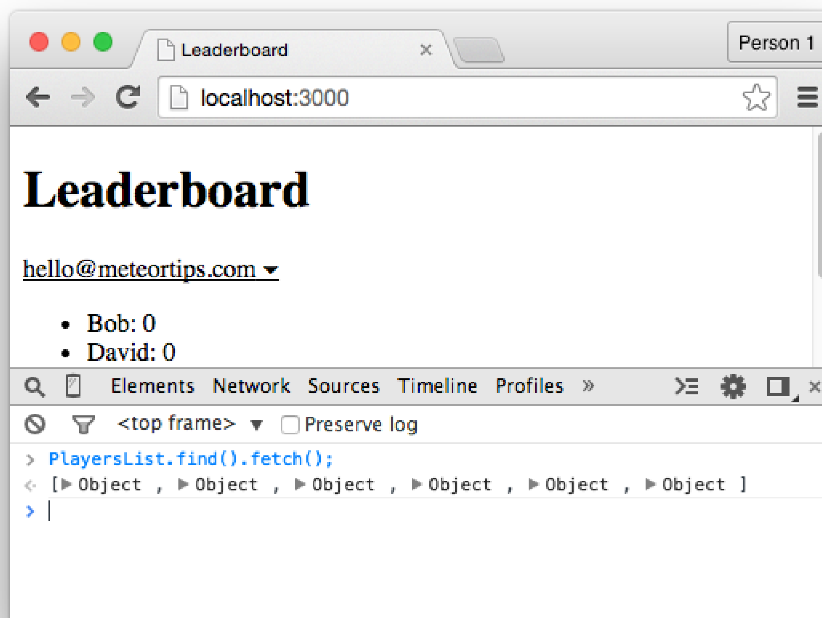
This is the `Meteor.subscribe` function, and the only argument we need to pass through is the name of a `publish` function:

```
Meteor.subscribe('thePlayers');
```

Save the file, then use the `find` and `fetch` function on the "PlayersList" collection:

```
PlayersList.find().fetch();
```

You'll notice that, once again, we have access to all of the data from the project's database, meaning our application is back to its original state. This still isn't what we want, but it's another important step.



Publishing all of the data.

Publications, Part 2

The goal now is to make it so the `Meteor.publish` function only publishes data from the server that belongs to the currently logged-in user.

This means:

1. Logged-in users will only have access to their own data.
2. Logged-out users won't have access to any data.

In the end, the application will be fully functional while being protective of potentially sensitive data.

To achieve this, we'll need to access the unique ID of the currently logged-in user from within the `Meteor.publish` function. While inside this function though, we can't use the `Meteor.userId()` function from before. Instead, we have to use the following statement:

```
this.userId;
```

But while the syntax is different, the end result is the same. This statement returns the unique ID of the currently logged-in user.

Place the statement inside a "currentUserId" variable:

```
Meteor.publish('thePlayers', function(){
    var currentUserId = this.userId;
    return PlayersList.find()
});
```

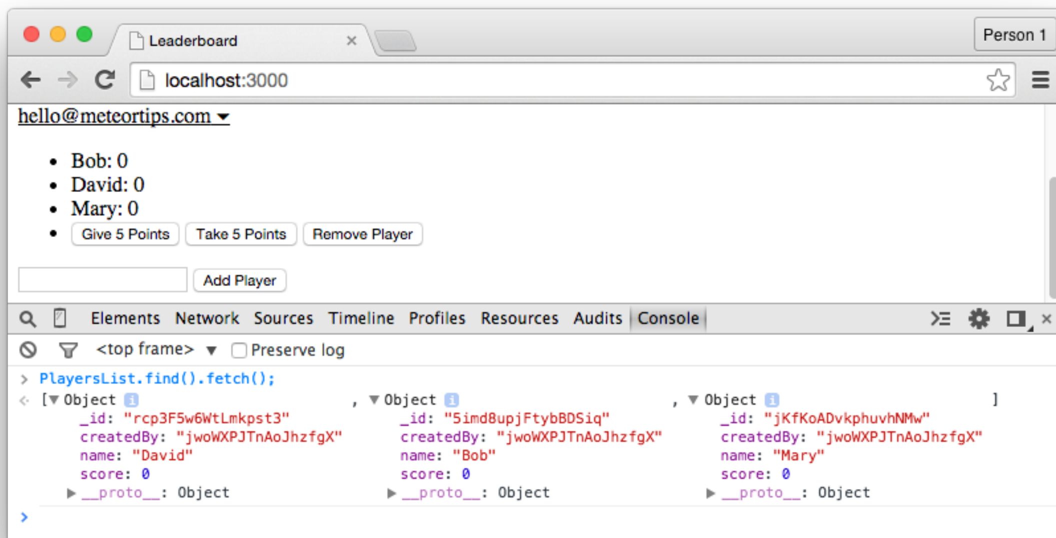
Then change the `find` function so it only retrieves documents where the `createdBy` field is equal to the ID of the currently logged-in user:

```
Meteor.publish('thePlayers', function(){
    var currentUserId = this.userId;
    return PlayersList.find({createdBy: currentUserId})
});
```

Save the file, then use the `find` and `fetch` functions on the "PlayersList" collection:

```
PlayersList.find().fetch();
```

If you're logged-in, you'll only see the data that belongs to the current user's account, and if you're not logged-in, you won't see any data. This is because the `return` statement inside the `Meteor.publish` function can only return documents that contain the unique ID of the current user.



Returning a limited selection of data.

Also know that we can now simplify the `player` function from this:

```
'player': function(){
    var currentUserId = Meteor.userId();
    return PlayersList.find({createdBy: currentUserId},
                            {sort: {score: -1, name: 1}});
}
```

...to this:

```
'player': function(){
    var currentUserId = Meteor.userId();
    return PlayersList.find({}, {sort: {score: -1, name: 1}});
}
```

Why?

Because the `return` statement inside the `player` function can only ever retrieve data that is being published from the server. Therefore, specifying that we want to retrieve the user's data in two places is redundant. We only need to define the returned data within the `Meteor.publish` function.

Summary

In this chapter, we've learned that:

- By default, all the data inside a Meteor project's database is available to all users of that application. This is convenient during development, but it's also a big security hole that need to be fixed before deployment.
- This default functionality is contained within an "autopublish" package. If we remove this package, the project will be more secure, but it'll also break and need to be fixed.
- The `Meteor.publish` function is used on the server-side to define what data should be available to users of the application.
- The `Meteor.subscribe` function is used on the client-side to retrieve the data that's published from the server.
- Inside the publish function, we can't use the `Meteor.userId()` function, but we can retrieve the current user's ID with `this.userId`.

To see the code in its current state, check out the GitHub commit.

Methods

In the previous chapter, we talked about the first of two major security issues that come included with every Meteor project by default. That issue was the ability for users to navigate through all of the data inside the database until we removed the "autopublish" package. Based on these changes we made, users now only have access to the data that "belongs" to them.

To demonstrate the second major security issue, enter the following command into the Console:

```
PlayersList.insert({name: "Fake Player", score: 1000});
```

See what's wrong?

Although we've made it so users can't navigate through all of the data inside the database, users are still able to freely insert data into the database by using the Console. This means a user could:

1. Exploit the application to their own advantage.
2. Fill the database with useless, unwanted data.

Users also have the ability to modify and remove data from the database, meaning by default, they basically have full administrative permissions.

As with the previous security issue, this feature is convenient when we're developing an application, since it makes it easy to create and manage data, but it's a feature we'll need to turn off before deployment.

This functionality is contained within an "insecure" package, and we can remove it from the project with the following command:

```
meteor remove insecure
```

After removing the package, switch back to Chrome and try playing around with the application. You'll notice that:

- We can no longer give points to the players.
- We can no longer take points from the players.
- We can no longer remove players from the list.
- We can no longer add players to the list.

All of the `insert`, `update`, and `remove` features have stopped working — both through the interface and the Console — so the application is a lot more secure as a result, but we will have to fix quite a few things.

Create a Method

Up until this point, all of the `insert`, `update`, and `remove` functions have been inside the `isClient` conditional. This has been the quick and easy approach, but it's also why our application is inherently insecure. We've been placing these sensitive, database-driven functions on the client-side.

The safer approach is to move these functions to the `isServer` conditional, which means:

1. Database code will execute within the trusted environment of the server.
2. Users won't be able to use these functions from inside the Console, since users don't have direct access to the server.

To achieve this, we'll create our first *methods*, and methods are blocks of code that are executed on the server after being triggered from the client. If that sounds weird though, fear not. This is one of those times where following along by writing out the code will help explain a lot.

Inside the `isServer` conditional, write the following:

```
Meteor.methods({
    // methods go here
});
```

This is the block of code we'll use to create our methods. You'll notice that the syntax is similar to how we create both helpers and events.

To demonstrate what a method does, create a "sendLogMessage" method:

```
Meteor.methods({
    'sendLogMessage'
});
```

Then associate this method with a function:

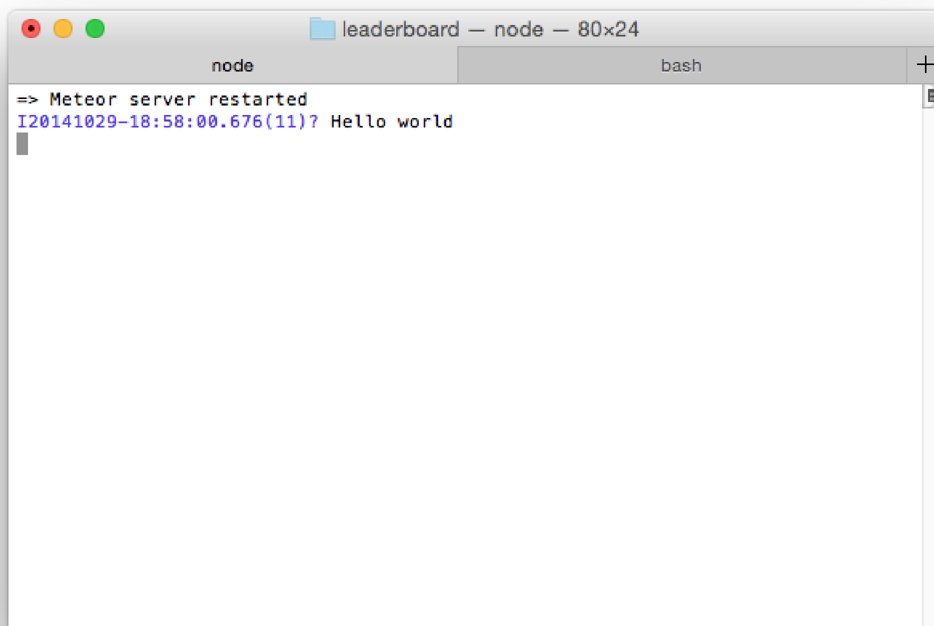
```
Meteor.methods({
    'sendLogMessage': function(){
        console.log("Hello world");
    }
});
```

Next, "call" this method from bottom of the `submit form` event that's attached to the "addPlayer-Form" template:

```
'submit form': function(event){
    event.preventDefault();
    var playerNameVar = event.playerName.value;
    var currentUserId = Meteor.userId();
    PlayersList.insert({
        name: playerNameVar,
        score: 0,
        createdBy: currentUserId
    });
    Meteor.call('sendLogMessage');
}
```

By using this `Meteor.call` statement, and passing through the name of the method we created, we're able to trigger the execution of the method whenever the "Add Player" form is submitted.

Save the file, switch back to Chrome, and submit the "Add Player" form. The core functionality of this form is still broken, but if you switch over to the command line, you'll see that the "Hello world" message appears each time the form is submitted. The submission of the client-side form is triggering the method, but the actual code inside the method is being executed on the server.



Code executed on the server, triggered when we submitted the form.

This basic principle is what we'll using throughout the rest of this chapter.

Inserting Data (Again)

To get the application working again, we'll first move the `insert` function that's inside the `submit form` event from the client and to the server.

This means:

1. The `insert` function will successfully (and securely) run on the server.
2. Users still won't be able to insert data through the Console.

In other words, when the "Add Player" form is submitted, the `insert` function will trigger on the server after being triggered from the client-side.

First, change the name of the "sendLogMessage" method to "insertPlayerData", and get rid of the `console.log` statement:

```
Meteor.methods({
    'insertPlayerData': function(){
        // code goes here
    }
});
```

Inside the method, grab the unique ID of the currently logged-in user:

```
Meteor.methods({
    'insertPlayerData': function(){
        var currentUserId = Meteor.userId();
    }
});
```

Then add a familiar `insert` function beneath this statement:

```
Meteor.methods({
    'insertPlayerData': function(){
        var currentUserId = Meteor.userId();
        PlayersList.insert({
            name: "David",
            score: 0,
            createdBy: currentUserId
        });
    }
});
```

Here, we're passing through a hard-coded value of "David", which isn't exactly what we'll want, but it's good enough for the time being.

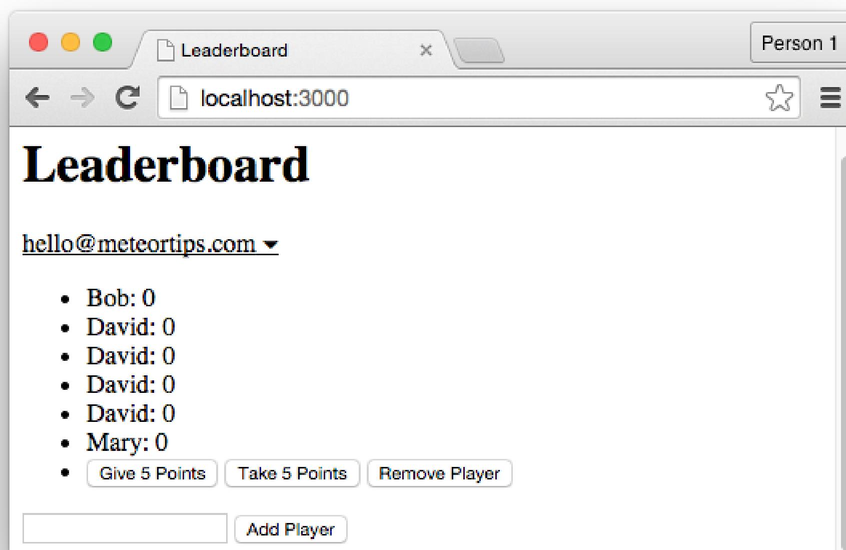
Return to the `submit form` event and remove both the `currentUserId` variable and the `insert` function. The event should then resemble:

```
'submit form': function(event){
    event.preventDefault();
    var playerNameVar = event.target.playerName.value;
    Meteor.call('sendLogMessage');
}
```

But make sure to also pass the correct method name through the `Meteor.call` statement:

```
'submit form': function(event){
    event.preventDefault();
    var playerNameVar = event.target.playerName.value;
    Meteor.call('insertPlayerData');
}
```

Based on these changes, the "Add Player" form will now kind of work. If we submit the form, a player will be added to the "PlayersList" collection. We can only add players named "David", but we'll fix that in the next section.



The `insert` function is sort of working again.

For now, the important part is that, while users can add players to the list by using the form, they're not able to use the `insert` function from inside the Console. This means we're gaining control over how users interact with the database, which is a vital part of keeping an application secure.

Passing Arguments

A problem with the "Add Player" form is that the value of the text field is not being passed into the method. As such, when we submit the form, the name of the player that's created will always be set to "David".

To fix this, pass the "playerNameVar" variable through the `Meteor.call` statement as a second argument:

```
'submit form': function(event){
    event.preventDefault();
    var playerNameVar = event.target.playerName.value;
    Meteor.call('insertPlayerData', playerNameVar);
}
```

Then allow the method to accept this argument by placing "playerNameVar" between the brackets of the method's function:

```
Meteor.methods({
    'insertPlayerData': function(playerNameVar){
        var currentUserId = Meteor.userId();
        PlayersList.insert({
            name: "David",
            score: 0,
            createdBy: currentUserId
        });
    }
});
```

Because of this, we can now reference "playerNameVar" to reference the value that the user enters into the form's text field:

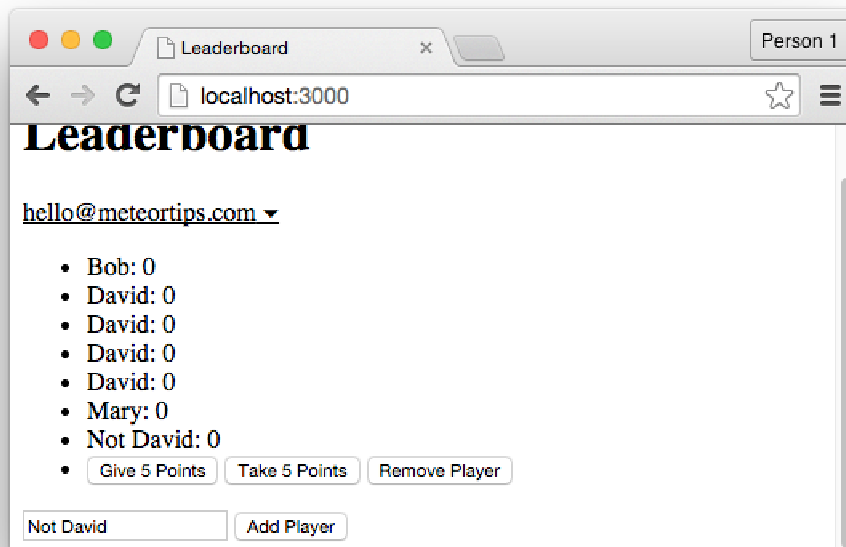
```
Meteor.methods({
    'insertPlayerData': function(playerNameVar){
        var currentUserId = Meteor.userId();
        PlayersList.insert({
            name: playerNameVar,
            score: 0,
            createdBy: currentUserId
        });
    }
});
```

In the end, here's what's happening:

First, when the form is submitted, the "insertPlayerData" method is called, and the value of the form's text field is attached to the call.

Second, the "insertPlayerData" method is executed. This method accepts the value of the "playerNameVar" variable, and the variable is then referenced from inside the method's function.

Third, the `insert` function executes inside the method, and because this code runs on the server, it can run without the "insecure" package. Unlike a moment ago, this function uses the value from the form's text field, rather than the hard-coded value of "David".



Creating players with original names.

The form will once again work as expect, but there'll still be no ways for users to manipulate the data through the Console.

Removing Players (Again)

In the same way we created an "insertPlayerData" variable, we're going to create a "removePlayerData" method that we'll attached to the "Remove Player" button that's inside our interface.

Just like how we create helpers and events, we'll place our methods inside a single block of code, remembering to separate the methods with commas:

```
Meteor.methods({
    'insertPlayerData': function(playerNameVar){
        var currentUserId = Meteor.userId();
        PlayersList.insert({
            name: playerNameVar,
            score: 0,
            createdBy: currentUserId
        });
    },
    'removePlayerData': function(){
        // code goes here
    }
});
```

Then we'll make two changes to the `click .remove` event:

First, get rid of the `remove` function:

```
'click .remove': function(){
    var selectedPlayer = Session.get('selectedPlayer');
}
```

In its place, create another `Meteor.call` statement:

```
'click .remove': function(){
    var selectedPlayer = Session.get('selectedPlayer');
    Meteor.call('removePlayerData');
}
```

Pass through the "selectedPlayer" variable as the second argument:

```
'click .remove': function(){
    var selectedPlayer = Session.get('selectedPlayer');
    Meteor.call('removePlayerData', selectedPlayer);
}
```

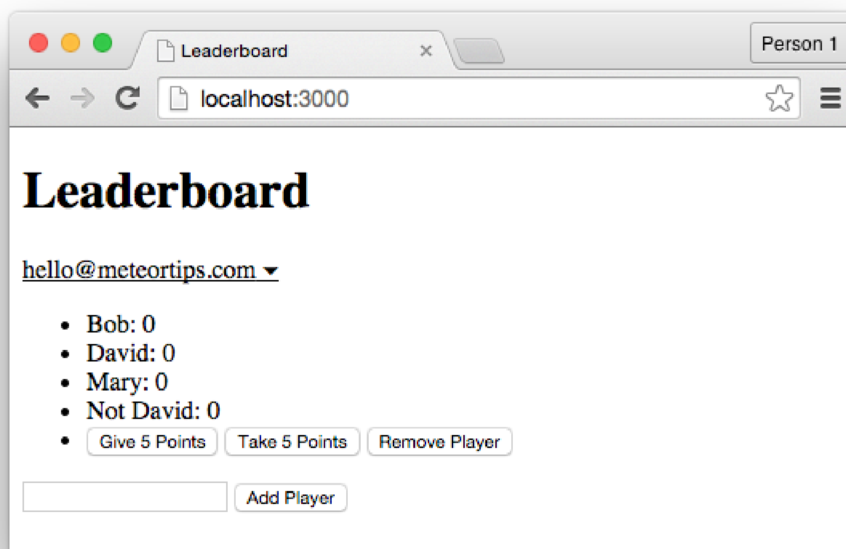
Allow the method to accept this argument:

```
'removePlayerData': function(selectedPlayer){
    // code goes here
}
```

Then recreate the `remove` function from inside the method:

```
'removePlayerData': function(selectedPlayer){
    PlayersList.remove(selectedPlayer);
}
```

The "Remove Player" button will now work as expected, but users still won't have complete administrative access to database-related functions from inside the Console.



Clearing out old data.

There is, however, a lingering problem…

Because the "removePlayerData" method is executed from the client, users are able to execute that same call themselves, from the Console:

```
Meteor.call('removePlayerData', '8sad8a90d8s9ad');
```

So while they don't have access to the full range of what `insert`, `update`, and `remove` functions can do, they can still do some damage. They could, for instance, run this command to remove a player from another user's list.

This isn't a tragically huge security hole, but it's something we should fix for the sake of being prepared for security holes in the future.

The best course of action is to change the "removePlayerData" method from this:

```
'removePlayerData': function(selectedPlayer){
    PlayersList.remove(selectedPlayer);
}
```

…to this:

```
'removePlayerData': function(selectedPlayer){
    var currentUserId = Meteor.userId();
    PlayersList.remove({_id: selectedPlayer, createdBy: currentUserId});
}
```

With this code in place, the method will only allow a player to be removed from the list if that player belongs to the current user.

Note: You might think it unlikely that a user would make such an effort to mess with another user's leaderboard, but when it comes to security, it's best not to underestimate people's ability and eagerness to wreak havoc.

Modifying Scores

Throughout this chapter, we've been using methods for the sake of security, but we can also use methods to reduce the amount of code in our project.

To demonstrate this, we're going to combine the `click .increment` and `click .decrement` event into a single method. This is possible because there's a lot of shared code between these events:

```
'click .increment': function(){
    var selectedPlayer = Session.get('selectedPlayer');
    PlayersList.update(selectedPlayer, {$inc: {score: 5} });
},
'click .decrement': function(){
    var selectedPlayer = Session.get('selectedPlayer');
    PlayersList.update(selectedPlayer, {$inc: {score: -5} });
}
```

The only difference, in fact, is that inside the `click .increment` event, we're passing a value of "5" through the `inc` operator, while inside the `click .decrement` event, we're passing through a value of "-5".

To improve this code, let's first focus on the `click .increment` event.

Inside the event, remove the `update` function and replace it with a `Meteor.call` statement:

```
'click .increment': function(){
    var selectedPlayer = Session.get('selectedPlayer');
    Meteor.call('modifyPlayerScore', selectedPlayer);
}
```

Here, we're calling this "modifyPlayerScore" method, which is a method we'll create in a moment, and we're passing through "selectedPlayer" variable.

Create the "modifyPlayerScore" method inside the `methods` block:

```
'modifyPlayerScore': function(){
    // code goes here
}
```

...and allow this method to accept the value of "selectedPlayer":

```
'modifyPlayerScore': function(selectedPlayer){
    // code goes here
}
```

Then, within this method's function, recreate the `update` function that we deleted a moment ago:

```
'modifyPlayerScore': function(selectedPlayer){
    PlayersList.update(selectedPlayer, {$inc: {score: 5} });
}
```

Based on this code, the "Give 5 Points" button will work as expected. To make the method more flexible though, return to the `click .increment` event and pass a third argument of "5" through the `Meteor.call` statement:

```
'click .increment': function(){
    var selectedPlayer = Session.get('selectedPlayer');
    Meteor.call('modifyPlayerScore', selectedPlayer, 5);
}
```

Allow the method to accept this third argument:

```
'modifyPlayerScore': function(selectedPlayer, scoreValue){
    PlayersList.update(selectedPlayer, {$inc: {score: 5} });
}
```

Then replace the value of "5" that's inside the method with a reference to this newly created "scoreValue" property:

```
'modifyPlayerScore': function(selectedPlayer, scoreValue){
    PlayersList.update(selectedPlayer, {$inc: {score: scoreValue} });
}
```

Because of this change, the method is now flexible enough that we can use it for both the "Give 5 Points" button and the "Take 5 Points" button.

Here's how…

First, delete the `update` function from inside the `click .decrement` event:

```
'click .decrement': function(){
    var selectedPlayer = Session.get('selectedPlayer');
}
```

Second, place a `Meteor.call` statement inside this event:

```
'click .decrement': function(){
    var selectedPlayer = Session.get('selectedPlayer');
    Meteor.call('modifyPlayerScore');
}
```

Third, pass through the value of the "selectedPlayer" variable:

```
'click .decrement': function(){
    var selectedPlayer = Session.get('selectedPlayer');
    Meteor.call('modifyPlayerScore', selectedPlayer);
}
```

Fourth, pass through a value of "-5" instead of just "5":

```
'click .decrement': function(){
    var selectedPlayer = Session.get('selectedPlayer');
    Meteor.call('modifyPlayerScore', selectedPlayer, -5);
}
```

See what we've done?

We've made it so the utility of "modifyPlayerScore" method depends on what we pass through as the third argument:

1. If we pass through a value of "5", the `update` function will increment the value of the "score" field.
2. If we pass through a value of "-5", the `update` function will decrement the value of the "score" field.

Therefore, the method allows for code that is both flexible and secure.

We do have the same security flaw from the previous section though, in that a user could enter the following command into the Console to modify the score of any player within the database:

```
Meteor.call('modifyPlayerScore', '8sad8a90d8s9ad', 100);
```

But the solution is also the same.

Just change the contents of the method from this:

```
'modifyPlayerScore': function(selectedPlayer, scoreValue){
    PlayersList.update(selectedPlayer, {$inc: {score: scoreValue} });
}
```

...to this:

```
'modifyPlayerScore': function(selectedPlayer, scoreValue){
    var currentUserId = Meteor.userId();
    PlayersList.update( {_id: selectedPlayer, createdBy: currentUserId},
                        {$inc: {score: scoreValue} });
}
```

Again, we're retrieving the unique ID of the currently logged-in user, and within the `update` function, making sure the player being updated "belongs" to that user. If the logged-in user doesn't "own" the player, that player's document won't be found by the `update` function.

Summary

In this chapter, we've learned that:

- By default, it's possible for users to insert, update, and remove data from a collection using the JavaScript Console. This is convenient for development, but a security risk for a deployed web application.
- To fix this security risk, we must move our database-related code from the client-side, to the trusted environment of the server. Here, users won't have any direct access to (or control over) the database.
- This security risk is contained within an "insecure" package. By removing this package, the application will become a lot more secure, but it'll also break, since none of the database-related features will work.
- By using methods, we're able to write code that runs on the server after first being triggered from the client. This is how we can fix the project's broken features.
- To create methods, we can define them within a `methods` block, and then trigger them from elsewhere in the code using `Meteor.call` statements.
- We can pass data from the `Meteor.call` statements and into the method, allowing us to use submitted data from a form inside the method.
- Users can execute client-side `Meteor.call` statements through the Console, so we need to be mindful of what those statements allow the users to do.
- Methods are not only useful for security. They're also useful for combining similar chunks of functionality into a small and repeatable fragment of code.

To gain a deeper understanding of Meteor:

- In a new project, start by removing the "insecure" package, and from the beginning, place all of the database-related code inside methods.

To see the code in its current state, check out the GitHub commit.

Structure

Throughout this book, we've placed all of the project's code within just three files:

- leaderboard.html
- leaderboard.js
- leaderboard.css

This has allowed us to focus on the fundamentals of building software with Meteor without worrying about how the code is organized, and the project is simple enough that we need don't need other files, but:

1. When building larger applications, it makes sense to spread the project's code across a number of files.
2. Meteor has a number of convention for structuring a project's files.

Before we continue though, there's two things to consider:

First, **Meteor doesn't have hard and fast rules about how to structure a project**. There are certain guidelines that Meteor encourages, but nothing strict that you *must* do. Your preferences are ultimately in control.

Second, **people are still figuring out the best practices when working with Meteor**. As such, there's no reason to form dogmatic views about how projects "should" be structured. Allow yourself to experiment.

This chapter is also different in that we won't be re-structuring the Leaderboard project step-by-step. Instead, we'll talk about the principles of structure, and it's your job to put those principles into practice.

Don't worry though.

Based on everything we've covered so far, getting a handle on how to structure a Meteor application is going to be a piece of cake.

Your Project, Your Choice

Like I said, Meteor doesn't have hard and fast rules about how to structure a project. It doesn't care how your files and folders are organized, and if you wanted to create a large project with just three files, you could.

With this flexibility though, comes the paradox of choice:

If you're not constrained to precise rules, how should you structure your project?

If you're a beginning developer, structure your projects as simply as possible for as long as you can. This means spreading your code across just three files — the HTML, JavaScript, and CSS files — until those files become too bloated to easily manage. It's rare for a real-world application to be contained within such a small structure, but if you're just getting started with Meteor, it's not productive to obsess over the "perfect" structure.

Still read through this chapter — there's some important conventions to be aware of — but don't feel the need to implement every detail right away. Best practices are easier to learn once the fundamentals have been absorbed, so as long as you're a beginner, you're allowed to be scrappy.

If you're *not* a beginning developer — meaning, if you have web development experience, and haven't had any trouble following along with this book — then you'll have an easy time implementing the conventions we're about to discuss.

Thin Files, Fat Files

When creating a Meteor application, the project's files can be as "thin" or as "fat" as we want them to be. This means:

1. We can spread the code across many files.
2. We can pack a lot (or a little) code into each file.

For example, let's consider the "leaderboard.html" file. It's not exactly fat, but it does contain three components that, although connected, don't need to be contained within single file:

- The HTML structure of the page (`head` tags, `body` tags, etc).
- The "leaderboard" template.
- The "addPlayerForm" template.

If this project were to grow larger, then, it would make sense to split these components into three separate files. You might, for instance, want to:

1. Leave the HTML structure in the "leaderboard.html" file.
2. Move the "leaderboard" template to a "leaderboardList.html" file.
3. Move the "addPlayerForm" template to a "addPlayerForm.html" file.

As a result, it'd be easier to navigate through the project's files since the name of each file is suggestive of what that file contains.

To be clear though:

1. There's no extra step involved. Just place the code wherever you want in the project and Meteor will know how to put the pieces together.
2. The file names are arbitrary. There are some "special" file names to be aware of, but generally, it's fine to name the files however you like.

You're also free to place files within folders and sub-folders (and deeper structures, too), but there are certain conventions that encourage certain ways of naming these folders.

Folder Conventions, Part 1

A lot of the code inside the “leaderboard.js” file is within the `isClient` conditional. If we spread this code across multiple files though, it’d be inelegant to reuse this conditional over and over again.

Luckily, Meteor has a convention where **any code that’s placed within a folder named “client” will only run on the client**.

To demonstrate this:

1. Create a folder named “client” within your project’s folder.
2. Create a JavaScript file within this folder.
3. Cut and paste all of the client-side code from the “leaderboard.js” file into the new file, but without the `isClient` conditional.

After saving the file, the application will continue to work as expected.

Because of this convention, it’s best to place templates, events, helpers, and the `Meteor.subscribe` function inside a “client” folder.

Conversely, Meteor has a convention where **any code that’s placed within a folder named “server” will only run on the server**. This is where we’d place the project’s methods, and the `Meteor.publish` statement.

After shuffling the code around inside the Leaderboard application, the only that left inside the original “leaderboard.js” file will be the statement that creates the “PlayersList” collection. We want this code to run on both the client and the server, so make sure to leave this statement outside the “client” and “server” folders. A common convention is to place this code inside a “collection.js” file, but that file name has no special meaning.

Folder Conventions, Part 2

When getting started with Meteor, most a project's files will probably end up in the "client" or "server" folders. There are, however, some other folder names that can be used for different purposes:

- Files stored in a "private" folder will only be accessible to code that's executed on the server. These files will never be accessible to users.
- Files stored in a "public" folder are served to visitors. These are files like images, favicons, and the "robots.txt" file.
- Files stored in a "lib" folder are loaded before other files.

But if all of this seems like too much remember, fear not. These details are worth knowing for future reference, but it'll be a while before you need to put them into practice. For the moment, it's fine to focus on the basics.

Boilerplate Structures

An effective way to learn how to structure a Meteor project is to learn from other developers who have likely encountered many of the problems that you'll one day encounter yourself.

Consider, for instance, the "iron" tool from Chris Mathers of Evented Mind:

> A command line scaffolding tool for Meteor applications. It automatically creates project structure, files and boilerplate code.

This tool can be used to quickly create a project with the following structure:

```
my-app/
 .iron/
   config.json
 bin/
 build/
 config/
   development/
     env.sh
     settings.json
 app/
   client/
     collections/
     lib/
     stylesheets/
     templates/
     head.html
   lib/
     collections/
     controllers/
     methods.js
     routes.js
   packages/
   private/
   public/
   server/
     collections/
     controllers/
     lib/
     methods.js
     publish.js
     bootstrap.js
```

You'll notice that everything has its place. There are folders for collections and stylesheets and templates, and every other component of a project.

Is this the *best* way to structure a Meteor project?

For some, it might be.

I'd say it's a little complicated for someone who's just getting started with Meteor, but when you're ready to move onto a bigger projects, it might be exactly what you need to manage a larger base of code.

At this stage, the point isn't to make any hard and fast decisions about your preferences. But it does help to be aware of what's at least possible.

Other boilerplates worth checking out and learning from include meteor-boilerplate and Void.

Summary

In this chapter, we've learned that:

- Meteor doesn't enforce a precise file structure upon projects. There are simply conventions that we're encouraged to follow for our own benefit.
- By naming certain folders in certain ways, we're able to avoid writing some application logic based on how Meteor handles those folders.

To gain a deeper understanding of Meteor:

- Search through other people's Meteor projects on GitHub and see how real-world applications are being structured.
- Imagine you're creating a blogging application like WordPress. How would you structure that application? Plan it out on a piece of paper.

To see the code in its current state, check out the GitHub commit.

Deployment

Throughout this book, we've made a lot of progress. We've decided on a project to build, created all of the fundamental features, added a few extra niceties, and even talked about a couple of common security issues.

As such, we're ready to *deploy* the application to the web.

This is where we can share our creation with the world and then wait for hordes of strangers to marvel at our genius. But deployment isn't simply a matter of uploading files to a web server. There's a bit more involved.

Here's a broad overview of the deployment process:

1. Create a server on a site like DigitalOcean.
2. Install the required software on that server (Node, MongoDB, etc).
3. Configure all of that installed software.
4. Upload the project's files to the server.
5. Cross your fingers that nothing breaks.

Sounds complicated, right?

Don't worry though, I've got you covered.

In the first two editions of this book, I explained the absolute basics of deploying Meteor applications to the web but have since put together a much more comprehensive guide, separate from this book.

You can read it online for free:

meteortips.com/deployment-tutorial

If you're looking to deploy something as quickly as possible, you'll only need to read the first couple of chapters, but when you're ready to launch something more interesting than a thrown-together prototype, the book will gently guide you through every step of the process,

As with everything else that I publish online, you can also expect to see plenty of updates as deployment practices change and evolve.

That said, if you're not looking to deploy your application to the world just yet, that's fine. There's nothing about deployment that you need to know at this point. The process is fairly distinct from the development process and, until you've developed a couple of applications, your time is better spent in a text editor, writing code and making things.

Conclusion

Congratulations. You've reached the final page of this book. Like I said in the first chapter though, this book isn't a tome. There is a lot more to learn about building cool things with the Meteor framework.

Here's what I'd suggest:

1. If you haven't already, actually build the Leaderboard application that we've been discussing in this book. There is no better way to learn how to code than by writing out every line, step-by-step.
2. Read the official documentation for Meteor. You might not understand every little bit of it, but it does provide insight into how and why certain features work the way they do.
3. Follow the "How To Learn Meteor Properly" road-map. It's a thorough curriculum for becoming a well-rounded Meteor developer (and it just so happens to recommend this book).

And, if you haven't already, visit meteortips.com and sign up for the email newsletter. That's the best way to here about whatever else I'm working on that will help you make bigger, faster progress with Meteor.

That's it for now though.

Good luck, and talk soon.

â€" David Turnbull

P.S. If you liked this book, feel free to leave a review on Amazon.com. Your support allows me to dedicate more time to working on new material.