MANNING

MEAP

Meteor IN ACTION

Stephan Hochhaus
Manuel Schoebel

MEAP Edition
Manning Early Access Program
Meteor in Action
Version 6

Copyright 2014 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Welcome

Hello, and welcome to our book! First of all we would like to thank you for purchasing *Meteor in Action* even before it is fully written. We are super excited to show you what we have been working on for the past months. We are also glad to have you on board for the next months during which we will gradually keep adding to the contents until we provide a complete introduction to the Meteor platform.

Our main goal with *Meteor in Action* is to give you a solid understanding of how Meteor works and how to use each component in order to achieve its often-quoted magic. Even though Meteor is very well suited for beginners in the field of programming we assume that you have already worked on web applications before and you know the basics of JavaScript. With that knowledge you will be able to follow along all topics and examples throughout the book.

Meteor itself is a relatively young project but already it has a very active community that not only provides support for working with the platform but also extends its functionality with routing, testing frameworks, and other tools. We are very proud to be part of such a community and we want to not only tell you about how to work with the core functionality of Meteor but also how to integrate the power the community to leverage its full potential.

We have thought long and hard whether we should put all the content we want to teach you in the context of building a single application. We did not do it because we do not think a single application can be used to cover all the topics necessary. Additionally, it makes it much harder to look up something on a specific topic if it is deeply embedded into a different context. As such all chapters are intended to serve both as a sound introduction as well as a place of reference to come back to when building your own applications.

We start in Chapter 1 with an introduction to the characteristics of modern web applications, the main concepts behind the Meteor platform, and take a closer look at its strengths and weaknesses. In Chapter 2 we start coding with a simple application that shows you the most important principles in action. The following chapters are going to teach you all about using templates, reactive data, and how to combine templates and data.

As you're reading, I hope you'll take advantage of the Author Online forum. We'll be reading your comments and responding, and your feedback is helpful in the development process. Let us know if there is anything about the book structure, content, or code examples that you think could be improved.

—Manuel and Stephan

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

brief contents

1 A better way to build apps

2 My fridge! A first application

3 Working with Templates

4 Working with data

5 Fully reactive editing

6 Users, Authentications, and Permissions

7 Exchanging Data

8 Routing

9 The Packages System

10 Advanced Reactivity

11 Isobuild

12 Going into Production

APPENDIXES:

A Installing Meteor

B The anatomy of MongoDB

C Setting up nginx

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

1

A better way to build apps

This chapter covers

- The story behind Meteor
- The Meteor stack
- Fullstack JavaScript, Reactivity, and Distributed Platforms
- Core components of the Meteor platform
- The pros and cons of using Meteor
- The anatomy of Meteor applications
- Dissecting a standard application

Meteors have a reputation of changing life as we know it. They are capable of making Dinosaurs extinct or forcing Bruce Willis to sacrifice his life for mankind. This book is about a Meteor that impacts web development, but it does not threaten to destroy anything. On the contrary, it promises to offer a better way to build applications. Meteor takes several existing tools and libraries, combines them with new thoughts and additional libraries, standards, and services, and bundles them to provide an entire ecosystem for developing web and mobile applications that are a delight to use.

As a developer you know that once you open the source view of your browser all web applications are just a combination of HTML, CSS, and JavaScript. Giants like Google, Twitter, or Facebook achieve impressive results that look and feel like desktop applications rather than web sites. The smoothness of Google Maps and the directness of Facebook's Messenger led to users having much higher expectations towards all sites on the internet. Meteor allows you to meet these high expectations because it provides all the infrastructure functionality like data subscriptions and user handling, allowing you to focus on implementing business functionality.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

This chapter will tell you how Meteor makes your life as a developer easier. After a short look at why it was created we will focus on what it consists of and how you can use it to build your own applications that may only take a fraction of the time

1.1 Introducing Meteor

When you look at the state of web development in recent years there is a clear trend visible. First, applications become more powerful, often indistinguishable from desktop applications. Frankly, users do not care what the technology is that works behind the scenes, they simply expect a great user experience. This includes instant feedback on clicks, realtime interaction with other users, and integration with other services. The second trend is that the number of languages, libraries, tools, and workflows is increasing so quickly, it becomes impossible for developers to keep up with all trends. As a result, we can summarize the current state of web development:

1. Users expect more convenience from applications.
2. Developers expect to worry less about making different libraries work well together or writing plumbing code.

1.1.1 The story behind Meteor

When Geoff Schmidt, Matt DeBergalis, and Nick Martin got accepted into Y Combinator, they planned to build a travel recommendation site. However, when they talked to fellow start-up companies they realized how much they struggled with the same challenges they had already solved when they worked on Asana, an online-platform for cooperative project- and task-management. Instead of working on a travel site they changed their plans and decided to come up with an open-sourced platform to provide a sound foundation for web applications that are just as smooth to use as desktop applications.

On the 1st of December 2011 the Meteor Development Group (or short: MDG), announced the very first preview release of Skybreak[1], which soon after got renamed to Meteor. Only eight months later the project had arranged for $11.2 million in funding from big names in the industry such as Andreessen Horowitz, Matrix Partners, Peter Levine (former CEO of XenSource), Dustin Moskovitz (co-founder of Facebook), and Rob Johnson (founder of SpringSource). The Meteor github repository has stayed in the top 20 most popular repositories since then and rose to become the 11th most popular repository on github just days after its 1.0 release, having more stars than the *linux kernel*, the Mac OS X package manager *homebrew*, and *backbone.js*.

Why did Meteor create such interest with developers? Because it takes away the need to create low-level infrastructure like data synchronization or build pipelines to minimize and compile code, but rather lets developers focus on actual business functionality. With over $11

[1] https://www.meteor.com/blog/2011/12/01/first-preview

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

million also investors find Meteor very appealing. Similar to Xen, the free hypervisor for server virtualization or JBoss, the Java application server, the Meteor Development Group will eventually offer additional tools targeted at larger enterprises.

The Meteor Development Group divides their projects into four areas:

- **Tools** such as a command line interface (CLI) – a hybrid between a build-tool like `make` and a package manager such as `npm`, that takes care of entire build flows that prepare an application to deploy for the web or mobile devices
- A collection of software **libraries** – a suite of core packages provide functionality that can also be extended with custom packages or node.js modules installed via `npm`
- **Standards** like the Distributed Data Protocol (DDP), a websocket-based data protocol
- **Services** such as an official package server or a build farm

All of Meteor's projects are accessible using a unified API, so that developers do not need to know which components make up the entire Meteor stack.

1.1.2 The Meteor Stack

Simply put, Meteor is an open-source platform for creating rich web applications entirely in JavaScript by providing all required tools under one umbrella. As such it is more than a combination of server processes and libraries. Some like to refer to it as an entire ecosystem rather than a framework. But even if it is goes beyond what other web frameworks offer, at its core it still relies on a stack to run applications.

Figure 1-1: The Meteor stack runs applications powered by Smart Packages on top of Node.js and MondoDB

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

The Meteor stack (see Figure 1-1) is a member of the MEAN[2] family, which means it is powered by **Node.js** on the server side. Node.js is an event-driven, highly scalable runtime for JavaScript on the server. It serves the same purpose as an Apache webserver in the LAMP (Linux, Apache, MySQL, PHP) stack.

All data is typically stored inside a **MongoDB**, a document-oriented NoSQL database. There are plans for Meteor to support other (SQL-based) database systems, but currently the only suggested DB is Mongo. It provides a JavaScript-API that gives access to all stored content in form of documents or objects. The same language used inside the browser can be used to access data, which Meteor takes advantage to implement true fullstack development.

All required software and libraries required to create web applications from scratch are bundled in the shape of Smart Packages, so developers can get started right away. These packages include a user interface library (`Blaze`), managing user accounts (`accounts`), and keeping all clients reactively updated in realtime (`Tracker`).

The **Meteor CLI tool** allows developers to quickly set up an entire development environment. There is no need to know how to install or configure any server software, Meteor takes care of the infrastructure aspect entirely. It is also both a build tool, comparable to `make` or `grunt`, and a package manager, such as `apt` or `npm`. For example it can compile *LESS* or *CoffeeScript* on the fly, without first setting up workflow or add authentication via Facebook oAuth with a single command. Finally the CLI tool bundles an application to run on different client platforms, inside a web browser or as native mobile apps.

All parts of the stack integrate seamlessly, all core packages are designed and tested to work well together. On the other hand, it is entirely possible to switch out parts of the stack for others, should the need arise. Instead of using Meteor in full you could decide to only use the server components and use e.g. Angular.js on the client side, or use a Java-backend that uses Meteor on the front-end to provide realtime updates to all clients.

1.1.3 Isomorphic Frameworks – Fullstack JavaScript

Meteor runs on top of Node.js and moves the application logic to the browser, which is often referred to as *Single Page Applications*. The same language is used across the entire stack, which makes Meteor an isomorphic platform. As a result the same JavaScript code can be used on the server, the client, and even in the database.

While many frameworks use the same language on both client and server, most of the times they cannot share code between the two instances because the frameworks are not tightly integrated, for example they use Angular on the frontend and Express.js on the backend. Meteor is truly fullstack because it uses a simple and unified API that exposes all core functionality and can be used on the server, in the browser, and even to access the

[2] The MEAN stack refers to all applications built on top of MongoDB, Node.js, Angular, and Express.js. There are several variations of the MEAN stack, such as MEEN – MongoDB, Ember.js, Express, and Node.js. Sometimes the term is used loosely to indicate any infrastructure running on Node.js in combination with a NoSQL database.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

database. To get started you do not have to learn multiple frameworks and results in much better re-usability of the code than only using the same language.

To access the database from the browser, Meteor includes mini databases. They simulate the exact same API of a database. Inside the browser *Minimongo* allows developers to use the same commands as they would in a MongoDB console.

All Meteor applications run on top of Node.js, a server that interprets application code written in JavaScript. In contrast to many other application servers it only uses a single thread. In multi-threaded environments a single thread that writes to disk may block all other threads and put all further client requests on hold until a write operation finishes. Node.js on the other hand is able to queue all write requests and continue taking and processing requests, effectively avoiding race conditions (i.e. two operations try to update the same data simultaneously). The application code runs in sequence from top to bottom, or synchronously.

Long running operations such as I/O to disk or database may be split off from the synchronous sequence. They will be processed in an asynchronous way. Node.js does not wait until these finish, but it attaches a callback and revisits the results of an operation once it finished, meanwhile working on the next requests in line. To better understand synchronous and asynchronous events let's consider a familiar programming scenario: Heating up a frozen pizza.

Figure 1-2 details all steps required to prepare food from the freezer. Each step is an event, albeit a pretty small one in our lives. Every event that requires our attention takes place in a synchronous stream of events: we take the pizza from the freezer, unwrap it, pre-heat the oven, put the pizza in, and set an alarm clock. That is the point when we actually branch off a sub-process. At its core, cooking the pizza in the oven is a long-running I/O process. We set the alarm clock to be notified when it is done so we may attend to more important matters, like learning Meteor. When the alarm clock goes off, it calls our attention and puts the result of the sub-process back in our synchronous flow. We can take the pizza out and move on.

Figure 1-2: Synchronous and asynchronous events when heating up a pizza

As you can see from this example cooking the pizza does not block your flow. However, if your colleague also wants a pizza and there is only room for one inside the oven he will need

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

to queue his request – this effectively blocks all others in the office from heating up their pizza.

In Node.js the synchronous flow takes place as long as the server runs. It is called the event loop. Figure 1-3 shows how the event loop deals with processing user requests. It takes one event at a time from a queue. The associated code is executed and when it finishes, the next event is pulled into the loop. However, some events may be offloaded to a thread pool, for example operations writing to disk or database. Once the write operation is finished, a callback function will be executed that returns the result of the operation back into the event loop.

Figure 1-3: The Node.js event loop

Typically, developers need to know how to write code that takes full advantage of the event loop and what functions run synchronously and which asynchronously. The more asynchronous functionality is used, the more callbacks are involved and things can become quite messy.

Fortunately, Meteor leverages the full power of the event loop, but it makes it easy by not having to worry so much about writing asynchronous code. It uses a concept called *Fibers* behind the scenes. Fibers provide an abstraction layer for the Event Loop that executes asynchronous functions (tasks) in sequence. It removes the need for explicit callbacks so that a familiar synchronous style may be used.

1.1.4 Processing in the browser – Running on distributed platforms

When using a backend running a Java, PHP, or Rails application, processing takes place far away from the user. Clients request data by calling a URI. In response the application fetches data from a database, performs some processing to create HTML and sends the results to a

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

client. The more clients request the same information, the more clever caching mechanisms can be used by the server. News sites work particularly well with this paradigm.

In a scenario where each user has the means to create highly individualized views a single processing instance can quickly become a bottle neck. Consider Facebook as an example: No two people will ever see the exact same wall, it needs to be computed for each user individually. That puts a lot of stress on the servers while clients idle most of the time, waiting for a response.

When the processing power of clients was relatively limited this made perfect sense, but these days a single iPhone already has more computing power than most super computers in the early days of the web. Meteor takes advantage of that power and delegates most of the processing to the clients. Smart front-ends request data from the server and assemble the DOM only in the browser or mobile device (see Figure 1-4).

This client-centric approach brings two significant advantages:

1. Less data needs to be transferred between server and client which essentially translates into quicker response times
2. Processing is less likely to be blocked by other users due to long-running requests, as most of the work is done on each individual client

Figure 1-4: The difference between traditional and modern rich web applications

Traditional client-server architectures are based on stateless connections. Clients request data once, the server responds and the connection is closed again. Updates from other clients may happen, but unless users explicitly make a server request again, they will not see the updates but an historic snapshot of the site. There is no feedback channel from the server to the client to push out updated content.

Imagine you open your local movie theatre's site and see there are only two seats left to the new Joss Whedon movie premiere. While you still debate whether you should go, someone else buys these tickets. Your browser keeps telling you until you decide to click again, only to find out that the tickets are gone. Bummer.

Moving the processing from a single server to multiple clients involves moving into the direction of distributed computing platforms. In such distributed environments data needs to

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

be sent in both directions. In Meteor, the browser is a smart client. Connections are not stateless anymore, the server may send data to the client whenever there are updates to subscribed content. Figure 1-5 shows the different architectures. To allow bi-directional communication between server and client Meteor uses WebSockets. A standardized protocol named *Distributed Data Protocol* (DDP) is used to exchange messages. DDP is simple to use and can be used with many other programming languages like PHP or Java as well.

Figure 1-5: Traditional client-server architectures compared to distributed application platforms

As a consequence of moving applications to the browser, all clients essentially become nodes of an application cluster. This introduces new challenges already familiar from distributed server farms, most importantly synchronizing data between all nodes. Meteor takes care of this by means of its reactive nature.

1.1.5 Reactive Programming

Applications created by traditional programming paradigms are much like a Golem[3] you sent off with a plan. No matter what happens, the Golem will keep walking and follows directions. As its creator you must be diligent about each and every step you command them. Very similar to an application where you must define what happens if the user selects a value from a dropdown list, and what to do if the associated entry has been deleted by another user

[3] A Golem is a mythical creature. Usually made from clay it is magically brought to life and carries out its masters wishes to the letter. Terry Pratchett's „Feet of Clay" is a good first introduction if you are a fan of the fantastic.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

meanwhile. In other words, traditional programming hardly reacts to the world, but follows orders given to it in code.

The real world happens to be slightly different. Things happen and the more complex usage scenarios get, the harder it is to foresee in which sequence events will occur. Especially on the web.

In a desktop environment reactivity is the norm. When you use an Excel spreadsheet and change the value in one cell, all other values depending on it will automatically recalculate. Even charts will be adjusted without the need to click on *refresh*. An event, e.g. changing a cell, triggers reactions in related parts of the sheet. All cells are reactive.

To illustrate how reactivity differs from procedural programming let us look at a simple example. We have two variables: a and b. We will store the result of adding a and b in a variable called c using the procedural way of doing things. With actual values it looks like this:

```
a = 2;
b = 5;
c = a + b;
```

The value of c is now 7. What happens if we change the value of a to 5? c will not change unless we explicitly call the addition code again. Hence a developer needs to include a checking method to observe whether a or b have changed. In a reactive approach the value of c will automatically be set to 10 because the underlying engine is taking care of observing change. There is no need to periodically check that neither a nor b have changed or even initiating a recalculation. The focus is on what the system should do and not how to do it.

In a web environment achieving the Excel-effect can be achieved in various ways. Using poll and diff you could check for changes every two seconds. In scenarios where a lot of users are involved and little change happens this puts a lot of stress on all components involved and is extremely ineffective. Increasing the polling interval makes the UI appear slow and sluggish. Alternatively, you can monitor all possible events and define actions writing a lot of code to mimic the desktop behavior. This option becomes a maintenance nightmare when you need to update various elements in the DOM, even if each event fires only a handful of update operations. A reactive environment offers a third option that enables low-latency UIs with concise, maintainable code.

Reactive systems need to react to events, load, failure, and users[4]. In order to do so they must be non-blocking and asynchronous. Remember when we talked about fullstack JavaScript? Then you will notice that reactivity and JavaScript are a natural fit. Also we discussed that Meteor applications run distributed, not a single server is responsible for creating a user's view. Load can still be scaled across multiple servers, but it also scales across each client. Should one of these clients ever fail, it does not bring down the entire application.

[4] The reactive manifesto defines how reactive systems should be designed and behave in production environments. http://www.reactivemanifesto.org/

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

While you can still build a less-than-optimal system by not taking into account the principles of reactive systems, reactivity is built into the core of Meteor. You don't have to worry about learning a new programming style, you can continue to use the same synchronous style you are used to. In many cases Meteor automatically hooks up reactivity without you even noticing it.

All components, from the database to the client UI, are reactive. This means all changes to data are synchronized between clients in realtime. There is no need to write any Ajax routines or code to push updates to users because this functionality is directly built into Meteor. Also, it effectively removes the need to write most of the glue code when integrating different components, thereby shortening development times significantly.

Reactive Programming certainly isn't the best approach for every scenario, but it fits perfectly into the way web applications work because in 90% of all cases we need to capture events and perform actions on it. Beyond the user experience it can help improve quality and transparency, reduce programming time and decrease maintenance.

1.2 A look under the hood

Once deployed on a server, Meteor applications can hardly be told apart from other Node.js-based projects. The platform's real strength comes to light when you look closely at how Meteor supplements the development process. A command line tool and a collection of packages enable developers to quickly achieve results and focus on adding functionality to an application. Infrastructure concerns such as data exchange between database and browser or integrating user authentication via external oAuth-providers are taken care of by adding packages.

Figure 1-6: Applications consist of business logic and various packages which are compiled for the target platform using Isobuild

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Figure 1-6 shows the anatomy of Meteor applications. Developers define business logic that consists of code, templates and styles, and assets such as image files. Meteor can leverage the power of external packages from the Node.js ecosystem via npm (node package manager) and PhoneGap for mobile devices. Additionally it defines its own package format called *isopacks*.

Isopacks are designed to work in both server and client environments and may contain also templates and images. They may even extend isobuild, the build process that outputs deployable code for all targeted platforms. Isobuild and isopacks are Meteor's core ingredients.

1.2.1 Core Projects

Meteor ships with a number of packages that provide commonly used functionality for web-based applications. A command line tool allows you to create a new project and add or remove packages with a single command. New projects contain all core packages already.

BLAZE

Meteor's user interface library is called Blaze. It serves the same purpose as other front-end libraries like Angular, Ember, or React. Using a simple templating language inspired by Handlebars it is much simpler than the previous and it is tightly integrated with the core reactivity of Tracker. This integrated makes it equally powerful while keeping templates free from all but basic logic.

The official documentation describes Blaze as a "reactive jQuery", a powerful library to update the DOM. However, it does not follow the same imperative style jQuery uses ("find element `#user-list` and add a new `li` node!"), but a declarative approach ("render all usernames from the db in this list using templates `users`!"). When content changes Blaze does not re-render the entire page, but only small fragments inside a template. It also plays nicely with other UI libraries such as jQuery-UI or even Angular.

TRACKER

The Tracker package provides the fundamentals of functional reactive programming (FRP). At its core Tracker is a simple convention that allows reactive datasources, such as data from the database, to be connected to data consumers. Remember this code from section 1.1.5:

```
c = a + b
```

a and b are reactive data sources, c is the consumer. A change to either a or b triggers a re-computation of c. Tracker achieves reactivity by setting up a reactive context with dependencies between data and functions, invalidating the given context whenever data changes and re-executes functions.

DDP

Accessing web applications is usually done over HTTP, which by definition is a protocol for exchanging documents. While it does have advantages for transferring documents it has several shortcomings when passing data only, so Meteor uses a dedicated protocol based on JSON called DDP (*Distributed Data Protocol*). DDP is a standard way to pass data over

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

WebSockets bi-directionally, without the overhead of encapsulating documents. This protocol is the foundation for all reactive functionality and is one of the core elements of Meteor.

DDP is a standard approach to solving the biggest problem facing client-side JavaScript developers: querying a server-side database, sending the results down to the client, and then pushing changes to the client whenever anything changes in the database. There are also DDP implementations in most major languages like Java, Python, or Objective-C available. This means you can use Meteor just as a front-end component for an application and use a Java backend to communicate with it via DDP.

LIVEQUERY

Distributed environments like Meteor need a way to push changes initiated by one client to all others without needing a refresh button. Livequery detects changes in the database and pushes all changes out to the clients currently viewing affected data. At 1.0 Meteor is tightly integrated with MongoDB, but additional databases support is already on the roadmap.

FULLSTACK DATABASE DRIVERS

Many of the tasks performed on a client rely on database functionality, like filtering and sorting. Meteor leverages a seamless *database everywhere* principle. This means as a developer you can re-use most of your code anywhere in the stack.

Meteor comes with mini databases that simulate an actual database inside the browser. The miniature database for MongoDB is called MiniMongo, an in-memory, non-persistent implementation of Mongo in pure JavaScript. It does not rely on HTML5 localStorage since it exists only in the browser's memory.

The in-browser database mirrors a subset of the actual server data and is used to simulate actions like inserts. It is also used as a cache for queries, so a client can directly access available data without any network activity. Because the connection between Minimongo and MongoDB is also reactive, the data is automatically kept in sync.

Latency is a key differentiating factor between desktop applications and the web. Since nobody likes to wait, Meteor uses prefetching and model simulation on the client to make it look like your app has a zero-latency connection to the database. The local MiniMongo instance will be used to simulate any database operations before actually sending requests to the server.

The client does not have to wait for the remote database to finish writing, but the application assumes it will eventually be successful, which roughly makes the vast majority of all use cases seem much quicker. In those cases where problems occurred when passing the write action to the server, the client needs to roll back gracefully and display an error message.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Figure 1-7: Data flow using latency compensation

A typical flow of events is shown in Figure 1-7. Once a user publishes a comment, it will be validated first and then immediately it gets stored in the MiniMongo database in the browser. Unless validation fails this operation will be successful and the user's view will be updated immediately. Because up to this point no network traffic is involved and all actions take place in memory, the user will experience no latency. However, in the background the action is still ongoing.

The comment gets sent to the server where again validations take place and afterwards the comment will be stored in the database. A confirmation is sent to the browser whether the storing operation was successful. A this point at least one full server roundtrip together with some disk I/O took place, but those have no impact on the user experience. From a user's perspective there is no delay in updating the view because latency compensation already took care of any updates in the fourth step. Eventually the comment gets published to all other clients as well.

Additional Packages

Besides the core packages there are many more packages available as part of Meteor and provided by the development community. They include functionality to easily integrate users and oAuth authentication via Twitter, Github, and others.

1.2.2 Isobuild and the Command Line Tool

On a computer with a Meteor installation, entering `meteor` on the command line will bring up the CLI tool. It is both a build tool comparable to `make` or `grunt` and a package manager such as `apt` or `npm`. It enables you to manage all tasks concerning your application:

- Create new applications
- Add and remove functionality in form of packages
- Compile and minify scripts and styles
- Run, reset, and monitor applications
- Access MongoDB shell
- Prepare an application for deployment
- Deploy applications to meteor.com infrastructure

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Creating a new project is a single command upon which the command line tool creates all essential files and folder structures for a simple application. A second command starts a complete development stack including a Node.js server and a MongoDB instance to enable a fully working development. Any file changes are monitored and directly sent to the clients if form of hot code pushes, so that you can fully focus on writing code instead of starting and restarting servers.

When starting a development instance or getting ready for production, Meteor gathers all source files, compiles and minifies code and styles, source mapping, and takes care of any package dependencies. As such it combines the power of `grunt` with `npm`.

There is no need to define processing chains if you use LESS instead of plain CSS, all it takes is to add the corresponding isopack. All `*.less` files will automatically be processed by Meteor:

```
$ meteor add less
```

Add the `coffeescript` package to enable transpiling from CoffeeScript to JavaScript.

1.2.3 Client code vs server code

When starting work with Meteor knowing which code should be executed in which environment is essential to writing applications. Theoretically all code can run anywhere in the stack, but there are some limitations. API keys should never sent to the client, event maps that handle mouse-clicks are not useful on the server. To let Meteor know where to execute specific code, files can be organized in dedicated folders or use a check to verify in which context they are running.

As an example, all code that handles mouse events might be placed inside a folder named `/client`. Also all HTML and CSS files will not be needed on the server side, which is why they will also be found inside the client folder. Access credentials to a mail server or API secret keys must never be sent to the client. They will be kept exclusively on the server (see Figure 1-8).

All content from the `/server` folder will never be sent to the client. To avoid redundancies shared code can be saved to a file inside the shared folders such as `/lib` and it becomes available in both contexts. As a side effect you can easily use front-end libraries like jQuery on the server as well.

Figure 1-8: A simple application

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Sharing code between both instances is especially helpful when it comes to input validation. The same method to validate a user entered a correct credit card number can be used to display an error message in the browser, and again on the server side to prevent inserting faulty data to the database on the server. Without Meteor you would have to define one method in JavaScript for validation in the browser and another method in your server context as everything coming from the browser must be validated before working with it to establish a certain level of security. If your backend is written in a language like Ruby, PHP, or Java there is not only redundant code, but also the same task needs to be done twice. Even if using JavaScript on the server side in other frameworks you would need to copy and paste the validation section to a file on the server and a second on the client. Meteor removes this need by processing the same file on both ends.

Figure 1-9: Data exchange between server and client via HTTP and DDP

During the initial page load all JavaScript, styles, and static assets like images or fonts are transferred to the client[5]. This happens using HTTP, which is also used as a fallback for browsers that do not support WebSockets. After the initial page load only data is exchanged using a protocol called DDP, which we will take a closer look at in a later chapter. It basically uses a WebSocket connection to exchange data via JSON (Figure 1-9).

1.3 Strengths and Weaknesses

As with any tool there are situations when Meteor will be a perfect fit, but there will always be scenarios in which using it might be a poor choice. Generally speaking, any application based around the principles of distributed application platforms will benefit greatly from using it, whereas the more static a site is, the less you will gain from using Meteor.

[5] Technically all JavaScript files are combined into a single *app.js* file, but for better traceability individual files shall illustrate the information flow.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

1.3.1 Where Meteor shines

The main advantages of Meteor are a single language across the entire stack, built-in reactivity, and a thriving eco-system of packages to extend existing functionality. In summary this translates to development speed. When you decide to use Meteor you get

- rapid results, which makes it ideal for building prototypes and MVPs[6]
- a complete solution for creating applications for servers, browsers, and mobile devices
- a reactive, latency-compensated UI that brings the comfort of desktop applications to the browser
- built-in realtime capabilities to minimize development efforts
- integrated build tools that make setup, development, and deployment extremely quick

Choosing a single language stack greatly increases developer productivity and can even offer performance benefits because there are no translations required. Every part of an application communicates in the same native language. That also makes recruiting much easier, as you only need to find someone for a JavaScript position and – within reason – can switch them between backend and front-end tasks.

As the possibilities expand it becomes logical to apply interactivity and collaboration to many different types of applications. Realtime capabilities are commonly expected in chats and games, but they can enhance almost any type of application, such as collaborative video editing. Even conservative genres like e-commerce can, if enhanced with a little realtime, be exciting and attractive to users again.

From an economical point of view you get more functionality for your money, which is why many startups already decided to take advantage of Meteor. Among the early adopters of Meteor are

- Assistant.io – An online meeting scheduler
- Lookback.io – Analytics tools for mobile applications
- ReactionCommerce.com – A complete E-Commerce solution
- Respond.ly – A shared team inbox for Email and Twitter
- Signwaive.com – Software for waivers and release forms
- Versoapp.com – Classroom learning app for iOS and Android
- Workpop.com – A platform for hiring hourly workers

Meteor's core characteristics also support growing applications. Much like JavaScript itself it is designed to be very lean and agile, allowing you to improve applications in constant iterations.

Having one language across the entire application stack, a protocol that is designed for data exchange, and simple unified APIs removes the need for additional JavaScript frameworks such as AngularJS or Backbone that talk to sophisticated REST backends. That

[6] Minimum Viable Products, see http://en.wikipedia.org/wiki/Minimum_viable_product

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

makes Meteor extremely well suited for projects that require fast results while still meeting high user expectations, such as prototypes or minimal viable products (MVP).

1.3.2 Challenges when using Meteor

Although you can use Meteor to build any type of site, there are situations when it's best to use alternatives. Given its relatively young age and positioning there are certain challenges when working with Meteor.

PROCESSING INTENSE APPLICATIONS

Especially when your application relies on heavy processing such as data crunching ETL-jobs Meteor will not be able to handle the load well. By nature any Node.js process is single-threaded, so it is much harder to take advantage of fast multi-processor capabilities. In a multi-tier architecture Meteor could be used to serve the UI, but it doesn't offer a lot of computing power.

The way to integrate more processing power into a Meteor application is similar to any other Node.js application, you delegate CPU-intense tasks to child processes. However, this is also a best-practice architecture for any language, where multiple tiers are used to separate the number-crunching from the user interface.

MATURITY

Meteor itself is relatively young and still has to prove itself in production environments in regards to scaling or search engine rankings. Especially scaling applications requires a lot of knowledge about the components involved and possible bottle-necks.

Although Node.js has proven that it is capable of scaling to large loads, Meteor still has to show it can handle large deployments and a high number of requests. Conservative users might argue that it is safer to rely on an established foundation. Just keep in mind that any server stack and framework is likely to be slow if the application is not written with scalability and performance in mind.

Even if the Meteor community is very friendly and helpful, it is in no way comparable with the huge resources available for PHP or Java. That also goes for hosting options, as there are not yet as many dedicated Node.js or Meteor solutions available as for PHP or even Python. If you plan on hosting your application on your own infrastructure there are several solutions available.

As with all young projects the number of tools available around the framework itself is rather limited with Meteor as of now. An official testing framework does not exist. Velocity is a community-driven effort to create a testing framework, which has very active developers but is not part of the core Meteor projects. Also debugging tools are not as convenient as the ones available for Java or PHP.

FEW CONVENTIONS ON STRUCTURE

There are only few suggestions for structuring applications and code in Meteor. This freedom is great for single developers that can quickly hack on code, but it requires good coordination between team members when applications grow in size. It is up to the developer's preference

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

whether they use a single file or hundreds of folders and files. Some may embrace this freedom, others will find it necessary to define clear structures before being able to start coding.

USING SQL AND ALTERNATIVE DATABASES

The roadmap shows that someday Meteor will support SQL databases, but for now you are tied to MongoDB with support for Redis in the near future. In consequence your app will not be able to connect to any MySQL engines to retrieve data. If relational data is high on your priority list, then Meteor is not yet for you.

SERVING STATIC CONTENT

Some sites rely heavily on static content, like newspapers and magazines. Those are the sites that profit most from server-rendered HTML and can utilize advanced caching mechanisms that speed up the site for all users. Also the initial loading times are much faster.

If initial loading times are important to your app or it serves mostly the same content for a large number of users you will not be able to leverage all the advantages of Meteor. In fact you will need to find ways to work around its standard behavior to optimize for your use case and therefore you might want to use a more traditional framework to build your site.

Who is using Meteor – From the horses' mouth

Despite its young history, Meteor is already powering many successful projects and even entire companies.

Adrian Lanning's **Share911.com** was one of the early adopters to the Meteor platform. In case of an emergency the application enables you to simultaneously alert the people you work with as well as public safety personnel. Main criteria for picking a technology was speed – both in realtime behavior as well as development time. Adrian researched the event-driven technologies Netty (Java), Tornado (Python), and Node.js. After further evaluation of Tower and Derby.js he decided to develop a prototype using Meteor, which took less than 10 days.

Happily Meteor has been solid and we haven't needed to make a change. We have included other technologies but I am confident Meteor will be the core web tier for us for a long time. (Adrian Lanning)

Workpop.com provides a job platform for hiring hourly-wage workers. With a team of only 2 developers and just five months time CTO Ben Berman managed to create a modern take on what job boards on the internet should look like. Over $7 million in funding prove that their decision to go with Meteor has paid off. Workpop's philosophy is to keep technology out of the way and focus on their goal of getting people hired. Although very performant, both Spring (Java) and ASP.net were found to be too intensive, and even Rails was dismissed because it encourages building RESTful applications.

By sticking to familiar Javascript and shipping with the best reactive UI kit on the web, Meteor delivers on its promise of rapid iteration for small teams. (Ben Berman)

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

With **lookback.io** it is possible to record mobile user experiences and get insight into how people use your application at the push of a button. The initial version was built using Django but lead developer Carl Littke switched to Meteor soon after. Achieving the same results using Django, Angular, and the associated REST APIs turned out to be a much more complex task than relying on Meteor's built-in reactivity, data APIs, and login. Speed of development was the most important aspect when choosing Meteor. This also made up for them in the areas where Meteor's young age is still showing.

The Meteor Development Group has done an exceptional job of developing a framework that solves some of the major pain points of developing web apps today. I wouldn't think twice about using Meteor for my next project. (Carl Littke)

Sara Hicks and Aaron Judd created the open-source shopping platform **ReactionCommerce.com**. They consider the Meteor's event-driven nature a perfect fit for enhancing sales using dynamic merchandising and realtime promotions and pricing. Having a single code-base for the web and mobile devices were a big plus. Meteor is not the only technology used for the Reaction platform, but it forms the foundation. Additional libraries are added into the project, thanks to the support for all Node.js packages via npm.

Slow speed can cost retailers as much as 13 percent of sales. Thanks to Meteor's latency compensation the screen will redraw right away. This translates into happier customers and better sales figures. (Sara Hicks)

Sacha Greif created the popular Hacker News clone **Telescope**. When looking for the right stack his choices narrowed down to Rails and Node.js. With Rails he was worried about managing a large number of moving parts with hundreds of files and gems and as a designer he was already familiar with JavaScript. He made the decision for Meteor despite its limited feature set at version back in 2012 already. Today Telescope is powering sites like crater.io (news about Meteor) and bootstrappers.io (a community for bootstrapped entrepreneurs).

What really appealed to me was the all-in-one aspect: all these things that required piecing together multiple solutions with other frameworks were provided out of the box with Meteor. (Sacha Greif)

1.4 Creating new applications

We have discussed a lot of theory, now it is time to look at the code. Before you proceed make sure you have installed Meteor on your machine. Refer to the appendix to guide you through the process.

Because Meteor is also command line tool, we will need to perform the initial setup of our application in a shell environment. This allows us to install the framework and create new applications. All the steps in this section will be performed inside a Terminal.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

1.4.1 Setting up a new project

When Meteor is installed, the command line tool is used to create a new project. Navigate to the folder you want to contain your application and type into the terminal:

```
$ meteor create helloWorld
```

```
helloWorld
MacBook:local stephan$ meteor create helloWorld
helloWorld: created.

To run your new app:
   cd helloWorld
   meteor
MacBook:local stephan$ cd helloWorld/
MacBook:helloWorld stephan$ tree -a
.
├── .meteor
│   ├── .finished-upgraders
│   ├── .gitignore
│   ├── .id
│   ├── packages
│   ├── platforms
│   ├── release
│   └── versions
├── helloWorld.css
├── helloWorld.html
└── helloWorld.js

1 directory, 10 files
MacBook:helloWorld stephan$
```

Figure 1-10: A basic Meteor application created with the Meteor command line tool

Meteor created a new project folder and three files automatically:

- `helloWorld.css` contains all styling information
- `helloWorld.html` contains all templates
- `helloWorld.js` contains the actual logic

NOTE Every project contains an invisible folder `.meteor` (see Figure 1-xx), where runtime files such as the development database, compiled files, meta information regarding used packages, and other automatically generated content goes. For development purposes we can ignore this folder.

You can now start creating your own application by changing the content of the existing files. For this project the three files will suffice, but for any other project that is even a little more complex it is better to create folders and split your code into separate files to maintain a

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

better overview. We will take a close look at how to structure your projects in the next chapter.

1.4.2 Starting the Application

The command line tool of Meteor also starts the application with the following command:

```
$ meteor run
```

You can also start a Meteor server by calling `meteor` without any arguments, `run` is the default behavior. Behind the scenes it starts both a Node.js server instance on port 3000 as well as a MongoDB listening on port 3001.

You can access the application with your web browser at http://localhost:3000 (see Figure 1-11).

Figure 1-11: Every new Meteor project is a simple application with a single button

Should you need to change the port Meteor is listening on you can do so by passing the argument `--port` to the meteor command. The following command starts Meteor on port 8080:

```
$ meteor run --port 8080
```

As we can see, the application is up and running and it has one button. If you click the "Click me" button the text below will update automatically, showing you exactly how many times you have clicked it since you loaded the web page. This is because the application already has an event binding included. Let's have a closer look at the file contents and find out how that binding works.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

1.5 Dissecting the default project

The *helloWorld* application at this state is very simple. Because all files are in the root folder of the project they are all executed on the server and sent to the client. Let's have a closer look what each file does.

1.5.1 helloWorld.css

By default, this file is empty. Being a CSS file you can use it to store your custom style information. If you put something into this file, the styles will immediately be applied to the application. Meteor automatically parses through all files ending with `.css` and will send those to the client for you. Try for example adding `body { background: red; }`. Save the file and you will see the background of your application in a beautiful red color.

1.5.2 helloWorld.html

This file contains the templates used in our project. Templates control the overall look and layout of an application. Although the file extension is `.html` the code inside is not fully valid html as you could expect. The content of the HTML templates is shown in Listing 1-1.

Listing 1-1: helloWorld template

```
<head> #A
  <title>helloWorld</title>
</head>

<body>#B
  <h1>Welcome to Meteor!</h1>

  {{> hello}}
</body>
<template name="hello"> #C
  <button>Click Me</button>
  <p>You've pressed the button {{counter}} times.</p>#D
</template>
```

#A The HTML head
#B The page's body – it prints a heading and imports a template named "hello"
#C The actual template named "hello"
#D counter is a helper that is filled dynamically

First of all we can see three different elements here: an HTML head, an HTML body, and a template named `hello`.

As you can see, the opening `<html>` tag of a valid HTML document is missing. Meteor adds it automatically, so we do not need to worry about it.

The body consists of only an `h1` heading and a placeholder using Handlebars syntax. Handlebars is a templating language that inspired Meteor's own templating language called *Spacebars*. The curly brackets let you know that you're dealing with some sort of dynamic content. The greater-than symbol indicates that another template will be injected into the document at this position. It is followed by the name of the template. Therefore the placeholder inserts a template named `hello` into the body element:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
{{> hello}}
```

When starting the server the Meteor parses all files with an `.html` extension to collect all templates. It recognizes and manages any references and inclusions. For this to work properly every template needs to have an opening `<template>` and closing `</template>` tag. The `name` attribute is needed to reference a template in order to include it. A template's name must always be unique.

You also need to be able to reference the template somehow in your JavaScript to extend it with some functionality, as we will see in the `helloWorld.js` file in the next section. Again, the name of the template is used to make that connection.

Finally we need a way to inject data from the JavaScript code into the template. That's the purpose of `{{ counter }}`. That is a so called *helper*. Helpers are JavaScript methods with a return value that is available to the template. If you look at your browser you find that instead of `{{ counter }}` you can see the number of clicks. Let's look at the corresponding code.

1.5.3 helloWorld.js

The JavaScript file of a basic project contains several fundamental concepts of meteor. The first snippet we want to look at is this:

```
if (Meteor.isClient) {
  //…
}

if (Meteor.isServer) {
  //…
}
```

There are two if-statements, both relating to a Boolean variable of the global `Meteor` object itself. Remember that all code that you write is available to both the client and the server unless you apply any restrictions. Being available also means code gets executed in both environments. However, sometimes we need to specify whether code should run only on the server or only on the client. Checking for these two attributes of the global `Meteor` object we can always find out where we are.

In any project, the code block in the first if-Statement will only run if the context is the client and the code block in the second if-Statement will only run if the context is the server.

You should be aware that the entire code from this file is available on the server and the client. That means you must NEVER put security related code (like private API keys) into an `if (Meteor.isServer)` block since it may be sent directly to the client as well. Anyone opening the source view inside their browser could simply read the lines of code and with those any security related information and you definitely do not want this to happen.

SECURITY NOTE When creating a new project, Meteor puts developer productivity first. That means initially projects will not be secure enough to be deployed into production. Throughout the book we will discuss how to develop production-ready and secure applications.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Of course there are simple and standard ways to handle sensitive code and we will cover this topic in the upcoming chapters, when we discuss how to structure a project. For now, we will only use this single JavaScript file. For simple applications, checking the current context is good enough.

The next snippet looks like this:

```
if (Meteor.isClient) {
  // counter starts at 0
  Session.setDefault("counter", 0);

  Template.hello.helpers({
    counter: function () {
      return Session.get("counter");
    }
  });
//...
}
//...
```

Here we see two global objects in use: `Session` and `Template`. `Session` allows us to store key-value pairs in memory. The `Template` object enables us to access all templates, which we defined in the HTML file, from our JavaScript files. Since both objects are only available on the client, they can not be called on the server. This would lead to reference errors which is why this code is wrapped inside the `isClient` context.

As long as `Session` variables are not declared they remain `undefined`. The `Session.setDefault()` command initiates a key-value pair inside the `Session` object with the key `counter` and a value of `0`.

In this snippet we access the `hello` template defined inside the `helloWorld.html` file and extend it with a so-called *Template Helper*. This template helper is named `counter` and is a function that returns the content of the `Session` value for the key `counter` as a `string`. Now we see why the `hello` template is different from what we actually see in the browser. The template helper in the `hello` template `{{ counter }}` is in fact a function, that returns the string that we see in the browser.

On the one hand we have templates to define the HTML that should be rendered and on the other hand we have template helpers that extend templates and make it possible to use functions and substitute placeholders with dynamic content.

Remember what happens when you click the button? This is where event binding comes in. If you click the button, a `click` event is fired. This in turn increases the counter on the page by 1. The following code increases the counter, which is stored inside the `Session` object:

```
if (Meteor.isClient) {#A
  Template.hello.events({
    'click button': function () {#B
      // increment the counter when button is clicked
      Session.set("counter", Session.get("counter") + 1);#C
    }
  });}
```

#A Handling mouse clicks is only useful on the client

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

#B Define a function to call when the input button is clicked
#C Increases the Session variable by 1

Every template has the function `events()` and with that we can define event handling for a specific template. The object to pass to the events function is called "event map" and this is basically a normal key-value JavaScript object where the key always defines the event to listen to and the value is a function that is called if the event is fired.

In order to specify the event we always use a string in this form: `'event target'` where the target is defined by standard CSS-Selectors. You can easily change the example above to use a CSS class or ID to further specify the button. Also note that these events are only fired in the context of this template. This means any input in a different template such as clicking on an input element would not call the function specified here.

You can go ahead and click the button a couple of times and you will note how the browser renders your new string. Only the placeholder is updated, not the entire page.

Notice that there is no code involved that updates the template directly. We rely on the reactive nature of `Session`. The template helper `counter` is re-run whenever the value inside the `Session` object changes. The event simply changes the data source, Meteor takes care that all places that use this value are re-computed instantly.

The last snippet we look at is this one:

```
if (Meteor.isServer) {
  Meteor.startup(function () {
    // code to run on server at startup
  });
}
```

As the comment indicates, you can define a function that should be run at the startup of your application. You could also call the `Meteor.startup` function multiple times and pass different functions in order to run multiple different functions at the startup of the application. `Meteor.startup` can also be used on the client side in order to run functions at the start of the client side application. This sample application does not use any server side code, so this block and the startup function remain empty.

Now that we have looked at the `helloWorld` example code and we have a solid understanding of the basic concepts we will extend these files to develop our own first Meteor application.

1.6 Summary

- Meteor is a fullstack or isomorphic JavaScript platform, similar to the MEAN stack
- Developers can run the same code only on the server, the client, or in all contexts
- Clients are active parts of the application logic, which means Meteor applications leverage the power of distributed computing environments
- Using a standard protocol called DDP servers and clients communicate via WebSockets instead of HTTP, enabling bi-directional message exchange
- Meteor uses the principle of reactive programming to minimize the need for

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

infrastructure code

- Development productivity is enforced by re-usable packages isopacks which are used to provide common or specialized functionality
- One codebase is used to provision HTML apps for the browser or hybrid applications for mobile devices on iOS, Android, or Firefox OS

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

2

My fridge! A reactive game

This chapter covers

- Building a reactive app with Meteor
- Basic Architecture for Meteor projects
- Using the Meteor Command Line Interface

In this chapter we are going to build a first Meteor application. While Meteor comes with several example applications we will create a small reactive game from scratch. Overall we are going to write less than 50 lines of JavaScript and about the same amount of HTML to create a game that stores data to a database and updates all connected clients in real-time.

Building this simple reactive application will help you understand the basic parts of Meteor and introduces the main concepts of working with them while we walk through the code line by line. At the end of this chapter you will be able to create your own application that talks to a database and distributes data in real-time.

2.1 Application Overview

My fridge! is a small real-time application that allows you to drag products from a product list into a fridge and back. Unlike a physical fridge, this one can be accessed from anywhere at the same time and all changes will be visible to every client that is connected to the server.

While our application is merely a simulator, you could use My fridge! as a notepad to remember what's in your actual fridge at home. So if your friends take out the last bottle of juice you can check this from your office and grab some OJ on the way back home.

We will be storing all products and their current location in a MongoDB database and add an image to the application to represent it. Using jQuery-UI we will enable dragging items and make the fridge on the left and the product list on the right side a drop container. Meteor's own reactivity will take care of keeping all clients synchronized and updating the views. All we

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

need to take care of is the actual functionality, i.e. the dragging and dropping to update the location attribute for a product inside the database.

We will start by creating the basic layout using templates. Then we will use JavaScript functions to add and show items in the browser. Once these are working we shall create an event map to enable user interaction with the fridge.

Figure 2-1: The goal for the My Fridge! application

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

2.2 Initial Setup

Before you proceed make sure you have installed Meteor on your machine. Refer to the appendix to guide you through the process. Because Meteor is also command line tool, we will need to perform the initial setup of our application in a shell environment. This allows us to install the framework and create new applications. All the steps in this section will be performed inside a Terminal.

2.2.1 Setting up a new project

With Meteor installed its command line tool is available and with this creating a Meteor project is also just one line to type into the terminal:

```
$ meteor create <projectName>
```

This will create a new folder named *projectName* with a standard "hello world" application consisting of three files. Our project shall be named myFridge.

Figure 2-2: A basic Meteor application created with the Meteor command line tool

Meteor created a new project folder and three files automatically:

- *myFridge.css* contains all styling information

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

- *myFridge.html* contains all templates
- *myFridge.js* contains the actual logic of our "hello world" application

NOTE Every project contains an invisible folder `.meteor` (see Figure 2-2), where runtime files such as the local database, compiled files, included packages, and other automatically generated content goes. For development purposes we can ignore this folder.

You can now start creating your own application by changing the content of the existing files. For this project the three files will suffice, but for any other project that is even a little more complex it is better to create folders and split your code into separate files to maintain a better overview. We will take a close look at how to structure your projects in the next chapter.

Also note, that at this point we don't yet have a proper place to put static content such as images, we will add that later when we work on styling the templates.

2.2.2 Starting the Application

The command line tool of Meteor also starts the application with the following command:

```
$ meteor run
```

You can also start a Meteor server by calling `meteor` without any arguments, `run` is the default behavior. Behind the scenes it starts both a Node.js instance on port 3000 as well as a MongoDB listening on port 3001 for you.

You can access the application with your web browser at http://localhost:3000.

Figure 2-3: The basic Hello World application

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Should you need to change the port Meteor is listening on just pass the argument `--port` to the meteor command. The following command starts Meteor on port 8080 instead:

```
$ meteor run --port 8080
```

As we can see, our application is up and running and it even has a button. If you click the "Click me" button the text below will update automatically, showing you exactly how many times you have clicked it since you loaded the web page. This is because the application already has an event binding included. Let's have a closer look at the file contents and find out how that binding works.

2.3 Dissecting the default project

The My Fridge! application at this state is just the basic "Hello World" application automatically generated by the `meteor create` command. Before we start making changes to the existing files we take a closer look of what each of them do and how they work together.

2.3.1 myFridge.css

In this file we can put our custom CSS. By default the file will be empty, meaning that there are no styles defined.

> **NOTE** If you put something into this file, the styles will immediately be applied to the application. Meteor automatically parses through all files ending with .css and will send those to the client for you. Try for example adding `body { background: red; }`. Then save the file and you will see the background of your application in a beautiful red color.

2.3.2 myFridge.html

This file contains the templates used in our project. Templates control the overall look and layout of your application. Although the file extension is .html the code inside is not valid html as one might expect.

Our basic html file is currently looking like this:

```
<head> #A
  <title>myFridge</title>
</head>

<body>#B
  <h1>Welcome to Meteor!</h1>

  {{> hello}}
</body>
<template name="hello"> #C
  <button>Click Me</button>
  <p>You've pressed the button {{counter}} times.</p>#D
</template>
```

#A The HTML head
#B The page's body – it prints a heading and imports a template named "hello"

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

#C The actual template named "hello"
#D counter is a helper that is filled dynamically

First of all we can see three different elements here: an HTML head, an HTML body and a template named `hello`.

As you can see, the opening `<html>` tag of a valid html document is missing. Meteor adds it automatically, so we do not need to worry about it.

The body consists of only an `h1` heading and a placeholder using Handlebars syntax. Handlebars is a templating language that inspired Meteor's own templating language called Spacebars. The curly brackets let you know that you're dealing with some sort of dynamic content. The greater-than symbol indicates that another template will be injected into the document at this position. It is followed by the name of the template. This placeholder inserts a template named `hello` into the body element:

```
{{> hello}}
```

In order to find all available templates, when starting the server the meteor command goes through all files with an .html extension. It recognizes all templates and manages any references and inclusions. For this to work properly every template needs to have an opening `<template>` and closing `</template>` tag. The "name" attribute is needed to reference this template in order to include it. A template's name must be unique.

You also need to be able to reference the template somehow in your JavaScript to extend it with some functionality, as we will see in our `myFridge.js` file later on. Again the name of the template is used to make the connection.

Finally we need a way to inject data from the JavaScript code into our templates. That's the purpose of `{{ counter }}`. This is a so called *helper*. Helpers are JavaScript methods that are available to the template. If you look at your browser you find that instead of `{{ counter }}` you can see the number of clicks. Let's look at the corresponding code.

2.3.3 myFridge.js

The JavaScript file of our basic project contains several important fundamental concepts of meteor and because of this we will look at the code step by step. The first snippet we want to look at is this:

```
if (Meteor.isClient) {
  //…
}

if (Meteor.isServer) {
  //…
}
```

There are two if-statements, both relating to a Boolean variable of the global Meteor object itself. All code that you write will be available to both the client and the server by default.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Being available also means code gets executed in both environments. However, sometimes we need to specify whether code should run only on the server or only on the client. Checking for these two attributes of the global Meteor object we can always find out where we are.

In our project, the code block in the first if-Statement will only run if the context is the client and the code block in the second if-Statement will only run if the context of this file is the server.

You should be aware that the entire code from our files will be available on the server and on the client as well. That means you must NEVER put security related code (like private API keys) into an `if (Meteor.isServer)` block since it will be sent directly to the client as well. Anyone opening the source view inside their browser could simply read the lines of code and with those any security related information and you definitely do not want this to happen.

SECURITY NOTE When creating a new project, Meteor puts developer productivity first. That means initially projects will not be secure enough to be deployed into production. Throughout the book we will discuss how to develop production-ready and secure applications.

Of course there are simple and standard ways to handle sensitive code and we will cover this topic in the next chapter, when we discuss how to structure a project. For now, we will only use the single JavaScript file. Checking the current context is good enough for this application.

The next snippet looks like this:

```
if (Meteor.isClient) {
  // counter starts at 0
  Session.setDefault("counter", 0);

  Template.hello.helpers({
    counter: function () {
      return Session.get("counter");
    }
  });
//...
}
//...
```

Here we see two global objects in use: `Session` and `Template`. `Session` allows us to store key-value pairs in memory. The Template object enables us to access all templates, which we defined in the html-file, from our JavaScript files. Since both objects are only available on the client, they must not be called on the server. This would lead to reference errors which is why this code is wrapped inside the `isClient` context..

As long as `Session` variables are not declared they remain undefined. The `Session.setDefault()` command initiates a key-value pair inside the `Session` object with the key `counter` and a value of `0`.

In this snippet we access the “hello” template from our “myFridge.html” file and extend it with a so-called “Template-Helper”. This template helper is named `counter` and is a function

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

that returns a string: The content of the `Session` value for the key `counter`. Now we see why the “hello” template is different from what we actually see in the browser. The template helper in the “hello” template `{{ counter }}` is in fact a function, that returns the string that we see in the browser.

On the one hand we have our templates to define the HTML that should be rendered and on the other hand we have template helpers that extend the templates and make it possible to use functions and make the templates dynamic.

Remember what happened when you clicked the button? I told you there was some event binding somewhere, because if you click it and the ‘click’ event is fired there is a console log that prints out “You pressed the button”. This happens in the next snippet:

```
if (Meteor.isClient) {#A
  Template.hello.events({
    'click button': function () {#B
      // increment the counter when button is clicked
      Session.set("counter", Session.get("counter") + 1);#C
    }
  });}
```

#A Handling mouse clicks is only useful on the client
#B Define a function to call when the input button is clicked
#C Increases the Session variable by 1

Every template has the function `events()` and with that we can define event handling for a specific template. The object to pass to the events function is called “event map” and this is basically a normal key-value JavaScript object where the key always defines the event to listen to and the value is a function that is called if the event is fired.

In order to specify the event we always use a string in this form: `'event target'` where the target is defined by standard CSS-Selectors. You can easily change the example above to use a CSS class or ID to further specify the button. Also note that these events are only fired in the context of this template. This means any input in a different template such as clicking on an input element would not call the function specified here.

You can go ahead and click the button a couple of times and you will note how the browser renders your new string. Only the placeholder is updated, not the entire page.

The last snippet we look at, before we actually start working on our own application, is this one:

```
if (Meteor.isServer) {
  Meteor.startup(function () {
    // code to run on server at startup
  });
}
```

As the comment indicates, you can define a function that should be run at the startup of your application. You could also call the `Meteor.startup` function multiple times and pass different functions in order to run multiple different functions at the startup of the application.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

`Meteor.startup` can also be used on the client side in order to run functions at the start of the client side application. This sample application does not use any server side code, so this block and the startup function remain empty.

Now that we have looked at the "Hello World" example code and we have a solid understanding of the basic concepts we will extend these files to develop our own first Meteor application.

2.4 Creating a layout

The first step of building our game requires us to think about the layout. For My Fridge! we need a simple layout with a fridge on the left side and a product shelf with a list of items on the right side. Items should then be draggable from one side to the other.

In order to create the layout we will create some templates, add images, and add a way to iterate through a list of products.

2.4.1 Setting the styles

As a basic structure we want to to have a fridge and a product list side by side as shown in Figure 2-4.

Figure 2-4: The three DIV containers for the MyFridge layout

At first we set the styles for the general layout inside myFridge.css according to Listing 2-1, so we can then focus on the HTML templates. We simply define the container width and make sure the fridge is displayed on the left, and the product list on the right is a little smaller.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Listing 2-1: General layout styles

```
.container{
  width: 95%;
  position: relative;
}

.left{
  float: left;
  width: 60%;
  margin-right: 2%;
}

.right{
  float: right;
  width: 37%;
}
```

That way we defined three classes to position the div containers. Since we want our application to work on a mobile phone as well we use percentages to make the layout responsive.

> **NOTE** There are more styles required to place the products properly inside the boxes. Since this book is about Meteor and not CSS we will not discuss the styling any further at this point. You can refer to the associated code example to browse through all styles.

2.4.2 Adding the templates

For each of the DIVs we are going to use an individual template, even for the container. When we are done we will have a head section, a body, and four templates:

- `container` – for the general layout
- `fridge` – to display the fridge on the left
- `productList` – to display the supermarket on the right
- `productListItem` – to show items on either side

Let's start with the first three templates. Update the contents of `myFridge.html` as shown in Listing 2-2.

Listing 2-2: Skeleton template structure for myFridge

```
<head>
  <title>myFridge</title>
</head>

<body>
  {{> container}}#A
</body>

<template name="container"> #B
  <div class="container">
    <div class="left">
      {{>fridge}} #A
    </div>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
    <div class="right">
      {{> productList}} #A
    </div>
  </div>
</template>

<template name="fridge"> #B
  <div id="fridge">
    ...
  </div>
</template>

<template name="productList"> #B
  <div id="supermarket" class="box">
    ...
  </div>
</template>
```

#A include contents from a subtemplates
#B subtemplates are referenced by name

The container template takes care of the basic layout of the application. It puts the fridge on the left and the product list on the right. Technically we could use a single template for all HTML code, but splitting things up provides a better overview and control over what happens where.

Once the skeleton layout is created we are going to make the left side look like an actual fridge. The fridge should be represented by an image of an open fridge.

At this point we first need to extend our project to be able to add image files. Because in the sample application no images were used, we need to create a new folder called “public” in the application root. This is where we will put all images.

THE PUBLIC FOLDER is treated in a special way by convention. Every file you put into the folder is accessible at the root path of your URL. If you put a file named “my-image.jpg” into the public folder you can access it via `http://localhost:3000/my-image.jpg`.

To include an image of the empty fridge in the browser we will use a file called `myFridge/public/empty-fridge.jpg`. To reference it from a template you do not need to include `/public` in the path, just use:

```
<img class="image-responsive" src="/empty-fridge.jpg" />
```

Our fridge should contain an unsorted list of items on the right, also represented by images. We cannot tell how many items will be in the fridge at any given time, so we need a flexible solution we can iterate through. Therefore we are going to use a dedicated template for the item list.

Iterating through an array of objects inside a template can be done with a helper named `{{#each}}`. The # in front indicates that it does not substitute the placeholder for a string but

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

provides some sort of logic. We create an unordered list and iterate over `products`. Listing 2-3 shows how it looks in action.

Listing 2-3: Looping through products in the productsList template with each

```
<template name="productList">
  <div id="supermarket" class="box">
    <ul id="products">
    {{#each products}}
          <li>{{> productListItem}}</li>
    {{/each}}
    </ul>
  </div>
</template>
```

For each object in the array we pass to this helper, everything inside the each statements is rendered once. In our case we want to render a list element for every product we pass to the helper. The list element we render contains again a template that we want to insert here `{{> productListItem}}`. The advantage of using a template inside the loop is that we can reuse the exact same template `productListItem` again for our product list on the right side of our application, which means we can write less code.

Add a new template to the bottom of your HTML file according to Listing 2-4.

Listing 2-4: Template for each productListItem

```
<template name="productListItem">
  <img src="{{img}}" data-id="{{_id}}" class="image-responsive product-image
      draggable" />
</template>
```

Next we are going to adjust the fridge template in the `myFridge.html` file so that it looks like the code in Listing 2-5.

Listing 2-5: Looping through each product inside the fridge template

```
<template name="fridge">
  <div id="fridge">
    <img class="image-responsive" src="/empty-fridge.jpg" />
    <ul>
      {{#each products}}
        <li>{{> productListItem}}</li>
      {{/each}}
    </ul>
  </div>
</template>
```

As we can see we reuse the `{{> productListItem}}` template for both the fridge and the supermarket. Next we want to be able to render some products, since the product list as well as the fridge is currently empty.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

2.5 Adding content to the database in real-time

Now that the layout is in place we can focus on the backend. All items, regardless whether they are in the fridge or not, should be available from our database. Once they are in the database, they should be sent to the client, so we need to add a connection between the database and the templates.

2.5.1 Storing items in the database

Meteor comes with a bundled MongoDB database that is used by default. In order to communicate with the database, we need a Collection. MongoDB uses collections instead database tables because it is a NoSQL or documents-oriented database. Collections contain data in form of one or multiple documents. We use `Meteor.Collection` to define a new database collection to insert, remove, update, and find documents. Because the collection is going to store products we shall name it accordingly and place it inside the JavaScript file as shown in Listing 2-6.

Listing 2-6: Declaring a products collections on client and server

```
Products = new Meteor.Collection('products');

if (Meteor.isClient) {
    //...
}

if (Meteor.isServer) {
    //...
}
```

You should put this line on top of the file, outside of either `Meteor.isServer`and `Meteor.isClient` block, since it should be available on both the client and the server.

All of our products will have three attributes:

- A name, e.g. Bread
- An associated image file in the `/public` folder, e.g. bread.png
- A current location, i.e. either ‘fridge’ or ‘supermarket’

Using the JavaScript console inside the browser

Due to the fact that Meteor applications run at least partly inside the browser we need to switch to the JavaScript console sometimes to see debugging output or be able to issue commands. All major browsers have developer tools that allow you to access the console by a simple keyboard shortcut.

Chrome:

- On the Mac press the keys Option-Command-J
- On Windows press the keys CTRL-Shift-J

Firefox:

- On the Mac press the keys Option-Command-K

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

- On Windows press the keys CTRL-Shift-K

Internet Explorer:

- On Windows press the F12 and click on the `Scripts` tab

Opera:

- On the Mac press the keys Option-Command-I
- On Windows press the keys CTRL-Shift-I

Safari:

- On the Mac press the keys Option-Command-C

In order to add some data we can simply open up the JavaScript command line tools in a browser and add a product using:

```
Products.insert({img: "/bread.png", name: "Bread", place: "fridge"});
```

As a response of this function call you get the document-ID of the newly inserted product. With this ID you can also get the document out of the database like this:

```
Products.findOne({_id: "X6Qw8v3ChcsZKaKan"});
```

Because we know only a single object may have this ID we use the `findOne()` function. The response is a single product object from the database.

Figure 2-5: Using the browser's JavaScript console to insert and lookup data in the database

Developer productivity versus security

While it is very convenient to use the Developer Console to add and remove items from the database it is also a security risk. If you can do it then anyone else using your application will also be able to do it.

New projects always contain a package called `insecure` which disables authentication checks and allows anyone to read and write to any Collection. Together with its sibling `autopublish` – which automatically makes all Collections from the server side and their content available to the client – it makes developer's lives much easier because it lets them focus on building functionality first, without having to struggle with authentication issues during the early stages of development.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

During development you will most likely define a permissions scheme and limit only publish data to the client that the authenticated user has access to. When that time comes you can get rid of both packages using the command line:

```
$ meteor remove insecure
$ meteor remove autopublish
```

Querying data by its location is also quite simple. Instead of using the `_id` field we can look at individual attributes as well. Since we expect more than a single result, we will use `find()` and look for all database entries that have `fridge` as their `place` attribute.

```
Products.find({place: "fridge"});
```

Now that we added some data to the database we can access and view the data in the browser.

2.5.2 Connecting data to templates

In the fridge template, we want to iterate over all products where the attribute "place" is "fridge". In order to do so we extend the template with a helper called "products" that returns all products we want to show in the fridge. Let's look at the template again:

```
<template name="fridge">
  ...
</template>
```

We can access each template by its name with the global `Template` object that comes from Meteor itself. Every template object has a function called `helpers` that takes an associative array where the values are accessible in the template through its keys.

```
Template.fridge.helpers({
  products: function(){
     return [];#A
  }
});
```

#A Templates expect an array within an each loop

For the fridge we want to iterate through an array of products that we get from the database. The helper's job is to look up the data from the database and pass them to the template. Remember that to display the data we created a loop inside the templates using `{{#each products}}...{{/each}}`:

```
<template name="fridge">
  <div id="fridge">
    <img class="image-responsive" src="/empty-fridge.jpg" />
    <ul>
      {{#each products}}#A
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
          ...
        {{/each}}
      </ul>
    </div>
  </template>
```

#A Loop through all items returned from the products helper

To extend the template `fridge` with the `products` helper we pass it to the `helpers` function of the template object. To do so replace all code inside the `Meteor.isClient` block in the `myFridge.js` file with the code from Listing 2-7.

Listing 2-7: Setting up a products helper for the fridge template

```
Template.fridge.helpers({
  products: function(){
      return Products.find({place: 'fridge'});
  }
});
```

The `products` helper is a function that returns every product where the attribute `place` is `fridge`, just as we want it. Since we have added a product to the fridge already above, it should directly be added to our view. Make sure you also placed the associated image into the public folder.

Figure 2-6: Bread in the fridge

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

For the `productList` template on the right side we can now do the exact same thing but with a query that looks for a different value of the `place` attribute. Since we do not want to show all products from the fridge but from the supermarket we need to query the database as in Listing 2-8.

Listing 2-8: Setting up a products helper for the productList template

```
Template.productList.helpers({
    products: function(){
        return Products.find({place: 'supermarket'});
    }
});
```

If you insert some more products to the fridge or the supermarket using the JavaScript console you will see how they get added automatically to the screen. Also the insertion of products is done in real time already. Open up another browser and you will see products being added immediately.

Figure 2-7: Changes to the database are reflected to other browsers in real-time

So far we created the UI and set up the required data structures for the fridge application. The last thing we have to add is the user-interaction that allows visitors to put products into the fridge and remove them from it, too.

2.5.3 Adding a defined set of products

While it is sometimes helpful to be able to add products manually we can use the server to add a defined set of products upon startup. That way we have a known state to work with, regardless of how much testing has been done in a previous run.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Inside the JavaScript file the code from Listing 2-9 will remove all products from the database every time the server starts. Afterwards it will put milk in the fridge and bread in the supermarket:

Listing 2-9: Adding a defined data set to the database upon server start

```
if (Meteor.isServer) {
  Meteor.startup(function () {
    // code to run on server at startup

    // clear the data base if the server starts
    Products.remove({});

    // fill the database with some products
    Products.insert({
      name: 'Milk',
      img: '/milk.png',
      place: 'fridge'
    });

    Products.insert({
      name: 'Bread',
      img: '/bread.png',
      place: 'supermarket'
    });
  });
}
```

Stop your Meteor server using CTRL-C and start it again. If you copied over the images for milk and bread from the sample code into your public folder you can now see a bottle in the fridge and bread on the right. As a final touch we will add interactivity via drag-and-drop.

2.6 Moving items into the fridge

Our goal is to make the products draggable from the product list into the fridge and vice versa. This is not specific to Meteor but a standard front-end task. We are going to use the jQuery-UI library for the dragging and dropping and make the necessary connections to our templates. Since we also want to update the database as a result of dragging and dropping we need to supplement the front-end actions with a backend that is capable of storing content into the database.

We will start by adding the jQuery-UI library to the existing project. It will provide drag-and-drop functionality that works across all major browsers. Once the library is available we will define both the fridge and the product list as possible drop targets where items may be placed. Finally each product list item will be marked as draggable, so that it may be moved to either of the drop containers.

2.6.1 Adding jQuery-UI to the project

We do not have to add the jQuery-UI library to our project files, but we can include the online version instead. This works the same way in Meteor as in any other HTML file, by adding it to the <head> area of our `myFridge.html` file (see Listing 2-10).

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Listing 2-10: Loading jQuery-UI from a CDN

```
<head>
  <title>myFridge</title>
  <script src="//code.jquery.com/ui/1.11.2/jquery-ui.js"></script>
</head>
```

The library is now loaded from jquery.com. Obviously you need to have a connection to the internet in order to use the CDN to provide the JavaScript file. Alternatively you can download the library from http://jqueryui.com and put the file `jquery-ui.min.js` into a folder called `/client`. Meteor will automatically serve and load this file on the client. If you add the file make sure to remove the `script` tag from the head section of your template file.

2.6.2 Defining drop targets for items

We will use the jQuery API for drag and drop to define both `fridge` and `productList` as possible drop targets. Since the `fridge` and the `productList` have to be rendered into the DOM before we can use jQuery-UI to perform any modifications on them we have to wait until each template is rendered. A callback function available to every template makes this very easy:

```
Template.fridge.rendered = function() {
    $('#fridge').droppable({
      drop: function(evt, ui){
        // do something if dropped
      }
    });
}
```

When the fridge template is rendered, we define `<div id="fridge">` as a droppable target. This means DOM elements may be added to it dynamically by simply dragging them into the div area. Basically we are listening for an event (i.e. a user moving an item to the container), therefore we have to define an event handler to first of all listen whether something was dropped in the container. The event handler will also need to perform an update to the associated item in the database and update its location.

The event handler will be called `drop` and it will change the `place` attribute once a product has been dropped. To identify the database entry that corresponds to the product that was dropped we need to pass a data ID to it. In JavaScript event handler we use a function with two arguments, the event and a `ui` object. We will use the `ui` object of the `drop` callback to identify the ID of the dropped product. We can tell which HTML item was dragged by looking at the `ui.draggable` argument. The `ui.draggable` object will always be one of the `productListItem` items. It is easy to identify which HTML element was dragged, but we also need a link back to the database.

```
<template name="productListItem">
  <img src="{{img}}" data-id="{{_id}}" class="image-responsive product-image
      draggable" />
</template>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

As you can see, we added a data-attribute to the `<img>` named `data-id`. The value of the `data-id` attribute is set to `_id` which represents the database ID of a product. When we drag an image we have an easy way to identify which product in our database is affected and we can change its `place` attribute accordingly. Using a similar syntax like we did for inserting new products we can update existing ones by calling the `update()` function for our products collection. All we need is to pass the database ID and set the `place` attribute to the location it was dropped in.

Remember that `data-id` and with that the `_id` of the related product of the dragged image is available via the `ui.draggable` object.

Listing 2-11: Declaring fridge as a droppable target and updating the product location

```
Template.fridge.rendered = function () {
    $('#fridge').droppable({
        drop: function (evt, ui) {
            var query = {_id: $(ui.draggable).data('id')};#A
            var data = {$set: {place: 'fridge'}};#B
            Products.update(query, data);#C
        }
    });
};
```

#A Get the database ID from the HTML attribute data-id
#B Set the update statement to set place to fridge
#C Perform the database update

In the code in Listing 2-11 you can see how we create a query to address the correct product document in our products collection. The first argument passed to `Product.update()` works just like before, it returns a document based on its ID. The second argument uses the `$set` functionality to define that we want to update this document. The data we want to set is the `place` attribute and since this is the drop event that is called if an item is dropped over the fridge, we want to change the place attribute to `fridge`.

The product list is also a drop target and therefore needs almost the exact same code, but with two small yet important differences. Again the `rendered` function of the `productList` template has to be used to wait for the DOM. Also if an item is dropped over the `productList` template, the `place` attribute of the product document has to be changed. This time we need to set it to `supermarket` instead of `fridge`. Listing 2-12 shows the required code.

Listing 2-12: Declaring productList as a droppable target

```
Template.productList.rendered = function () {#A
    $('#supermarket').droppable({#B
        drop: function (evt, ui) {
            var query = {_id: $(ui.draggable).data('id')};
            var data = {$set: {place: 'supermarket'}};#C
            Products.update(query, data);
        }
    });
};
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

#A productList is the template name used by Meteor
#B supermarket is the div ID
#C Set the place attribute to supermarket when products are droped

2.6.3 Allowing items to be draggable

We have set up the droppable targets but we also have to define elements that may be dropped: The draggable items. In this application each `productListItems` element should be draggable. Again we require jQuery-UI to mark the list items as draggable, so we use the `rendered` function of the `productListItem` again. For every `productListItem` that is rendered, this `rendered` callback will be executed once, effectively making each food movable.

Listing 2-13: Declaring productListItems as a draggable items

```
Template.productListItem.rendered = function () {
    $(this.find('.draggable')).draggable({
        cursor: 'move',
        helper: 'clone'
    });
};
```

Note in Listing 2-13 that we use `this.find('.draggable')` in order to access the HTML element that is dragged. In our case this is the `<img>`. Using the `find` method inside a rendered callback of the `productListItem` template allows Meteor to only search for an element inside of the associated template, not the entire DOM tree, which makes it more effective.

2.7 Deploying to meteor.com and using the fridge

We now have an application in which you can drag an image from a product list on the right, to the fridge on the left and vice versa. When the image is dropped, the underlying product document will be updated accordingly. The changes reactively change the UI and the product is rendered in the correct place automatically.

In order to share your fridge with the world you can deploy it on one of the Meteor test servers with the deploy command. Just pick a name that is not yet taken that should be used as the subdomain under which your app shall be available. If the subdomain is taken already you will see an error message.

We will deploy our projecte to `mia-ch02-myfridge` by using

```
$ meteor deploy mia-ch02-myfridge
```

To make sure only you can access, update, or delete the application from the test server it will be associated with your personal Meteor developer account that is based on your e-mail address. Therefore you must provide your e-mail when deploying for the first time. Meteor will memorize your address on your working machine. Also you will get an e-mail response on how to secure your account with a password. When the deployment is finished you can access your

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

application on a meteor.com subdomain with the name you picked, in our case http://ch02-mia-myfridge.meteor.com.

Now you can share this URLwith a friend (or simply open up two browsers on the same computer) and start dragging and dropping products back and forth. You will see that all changes will be available on all clients almost instantly. Meteor takes care of keeping all connected clients in sync, even though we never defined any specific code for that. We also never had to write any code to poll the server for any database updates. This is full-stack reactivity in action.

In the next chapters we will look at how all the parts that seem to work magically fit together and see, that there is no magic behind all of this at all but just good old JavaScript.

2.8 Summary

- You create and start a project with the meteor command line tool
- Meteor runs Node.js and MongoDB in the background
- You use Collections to interact with the database
- Static files like images are served from the `public/` folder
- Changed data is reactively pushed to all clients
- Spacebars is the templating language used by Meteor

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

3

Working with Templates

This chapter covers:

- What are templates and how to create them
- Meteor's default templating syntax
- Organizing JavaScript and HTML
- Using event maps to make templates interactive
- The lifecycle of a template

Everything your web browser renders is HTML eventually. If you open any web site in source view you will find that each page has hundreds of lines of HTML. Writing everything by hand is not only tedious, error-prone, and highly inefficient, it is also impossible for web applications because much of the content is dynamic and you cannot know in advance what exact code needs to be rendered.

Templates allow you to define blocks of HTML that can be reused as often as you like. As a result you need to write less code which is easier to read and much simpler to maintain.

In this chapter we will understand how to write templates and what we can do with them. In order to do this, we will first take a look at Blaze, Meteor's UI system, and Spacebars, its default templating language.

3.1 Introduction to templates

Templates are the building blocks for our UI. Basically templates are snippets of HTML that contain placeholders which can be filled by content returned by the application logic. They can be reused to ensure the same look and feel on all sites and all elements.

Another important aspect of templates is separation of concerns. This means a template is only for the representation of data and gets ultimately rendered as HTML to the DOM. A

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

template should have as little programming logic inside of it as possible. Everything that needs computation and creates a specific output should not be inside of a template but separated into a JavaScript file. This helps to increase the readability a lot and it is basically the same concept of separating the styling of html into its own CSS files instead of cluttering the html with all styling definitions.

Think about a simple list of users as shown in Figure 3-1. For each person you want to display a name along with their phone number. Since we do not know how many people need to be shown in this list, we are going to use a dynamic template to generate the necessary HTML code.

Figure 3-1: A list of people and their phone numbers

In this example we use a <h5> element that wraps around the name and put the phone number inside a <p> element. In order to tell the template processor where we want the content to go we use placeholders that use double curly brackets. Using pseudo-code the required template would look like this:

```
<h5>{{NAME}}</h5>
<p>Phone: {{PHONE}}</p>
```

Because the placeholders may contain multiple elements we need the template to be able to create any number of list items. Just like in JavaScript code we can use a loop that takes an array as input and iterates over each of the array contents, taking care of filling the placeholders for *name* and *phone* with information about the person. In our case we create a list of people that reuses the profile template. For each person it should contain:

```
<ul>
    <!-- for every user render this: -->
    <li>{{ LOOP_THROUGH_PERSON_LIST_USING_TEMPLATE }} </li>
</ul>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Hooking up the templates with the backend logic will return the HTML code shown in Listing 3-1.

Listing 3-1: Rendered HTML from template

```
<ul>
    <li>
        <h5>Christina</h5>
        <p>Phone: 1234-567</p>
    </li>
    <li>
        <h5>Stephan</h5>
        <p>Phone: 666-999-321</p>
    </li>
    <li>
        <h5>Manuel</h5>
        <p>Phone: 987-654-321</p>
    </li>
</ul>
```

The advantage of using templates is you write them only once and use them for an infinite amount of data (in our case: people) you want to render. Having a single place to manage the appearance of an element is also very convenient. Think about what you need to do if you do not want the <h5> element but only a normal <p> element for a person's name. You only have to change it in a single template and it instantly updates for any person thatis rendered with the help of the `profile` template.

3.2 Working with Templates

Unless you are writing a server-only application every web application written in Meteor should have at least one template. Displaying content in a browser without using a template is not possible. This section discusses the foundations of working with templates.

THE BLAZE ENGINE

Behind the scenes Meteor uses a UI engine called *Blaze*. It is responsible for processing templates and is one important part of what is often referred to as the magic of Meteor. As you can see in Figure 3-2 Blaze consists of two major components:

1. Runtime API
2. Build Time Compiler

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Figure 3-2: The Blaze components

The *runtime API* renders elements, keeps track of their dependencies, and updates them through their complete lifecycle as those dependencies change. That means if the phone number of a person is changed in the database and a user currently views a page listing this person's information, the number gets automatically updated on the screen. This is because the placeholder depends on the actual value stored in the database, which is a reactive data source.

The runtime API works with JavaScript in order to apply reactivity, it cannot deal with HTML directly, which is why the second component of Blaze is a *Build Time Compiler*. Meteor ships with a compiler, which translates HTML into JavaScript, or more specifically HTMLJS. By default it processes templates using Spacebars. If needed, it is possible to use packages to switch the compiler to a different templating language such as Jade.

Both components work separately, so it is entirely possible to bypass the build time compiler and not use any templates but write code that directly targets the runtime API. Because this is not very practical for most users we will not worry about the runtime API itself but concentrate on using Spacebars.

NOTE Meteor's templating language is called Spacebars. It is a direct descendant of Handlebars, with which it shares most of its syntax and functionality.

Blaze uses live page updating, so whenever a template-related file is changed on the server (either HTML, CSS, or JavaScript) it gets updated instantly in the browser. How does Blaze find templates inside a project?

Organizing Template Files

When working with templates you usually have 4 types of files to deal with, while only one is technically required for a template:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

- The actual template stored in an `.html` file
- Optional JavaScript code in a `.js` file that runs inside the client context and provides functionality to templates
- Optional styles in one or more `.css` files[9]
- Optional static resources such as images or fonts inside the `/public` folder

Without a JavaScript counterpart templates may only be static and cannot be filled with dynamic content, which is why in most cases you will need at least two files for working with templates. To keep everything organized it is best to keep each individual template in a dedicated `.html` file. Meteor will find them anywhere in your project folders[10]. All the frontend code can be stored in a single `.js` file or you may use `html/js` pairs so that each template consists of two files, which is recommended if your projects gets more complex:

```
<template name>.js
<template name>.html
```

In this chapter we are not going to worry about styling elements as this works just like in any other web application. We have covered the `/public` folder in the last chapter, so we will focus solely on the first two elements: HTML and JavaScript files.

3.3 Creating dynamic HTML templates

Meteor comes with its own templating language called Spacebars. If you are familiar with Handlebars or Mustache you will already know enough to start using it immediately. Even if you did not yet use any of these languages you will find that the general syntax is fairly simple to work with.

Templates written using Spacebars look almost like regular HTML. Template tags are easily spotted because they are always enclosed in multiple curly braces. There are four major types of template tags:

- Double-braced tags `{{ … }}`
- Triple-braced tags `{{{ … }}}`
- Inclusion tags `{{> … }}`
- Block tags `{{#directive}}` … `{{/directive}}`

3.3.1 Double and Triple-Braced Tags (Expressions)

Template tags that can be used to enhance static HTML code by substituting them with dynamically generated content are also called *expressions*. They depend on a data source or some kind of application logic that returns a value.

[9] Also less, sass, and other styling languages can be used, css is just as an example.
[10] There are exceptions to this rule which we will explore in depth in chapter 10.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

They can only be used inside a template context. A basic template tag is shown in Listing 3-2.

Listing 3-2: Double-braced template tags

```
<template name="expressions">
    {{ name }}
</template>
```

As you can see every template has an opening and closing `<template>` tag and a mandatory name attribute. The name attribute has to be a unique identifier for a template inside the application. Using the name of a template you are able to access it from a JavaScript file, which we will do later on.

DOUBLE-BRACED TAGS

Double-braced template tags are used to insert strings into the HTML. No matter what return value they are dealing with - an array, an object, a string - it is always rendered as a string. Let's assume you have a template named expressions from Listing 3-2 and you want to replace the template tag `{{ name }}` with `<strong>Michael</strong>`.

The accompanying JavaScript code must return the substitute string as in Listing 3-3. Remember that the code must be wrapped in a `Meteor.isClient` environment as it cannot be run on the server side because templates are not available inside the server scope.

Listing 3-3: JavaScript code for a helper called name

```
if (Meteor.isClient) { #A
    Template.expressions.helpers({
        name: function () {
            return "<strong>Michael</strong>";
        }
    });
}
```

#A Template may only be used on the client, it is undefined on the server

Together HTML and JavaScript code will render as shown in Figure 3-3.

Figure 3-3: Tags in double curly braces always render strings and escape HTML and script tags

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

As you can see, double braces process the string and escape all possibly unsafe characters. This is very useful if you want to avoid return values to be accidentally interpreted as HTML or even JavaScript. However, sometimes you may want to avoid any processing of the string. In that case triple brackets should be used.

TRIPLE-BRACED TAGS

If your template tag starts and ends with three curly braces `{{{ … }}}` its contents will be rendered,exactly as you passed them to the template tag. Extending the example from Listing 3-3 we will now use `{{{ name }}}` instead of `{{ name }}`. Meteor will not escape any characters or tags and pass the output to the browser unaltered (see Figure 3-4).

Figure 3-4: Strings in triple curly braced tags are rendered as HTML

As you can see, the HTML tags are rendered as HTML into the DOM and not as simple Strings.

BE CAREFUL if you use triple curly braces to display user generated content you must ensure that it has been sanitized first (i.e. check for potential malicious scripting content). If you don't, your website is vulnerable to Cross-Site-Scripting attacks. The simplest way to deal with user generated data is to let Meteor sanitize it for you before it gets displayed and stick with double curly brackets whenever possible.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

A look into Blaze's build time compiler - Turning HTML into HTMLJS

Blaze uses its runtime API to transform HTML code from a Spacebars template into JavaScript. Each compiled template file can be found in the directory `.meteor/local/build/programs/web.browser/app`.

The template in Listing 3-2 produces the following code:

```
Template["profile"] = new Template("Template.profile", (function() {#A
  var view = this;
  return HTML.STRONG(Blaze.View(function() {#B
    return Spacebars.mustache(view.lookup("name"));#C
  }));
}));
```

#A The template named profile is now converted into a function
#B The template tag inside the tags is also wrapped inside a function
#C lookup("name") will retrieve the value to be displayed

The above HTMLJS code enables Meteor to reactively update templates and even just parts of it. Every template is accessible through the global `Template` object by its name attribute.

Although it is possible to call `Blaze.View` or even `HTML.STRONG` directly from inside a template helper there is rarely a need to do so. Only when you decide to build your own runtime API for Blaze to replaces Spacebars with something else such as Jade or Markdown you will need to familiarize yourself with the internal processing structures.

Until an API documentation for this becomes available you can find out more by looking at the contents of the `blaze` and `spacebars` packages, which are part of Meteor's core.

3.3.2 Inclusion Tags (Partials)

Beside inserting strings or HTML you can also insert a template into another template. As they represent only a part of the whole template, these subtemplates are also known as *partials*. Inserting a template into another is achieved using the > sign enclosed within two curly braces like this:

```
{{> anotherTemplate }}
```

Inclusion tags are an important tool to keep templates small and let them represent only one thing. If you want to render a complex UI it is advisable to split everything the user can see into smaller, logically encapsulated templates and subtemplates.

MONOLITHIC TEMPLATES

For example you may require a profile page that not only displays an avatar image and the name of a person but also a news stream. You could put everything into one template as in Listing 3-4.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Listing 3-4 Full profile page template

```
<template name="partials"> #A
    <div class="left"> #B
        <img src="{{image}}">
        <p>{{name}}</p>
    </div>
    <div class="right"> #C
        <ul class="news-stream">
            <li class="news-item">Yesterday I went fishing, boy this was a blast</li>
            <li class="news-item">Look, cookies! <img src="cookies.jpg"></li>
        </ul>
    </div>
</template>
```

#A The template representing the profile page
#B Some CSS that places the div on the left side
#C Some CSS that places the div on the right side

If your UI gets more complex you can imagine that your profile template could become big. This is really bad for readability and code maintainability. It is better to separate everything with its own logical function into a dedicated template and combine them in a main template. This way your templates stay small and are easier to read and maintain. That way two developers could split up the work on profile and news stream and make independent changes much more easily.

MODULAR TEMPLATES

Listing 3-5 shows a first template `partialsSplit`. This is the main template that represents the site and contains two smaller templates. Two small templates represent the actual user profile (`partialsUserProfile`) and the news stream (`partialsNewsStream`). Also note, that the layout is contained in the main `partialsSplit` template. In this case the simple `<div class="left"></div>` and `<div class= "right"></div>`.

Listing 3-5: Splitting the profile page template

```
<template name="partialsSplit"> #A
    <div class="left">
        {{> partialsUserProfile}} #B
    </div>
    <div class="right">
        {{> partialsNewsStream}} #C
    </div>
</template>

<template name="partialsUserProfile">
    <img src="{{image}}">
    <p>{{name}}</p>
</template>

<template name="partialsNewsStream">
    <ul class="news-stream">
        <li class="news-item">Yesterday I went fishing, boy this was a blast</li>
        <li class="news-item">Look, cookies! <img src="cookies.jpg"></li>
    </ul>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
</template>
```

#A The template that represents the profile page
#B The user profile is included into the parent template as a sub template
#C The news stream is also included into the parent template as a sub template

TIP Avoid putting layout information into subtemplates. Let the parent template define the look and feel as well as the size of the elements they include.

Not having layout definitions buried inside subtemplate greatly improves reusability. Because there are no layout definitions in the `partialsUserProfile` template you can easily reuse it on another page template and put it onto the right side into a `<div class="right">{{> partialsUserProfile }}</div>`.

DYNAMICALLY INCLUDED TEMPLATES

Besides using static texts to include subtemplates it is also possible to dynamically include a template based on the return value from a helper. That way you can reactively switch templates without having to maintain complex if/else structures inside of templates. In combination with reactive data sources like a session variable they can be quite powerful.

Listing 3-6: Inserting subtemplates dynamically with helpers

```
// HTML file
<template name="dynamicPartials">
    <div class="left">
        {{> Template.dynamic template=templateNameLeft }} #A
    </div>
    <div class="right">
        {{> Template.dynamic template=templateNameRight }} #A
    </div>
</template>

// JavaScript file
Template.dynamicPartials.helpers({
    templateNameLeft: function () {
        return "partialsUserProfile"; #B
    },
    templateNameRight: function () {
        return "partialsNewsStream"; #B
    }
});
```

#A The name of the subtemplate comes from a helper
#B Helpers return a string can be dynamically generated or static

3.3.3 Block Tags

In contrast to *expressions* or *partials*, which are essentially placeholders, *block tags* change the behavior of the enclosed block of HTML. They start with double curly braces and a #. A sample block template tag may look like this:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
<template name="myTemplate">
  {{#name arguments}} #A
      <p>Some content</p> #B
  {{/name}} #C
</template>
```

#A The opening of a block template tag has a name and has optional arguments
#B The content of the block template tag
#C The closing of a block template tag must always have the same name as in the opening

Block tags are not only used to display content, but to have control over the processing of a template. You can either define your own block tags or use the Spacebars own:

- `#if` – executes a content block if a condition is true or the else block if not
- `#unless` - executes a block if a condition is false or the else block if not
- `#with` – sets the data context of a block
- `#each` – loops through multiple elements

THE IF/UNLESS TAG

One of the built-in tags is the `#if` block tag. It works like a normal `if` block in JavaScript. It checks a condition and if it evaluates to true, the contents of the block are processed, i.e. they are rendered. Any value that is considered to be true in a regular JavaScript context is also considered to be true by the `#if` tag. If a common falsey JavaScript value like `null`, `undefined`, `0`, `""` (empty string), and `false` is passed to `#if` the following block will not be rendered (see Table 3-1).

Listing 3-7 only shows an image tag if there is an image helper and it evaluates to true.

Listing 3-7: Using an if-block

```
<div class="cookies">
    <p>Look, more cookies!</p>
   {{#if image}} #A
       <img src="{{image}}" />
   {{/if}}
</div>
```

#A The <img> tag is rendered if image evaluates to true

Table 3-1: Input values and their interpretation by `#if`

Input value	Evaluates to
false, 0 (zero), '' (empty string), null, undefined, NaN, [] (empty array)	`false`
Strings (including "0"), arrays, objects (including empty objects)	`true`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

The counterpart to an #if block tag is the #unless block tag. It processes the block contents only if the condition evaluates to false (Listing 3-8).

Listing 3-8: Using an unless-block

```
<template name="unlessBlock">
    {{#unless image}} #A
        <p>Sorry, no image available.</p>
    {{/unless}}
</template>
```

#A the <p> tag is rendered if image evaluates to false

Both #if and #unless can be combined with an else tag in order to render one thing if the condition is true and another if it is false. Listing 3-9 uses #if but it works just the same with #unless.

Listing 3-9: Using else in an if-block

```
{{#if image}} #A
   <img src="{{image}}" />
{{else}}
   <p>Sorry, no image available.</p> #B
{{/if}}
```

#A Whether if or unless are used defines whether the case for true or false is handled first
#B The <p> tag is rendered if {{image}} returns a falsey value

> **NOTE** There is no tag {{elseif}}. In order to deal more cases than true and false you will either need to use nested if-else-structures inside your template or preferably adjust your JavaScript code so that it can do the processing instead of the template.

Block tags and template tags always have to include valid HTML, otherwise Meteor will run into an error. That means you have to take care that tags that are opened are closed as well. Also, every element that opens inside a block must also be closed inside that block. You cannot for example include an opening tag in an #if block tag and close after the {{else}} because this would result in invalid HTML for your page upon rendering. If you have errors in your templates the application will crash and produce an error message. Figure 3-5 shows the resulting error message for the following code block.

```
{{#if highlightBox}}
    <div class="box box-highlighted"> #A
{{else}}
    <div class="box"> #A
{{/if}}
<p>Welcome!</p>
</div> #B
```

#A This is not possible since it is not a complete HTML tag
#B A closing div tag without an opening one is not allowed inside a template

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Figure 3-5: Output for invalid HTML in templates

It is perfectly alright, however, to use a block tag inside of an HTML attribute value:

```
<div class="box {{#if highlightBox}}box-highlighted{{/if}}"> #A
    <p>Welcome!</p>
</div>
```

#A If showBox is a truthy value the box string is included into the class attribute

THE EACH/WITH TAG

If you wish to pass multiple values to a template the most common way is to use an array. When an array is passed to a template you can iterate over its contents using an #each tag. #each takes an array as its argument and renders its block content for every item inside the array. In Listing 3-10 skills is passed as an argument. This defines the so-called data context of the block. Without a data context #each does not render anything.

Listing 3-10: Using the #each tag

```
// HTML file
<template name="eachBlock">
    <ul>
        {{#each skills}} #A
            <li>{{this}}</li> #B
        {{/each}}
    </ul>
</template>

// JavaScript file
Template.eachBlock.helpers({
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
    skills: function(){
        return  ['Meteor',  'Sailing',  'Cooking'];
    }
});
```

#A The #each block tag takes an array as a parameter
#B You can access the current object of the array with "this"

While #each requires the template to have a data context, `#with` allows you to define a data context. The data context is the actual association between a template and any data.

Setting the data context using the `#with` tag requires a single attribute that will become the data context for the following block. Listing 3-11 shows an example where the data context for the template `withBlock` is explicitly set to `profileJim`.

Listing 3-11: Using the #with tag

```
// HTML file
<template name="withBlock">
    <ul>
        {{#with profileJim}} #A
            <p>{{name}}</p>
            {{#each skills}}
                <li>{{this}}</li>
            {{/each}}
        {{/with}}
    </ul>
</template>

// JavaScript file
Template.withBlock.helpers({
    profileJim: function () {
        var jim = {
            name: 'Jim "Sailor Ripley" Johnson',
            skills: ['Meteor', 'Sailing', 'Cooking'],
        };
        return jim;
    }
});
```

#A The profileJim object is defined as the data context

There is no need to explicitly specify a data context, using a helper automatically provides a context. There are some more advanced use cases which require you to do so. We will look at them when we talk about reactive data sources in the next chapter.

Technically all built-in block tags are helpers. Let us have a look at how you can create your own template helpers in the next section.

3.3.4 Helpers

When dealing with templates it is often necessary to apply the same functionality again, such as formatting seconds into the HH:mm:ss format or applying control structures. This is when helpers are needed.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Helpers are JavaScript functions that can perform any kind of processing. They can either be limited to a single template or be made available globally. Global template helpers may be reused in all available templates and it is good practice to define them in a dedicated JavaScript file instead of the individual template `.js` files.

Local Template Helpers

A local template helper is used to extend one specific template only. It cannot be shared between other templates and exists only within the namespace of a template. In their simplest form they look similar to *expressions*.

Every `Template` object has a `helpers` function which takes an object that can contain multiple key–value-objects. Typically the key represents the placeholder name that can be used inside a template while the value is a function that returns something. The function's return value does not have to be a string, but it can be any static value like a number, an array, an object, or even a function that returns another value.

In order to simplify things both the contents from the HTML file as well as the JavaScript file are shown in Listing 3-12. Some of the helpers return only a static value (`name`), others return an array (`skills`), an object (`image`), and even a function (`hasMoreSkills`). The rendered HTML code can be seen in Figure 3-6.

Listing 3-12: A profile template using different local helpers

```
// HTML code
<template name="localHelpers">
    <p>{{name}}</p> #A
    {{#if image}} #B
        <img src="{{image.thumb}}"> #C
    {{/if}}
    {{#if skills}} #D
        <p>Primary Skill: {{skills.[0]}}</p> #E
        {{#if hasMoreSkills skills}} #F
            <a href="/skills">see more…</a>
        {{/if}}
    {{/if}}
</template>

// JavaScript code
Template.localHelpers.helpers({
    name: 'Jim',
    image: {
        large: '/jim-profile-large.jpg',
        thumb: '/jim-profile-thumb.jpg'
    },
    skills: ['Meteor', 'Sailing', 'Cooking'],
    hasMoreSkills: function (skills) {
        return skills && skills.length > 1; #G
    }
});
```

#A Accesses the local helper name that is the simple string 'Jim'
#B Image is an object and thus a truthy value, the content block is rendered
#C You can access object values as you would in normal JavaScript by a .
#D If the array is empty it would not render the block content

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

#E Skills is an array and you can access a special index with via array.[index]
#F The #if block content is rendered if the returned value of the helper function hasMoreSkills returns true. In this case hasMoreSkills has the skills helper as a parameter
#G Returns true if the skills parameter is passed and has more than just one skill

Figure 3-6: Resulting code from local helpers

In order to pass parameters to a helper function simply write the parameters you want to pass after the helper function (separated by a whitespace). The order in which you pass the parameters will be the same as defined in the function itself.

Have a look at Listing 3-12 again. The built-in helper `#if` evaluates the following expression to determine whether it is true or false:

```
{{#if hasMoreSkills skills}}
```

In this case it checks `hasMoreSkills`, which is a function that itself requires an input value. Therefore, instead of the standard behavior of using `if` followed by a single expression a second placeholder is passed: `skills`. The content of the `skill` object is passed to the function represented by `hasMoreSkills`. If a person has multiple skills it will return `true`, so the `if` condition passes.

Global Helpers

Oftentimes you have some helper that you need in more than just one template but you want to write it only once, of course. Let's say you want to create a helper that returns `true` if an `array` has more than `n` items. We shall call this helper `gt` (as in greater than). This helper should be used in multiple templates, so we create a `globalHelpers.js` file and put the helper code in there. Remember to wrap it in an `if(Meteor.isClient){…}` block since helpers – just like templates – are only available in the client scope.

Because we want our new helper to be available in all templates we cannot use `Template.<templateName>` to define it, but we will use `Template.registerHelper` instead. Listing 3-13 shows how local and global helpers are combined.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Listing 3-13: Using global helpers to determine array length

```
// HTML file
<template name="globalHelpers">
    {{#if gt skills 1}} #A
        <a href="/skills">see more…</a>
    {{/if}}
    {{#if gt images 4}} #B
        <a href="/images">see more…</a>
    {{/if}}
</template>

// JavaScript file
Template.globalHelpers.helpers({ #C
    skills: function () {
        return ['Meteor', 'Sailing', 'Cooking'];
    },
    images: function () {
        return ['/jim-profile-large.jpg', '/jim-profile-thumb.jpg'];
    }
});

// globalHelpers.js
if (Meteor.isClient){
    Template.registerHelper('gt', function(array, n){ #D
        return array && array.length > n;
    });
}
```

#A skills contains 3 items so the global gt helper returns true
#B images contains only 2 items, the global gt helper returns false
#C local helpers for the templates
#D Using the registerHelper function you can create helpers that are available to all templates

CUSTOM BLOCK HELPERS

Also globally available and very useful are *custom block helpers*. They allow you to build re-usable UI components or widgets. Note that the actual helper may be used even without any JavaScript. We define a new block helper named `#customHelperBlock`. We also need to define a template of the same name. Inside the template we use the partials syntax to include the output of `Template.contentBlock`.

When `#customBlockHelper` is called from within a template it places its own contents between `{{#customBlockHelper }}` and `{{/customBlockHelper }}` where `Template.contentBlock` is included. Listing 3-14 shows how `contentBlock` can be used to wrap block contents in any template using the `{{#customHelperBlock}}` in a `<div>`. Adding `{{> customHelperBlockTemplate }}` to the body of an application creates the output shown in Figure 3-7.

Listing 3-14: Custom block helpers with Template.contentBlock

```
<template name="customHelperBlockTemplate">
    {{#customHelperBlock}} #A
        Surrounded by a box.
    {{/customHelperBlock}}
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
</template>

<template name="customHelperBlock">
    <div class="box custom-helper-block">
        {{> Template.contentBlock}} #B
    </div>
</template>
```

#A We use the custom purpleBox block helper and in its content we put some image
#B The block helper is defined as a normal template, but it has a special inclusion tag. It uses the Template.contentBlock to relate to the content of the block helper. In this case it would be the img element

Figure 3-7: The surrounding div is created by the custom block helper

Besides the `Template.contentBlock` there is also a `Template.elseBlock` that relates to the content block after an `{{else}}` template tag. This way we can enhance block helpers with simple control structures.

Listing 3-15: Using Template.elseBlock

```
// HTML file
<template name="templateElseBlock">
    {{#isFemale gender}} #A
        Mrs.
    {{else}}
        Mr.
    {{/isFemale}}
</template>

<template name="isFemale">
    {{#if eq this 'w'}} #B
        {{> Template.contentBlock}} #C
    {{else}}
        {{> Template.elseBlock}} #D
    {{/if}}
</template>

// JavaScript file
Template.templateElseBlock.helpers({
    gender: function () {
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
        return 'm';
    }
});

Template.isFemale.helpers({
    eq: function (a, b) { #E
        return a === b;
    }
});
```

#A Use of the custom block helper called isFemale to which we pass the gender variable that comes from the template helper
#B The isFemale block helper also has a helper function eq that takes to arguments and checks if they are both equal. "this" relates to the gender parameter in this case.
#C If the gender is 'w' then the #if statement is truthy and the contentBlock should be rendered
#D If not, the elseBlock should be rendered. Since the gender is 'm' the elseBlock would be rendered.

MOVING LOGIC INTO HELPERS

Using helpers to return a value that should be rendered is generally the better way to dynamically display content. The less logic exists within a template the easier it is to troubleshoot and extend an application.

If you do not require different HTML code for different cases it is best to define a helper that calculates the correct content to be displayed. This way you can avoid using `#if` and `#unless` in your templates. Listing 3-16 uses a template `logicByHelper` that displays the correct salutation based on a single character gender value that may be stored in a database. All processing is done by the helper function, not the template itself.

Listing 3-16: Using the #each helper

```
// HTML file
<template name="logicByHelper">
    {{genderLabel gender}} #A
</template>

// JavaScript file
Template.logicByHelper.helpers({
    gender: 'm',
    genderLabel: function (gender) { #B
        if (gender === 'm') {
            return 'Mr.';
        } else {
            return 'Mrs.';
        }
    }
});
```

#A In the double braced tag we call the helper function genderLabel and pass the parameter gender
#B The genderLabel helper function returns 'Mr.' in this case, since gender has the value 'm'

As we can see, we can define lots of different helpers for a template that can be static values or even functions that return some computed value.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

We learned a lot about how to use spacebars to create and extend templates so that is easy to generate some html at the end. Now that we know how to create the HTML code of an application we are going to enable the user to interact with the rendered HTML using events next.

3.4 Handling Events

One of the main differences between a static site and an application is that applications allow for user interaction. They need to handle events such as a click on a button and react on it. The response most of the time is basically the modification of the DOM in order to show the user something that is useful for him. In order to achieve this there must be a way to do two things:

1. Define the events the application listens to
2. Define the actions that are triggered by an event

Meteor uses event maps to both define events and their actions. DOM events are used in conjunction with CSS selectors to specify which elements and what events should be watched. Although you may use any DOM event for an event map their behavior might differ between various browsers. The events in Table 3-2 should however behave the same across all major browsers.

Table 3-2: Event types that work alike across all major browsers

Event type	Use
`click`	Mouse click on any element, including links, buttons, or divs
`dblclick`	Double-clicks using the mouse
`focus`, `blur`	Text input or other form control fields that gain or lose focus. Any element with a `tabindex` property is considered focusable.
`change`	Checkboxes or radio buttons that change state
`mouseenter`, `mouseleave`	Entering or leaving an element with the mouse pointer
`mousedown`, `mouseup`	Pressing and releasing the mouse button
`keydown`, `keypress`, `keyup`	Pressing and releasing keys on a keyboard, `keydown` and `keyup` are mostly used for modifier keys such as shift

3.4.1 Template event maps

Every template has its own event map. It is defined inside a JavaScript file similar to Listing 3-17.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Listing 3-17: Event map for layout template

```
if (Meteor.isClient) {
    Template.layout.events({ #A
        'click button': function (event, template) { #B
            $('body').css('background-color', 'red'); #C
        },
        'mouseenter #redButton': function (event, template) { #D
            // start fancy animation
        }
    });
}
```

#A Each template has an events function that takes the event map object as its parameter
#B The key of the object defines what and from whom the event should be triggered
#C The value of the object is a function that is called in case the button is clicked. This event handler takes the event object itself as the first parameter and the template instance as the second parameter
#D Moving the mouse cursor over the element with id redButton might start an animation

Meteor uses jQuery to call the actual event handler functions. In this example, if the user clicks on any button within the layout template, the corresponding event handler is called and the background-color of the body is set to red. A click on any button outside of the layout template would not trigger an action since the associated event map is only relevant for anything inside the layout template. But what happens if we use a child-template and send an event to it? Change the code according to Listing 3-18 and click either button.

Listing 3-18: Listening for events in a subtemplate

```
// HTML file
<body>
    {{> layout}}
</body>
<template name="layout">
    <button>Turn red</button>
    {{> green }}
</template>

<template name="green">
    <button id="green">Turn green</button> #A
</template>

// JavaScript file
Template.layout.events({
    'click button': function (event, template) {
        $('body').css('background-color', 'red');
    }
});
Template.green.events({
    'click button': function(event, template) {
        $('body').css('background-color', 'green');
    }
});
```

#A Clicking the button would make the bodies background-color red because the event would also call the handler of the layout template after the green handler was executed.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Even though we have two distinct event maps and two buttons, any click on a button in the updated code will turn the background to red. Even clicking on the `Turn green` button. Why is that?

3.4.2 Event propagation

The phenomenon you can see at work here is called *event propagation* or `event bubbling`. That means every event is first processed where it occurs and then passed upwards in the DOM tree. There it may trigger another action.

In a best case scenario you would cleverly use this as chaining, in the worst case – which this example shows – you will end up overwriting the action you wanted to take place with another.

NOTE Event propagation may have unexpected side effects when working with event maps. Remember to stop it, otherwise it will bubble up the DOM tree.

Figure 3-8: Action sequence for a click in green template

As you can see in Figure 3-8 there are three template instances, the body, a `layout` and a `green` template. If the user clicks on `<button>Turn green</button>` the event listener of the `green` template is called, since it listens for clicks on buttons inside its template scope. The first action that takes place sets the `background-color` attribute of body to green. But the event propagation is not done yet.

The event is passed upwards to the `layout` template, which also acts on click events. It calls its own event handler. Now a second action takes place which sets the `background-color` attribute to red.

Technically the background color was green for a brief moment in time, but only as long as it took the event to propagate up. Therefore the event handler of the `green` template has no visible effect.

Finally, the event is passed to the body element and if it had an event map defined it might even trigger a third action.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

If you do not want an event to be handled by multiple templates you can – and should - always stop the event propagation. Add `stopImmediatePropagation()` to your template's event map to prevent events from bubbling up the DOM. Fix the event map for the green template by updating your code to Listing 3-19.

Listing 3-19: Stopping event propagation within an event map

```
Template.green.events({
    'click button': function(event, template) {
        event.stopImmediatePropagation(); #A
        $('body').css('background-color', 'green');
    }
});
```

#A This stops the event from bubbling up the DOM

Now clicking the button turns the background color to green, regardless whether the layout template also listens to button clicks. If you want to have even more control over event handling you can also call `evt.stopPropagation()`. This does not prevent other event handlers from being executed, but instead you can call `evt.isPropagationStopped()` to check whether `stopPropagation()` was called somewhere in the event chain. Using this technique you could add an event handler to the body and react to the click event in the green template as well as in the body but not trigger the `layout` event handler.

3.4.3 Preventing the browser's default behavior

Something you also want to prevent in many circumstances is the browser's default event handling. For example if you click on a normal link (`<a href="url">Go To</a>`), the browser would open the URL specified in the `<a>` element's `href` attribute and reload the page. Of course in an application you build with meteor you do not want the browser to reload the page at any time. In order to prevent this we can simply call `event.preventDefault()` and the browser's default behavior is prevented.

Listing 3-20: Preventing the browser's default behavior

```
Template.profile.events({
    'click a': function(event, template){
        event.preventDefault(); #A
        // do not follow the link
        // do something else instead
    }
});
```

#A Prevents the browser from executing the default action

Before we move on to see how to integrate data into a template the last topic we need to cover is the template lifecycle.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

3.5 The Template Lifecycle

Putting a template inside the DOM and making it visible for the user is only part of its lifecycle. Overall there are three main steps each template goes through in order to get rendered in the browser. Each stage has an associated callback which is useful to add custom behavior.

Figure 3-9: The template lifecycle

The first step when inserting a template into the DOM is called `created`. Although the actual template is not yet visible, the template instance is already accessible. The associated callback is especially useful if you want to initially create some properties for the template instance before it is rendered and visible to the user. All properties that you set in the `created` callback are available in the other life cycle callbacks as well. You can even use them in your helpers and event handlers. In order to access a template instance from within a helper or event handler use `Template.instance()`.

The second step is called `rendered`. This callback is typically used to initiate objects that are inside the DOM already. Typical examples are jQuery plugins like datepickers, calendars, or datatables. They require a rendered DOM element, so they are initiated in the `rendered` callback as shown in Listing 3-21. Here we extend all objects with a class `.dateinput` with a datepicker[11].

Listing 3-21: Initializing a jQuery plugin to create a datepicker on an input element

```
Template.formTemplate.rendered = function(){
    $('.dateinput').datepicker({
    // additional options
    });
};
```

The final callback `destroyed` is used to clean up anything that you set up during the lifetime of a template. After it executes the template instance is neither visible nor accessible anymore.

All three callbacks are just executed once and will not be repeated, even if data on the page changes.

Let's consider a simple scenario with a single placeholder expression:

```
<body>
    {{> profile}}
</body>
```

[11] In order to actually use a datepicker you also need to add the required datepicker libraries to your project.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
<template name="profile">
    {{!-- demonstrating the lifecycle --}} #A
<p>{{placeholder}}</p>
    <button>Button</button>
</template>
```

#A You can use comments inside templates like this

In Listing 3-22 we add an explicit callback to each stage in a template's lifecycle. When the `profile` template is created we attach a property `lastCallback` to the template object and set it to `created` and print the object in the JavaScript console. This is also the place where you could already read the data context of a template. Inside the `rendered` callback we change the value for `lastCallback` to `rendered`. Using `Template.instance()` the helper can read the value for `lastCallback` and the button click is able to update its value. The `destroyed` callback cannot be observed inside the browser console. All console messages are shown in Figure 3-9.

Listing 3-22: The template life cycle callbacks

```
Template.profile.created = function () {
    this.lastCallback = 'created';
    console.log('profile.created', this); #A
};
Template.profile.rendered = function () {
    this.lastCallback = 'rendered';
    console.log('profile.rendered', this); #A
};
Template.profile.destroyed = function () {
    this.lastCallback = 'destroyed';
    console.log('profile.destroyed', this); #A
};
Template.profile.helpers({
    placeholder: function () {
        console.log('profile.placeholder', this); #B
        console.log('profile.tplInstance', Template.instance().lastCallback); #C
        return 'This is the {{placeholder}} helper';
    }
});
Template.profile.events({
    'click button': function (event, template) {
        Template.instance().lastCallback = 'rendered and clicked'; #C
        console.log('profile.clicked', this); #B
        console.log('profile.clicked.tplInstance', template); #D
    }
});
```

#A Prints out the instance of the profile template. You can set variables to the template instance like this.foo = 'bar' that you could use later. You can also read the data context, but you cannot set it here.
#B Prints out only the data context. You have no access to the template instance via this.
#C You can still access the template instance in a template helpers and events
#D In the event handlers you do not need Template.instance() since the template instance is passed as the second parameter directly

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
Elements  Network  Sources  Timeline  Profiles  Resources  Audits  Console
<top frame> ▼  Preserve log
profile.created
▼ Blaze.TemplateInstance {view: Blaze.View, data: null, firstNode: null, lastNode: null,
  lastCallback: "created"…}
    data: null
  ▶ firstNode: h1
    lastCallback: "rendered"
  ▶ lastNode: button
  ▶ view: Blaze.View
  ▶ __proto__: Blaze.TemplateInstance
                                        section35.js?4417913ba495ac82c2645040a8eb25c8790e8ac3:4
profile.placeholder Object {}           section35.js?4417913ba495ac82c2645040a8eb25c8790e8ac3:16
profile.tplInstance created             section35.js?4417913ba495ac82c2645040a8eb25c8790e8ac3:17
profile.rendered
▶ Blaze.TemplateInstance {view: Blaze.View, data: null, firstNode: h1, lastNode: button,
  lastCallback: "rendered"…}
                                        section35.js?4417913ba495ac82c2645040a8eb25c8790e8ac3:8
profile.clicked Object {}               section35.js?4417913ba495ac82c2645040a8eb25c8790e8ac3:24
profile.clicked.tplInstance
▶ Blaze.TemplateInstance {view: Blaze.View, data: null, firstNode: h1, lastNode: button,
  lastCallback: "rendered and clicked"…}
                                        section35.js?4417913ba495ac82c2645040a8eb25c8790e8ac3:25
```

Figure 3-10: Console messages for template callbacks

3.6 Summary

- Meteor uses its own UI engine called Blaze
- Spacebars is Meteor's default templating language which is an extended variant of Handlebars
- Spacebars uses expressions, partials, blocks, and helpers to create small and modular templates
- Helpers may be limited to a single template or available globally
- Event maps are used to associate actions to events and elements
- The templates life cycle goes through three stages and each uses a dedicated callback

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

4

Working with Data

This chapter covers

- Meteor's default data sources
- The principles of reactive data and computations
- The `Session` object
- Working with a MongoDB database using `Collections`
- CRUD operations

As we have seen in chapter 1, Meteor does not rely on a traditional, server-focused architecture. It runs code and processes data on each client as well. In order to do so it uses a mini database to mimic the API of a real database inside the browser. That means you can access data in the same manner, regardless whether you are making a database lookup or access the results of a query on the browser.

All data that is available only in a single client will obviously be lost once the client disconnects and has not sent its updates back to a central server. Meteor will take care of persisting data automatically by synchronizing client and server state.

Some information is only relevant for a single client, such as state information, which tabs have been clicked or what value was selected from a dropdown list. Information that is only relevant to an ongoing user session does not need to be stored on the central server and does not get synchronized. Figure 4-1 illustrates the general architecture. At the end of this chapter you will be able use `Collections` for data inside databases and `Session` for client-only information.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Figure 4-1: Database everywhere means persistent data on the server, but volatile data on the client

At its core, each web application captures, stores, and processes data. Creating, reading, updating, and deleting data, often referred to as CRUD, is the foundation upon which advanced functionality can be built. CRUD in itself is only the most basic functionality, though. When two users read the same document from the database and one of them performs an update we want this update to be published to the other client immediately. At least we should tell the second user that the document has changed since they first accessed it. In most languages and frameworks you must manually set up a synchronization process that ensures all clients work with fresh data. Meteor takes care of managing the data flow properly by means of *reactivity*.

This chapter introduces you to the key components of reactivity and how to leverage it for CRUD operations in Meteor. To illustrate the principles we are going to turn a real world problem into an application: Imagine you go on a trip and ask a friend to take care of your plants and leave him with a note to water the red flowers once a week. We are building a house-sitter application.

houseSitter – Become a better friend using Meteor

When Manuel goes on vacation he asks his friend Stephan to water the red flowers once a week. He left his instructions on a post-it note requesting a simple action: watering the flowers. When should

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

this action be carried out? When a week has passed. Although this seems like a simple task, we should build an application that helps us keep track of when and which plants we need to water.
To comply with Manuel's request any good friend will have to factor in two data sources – the flowers and a calendar. The latter is not explicitly mentioned, but plays an important role in determining whether an action should be executed on the flowers. As with most situations in the real world, these instructions leave out some details. For starters, they do not define when the week starts or even how much water to use.

Naturally, a true friend will provide the missing context themselves. In order to determine whether to water the flowers, Stephan relies on his calendar. A calendar is just a regular data source in itself, but adding a dependency to water the flowers each week turns it into a reactive data source. While Manuel is gone Stephan is going look at his calendar each day and if it's a Monday he decides he will go over and water the red flowers. In order to react to changed data Stephan created a mental dependency. The event "Monday" is linked to an action "water the flowers".

Although there is room for error, it is much more convenient that Stephan is capable of making some sensible assumptions which allows Manuel to use a post-it instead of writing a 1000 word essay on what to do while he's gone.

Meteor also links data sources and actions and provides an out-of-the-box behavior that allows us developers to work with reactive data.

When you have finished this chapter you will be familiar with the most important data sources in Meteor. You will also be able to work with both volatile and persistent data, meaning you can store and retrieve data from a database, and perform full CRUD-operations. We will also discuss how to implement two-way data binding using Meteor's reactive approach by means of a so-called edit object.

We are going to implement functionality into an application by making use of two standard packages in Meteor: `autopublish` and `insecure`. As the names suggest they make development easier by taking care of publishing data automatically across all clients and giving us more freedom during the development process by waiving strict security. That way we can focus fully on adding functionality without constantly having to refactor security settings. Eventually, when we get to chapter 6, we will discuss how to remove them to prepare an application for deployment.

4.1 Meteor's default data sources

In web application we typically deal with two types of data that are each associated with a specific type of data source

1. Volatile data or short-term memory (e.g. RAM)
2. Persistent data or long-term memory (e.g. files and databases)

Volatile or short-lived data is used to deal with things like accessing the currently logged in user. There is no reason to put this data into a database and share it across all clients, so it is generally only available in the client instance where the session takes place.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Once a browser window is closed all volatile data is typically gone, unless it is stored in form of a cookie or the browser's local storage. However, a user may have configured their browser to delete those on quit, so it is not safe to assume the data stored inside either is still available the next time a user visits a site.

Persistent data is anything that is actually stored by an application. This can include blog articles, comments, user profiles, or products in a web shop. Persistent data sources are available to some or all users of a web application. Meteor's default behavior is to share all persistent data sources across all connected clients. This is great during the early stages of development, but not if the amount of data grows into hundreds or even thousands of datasets. Transferring everything to a client regardless whether they will ever see the data can be avoided by using customized publications that clearly define what data gets transferred. They can also help adding a security layer to avoid sending sensible data that should only be available for some users to all connected clients. Again, this is a topic we are going to cover in chapter 6.

As Meteor is designed to work well with NoSQL databases it does not use tables like MySQL or Oracle do, but it stores data as documents. Collections are comparable to database tables and may hold one or multiple documents. Later in this chapter we will talk more about databases.

Table 4-1: Most common data sources and their typical usage

Data Source	Typical Usage	Type
Session variables using the `Session` object	Selections or current step in multi-step actions	Volatile
Collections (Database queries)	Database content	Persistent

NOTE By default Meteor publishes all data from a database to all clients unless the `autopublish` package is removed. We will do this in chapter 6 when we cover publications.

Regardless whether they are used to store volatile or persistent data, all built-in data sources in Meteor are reactive by default. Table 4-1 gives an overview of the most common data sources and what they are used for. Let's take a closer look to understand what makes them different from non-reactive data sources.

4.1.1 What makes a data source reactive?

If something happens in consequence to another thing that happened before it is commonly known as a *reaction*. This holds true also for Meteor. In order to apply reactivity we need data and actions, and create a trigger mechanism that links both together.

There is no need to constantly evaluate whether the flowers in a friends house need water. Instead, by using a calendar we already have a data source that can be used to determine

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

whether an action is required. We have an action "check whether the flowers should be watered" which we defined to be executed each Monday. As such we need to use a calendar as our data source in order to tell if today's day of the week has changed. If it changed we must perform the check once and can then wait for the next day until we check again.

It is not effective to create a large list of all actions and every relationship that eventually may occur as it becomes tedious to keep such a list up to date. Also what happens if we forget to check the calendar? In most frameworks we would have to implement a frequent check to monitor possible changes to the calendar which is similar to sitting at a desk and constantly looking at the clock in order not to miss the next day to possibly water the flowers.

Meteor makes things easier by using a declarative approach to define relationships between data and functions. A regular calendar becomes reactive due to the linked dependency on the check action and it behaves just like an alarm clock for weekdays. That way we can take advantage of reactivity, which means we execute the check based on the alarm initiated by our data source (see 4-2). There is no need to explicitly check whether the current day has changed, we will be notified by the calendar when it happens.

Figure 4-2: Data changes trigger related actions

By adding a relationship in form of a dependency any regular data source may be turned into a Reactive Data Source. A reactive data source cannot only be passively accessed but it is also capable of actively initiating a function to run by invalidating it. The `Tracker` package is responsible for creating and tracking dependencies as well as managing computations, which are the foundation of the platform's reactivity.

Tracker – Dependency tracking behind the scenes

The fact that all built-in data sources are reactive by default means that Meteor automatically takes care of creating and tracking dependencies. It does so via a package named *Tracker*. This package is used to declare dependencies for a data source, invalidate computations, and trigger re-computations.

As long as you only work with `Session` and `Collection` objects inside of templates you might not need to use Tracker directly. For advanced techniques it is very helpful to understand the basic principles behind this small package that consists of less than 1k of code. Chapter 9 will introduce you to the inner workings of Tracker. For the moment we will rely on Meteor keeping track of all dependencies for us without explicit declaration.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

4.1.2 How reactive data is connected to functions

Although we have said before that reactivity is built into Meteor and you get it for free, it will only be used if you set your code up to do so. For that matter, Meteor provides *reactive contexts* in which reactivity takes place. These contexts can be created by using one of

- `Templates`
- `Blaze.render` and `Blaze.renderWithData`
- `Tracker.autorun`

We looked at `Templates` and `Blaze` in chapter 3. We will be using `Tracker.autorun` in section 4.3 when we look at the `Session` object.

Once you have created a reactive context, a function inside will become a *computation*. A computation gets executed once and is executed again if it gets invalidated. Invalidating a function happens when a reactive data source is changed.

When computations are invalidated they are re-run, which makes them valid again. That prevents functions from running constantly and creating a chaotic and non-deterministic state of the application. The invalidation is a direct consequence of a data change and triggers an action. As long as the data does not change, the computation is not invalidated and hence not executed again. Because there can be various different reactive contexts with varying dependencies Meteor keeps track of all dependencies in a list (see Figure 4-3).

Figure 4-3: Changes to reactive data sources invalidate computations and trigger them to re-run

When starting to work with Meteor you may not be aware that you are using reactive computations. When sending data to a template, Meteor takes care of re-rendering if any of the data changes. For example if you have a template that displays a reminder to water the flowers it would get automatically updated if a reactive data source is used as shown in Listing 4-1.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Listing 4-1: Setting up a reactive context using Template helpers

```
Template.friendsHouse.helpers({#A
    waterTheFlowers: function () {#B
        var day = Session.get("today");#C
        if (day === "Monday") {
            return "Watering the flowers";
        }
    }
});
```

#A Creates a reactive context
#B Functions inside reactive contexts are called computations
#C Session is a reactive data source that will invalidate the computation when its contents change

NOTE Functions inside reactive contexts are called computations. Reactive data sources invalidate computations when data changes, which causes computations to re-execute. All reactive data sources used inside a computation are automatically associated with the computation

Now that we have looked at the things Meteor will do automatically for us we will focus on the things we have to do to work with and display data.

4.2 Building a house-sitting app

Let us revisit the example of our friend watering the flowers. He is not only taking care of our plants, but he is now a professional house-sitter for many people and he tracks all of them in a database. This is also where he keeps all instructions for watering the plants. He is going to use a simple web application to look up each house and make a note when he is done with a visit. Such an application would use the following data sources:

- A database to store all friends, instructions, and notes
- Session variables to store currently selected house

Figure 4-4 shows the user interface along with some annotations. All data coming from a collection is marked in green, all temporary data inside a `Session` object is colored orange. Looking at it from a high level the app looks up all entries in a database and retrieves a single full document based on the value of a temporary session variable. Eventually it lets the user store changes made to the database.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Figure 4-4: User Interface and data sources for the house-sitter app

Create a new project using the meteor command line tool:

```
$ meteor create house-sitter
```

We are going to organize our code in different folders to make it easier to follow which code goes where. That removes the need to add `Meteor.isServer()` or `Meteor.isClient()` around any of the code.

We will have code that should only be executed on the client. This goes into `client/client.js`. All templates will be put inside `client/templates.html`. Code that is only executed on the server goes to `server/server.js` and collections will be stored inside `collections/houses.js` as they should be available on both client and server. See Figure 4-5 for reference.

Figure 4-5: Organization of the House-Sitter app

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

4.2.1 Setting up templates

Before we start working with reactive data we will set up a skeleton structure. Listing 4-2 shows the actual body of our site. It includes three sub-templates: `selectHouse` that will allow users to select a house, `showHouse`, which displays all relevant details of a database entry, and `houseForm` to add and edit database entries. These will be defined inside the same file in the upcoming sections as all templates will remains rather small. Splitting them up is not necessary.

Listing 4-2: Basic template structure in client/templates.html

```
<head>
    <title>The House-Sitter</title>
</head>

<body>
   <h1>The House-Sitter App</h1>
   {{> selectHouse }}
   {{> showHouse }}
   {{> houseForm }}
</body>

<template name="selectHouse"> #B
</template>

<template name="showHouse"> #B
</template>

<template name="houseForm"> #B
</template>
```

#A inclusion tags for additional templates
#B These templates remain empty for now

> **NOTE** If you are including templates that are not available Meteor will show an error. You can avoid running into these errors by either creating an empty templates or removing the inclusion tags from the body until they are actually needed.

The application will does not need any styling definitions, therefore the `client/style.css` file remains empty. If you wish to add styles to make the house-sitter app prettier, this is where you could put them.

4.2.2 Connecting to a database and declaring Collections

Although we are going to look at Collections at section 4.4, we need to define it now because we require some data to be available. We will discuss the details of working with Collections later in this chapter.

MongoDB is a document-oriented or NoSQL database. It does not store its contents in tables, but in documents. Multiple documents are grouped in collections. Therefore we are going to define a new Collection object called `HousesCollection` that will store its contents

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

inside a Mongo database in a collection named houses. Create a file inside a collections directory and put the code from Listing 4-3 inside.

Listing 4-3: Collection declaration in collections/houses.js

```
HousesCollection = new Mongo.Collection("houses");
```

Also, we want to make sure there is some data inside the database for us to work with, so we are adding some server-side code that checks upon start whether there is any data inside the `HousesCollection` available. If there are no entries, our code will insert a new document into the database (see Listing 4-4). You can add more houses to the houses array if you require more data to be available.

> **NOTE** Wrapping code in a `Meteor.startup()`block works on both server and client. While on the server the code gets executed once when the Node.js instance starts, it gets executed on each client once the DOM is ready.

Wrapping the code inside a `Meteor.startup()` block makes sure it only runs when the server starts. Theoretically we could also add this code to the client, but `Meteor.startup()` would be executed every time a client successfully connects. Due to the conditional `if` nothing would happen, so we can just as well limit the fixtures to the server only.

Listing 4-4: Adding fixtures in server/server.js

```
Meteor.startup(function () { #A
    if (HousesCollection.find().count() === 0) { #B
        var houses = [ #C
            {
                name: 'Stephan',
                plants: [
                    {color: "red", instructions: "3 pots/week"},
                    {color: "white", instructions: "keep humid"}
                ]
            }
        ];
        while (houses.length > 0) {
            HousesCollection.insert(houses.pop()); #D
        }
        console.log("Added fixtures");
    }
});
```

#A only execute once when the server starts
#B check whether there are any entries in the Collection yet
#C define all fixtures as array elements
#D insert all objects from the houses array into the database

4.3 Working with the Session object

Traditionally accessing a web site via HTTP is stateless. A user requests one document after another. Because there is often the need to maintain a certain state between requests, for

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

example keep a user logged in, the most essential way to store volatile data in a web application is the session. Meteor's concept of a session is different from languages such as PHP, where a dedicated session object exists on the server or in a cookie. Meteor does not use HTTP cookies but the browser's *localStorage* instead, for example for storing session tokens to keep a user logged in.

A dedicated `Session` object that is only available on on the client and lives in memory only is useful for keeping track of current user contexts and actions.

4.3.1 The Session object

The `Session` object holds key-value pairs, which can only be used on the client. Technically it is a reactive dictionary that provides a `get()` and a `set()` method. Until a Session key is associated via `set()` it remains `undefined`. This can be avoided by setting a default value using `setDefault()` which works exactly as `set()`, but only if the value is currently `undefined`. As a frequent operation is to check for a session value the `Session` object provides an `equals()` function. It is not necessary to declare a new Session variable using the var syntax, it becomes available as soon as a `set()` or `setDefault()` command is used.

The corresponding syntax is shown in Listing 4-5**Error! Reference source not found.**.

Listing 4-5: Using the Session object

```
Session.setDefault("key", "default value"); #A
Session.get("key"); #B
Session.set("key","new value"); #C
Session.equals("key","default value"); #D
```

#A setDefault() only sets a value for a key if the key is undefined
#B returns default value
#C assigns a new value to a key
#D translates to Session.get("key") === "default value" but is more efficient

GOOD TO KNOW Although a Session variable is typically used with strings, it can also hold arrays or objects.

Let's see how we can apply the `Session` object to the house-sitting app. Consider `Session` to be the app's short-term memory to keep track of the currently selected house.

4.3.2 Using Session to store selected dropdown values

For the `selectHouse` template all we need is a dropdown list to select a house from the database. The idea is to retrieve all documents from the database and show all available names. Once a name is selected it is going to define the context of all other templates and a single house is displayed. We will be using the code shown in Listing 4-6.

An `{{#each}}` template helper is used to iterate through all houses returned from the database. The data context is set explicitly by passing `houseNameId` as an argument. Both `{{_id}}` and `{{name}}` are attributes of the house object coming from the database, so there is no need to define helpers for them.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Listing 4-6: Dropdown list code in selectHouse template

```
<template name="selectHouse">
    <select id="selectHouse">
        <option value="" {{isSelected}}></option>
        {{#each houseNameId}}#A
            <option value="{{_id}}" {{isSelected}}>{{name}}</option>
        {{/each}}
    </select>
</template>
```

#A Begin the list with an empty option to select
#B each iterates over all objects returned by a helper called houseNameId

Inside the JavaScript file we will define a helper that will provide the `houseNameId` data context. Because we haven't looked at the details of working with Collections yet we will simply return all documents and fields for now. Because `houseNameId` is defined inside a `Template` object it is reactive. This means if documents are added or removed from the database the return value will automatically be adjusted and the select box will reflect the changes without the need to write dedicated code!

We will now use a `Session` variable called `selectedHouse` to store the dropdown selection. As the select box should reflect the actual selection it needs to add a `selected` attribute to the currently selected option. In order to do so we define a second helper named `isSelected` that returns either an empty string or `selected`, if the value of `_id` equals that of our Session variable.

As the last step we need to set the value for the `Session` variable based on the user's selection. Because it involves an action coming from the user this this requires an event map.

Whenever the value for the DOM element with the ID `selectHouse` changes the event handler shall set the `selectedHouse` variable to the value from the selected option element. Note that we need to pass the event as an argument to the JavaScript function that sets the `Session` value in order to access its value (see Listing 4-7).

Listing 4-7: JavaScript code for selecting houses

```
Template.selectHouse.helpers({
    houseNameId: function () {#A
        return HousesCollection.find({}, {});
    },
    isSelected: function () {#B
        return Session.equals('selectedHouse', this._id) ? 'selected' : '';
    }
});
Template.selectHouse.events = {
    'change #selectHouse': function (evt) {#C
        Session.set("selectedHouse", evt.currentTarget.value);
    }
};
```

#A returns all documents from the collection
#B returns selected if the _id for the currently processed house equals that stored inside the Session variable

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

#C remember to pass the event as an argument so the function can assign the selection value to the Session variable

You can test that everything works correctly by opening the JavaScript console inside a browser and selecting a value from the dropdown list. You can get and set values for the variable directly inside your console as well. If you change the value to a valid `_id` you can see that the dropdown list instantly updates itself due to the `isSelected` helper as you can see in Figure 4-6.

Figure 4-6: Getting and setting the Session variable via the JavaScript console

4.3.3 Creating a reactive context using Tracker.autorun

When working with JavaScript code you will often need to check for the value of a variable to better understand why an application behaves the way it does. Using `console.log()` to keep track of variable contents is one of the most important tools for debugging. Since we are dealing with reactive data sources we can also take advantage of computations to monitor the actual values of those source.

In section 4.1 we saw that besides `Templates` and `Blaze` there is a third context which enables reactive computations: `Tracker.autorun()`. Any function running inside such a block is automatically rerun whenever its dependencies change. Meteor automatically detects which data sources are used and sets up the necessary dependencies.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

We can keep track of the value for `Session.get("selectedHouse")` by putting it inside an `autorun`. We place this code at the very beginning of the `client.js` file, outside of any `Template` blocks (see Listing 4-8). Whenever we use the drop down list to select another value the console immediately prints the currently selected ID. If no house is selected it will print `undefined`.

Listing 4-8: Using Tracker.autorun to print a Session variable to the console

```
Tracker.autorun(function() {
    console.log("The selectedHouse ID is: " + Session.get("selectedHouse"));
});
```

As you can see the `Session` object is very simple to work with and can be extremely useful. It can be accessed from any part of the application and maintains its values even if you change source files and Meteor reloads your application (hot code pushes). If a user initiates a page refresh all data is lost though.

Keep in mind though that the contents of a Session object never leave the browser, so other clients or even the server may never access its contents. This is what Collections are used for instead. In order to display the data for a selected house we need to take a closer look at working with collections first.

4.4 Working with MongoDB Collections

Meteor comes with its own instance of MongoDB, an open source document-oriented NoSQL database. Each time you start your server with the `meteor run` command, a dedicated database server is also started, listening to connections on port `3001`. By default Meteor uses this instance as its database engine and stores all content inside a database called `meteor`. There is no need to define any database connection strings. However, you can use environment variables like `MONGO_URL` to point your Meteor server to another database instance. Chapter 12 goes into detail on how to use external databases instead of the default local one.

What is a document-oriented database?

MongoDB is a document-oriented NoSQL database. In contrast to relational (SQL) databases every entry is self-contained and not spread across multiple tables to specify relationships.

To better understand document-oriented databases it helps to consider all data you want to store to be written on a piece of paper ("a document"). If you want to keep track of all the houses you are house-sitting for, you would create one form per house, then write all the instructions down. The advantage of a document storage is that all relevant information is in single place. Just grab a single paper and you have all the info you need to take care of that house. The downside is, that if multiple houses have the same plants and it turns out that the red flowers in fact need 4 instead of 3 pots of water you would have to change the amount on every single piece of paper.

Every document is self-contained and may even have different information on it ("fields") which reflects that at some houses you take care of the plants while in others you might also need to feed

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

the cat. You do not write down any areas that you do not need to take action in, so even though two documents may be from the same collection, they do not necessarily contain the same fields:

```
Name: Stephan
Plants:
  - Color: Red
    Instructions: 3 pots/week
  - Color: White
    Instructions: water daily
Animals:
  - Name: Danbo
    Instructions: 1 carrot/day
```

```
Name: Manuel
Plants:
  - Color: Red
    Instructions: 3 pots/week
  - Color: Yellow
    Instructions: keep humid
```

If you have worked with SQL databases such as MySQL or Oracle previously, the following table translates common SQL terms to their document-oriented equivalents.

SQL term	Document-oriented term
database	database
Table	collection
Row	document
column	Field

If data is meant to be stored for an extended period of time or if it should be shared across clients, `Collections` are the way to do it. New database collections are declared using the `Mongo` object. The following statement makes the content from a MongoDB collection named `mycollection` available inside a Meteor application as `MyCollection`:

```
MyCollection = new Mongo.Collection("mycollection");
```

As the collection should be accessed from both the server and the client, we need to make sure this line is executed in both contexts, so it should not be wrapped inside any `isClient()` or `isServer()` blocks. Also note that it does not use a `var` declaration which would limit its scope to a single file.

NOTE Collection names in Meteor usually start with a capital letter and have plural names. If you want to be more explicit you can add `Collection` to their names to make your code more readable. Collections are best defined in one or more dedicated files accessible to both client and server.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

The fundamentals of working with a collection are based on the way MongoDB works, so if you are already familiar with the syntax to query Mongo databases you can re-use this knowledge with Meteor, even from inside the browser.

4.4.1 Querying Documents in MongoDB

Querying documents in MongoDB is done either using `find()` or `findOne()`. The first returns all matching documents, the latter only retrieves the first document that matches specified search criteria. These criteria are passed to the query as an object called *query document* or *selector*. If no selector is defined, then all documents match.

To find one document where the name is "Stephan" the selector would need to contain the search field (name) and the desired value ("Stephan"). Fields do not require quotes:

```
MyCollection.findOne({name: "Stephan"});
```

The `findOne()` operation is executed on the collection named `MyCollection`. It matches and returns the first document where the value of the field `name` equals "Stephan".

To find all documents that have instructions for watering a white plant requires a more advanced query. This time `find()` is executed, so all matching documents inside `MyCollection` are returned. The query document specifies that inside a `plants` key must be another key called `color`. The term `$exists: 1` translates into the field being present inside all matches.

```
Collection.find({"plants.color" : {$exists: 1 } });
```

To check whether the field contains a specific value `$in` may be used instead of `$exists`. If we wanted to find all documents that contain a plant with a `color` attribute of "White" the following query would be used:

```
Collection.find({"plants.color" : {$in: ["White"] } });
```

In addition to search criteria a second object may be passed to the query operations. It is called the *projection* and it may be used to limit the fields that should be returned, change the sort order, or apply any kind of operation on the search results before actually returning them. Projections may be used with or without a query document. If no search criteria are needed, an empty selector is passed to the `find()` function.

The following query only returns the `name` field and value from each document. The term `name:1` can be read as "set the field name to visible" since `1` represents true. Using `0` you define fields to exclude from retrieval.

```
Collection.find({},{name:1})
```

As you can see from these examples using collection queries is quite different from using SQL. It is helpful to keep in mind that you are not dealing with tables and rows, but documents that behave like objects.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

NOTE More details on using MongoDB can be found in the official documentation at http://docs.mongodb.org/.

4.4.2 Working With Meteor Collections

At the most basic level we can store data to a collection as a document and search and retrieve one or more documents when we need to display data. Let's start by populating the houses collection.

The most important functions when working with Collections are listed in Table 4-3.

Table 4-3: Overview of most important functions when working with Collections

Function	Use case	Return value
`Collection.find()`	Finds all documents that match the selector	Cursor
`Collection.findOne()`	Finds the first document that matches selector and projection criteria	Object
`Collection.insert()`	Inserts a document in a collection	String (document _id)
`Collection.update()`	Modifies one or more documents in a collection	Number of documents affected
`Collection.upsert()`	Modifies one or more documents in a collection or inserts a new document if no match is found	Object
`Collection.remove()`	Removes documents from a collection	Object

INTRODUCING CURSORS - THE DIFFERENCE BETWEEN FIND() AND FINDONE()

A single document may be retrieved from a collection by using `findOne()`. This returns a JavaScript object that can be dealt with just like any other object. Retrieving multiple documents using the `find()` function does not return any documents, but a *cursor*. Strictly speaking the cursor is a reactive data source, not the collection.

You can think of a cursor as being the actual query that eventually gets executed against the database. Cursors allow to send data in batches. When dealing with large datasets it is not efficient to always return all documents from a query, but allow to iterate over results and send them in batches.

For now we are not going to deal with cursors directly as Meteor is capable of knowing how to handle the results from a `Collection.find()` just fine. We will revisit this topic once we discuss more advanced use cases in chapter 9.

4.4.3 Initializing a collection

For each house we need to take care of we will have one document inside the database. It will include the name and plants to take care of. In order to have a better traceability throughout

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

the application code a verbose name helps keeping track of the collection object. We have added the necessary code in section 4.2.2 already, so there is no need to add this line again. `HousesCollection` will provide an interface to all database entries:

```
HousesCollection = new Mongo.Collection("houses");
```

There is no need to create any data structures inside a Collection. When the first entry is added to a collection Meteor automatically creates a collection inside the database if it is not yet present.

The server code we defined during the set up of our application in section 4.2.2 takes care of both creating a database collection and filling it with data. As we have seen in chapter 2 with the fridge example we may also use the browser console to add new data. We will add security mechanisms that prohibit this when we discuss publications and methods in chapter 6. For now we will focus on adding functionality instead of getting ready for production.

An important side effect of using fixtures is that you have an idea of what the data structure looks like. Since we are dealing with a NoSQL database each document may have a completely different structure and it is good to have reference entries at hand. Although there may be other fields (such as animals or kids) the expected fields for each house document are shown in Table 4-4. We are only taking care of plants in this example.

Table 4-4: Expected fields for the houses collection

Field name	contains	notes
`_id`	Unique ID for each house, string	Automatically assigned by MongoDB
`name`	Display name for each house, string	
`lastvisit`	Timestamp for last action, `Date`	Generated by application logic
`plants`	Household plants to take care of, array of objects	
`plants.color`	Unique plant color per house, string	No database constraint, uniqueness must be ensured by application logic
`plants.instructions`	Watering instructions for a plant, string	

4.4.4 Querying collections

It is very convenient to rely on everything being reactive in a development environment, but with larger datasets having every function react on any data change will have visible impact on performance. For the drop down list of houses it does not matter whether someone adds or removes plants to a document, so we are going to limit reactivity to the `name` and `_id` fields only.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

RETURNING SPECIFIC FIELDS ONLY

In the previous section we defined the `houseNameId` helper to return everything from `HousesCollection`, now we will limit the return value to only the fields `name` and `_id` as shown in Listing 4-9. The selector object remains empty, but we pass a second object with a `fields` property to the cursor. Inside it we may set individual fields to 1, which means they will be returned. Alternatively we could set fields to 0 to exclude them from being returned.

Listing 4-9: Limiting the fields returned to the drop down list

```
Template.selectHouse.helpers({
    houseNameId: function () {
        return HousesCollection.find({}, {fields: {name: 1, _id: 1} });
    },
    //...
});
```

RETURNING A FULL DOCUMENT

Once the user made a selection from the drop down list we want to show the house document in full. In this case we obviously need to return all the fields inside the document. We are going to create another helper that will return exactly one document with all fields based on the value of the session variable `selectedHouse`: This helper is going to be used inside a template called `showHouse`, so we need to add this block to the `client.js` file as in Listing 4-10.

Listing 4-10: Returning a database document to a template based on ID

```
Template.showHouse.helpers({
    house: function () {
        return HousesCollection.findOne({
            _id: Session.get("selectedHouse")
        });
    }
});
```

This time we are using a query document but pass no options to the `findOne()` function. As a result we will get the full document stored inside MongoDB as an object. This object can be accessed like any other in JavaScript.

4.4.5 Display collection data in a template

Meteor templates make it easy to access data inside an object returned by a helper. All we need is a double-braced tag referencing the name of the object and the specific field to be displayed. In order to show the name stored inside the document returned by house we can simply use `{{house.name}}`. However, to eliminate the need to prefix each object property with `house`. a `#with` block that sets the data context is useful because it makes our template more readable.

To enhance the user experience we add a conditional to check whether a valid selection has been made. If not the template should ask the user to make a selection.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Each plant should be displayed with information regarding their color, instructions for watering and a button to mark a plant as watered. We will add

For displaying the document's contents we can either put everything inside the `showHouse` template or use a dedicated sub-template. Using a subtemplate gives us greater flexibility and manageability in the future when we might want to support pets, kids, or cleaning.

When including a template it inherits the parent's data context. That way we do not have to define a new helper and require less code. In the example in Listing 4-11 you can see that the data context is not taken from the parent directly, but is further specified by `{{#each plants}}`. From the subtemplate the current `plant` object from the loop is the defined context. We can still use the same expressions as in we would in the parent, only remember that now we are at a deeper level inside house. To access parent attributes such as the house ID the `../` notation is used.

Listing 4-11: Template code for showing a house with all plants

```
<template name="showHouse">
    {{#with house}} #A
        <h2>Taking care of {{name}}'s house</h2> #B
        {{#each plants }} #C
            {{> plantDetails }} #D
        {{/each}}
    {{else}}
        You need to select a house.
    {{/with}}
</template>

<template name="plantDetails"> #D
    <p><strong>Plant color:</strong>  {{color}}</p>
    <p><strong>Instructions:</strong> {{instructions}}
        <button class="water" data-id="{{../_id}}-{{color}}"
      {{isWatered}}>Done</button>
    </p>
</template>
```

#A setting the data context for the template explicitly
#B Each database field is a property of the house object and can be accessed using dot notation
#C each further narrows the data context for the subtemplate
#D a subtemplate plantsDetails is included

Adding both templates to the application we can now select a house and view its contents, regardless of the number of plants.

EVENT MAPS AND DATA ASSOCIATION

Besides simply presenting the data from a document we also want to be able to mark a plant as watered. By pressing a button an event shall be triggered that sets a plant to `watered`. However, if we switch between houses we want to keep the state of a plant, so we are going to use a Session variable again. This time we cannot set the name in the application code because we do not know how many plants each house has. Therefore we are going to dynamically create an id for each plant, consisting of the document `_id` in combination with

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

the `color` attribute. We can do so because we defined `color` to be a unique identifier for each plant inside a house.

It is common practice to use the HTML attribute `data-id` to pass unique element IDs to the application code. The event map watches for a click on any button with the class `water` and stores the value of `data-id` for the currently clicked button consisting of both document ID and `color` value. The event map shown in Listing 4-12 can simply use `data-id` without having to create the compound ID itself.

Once the button is clicked, a session variable with the new compound ID value is set to true. There is no need to set a default value for the session variable. Remember, technically it is a key-value store inside the `Session` object, hence we can add new keys at any time.

Listing 4-12: Event map for watering the plants

```
Template.plantDetails.events({
    'click button.water': function (evt) {
        var plantId = $(evt.currentTarget).attr('data-id'); #A
        Session.set(plantId, true);
    }
});
```

#A data-id contains a unique ID for each plant

Whenever a plant is watered we want to disable the button as an indicator that this plant does not require any more attention. We will do this with a similar helper that we used to determine the currently selected house in the dropdown list (Listing 4-13). Because we are using globally available `Session` variables we can water the red plant in Manuel's house, switch to Stephan's house and back to Manuel and still find the button disabled.

Unless the user forces a page reload all contents stored inside a `Session` object will remain available globally throughout an application.

Listing 4-13: Template helper for disabling the Done button

```
Template.plantDetails.helpers({
    isWatered: function () {
        var plantId = Session.get("selectedHouse") + '-' + this.color;
        return Session.get(plantId) ? 'disabled' : '';
    }
});
```

4.4.6 Updating data in a collection

Updating data inside a collection is done via the `Collection.update()` function. Although it can modify one or more documents at the same time when invoked on the server, the `update` function is limited to manipulating a single document based on its `_id` when running on the client side. This is to avoid accidental bulk operations that may block the server for all users. If you need to update more than a single document at a time you can do so using server side methods which we will look at in chapter 6.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

When using `update()` you need to specify which documents to update, how to update them, and optionally define options as well as a callback (using an error as the first return value and the number of affected documents as a second value):

```
Collection.update(selector, modifier, options, callback);
```

There are only two options available, both of which are Booleans:

- `multi` – default: false

  if set to true all matching documents are updated, otherwise only the first match

- `upsert` – default: false

  if set to true it inserts a new document if no matching documents are found

In order to call `update()` from the client we need to provide a single `_id` as the first argument. This is part of the first argument, which is called the selector. It can either be an object with an `_id` attribute or a string that holds a valid document id. The modifier uses standard Mongo syntax to define how to modify present data. Table 4-5 gives an overview of some of the most common operators. The following command updates a document with an `_id` of `12345` and sets the field name to `Updated Name`:

```
Collection.update({_id: "12345"}, {$set: {name: "Updated Name"}});
```

NOTE Not all functionality available in MongoDB is available in Minimongo, the client implementation. You can check the `NOTES` file inside the `minimongo` package for an overview of the current limitations.

Table 4-5 Overview of frequently used update modifiers for collections

Update operator	Description
`$inc`	Increments the value of a field by the specified amount
`$set`	Sets the value of a field in the document
`$unset`	Removes a field from the document
`$rename`	Renames a field from a document
`$addtoSet`	Adds elements to an array if they do not already exist
`$push`	Adds an item to an array
`$pull`	Removes all array elements that match a specified query.

TRIGGERING UPDATES VIA EVENTS

So far we are only using Session variables to store data, which means all data is gone once the browser window is closed or the user forces a reload. To keep track when each house was last visited we will extend the event handler to actually store the current date to the database.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

A field lastvisited will be used inside each house document. Again, we do not have to define the database structure before we can add data, we may simply add a new field to an existing document.

Inside the client.js file we will extend the existing event handler for the plantDetails template with two lines (Listing 4-14). A new variable called lastvisit will be assigned a current timestamp. It will be used in the update() function for the current document[10]. Watch out as we now have to deal with two IDs: one for the plant and another for the house. We will leave plantId untouched and for the document ID we will use the selectedHouse session variable to specify the ID in the update statement.

Listing 4-14: Extended event map for adding a last visited date

```
Template.plantDetails.events({
    'click button.water': function (evt) {
        //...
        var lastvisit = new Date();#A
        HousesCollection.update({_id: Session.get("selectedHouse")},{$set:
      {lastvisit: lastvisit }});
    }
});
```

#A lastvisit contains a current timestamp
#B the lastvisit field inside the currently selected document is set to the current timestamp

Every time any of the plants inside a house is watered now (i.e. the Done button is clicked) the lastvisit field gets updated. To verify that this update happens we extend the showHouse template to display the value of lastvisit as shown in Listing 4-15.

Listing 4-15: Adding the lastvisit timestamp to the showHouse template

```
<template name="showHouse">
   {{#with house}}
        //...
        <p>Last visit: {{lastvisit}}</p>#A
        //...
    {{else}}
   //...
</template>
```

#A Add this line to show the timestamp stored as lastvisit

Clicking the button for either plant will automatically disable itself and update the timestamp in the database as shown in Figure 4-7.

[10] The browser implementation Minimongo does not support the full feature set provided by Mongo itself. Therefore using the $currentDate modifier is not possible on the client.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Due to latency compensation the value inside the local Minimongo instance is updated first and the result is shown immediately. At the same time the update is sent to the server, where the last visit date will be persisted.

Figure 4-7: The timestamp gets updated on screen and inside the database by each click on a Done button

4.4.7 Inserting new data into a collection

As business grows new houses are added and new documents must be added to a collection. Generally adding documents is done using the `insert()` function. Each field of the document must be specified individually while the value for `_id` is left out as it is assigned automatically by the database:

```
Collection.insert({field: "value"});
```

Adding a new house to our collection requires a new template as well as another event map to trigger the insertion. In section 4.2.2 we already set up a template called `houseForm`, which we can extend.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Listing 4-16 shows the `houseForm` template, which basically consists of a view for a house document that displays all editable fields inside a form. To keep things simple we will limit the form to only add a single plant to a house[11]. All input fields are going to be identified by their ID.

Listing 4-16: Adding new houses using a form

```
<template name="houseForm">
    <h3>Add a house</h3>
    <form id="houseForm">
        Name <input id="house-name" type="text" placeholder="Name"><br/>
        Plant<br/>
        Color <input id="plant-color" type="text">
        Instructions <input id="plant-instructions" type="text">
        <br/>
        <br/>
        <button id="saveHouse">Save House</button>
    </form>
</template>
```

Create an event map for handling the form (Listing 4-17). When the button for the form is clicked the default behavior is to send a form and reload a page. Because we are implementing our own functionality for the button we need to suppress the default behavior by calling the `preventDefault()` method on the event. Using jQuery we retrieve all values for the input fields and put them in local variables.

The final line of code does two things. First it inserts a new document with field values from the input fields into the `HousesCollection`. That returns the ID of the new document which is assigned to the `selectedHouse` session variable. That way the entire page instantly displays the new house in the selection and detail view.

Listing 4-17: Event map for adding a new house

```
Template.houseForm.events({
    'click button#saveHouse': function (evt) {
        evt.preventDefault(); #A
        var houseName = $("input[id=house-name]").val(); #B
        var plantColor = $("input[id=plant-color] ").val(); #B
        var plantInstructions = $("input[id=plant-instructions]").val(); #B
        Session.set("selectedHouse", HousesCollection.insert( #C
            {
                name: houseName,
                plants: [{color: plantColor, instructions: plantInstructions}]
            }
        ));
    }
});
```

#A prevent sending the form and reloading the page

[11] Do not worry, we will soon create a more powerful way to add as many plants as you like.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

#B retrieve the input field values using jQuery
#C insert a new document and assign the return value to the selectedHouse variable to immediately show the contents of the new document

This is only a very simple way to add houses that has several shortcomings. The form does not allow the user to enter multiple plants for a house. Also the form is only capable of adding new houses while we could easily re-use it to allow editing exisiting houses as well. We will enhance this example in the upcoming section to improve on these shortcomings and refactor some of the code for better efficiency. But first let's but the finishing touches on basic CRUD.

4.4.8 Removing data from a collection

The final step for full CRUD[12] functionality is removing an entry from the database. The relevant function is called `remove()`:

```
Collection.remove(id);
```

Similar to the `update()` method `delete()` requires a unique ID to know which document to remove. Multiple documents may only be deleted from the server, which we will look at in a bit when we discuss methods.

For the HTML all we have to do to allow for houses to be deleted app is to add a button to the `showHouse` template.

When the button is clicked we shall identify the house by its ID. There are at least three ways to do this in Meteor:

1. If the HTML button had a `data-id` attribute we could query its contents in the same way we determined the plant ID.
2. If the currently selected house ID is stored inside the `Session` object we can use it as we did with the "Done" button used for updating houses.
3. If the template has a data context that provides the ID we access it directly.

We already covered the first two. This time, the ID is already part of the data context in which the `click` event takes place. That means we can simplify the way we determine which document to delete and chose option #3 to go ahead. Listing 4-18 shows how to add the button to the template.

NOTE Although it is not needed you should consider adding a `data-id` attribute to the `delete` button to provide some traceability to make debugging easier.

[12] Create, Read, Update, Delete – a common name for basic operations on persistent data.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Listing 4-18: Adding a delete button to remove houses from the database

```
<template name="showHouse">
    ...
        <button id="delete">Delete this house</button> #A
    {{else}}
    ...
</template>
```

#A Add the button just before the else tag

As there is no event map yet for the showHouse template clicking the buttons does not do anything yet. We are going to create a new event map inside `client.js`. Listing 4-19 shows the code that essentially performs a `Collection.remove()`.We will wrap it inside a confirmation dialog to avoid users to accidentally delete houses.

Notice that we do not have to capture the event and read out the `data-id` attribute like we did when we set a plant to a watered state? We can directly access all information contained in the currently selected house document, including `_id`.

Listing 4-19: Event map for deleting a house

```
Template.showHouse.events({
    'click button#delete': function (evt) {
        var house = this; #A
        var deleteConfirmation = confirm("Really delete this house?"); #B
        if (deleteConfirmation) {
            HousesCollection.remove(house._id); #C
        }
    }
});
```

#A this is the current data context: the selected house document
#B Show a confirmation dialog before actually removing the document
#C Removes the document from the collection on both client and server

Congratulations, you are now able to perform all basic data operations using Meteor! Roughly 50 lines of HTML code and about 100 lines of JavaScript are all it takes. Take a deep breath and pat yourself on the shoulder.

By now you are familiar with using the `Session` object and various ways to store, manipulate, and retrieve data from a MongoDB database using `Collections`. While the main functionality for this application is present, we still have some work to do. Depending on your own preferences you could start securing the application now, which would require users and accounts. A second option to go from here would be to add routing capabilities to the application so that the form for adding new houses is not shown on the same page as house details. The required techniques for both are covered in the corresponding chapters. However, we should address some shortcomings in regards to our application's functionality first: Adding and removing a custom number of plants to existing and new houses.

In the next chapter we will step things up a bit by taking a more reactive approach to overcome the current limitations of our application by leveraging Meteor's core principles of

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

reactivity instead of falling into old habits of performing complex DOM manipulations using jQuery.

4.5 Summary

- Reactive data sources are aware of dependent computations
- Reactive contexts allow functions to be re-run when data-sources change
- `Tracker.autorun()` can turn any function into a reactive computation by providing a reactive context
- Short-lived data can be stored inside the key value store of the `Session` object
- Persistent data is stored in MongoDB `Collections`
- Meteor automatically publishes all `Collections` to every client unless the `autopublish` package is removed
- Both `autopublish` and `insecure` must be removed before putting an application into production

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

5

Fully reactive editing

This chapter covers

- Two-way data binding
- Editing forms with an intermediary edit object
- The `ReactiveVar` package

The most common approach to front-end development is manual DOM manipulation. In most frameworks data is retrieved from a backend and needs to be inserted into a DOM somehow. If a new database entry is sent to the client a library like jQuery is utilized to add a new `li` element or table row. While this approach is simple and straight forward, it tends to make code overly complex and forces you to explicitly add changes for all occurences inside a page. That said, typical front-end coding is very DOM-heavy while Meteor simplifies things by following a data-centric approach powered by reactivity.

In the last chapter we created a fully functional application that allows users to select, view, add, and delete houses. Because we have used reactive data sources we did not have to manipulate the DOM tree ourselves, for example adding a new `option` element to the dropdown list when we created a new house – Meteor's reactivity took care of it for us. In this chapter we will enhance the existing application to add and remove any number of plants to a house.

At the end of this chapter you will be able to work with `ReactiveVar` as an alternative to `Session`. Because `ReactiveVar` combines variable scope to the magical powers of `Session` it allows you to render the same template twice, with different data contexts. As a result we will achieve a similar effect as two-way data binding in frameworks like Angular or Ember.

We are also going to minimize all DOM interaction to a minimum and explain how to take advantage of a data-driven approach using an *edit object*. This edit object is going to be an intermediary layer to represent a single house and all unsaved edits.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

5.1 Data-centric applications vs DOM manipulation

Most front-end engineers typically think about the DOM first. They consider how and where to place elements, how to serialize data and send it to REST interfaces. jQuery is a handy tool for these tasks and it is synonymous for many people with front-end development and DOM manipulation.

When you are using different languages and frameworks on the server and the client you often find yourself faced with the problem of how to map data received from the server to the actual view. If there is no easy integration available, there is always jQuery and DOM manipulation. Unfortunately changing the DOM manually becomes quite messy, even for small tasks.

Consider a simple form to add and remove plants to each of our houses. One button is used to remove an existing plant, another will add another form `fieldset` to allow entering plant details. As you can see in Figure 5-1, removing a DOM node is not very complicated. A click on a `button` may remove a form's `fieldset` (the parent of the clicked button). It does not affect any other occurences where the plant may be displayed, so you might need additional `remove()` operations on the DOM, for example when you offer a preview of the house document in the same view. Adding a new plant involves a lot more complexity – you need to inject the entire HTML necessary to define a plant. Not only is this line very long, it is also very fragile should you ever decide to adjust the form code by using different classes or adding more fields.

Figure 5-1: Using a DOM-centric approach for adding and removing plants to a form

DOM manipulation is nice for small front-end hacks, but when writing larger applications it will be a challenge to maintain that kind of code. Here is where Meteor's seamless integration between front-end and backend comes in handy. With Meteor we can directly work with data and let the framework do all the DOM mapping for us.

Each house is stored as a document or object inside the database already. Why not leave it at that and work with the object rather than worry about the DOM? With Meteor we can do

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

this easily so that not only we do not have to worry about mapping content with jQuery, but we get two-way-data binding for free (see the info box for more information on data binding). The HTML form will just be used as a different representational view of the house object. Clicking buttons will not trigger DOM-operations but change an object as shown in Figure 5-2.

Figure 5-2: Using a data-centric approach for adding and removing plants to an object

As you can see there is not only less code involved, it also does a much better job at separating data from presentation. As a JavaScript developer you are already familiar with manipulating objects. All that is left is walk through how Meteor can pass data as an object to the front-end.

Two-way data binding

Using a traditional server-side language such as PHP or Java the application retrieves data from a database, renders it into HTML, and sends it to the browser for display. It is easily possible for the server code to poll the database for changes, re-render, and send an updated view to the browser, e.g. using Ajax. If the data is displayed as a form, a user may perform as many changes as they like, it will not affect the actual database content. In fact, form data must be posted first to the server to process and update the data. The server will then store the data in the database, retrieve it, and send an updated view to the browser. Because data flows from one side to another this is sometimes referred to as 1-way data binding.

One-way Data Binding

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

With two-way data binding the flow is continuous. Every change in a view flows back into the actual data and vice versa. That means if a form field is updated by a user, the underlying data provider is changed as well. This behavior can be observed if form data is also displayed elsewhere on the site because it would update instantly.

Two-way Data Binding

Using a reactive data source is an easy way to implement two-way data binding with Meteor. There is no code required to update any views if data changes because it is taken care of by the Blaze engine.

5.2 Creating a reactive edit object

In the previous chapter we have used two different types of reactive data sources: `Session` and `Collection`. Before we implement any reactive editing we need to decide which to use for holding house data. We could easily use the `Collection` object as all houses are already stored inside. That way we could get rid of much of the code and always have the latest editing state inside the database. There are two disadvantages of using `Collection`:

1. Each change, i.e. keystroke, must initiate a database write operation, which puts stress on both the network as well as the server. A single client will do quite well, but if we expect thousands of house sitters to be updating their client's houses we can foresee a lot of load that we should best avoid.
2. There is no rollback if changes were made accidentally. We still want the user to press a button to persist data to avoid adding complex undo-routines.

We need an intermediary data storage that only holds house data for the duration of the editing process and hands it over to a `Collection` once the *Save* button is clicked. We already know that `Session` can be used to hold such data, but it also comes with a disadvantage: It lives in the global namespace. To avoid polluting the global namespace and making the application more flexible we will use a new data source: the `ReactiveVar`.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

5.2.1 ReactiveVar

The `ReactiveVar` package is part of Meteor core but it needs to be added manually. Since we require it as a substitute for `Session` add it via the command line:

```
$ meteor add reactive-var
```

Both `ReactiveVar` and `Session` use `get()` and `set()` functions, but `ReactiveVar` does not pollute the global namespace and can be limited to a local scope. We can re-use it with different values for the same template. Figure 5.3 illustrates how data flows between the database and the browser view.

Figure 5-3: House documents are put inside a reactive object for easier editing

Just like `Session` `ReactiveVar` is basically a key-value storage it may store whole objects as values. Updating objects inside the `ReactiveVar` container requires the usage of `set()`. To avoid cluttered and long code lines we will put the object's contents in a regular JavaScript object whenever we need to update data and then store the non-reactive object back into `ReactiveVar`. At the beginning of `client.js` we are going to add declarations for an empty house object, the temporary JavaScript object `editObject` and set the default value for a new `reactiveHouseObject` `ReactiveVar` object to an `emptyHouse` object as in Listing 5-1. The syntax for initiating a ReactiveVar variable is

```
myReactiveVar = new ReactiveVar(defaultValue);
```

Unlike from using `Session` there is no `setDefault()` function, but a default value may optionally be passed to `ReactiveVar` when a new instance is declared. The command will create a new variable called `myReactiveVar` that is in fact a `ReactiveVar` object. It holds whatever `defaultValue` is defined to be – string, integer, array, or object.

Listing 5-1: Setting up an edit object

```
var emptyHouse = {name: "", plants: []};
var editObject;
reactiveHouseObject = new ReactiveVar(emptyHouse);
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Within the house-sitter application we will use the `ReactiveVar` object as a middle layer. No template should directly use the result of `Collection.find()` calls anymore. When selecting a house we will set the ID of the currently edited house as well as populate the new `reactiveHouseObject` variable, which we use to temporarily store the edit object. We still need to store the selected ID so all existing code remains untouched. We need to extend the event maps as shown in Listing 5-2.

Listing 5-2: Assigning house data to the edit object via the dropdown list

```
Template.selectHouse.events = {
    'change #selectHouse': function (evt) {
        Session.set("selectedHouse", evt.currentTarget.value);
        reactiveHouseObject.set( #A
            HousesCollection.findOne({_id: Session.get('selectedHouse')})
            || emptyHouse); #B
    }
};
```

#A set the value for reactiveHouseObject to the result of the findOne command
#B if no document was found set reactiveHouseObject to the emptyHouse object

First you will notice that are about to move the database lookup from the `showHouse` template helper to the dropdown list. The result of the database lookup is directly put into `reactiveHouseObject`. That way we can take advantage of the reactive context provided to instantly update the house details according to form entries.

In case the empty option is selected from the dropdown list the database lookup will not return an object. In that case, the `emptyHouse` object is assigned to `reactiveHouseObject`.

Now that we want to add editing functionality to the form we need the full house details in two places: The `showHouse` template and the `houseForm`. It does not make sense to create another helper that provides `find()` functionality and returns an entire house. It is more efficient to use a single global helper that may be used in any template.

`Template.registerHelper()` allows creating global helpers, so we use it to make `{{selectedHouse}}` available from all templates in the application (Listing 5-3). Note that the helper does not perform a database lookup as before, it merely returns the contents of `reactiveHouseObject.get()`.

Listing 5-3: Global helper to return the edit object

```
Template.registerHelper('selectedHouse', function () {
    return reactiveHouseObject.get();
});
```

Next we can update the `showHouse` template to use the global helper instead of the template-specific house. Find the `{{#with house}}` tag and change it as shown in Listing 5-4.

Listing 5-4: Using the global helper selectedHouse to display the edit object

```
<template name="showHouse">
    {{#with selectedHouse}}
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
        <h2>Taking care of {{name}}'s house</h2>
        ...
    {{/with}}
</template>
```

To conclude the showHouse refactoring we can remove the code for `Template.showHouse.helpers` completely.

If you use the dropdown list to select a house, name and plant details should now be shown just like before, only this time coming from a `ReactiveVar` object. The `houseForm`, however, does not yet show any data for a selected House as it does not yet have a data context. We can either use the same approach as before (adding `{{#with selectedHouse}}` to the template) or provide the data context directly when including the template in the body. The latter only requires adding a single word and not two more lines to the code we will change the partial to

```
{{> houseForm selectedHouse }}
```

5.3 Enhancing the houseForm to display existing data

Still, no data shows in the `input` fields, we must add `value` attributes to each. Also we still need to allow for multiple plants to be displayed. Listing 5-5 shows the updated code.

For better visual organization we introduce `fieldsets` around each group of inputs. That way we can easily keep track of which instructions belong to which plant. Also we are going to split the templates in the same way we did for the `showHouse` template. A new `plantFieldset` template will be used for each plant. For removing plants we will add buttons to each plants' `fieldset` and place another button for adding new plants just before the save button of the form (Figure 5-4).

Figure 5-4: Updated form using fieldsets and buttons to add/remove plants

Finally in order to show the existing data inside the `input` fields each `input` gets a new attribute `value` with an associated expression. As a finishing touch we change the generic "Add a house" headline to mention current `name` instead.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Listing 5-5: Template code for displaying multiple plants in the HTML form

```
<template name="houseForm">
    <h3>Edit {{name}}'s house</h3> #A

    <form id="houseForm">
        <fieldset id="house-name"> #B
            Name <input class="name" type="text" placeholder="Name"
      value="{{name}}"><br/> #C
        </fieldset>
        <label>Plants</label>
        {{#each plants}}
            {{> plantFieldset}}
        {{/each}}
        <button class="addPlant">Add Plant</button> #D
        <br/>
        <button id="saveHouse" data-id="{{_id}}">Save House</button>
    </form>
</template>

<template name="plantFieldset">
    <fieldset> #B
        Color <input class="color" type="text" value="{{color}}"> #C
        Instructions <input class="instructions" type="text"
      value="{{instructions}}"> #C
        <button class="removePlant">Remove Plant</button> #D
    </fieldset>
</template>
```

#A Change the generic headline to mention the name
#B Use fieldsets to group inputs
#C Added value attributes show existing data
#D Buttons for adding and removing plants

Let's revisit the code. As we are aiming for two-way data binding there must be a way for the application code to map each attribute in the `ReactiveVar` object to a unique HTML element. We can verify that this is the case because each data attribute such as name and all plant details are correctly shown. There must also be a way to uniquely tie HTML elements back to the data object, which is missing so far. Apart from the input values each plant `fieldset` looks exactly the same, so we need to enrich it with an `index` value to allow for reverse mapping.

ADDING ARRAY INDEX INFORMATION TO AN #EACH LOOP

Since we are dealing with array contents the logical solution is to use an array index to uniquely identify plants inside our form. Unfortunately the Spacebars templating language does not offer an easy way to directly get the index of an element inside an each block, so we need to define our own function.

We are going to use a global helper, which we will call `withIndex`. In our case it will return the actual `plants` array with each `plant` as an object and enhance it with a new `index` attribute using the `map` function. As you can see in Listing 5-6 we are using the Underscore.js library to keep the required code concise. Underscore.js ships with Meteor and does not need to be added manually.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Listing 5-6: Using underscore.js to add index attribute to array objects

```
Template.registerHelper('withIndex', function (list) {
    var withIndex = _.map(list, function (v, i) {#A
        v.index = i;
        return v;
    });
    return withIndex;
});
```

#A Underscore's map function is used to iterate over a list

We can pass any array of objects to the `withIndex` function and use `{{index}}` inside a template to return the array position of an object inside a loop. That way we can uniquely identify each cycle and element created by an `#each` block. Inside the `houseForm` template adjust the `#each` block tag to pass plants to the `withIndex` function like this:

Listing 5-7: Adding an index to plants using template helpers

```
<template name="houseForm">
    ...
    <form id="houseForm">
        ...
        {{#each withIndex plants}} #A
            {{> plantFieldset}}
        {{/each}}
        ...
    </form>
</template>
```

#A Adjust this line, the rest remains as is

Also we need to enhance the `plantFieldset` template to add an `index` for each field (see Listing 5-7). We are going to use an attribute `data-index` to store it for each input, the `fieldset` itself will use `id` instead.

There is no more need for a compound ID mashed up of house ID and plant color, we can uniquely identify each plant by its position in the `plants` array of the current document by enhancing the template as shown in Listing 5-8.

Listing 5-8: Adding index information to the plantFieldset template

```
<template name="plantFieldset">
    <fieldset id="plant-{{index}}">
        Color <input class="color" type="text" value="{{color}}" data-
      index="{{index}}">
        Instructions <input class="instructions" type="text" value="{{instructions}}"
      data-index="{{index}}">
        <button class="removePlant">Remove Plant</button>
    </fieldset>
</template>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

WORKING WITH THE EDIT OBJECT

Have a look at Figure 5-5 to understand how working with our edit object works. We have a `ReactiveVar` object and a regular JavaScript object. `ReactiveVar` holds a single house document. It gets assigned inside the helper for the dropdown selection. `ReactiveVar` creates a reactive context, which is great for updating views, but the wrapper makes manipulating the content much harder than using plain JavaScript objects. For that reason we need to use a dedicated `editObject` that holds the contents of `reactiveHouseObject` for easier editing. Once the edit process is finished, we will assign the updated object back to `reactiveHouseObject`.

Figure 5-5: Working with the edit object

Let's look at the code. We begin with the event map for the `plantFieldset` template, as there is only a single button here. It is used to remove plants. In order to determine which plant to remove from the edit object we can look at the fieldset's `id` attribute. We can take the content from the data-id attribute and put it inside a variable called `thisPlantIndex`. The principle is the same as the one used for the "Done" button to indicate a plant is watered.

To make updates to the `reactiveHouseObject` object stored inside `ReactiveVar` we copy it to a local variable `editObject`. It is now a regular JavaScript object and we can perform all manipulation on it. The `splice()` function allows us to remove one object from the `plants` array at the index position retrieved from the template. Then we assign the content from the edit object back into the `ReactiveVar` object. Listing 5-9 shows how to do each step. Insert the code and then open your browser and check what happens when you click on the remove button for a plant.

Listing 5-9: Event map for removing plants from a house

```
Template.plantFieldset.events({
    'click button.removePlant': function (evt) {
        evt.preventDefault();
        var thisPlantIndex = $(evt.currentTarget).parent().attr("id").split('-')[1];
      #A
        editObject = reactiveHouseObject.get(); #B
        editObject.plants.splice(thisPlantIndex, 1); #C
        reactiveHouseObject.set(editObject); #D
    }
});
```

#A Query the array index from the fieldset
#B Create a local copy of the edit object

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

#C perform updates to the local edit object
#D store the local edit object back to ReactiveVar

As soon as the ReactiveVar object is updated, Meteor's reactivity is going to kick in. All instances where plants is shown will be updated immediately. That also affects the showHouse template. Removing a plant from the form will also – and instantly – remove it from the detail view. As it is an unsaved change only the current user will be able to see this. Any other user will see the state of the house from the database and still see the plant that was just removed.

That brings an undesirable side-effect: A user cannot tell whether they see the house as it is stored inside the database or just a local working copy.

HIGHLIGHTING UNSAVED CHANGES TO THE USER

There is a simple solution to letting the user know they are looking at a local working copy instead of the current database values – we can display a status notification bar everytime as long as the editObject is marked as edited. Let's start with the JavaScript code. We need a flag that symbolizes whether the object has been edited. Just before we store the contents of editObject back into ReactiveVar we add one line to set an edited flag:

```
editObject.edited = true;
```

If we place this inside a function we can reuse this technique later whenever we change the edit object, e.g. by adding plants or changing their colors. Listing 5-10 shows the storeReactiveHouseObject function that will set the edited flag to true before storing reactiveHouseObject back to ReactiveVar. Instead of calling set() on reactiveHouseObject you need to call storeReactiveHouseObject(editObject) with the object that should be stored inside the reactive context as the first argument. For our form this is always editObject.

Listing 5-10: Function for setting the edited flag and storing editObject to ReactiveVar

```
storeReactiveHouseObject = function(obj){
    obj.edited = true;
    reactiveHouseObject.set(obj);
};
```

Once the function from listing 5-10 is available you can change the event map for plantFieldset to use the function instead of calling set() directly (see Listing 5-11).

Listing 5-11: plantFieldset using a function to store the editObject

```
Template.plantFieldset.events({
    'click button.removePlant': function (evt) {
        //...
        storeReactiveHouseObject(editObject);
    }
});
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Inside the showHouse template we can now add a conditional statement that shows a message to the user if the edited property is true. We will also make sure that each time we store the object back into ReactiveVar to set the edited flag to true by calling storeReactiveHouseObject() whenever we are done working with the object.

Listing 5-12: Adding an edited message to the template

```
<template name="showHouse">
    {{#with selectedHouse}}
        <h2>Taking care of {{name}}'s house</h2>
        {{#if edited}}
            <p class="warning">There are unsaved changes!</p>
        {{/if}}
        ...
    {{/with}}
</template>
```

5.4 Adding plants to the edit object

All that is left to do is finalize the event map for the houseForm template. We need it to provide functionality for

- clicking the add plant button
- entering text into the input fields
- clicking the save button

We will tackle these one by one.

ADDING A PLANT

Adding a new plant requires nothing more than adding a new object to the plants array when working with an edit object.

As such the event handler will follow the basic principles of working with the edit object: Copy the object from ReactiveVar to a local object, push a new plant object into the plants array, and call the storeReactiveHouseObject() function to set the edited flag and put the local object back into ReactiveVar. The corresponding code is shown in Listing 5-13.

Listing 5-13: Extending the houseForm event map to add plants

```
Template.houseForm.events({
    //...
    'click button.addPlant': function (evt) {
        evt.preventDefault();
        editObject = reactiveHouseObject.get();
        editObject.plants.push({color: '', instructions: ''});
        storeReactiveHouseObject(editObject);
    }
});
```

Just like removing plants, adding a new plant will instantly update the showHouse template as well. Due to the edited flag which is set to true by the storeReactiveHouseObject function clicking the button will also trigger the warning message about unsaved changes.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

CHANGING INPUT FIELD VALUES

All the events for editing a house take place only locally inside the user's browser. Therefore we do not have to worry about performance issues or network latency and can set up the event listeners to react to each individual keypress. By listening to the keyup event we ensure that the whole content of an input field is captured properly. Add the code from Listing 5-14 to the event map for the `houseForm` template and watch what happens when you type in the `name` input field.

Listing 5-14: Listening to name changes using the event map

```
Template.houseForm.events({
    //...
    'keyup input.name': function (evt) {
        editObject = reactiveHouseObject.get();
        editObject.name = $(evt.currentTarget).val(); #A
        storeReactiveHouseObject(editObject);
    }
});
```

#A put the input field value into the name field of the edit object

Select a house and change the name value. You can now observe two-way data binding in action – each keystroke updates every instance where the `{{name}}` property of house is used. The field value updates the `showHouse` template directly; every keystroke updates the view (Figure 5-6).

Figure 5-6: Changes to the input field instantly update the value on the view above

Let's add another event map for `plantFieldset`. This time we need to listen to two different input fields: `color` and `instructions`.

Because the input field's class is the same as the field name inside the house object we can assign the class value to a `fieldName` variable so we know which property needs to be updated. Furthermore we need to determine `thisPlantIndex` to know which object inside the plants array is going to be affected (Listing 5-15).

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Listing 5-15: Listening to changes to color and instructions inputs in plantFieldset

```
Template.plantFieldset.events({
    //...
    'keyup input.color, keyup input.instructions': function (evt) {
        var thisPlantIndex = $(evt.currentTarget).attr("data-index");
        console.log("Index: ",thisPlantIndex);
        var fieldName = $(evt.currentTarget).attr("class");
        editObject= reactiveHouseObject.get();
        editObject.plants[thisPlantIndex][fieldName] = $(evt.currentTarget).val();
        storeReactiveHouseObject(editObject);
    }
});
```

We are almost done. The next section explains how to store the edit object inside the `ReactiveVar` back to the database.

5.5 Saving the edit object to database

Saving the edit object to database is triggered via the "Save house" button. Therefore we set up another event handler that will persist all house data into the `HousesCollection`. The main change is to switch from an `insert()` to an `upsert()`. Refer to Listing 5-16 for details.

Listing 5-16: Event map for saving the edit object to the database

```
Template.houseForm.events({
    'click button#saveHouse': function (evt) {
        evt.preventDefault();
        editObject = reactiveHouseObject.get();
        delete editObject.edited; #A
        editObject._id = HousesCollection.upsert( #B
            {_id: reactiveHouseObject.get()._id},
            reactiveHouseObject.get()
        ).insertedId || reactiveHouseObject.get()._id; #C
        Session.set("selectedHouse", editObject._id); #D
        reactiveHouseObject.set(editObject); #A
    },
    //...
});
```

#A remove the edited flag from the edit object
#B using an upsert will update an existing document if a match for _id is found, else it inserts a new document
#C if upsert does not return an ID use the _id from the ReactiveVar object
#D If a new house was created make sure it is selected

If you prefer you could also get rid of the save button and instantly save the edit object to database each time a change occurs. Because this adds a lot of network interactions and database operations it is better to avoid these frequent interactions between the client and the server if possible. Additionally you would need to provide some form of undo to revert accidental changes.

As a finishing touch remember to enhance the delete button to assign `emptyHouse` to the edit object when removing a document from the collection, otherwise you will still display a house even after it is deleted from the database.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Listing 5-17: Empty the edit object when houses are deleted

```
Template.showHouse.events({
    'click button#delete': function (evt) {
        //...
        if (deleteConfirmation) {
            HousesCollection.remove(id);
            reactiveHouseObject.set(emptyHouse); #A
        }
    }
});
```

#A add this line

Another detail you may wish to take care of is the watering button in the `showHouse` template. Right now it updates the database directly. Let's keep the instant database updates upon clicks, so we must ensure that the edit object is updated as whenever a plant is watered.

Make sure not to call the `storeReactiveHouseObject()` function because it will mark the object as different from the database content, which is not the case due to the instant update. Add the following code to the event handler for the `water` button in the `showHouse` template:

Listing 5-18: Watering the plants updates both database and edit object

```
Template.plantDetails.events({
    'click button.water': function (evt) {
        //...
        HousesCollection.update({_id: Session.get("selectedHouse")}, {$set:
      {lastvisit: lastvisit }});
        editObject = reactiveHouseObject.get();
        editObject.lastvisit = lastvisit;
        reactiveHouseObject.set(editObject);
    }
});
```

5.6 Input validation on the client – Keeping plant colors unique

The way we initially set up the house-sitter application it assumed that each house only had a single plant of each color. Due to the switch to identify plants based on their array position rather than a unique ID there is no need to forbid two flowers of the same color. From a user's perspective it still makes a lot of sense. If there are two red flowers, how do you know which instruction applies?

> **NOTE** This event-based validation can easily be bypassed using the JavaScript console. For a safer way to validate data before adding it to the database refer to Meteor methods, which we discuss in chapter 6.

We will implement input checking to the `plantFieldset` event map. Because it listens for keyup in either of the two input fields we add a conditional to only check for duplicate colors if the value in any `color` field changed. After setting the current color from the edit object we will iterate over all plants and count how often the current color is found. If the result is anything but 1 we set the input field's background red and disable the save button. Only if the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

value for color becomes unique again we reset the color and enable the save button again (Listing 5-19).

Listing 5-19: Avoiding duplicate entries for color by extending the event handler

```
Template.plantFieldset.events({
    //... #A
    'keyup input.color, keyup input.instructions': function (evt) { #B
        //...
        storeReactiveHouseObject(editObject);
        if (fieldName == "color") {
            var currentColor = editObject.plants[thisPlantIndex]['color'];
            plantsArray = reactiveHouseObject.get().plants; #C
            var colorCount = 0; #D
            for (i in plantsArray) {
                if (plantsArray[i]['color'] == currentColor) {
                    colorCount++;
                }
            }
            if (colorCount != 1) {
                $(evt.currentTarget).css("background-color", "red");
                $("button#saveHouse").attr("disabled", "disabled");
            }
            else {
                $(evt.currentTarget).css("background-color", "");
                $("button#saveHouse").removeAttr("disabled");
            }
        }
    }
});
```

#A The previous event handlers remain untouched
#B Each key stroke calls this routine
#C Each input field value with a color class is added to the array
#D colorCount holds the number of times each color occurs in the object

By now you are familiar with using the `ReactiveVar` object and various ways to store, manipulate, and retrieve data from a MongoDB database using `Collections`. While the main functionality is there, we still haven't covered some security basics every application on the internet should implement. One of the reasons is that in order to secure our data we need to introduce the concept of users, which we will address in the next chapter. Once we have discussed how Meteor handles users we will deal with more sophisticated ways to ensure your data stays safe.

5.7 Summary

- Using `ReactiveVar` behaves similar to `Session`, but it avoids polluting the global namespace
- Templates cannot return array index positions, a helper function is required
- Although not a core feature of Meteor, two-way data binding can be easily achieved by using reactive contexts such as `Session` or `ReactiveVar`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

6

Users, Authentications, and Permissions

This chapter covers

- Enabling user registration via username/password
- Connecting to an SMTP server to send e-mails
- Customizing e-mail messages for accounts-related messages
- Adding oAuth authentication via Facebook
- Managing permissions using allow/deny

Once an application is connected to one or more data sources it is capable of displaying dynamic content. In order to customize the actual content different users see, the application needs to know who is requesting data. Most like some users will be able to add content, but there will be some restrictions on both what data can be accessed and what can be done with it.

For these reasons we are going to introduce the concept of users and accounts. Up until now we kept things fairly simple and assumed a single type of user: the anonymous guest. Unless the application can identify a guest as a particular user we cannot show user-fitted content.

The process of identifying a user is called *authentication*. In this chapter we will discuss how users may sign up to an application and how they can identify themselves using either a combination of username and password or log in using an existing service like Facebook, Twitter, or Github to verify who they are.

The second main concept of working with accounts is called *authorization* and introduces security fundamentals to every application. In the simplest form logged-in users may be treated differently from anonymous guests. Typically applications require a more granular way

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

to define permissions so the concepts of roles such as members and admins become important.

In this chapter you will learn how to add users to any application using Meteor's core functionality for both password authentication and OAuth. Every step from registration to editing a user's profile data to deleting an account will be discussed. We will use what we learned in chapter 4 to use Collections to allow users to exchange messages and apply granular permissions for deleting and viewing these messages.

In order to walk through all user-related functionality we will be building a small application again. Meteor does not require a specific file structure. For the course of this chapter we are going to use the structure shown in Figure 6-1.

Figure 6-1: Application structure for userApp

Create a new project and set up these files and folders. Once you are all set we are ready, let's start by addressing the authentication aspect when dealing with users.

6.1 Adding users to an application

Our application should be able to tell who is currently using it. This will be the foundation upon which we can later on grant or restrict access and functionality. Fortunately Meteor makes it very easy to add user functionality, requiring hardly any code.

The most common scenario to add users on the web is to allow visitors to sign up or register themselves. That turns them from guests to users. As the identifier we will be using an e-mail address or a username and to verify a user we shall rely on passwords.

The basic workflow for user management includes:

1. User registration or sign-up

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

2. Log in or sign-in existing users
3. Password reset for existing users

6.1.1 Adding password authentication

Not every application needs accounts, so the accounts functionality is not available in a new project. It is, however, part of the core packages of every Meteor installation and can be added quickly using the command line tool. The following commands will extend an existing application to allow users to sign-up, log in, and perform all relevant actions for a basic user workflow:

```
$ meteor add accounts-password
$ meteor add accounts-ui
```

The first command adds the actual functionality for using passwords. The second command will add templates for user actions (registration/login/password reset) and associated styling information. If you do not want any styles applied to the templates because you are going to take care of this yourself simply add `accounts-ui-unstyled` instead.

Both commands will ensure that all dependencies are met as well. For example the `accounts-password` package enables users to reset their passwords. In order to do so it requires the `email` module to be able to send the reset link to a user, so it adds it to the application as well. Also the styles used for the default login mask rely on less, so the `less` package is added as well. Meteor will show detailed messages on which packages were added on the command line. Inside your project folder you can find information on all packages inside the `.meteor/packages` file.

Once the packages are added start the meteor server again.

ADDING USER TEMPLATES

Inside our `client/templates.html` file we can now add the user-related sub-templates. The `accounts-ui` package includes all required templates and all we need to do is add an inclusion tag anywhere inside the existing templates like this:

NOTE For the code in this chapter we have also added Bootstrap 3 using `meteor add mizzao:bootstrap-3` to provide a nicer look and feel.

Listing 6-1: Adding loginButtons

```
<head>
          <title>Working with users</title>
      </head>

      <body>
      <div class="container">
          <div class="navbar">
              {{>loginButtons }} #A
          </div>
          <h1>Working with users</h1>
      </div>
      </body>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

#A This includes the actual user templates

Figure 6-2 shows how the login feature is presented to the user. The `loginButtons` template creates an expandable overlay that provides login, sign-up, and password reset functionality inside a single container. When the user clicks on "Sign in" the dialog opens and provides three buttons. By default every user is identified by their e-mail address and a password. Requiring a username or making the e-mail address optional can be configured using `Accounts.ui.config`.

Before we change the default behavior let's use the login box to sign up a first user.

Figure 6-2: Login box

6.1.2 Registration and password reset

Ideally the sign-up process for an application should be as simple and quickly as possible. That encourages users to actually register and not cancel the process if there are multiple steps involved. Therefore the default Meteor user registration only requires a minimum of information from the user. Once a user is signed-up you may remind them to fill out a profile or answer additional questions.

The simplest way to sign up a user is to ask for their e-mail address and password. These two parts are sufficient to uniquely identify any user and keep their accounts somewhat safe. The `accounts-password` package requires all passwords to be at least 6 characters long

Use the login box to sign up your first user by filling in both fields. Then click on the small text link that says "Create account", just below the "Sign in" button. Notice how the big "Sign in" button now shows "Create Account" as well (Figure 6-3)? That's it, you just registered the first user for your application.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Figure 6-3: Using the login box to create an account

THE USERS COLLECTION

Users belong in the long-term memory of the application, so they are stored inside a database collection.

> **BEWARE** There are database users and application users. Application users are stored inside a real Collection, not inside the special Users folder. These are what we are dealing with here. Database users are needed to connect Meteor with MongoDB and there is typically only one user of that type needed.

When we look at a single user document we see that it now contains four top-level fields:

- `_id` – holds the unique database ID for the user, also accessible via `Meteor.userId()`
- `createdAt` – a timestamp when the user was created/registered for the application
- `emails` – an array of one or more addresses associated with the user. Each e-mail address may belong to only one user and may be verified or not.
- `services` - an object containing data used by particular login services, e.g. tokens used by forgot password links

Listing 6-2: A single user document stored inside the users collection

```
{
    "_id" : "xcwYNyvMhP8rq6EPp",
    "createdAt" : ISODate("2014-12-25T12:47:33.821Z"),
    "emails" : [
        {
            "address" : "stephan@meteorinaction.com",
            "verified" : false
        }
    ],
    "services" : {
        "password" : {
            "bcrypt" : "$2a$10$OsFJKxSApp68T9elfjKvtXBdBPSnYubmnXo2gg H9NhWo8aVsa"
        },
        "resume" : {
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
            "loginTokens" : [
                {
                    "when" : ISODate("2014-12-26T09:24:51.382Z"),
                    "hashedToken" : "sAMzRZMnqWrmXbmOCm7cpKzG5JR5vZAVEgqf8f9bUTo="
                }
            ]
        }
    }
}
```

For each authentication provider the `services` field holds the necessary information to perform authentication. Passwords are stored using *bcrypt* by default, the same that is used as the password hash algorithm for BSD and many Linux systems.

There are no restrictions on which fields may be stored inside a user document, so you can extend it as you like. There are two more standard fields that are only filled when needed. These are `username` and `profile`. Because the sign-up process does not require users to set a `username` the field is not used in our case. The `profile` field contains an object, which the associated user has full read and write access to by default. This object is the default storage for things like a real name, biography text, or phone numbers.

To allow users to share social information and protect their identity we will require usernames and also make sure every user has a profile that they can fill as they like.

CONFIGURING THE REGISTRATION PROCESS

Because the login box is only available on the client the corresponding configuration also needs to take place inside the client context. By adjusting the setting for `passwordSignupField` for the `Accounts.ui.config` object we can ask users to provide a username during the registration process. As you can see in Table 6-1 each setting has a different requirement as to which fields must be provided during registration.

Table 6-1: Possible values for passwordSignupFields

Setting	username	e-mail
`USERNAME_AND_EMAIL`	mandatory	mandatory
`USERNAME_AND_OPTIONAL_EMAIL`	mandatory	optional
`USERNAME_ONLY`	mandatory	n/a
`EMAIL_ONLY`	n/a	mandatory

For this chapter we are going to assume each user has a username. An e-mail address is not important for us – if they do not want to be able to recover their passwords we will not force users to provide an address. Listing 6-3 shows how to configure our application to ask for both username and an (optional) e-mail address during the registration process.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Listing 6-3: Configuring registration to ask for both username and e-mail

```
Accounts.ui.config({
  passwordSignupFields: 'USERNAME_AND_OPTIONAL_EMAIL'
});
```

Remember that this configuration code belongs in a file that is not executed on the server or at least wrapped in a `Meteor.isClient` block. If it is not, your application will produce errors. We will place it in `client/client.js`.

When the configuration is in place the login box will show four instead of two fields for creating a new account as shown in Figure 6-4. Because the user may decide to not provide us with an e-mail address they will not be able to reset their password and it becomes important to make sure they enter the password correctly. For this reason there is also a password verification field that was not shown when using a mandatory e-mail address.

Figure 6-4: Creating account with mandatory username and optional e-mail address

If you sign up new users without an e-mail address now you will notice that inside the MongoDB collection there is no field `emails`. This is because empty fields do not get created in NoSQL databases. This is different from relational databases like MySQL, where a fixed schema is used. New fields may be added at anytime, so there is no need to store empty fields to a document.

EXTENDING THE ACCOUNTS CREATION WITH PROFILE DATA

When registering a new user on a web page the form data is forwarded to the server to process and store inside a MongoDB collection. To extend the default behavior it is possible to hook into the creation process and add checks or enrich the data stored. This is done using `Accounts.onCreateUser()`. It takes a function as its argument. This function gets called everytime a new user is created and enables you to control the content of new user documents. The function itself can make use of two arguments: `options` and `user`.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
// server.js
Accounts.onCreateUser(function (options, user) {
   user.profile = options.profile;
   return user;
});
```

The options object comes from the authentication provider, in this case accounts-password. It comes from the client and should not be trusted. By default onCreateUser simply copies options.profile to user.profile and returns a user object representing the newly created user document in the database.

If we want to add profile information to each new user we need to place our code into the server/server.js file (or inside a Meteor.isServer block). We will keep the default behavior of onCreateUser to copy profile data from the authentication provider to the user document. When using passwords we know there is no profile data, but when we later add external logins with Facebook or Twitter we will be able to use the profile data they pass on. If there is no profile data provided from the sign-up process we will add it as an empty object. Before returning the user document a new profile property called rank will be added to each user, making them a "White belt". The required code is shown in Listing 6-4.

Listing 6-4: Adding profile information to new users

```
// server.js
Accounts.onCreateUser(function (options, user) {
    if (options.profile) {
        user.profile = options.profile;
    }
    else {
        user.profile = {};
    }
    user.profile.rank = "White belt";
    return user;
});
```

All users that sign up from now on will have a profile field in their user document. Existing users are not affected by that change because they already went through the createUser stage and therefore will not be affected by the updated functionality.

TIP If you want to clear the database use the command meteor reset. This will empty all collections and you can start over with no users.

6.1.3 Setting up E-Mail

Especially when an e-mail address is required to sign up you will want to ensure that it actually works. Therefore, and for resetting passwords, the Meteor server must be capable of sending out e-mails to users. If you do not configure the emails package then all messages will be shown on the server console but never actually sent out.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
=> App running at: http://localhost:3000/
I20141113-10:44:10.289(1)? ====== BEGIN MAIL #0 ======
I20141113-10:44:10.338(1)? (Mail not sent; to enable sending, set the MAIL_URL environment variable.)
I20141113-10:44:10.339(1)? MIME-Version: 1.0
I20141113-10:44:10.339(1)? From: "Meteor Accounts" <no-reply@meteor.com>
I20141113-10:44:10.339(1)? To: stephan@yauh.de
I20141113-10:44:10.339(1)? Subject: How to reset your password on localhost:3000
I20141113-10:44:10.339(1)? Content-Type: text/plain; charset=utf-8
I20141113-10:44:10.340(1)? Content-Transfer-Encoding: quoted-printable
I20141113-10:44:10.340(1)?
I20141113-10:44:10.340(1)? Hello,
I20141113-10:44:10.340(1)?
I20141113-10:44:10.340(1)? To reset your password, simply click the link below.
I20141113-10:44:10.340(1)?
I20141113-10:44:10.341(1)? http://localhost:3000/#/reset-password/LmpvMrBoEOBTlkdamjiomi-WC6luwN3LELCW98Pr6Dr
I20141113-10:44:10.341(1)?
I20141113-10:44:10.341(1)? Thanks.
I20141113-10:44:10.341(1)?
I20141113-10:44:10.341(1)? ====== END MAIL #0 ======
```

Figure 6-5: Unless an email server is configured Meteor will show e-mails on the server console

As you can see in Figure 6-5 Meteor requires the `MAIL_URL` environment variable to hold the connection string for an SMTP server. Using an environment variable is a great way to quickly adjust a configuration value. Most of the times it is more comfortable to add the mailserver configuration to a file, though.

ADDING A MAILSERVER

The connection details to a mailserver are considered highly sensitive. You do not want anyone to find out the credentials for your mailserver and start sending Spam from your machines. To avoid sharing login details with all users make sure to configure the mail server not inside a `Meteor.isServer` block but inside a dedicated file inside the `server/` folder of your application.

Meteor uses various environment variables for configuration, you can find an overview in chapter 12. All environment variables may either be passed directly to the server upon startup or can be set from inside the code using `process.env.<Name of Environment Variable`. To set `MAIL_URL` to a valid mailserver we put the command inside a `Meteor.startup()` function, so it gets executed everytime the Meteor server starts. To keep the code clean we use variables to set `username`, `password`, `server`, and `port` instead of writing the SMTP connect string directly. Some characters need to be escaped in a connection string, so each variable is processed by `encodeURIComponent`.

Once you have added the code shown in Listing 6-5 to your application your application will be able to send e-mails such as password reset links to all users who provided an e-mail address. Make sure to adjust the variable values to match your own mailserver configuration.

Listing 6-5: Configuring an SMTP server in server/smtp.js

```
Meteor.startup(function () {
    smtp = {
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
        username: 'meteorinaction@gmail.com',
        password: 'mySecretPassword',
        server: 'smtp.gmail.com',
        port: 587
    };
    process.env.MAIL_URL = 'smtp://' +
                   encodeURIComponent(smtp.username) + ':' +
                   encodeURIComponent(smtp.password) + '@' +
                   encodeURIComponent(smtp.server) + ':' +
                   smtp.port;
});
```

Now that the application is capable of sending e-mails, we can encourage users to verify their address upon registration.

SENDING ADDRESS VERIFICATION E-MAILS

Again, we will hook into the `onCreateUser` function to send a verification e-mail as soon as a user registers. The corresponding function is called sendVerificationEmail and it takes two arguments: the user ID and an optional e-mail address:

```
Accounts.sendVerificationEmail(user._id, email);
```

Typically only the first argument is needed, as the address will be part of the user document. However, we do not require users to provide an e-mail address, so we should take care that there is no attempt to send mails when the user does not have an address. Also, if we hook directly into the creation process we need to wait for Meteor to actually create a user document first before we can actually access it.

The code shown in Listing 6-6 first checks if the user provided an e-mail address, then sets a timeout of 2 seconds to wait for the account to be created, and finally sends the verification mail.

Listing 6-6: Sending verification e-mails upon user creation

```
Accounts.onCreateUser(function (options, user) {
    //...
    user.profile.rank = "White belt";
    if (options.email) { #A
        Meteor.setTimeout(function () { #B
            Accounts.sendVerificationEmail(user._id); #C
        }, 2 * 1000);
    }
    return user;
});
```

#A Only do this if the user provided an address
#B Give Meteor up to 2 seconds to actually create a user document
#C Send a verification mail

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

CUSTOMIZING MAIL MESSAGES

Meteor has default settings for the sender of all accounts-related e-mails, subjects, and body texts. You can adjust them to hold any text you like. Table 6-2 shows the settings stored inside the `Accounts.emailTemplates` object.

Table 6-2: Available fields for adjusting accounts-related e-mails

Field name	Description	Notes
`siteName`	Name of the application	Default value: DNS value of the app, e.g. usersApp.meteor.com
`from`	rfc5322 compliant sender name and address	Default value: `Meteor Accounts <no-reply@meteor.com>`
`resetPassword`	An object with three fields which take a function: `subject`, `text`, `html`	`text` and `subject` are mandatory, `html` is optional
`enrollAccount`	An object with three fields which take a function: `subject`, `text`, `html`	`text` and `subject` are mandatory, `html` is optional
`verifyEmail`	An object with three fields which take a function: `subject`, `text`, `html`	`text` and `subject` are mandatory, `html` is optional

As you can see the accounts package defines three different types of e-mails it can send. You can trigger them manually by using the corresponding send-functions:

```
Accounts.sendResetPasswordEmail()
Accounts.sendEnrollmentEmail()
Accounts.sendVerificationEmail()
```

During user creation we send the verification e-mail and we shall now adjust all e-mail settings and templates. To keep things simple you can add the code from Listing 6-7 to the existing `server/smtp.js` file. It must run inside a server environment, otherwise the browser will throw errors.

Listing 6-7: Customizing the accounts e-mail templates

```
Accounts.emailTemplates.siteName = "Meteor in Action userApp"; #A
Accounts.emailTemplates.from = "Stephan <stephan@meteorinaction.com>"; #B

Accounts.emailTemplates.verifyEmail.subject = function (user) {
    return 'Confirm Your Email Address, ' + user.username;
};
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
Accounts.emailTemplates.verifyEmail.text = function (user, url) { #C
    return "Welcome to the Meteor in Action userApp!\n"
    + "To verify your email address go ahead and follow the link below:\n\n"
    + url;
};

Accounts.emailTemplates.verifyEmail.html = function (user, url) { #C
    return "<h1>Welcome to the Meteor in Action userApp!</h1>"
    + "<p>To <strong>verify your email address<strong> go ahead and follow the link
      below:</p>"
    + url;
};
```

#A Adjusting the sitename has no effect on the URL users have to click
#B Users will see this as the sender of all e-mails
#C Defines the content of the verification e-mail

Even though you can easily define HTML e-mails keep in mind that both text and HTML are send to the recipient. If they set their e-mail client to prefer plain text to display content they will not see the content defined inside the `html` function. Therefore make sure that you always have the same amount of information in both `text` and `html` templates.

Register as a new user. You will now receive a customized e-mail with a personal verification link. As you can see it still points to http://localhost:3000. When deploying your application you will need to set the correct URL via an environment variable. If you use meteor deploy, it is automatically taken care of. If you use a different method to deploy your application you need to adjust the `ROOT_URL` to the correct URL like this

```
// server.js
Meteor.startup(function () {
    process.env.ROOT_URL = 'http://www.meteorinaction.com';
});
```

6.2 Authenticating users with 0Auth

Oftentimes usernames and passwords are not the only option you want to give your users to log into an application. Being able to use an existing account to log into a site lowers the barrier of signing up by not having to type in a single bit of information. Additionally it simplifies using an application by not having to remember additional usernames or passwords.

Meteor ships with multiple authentication providers that allow users to use a social network instead of a local username. These networks include

- Facebook
- Github
- Google
- Meetup
- Meteor Developer Account
- Twitter
- Weibo

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

All of the above are based on oAuth, a complex way to pass authentication data from one site to another. There are also many community packages available that enable other authentication providers such as LinkedIn or Dropbox. As the fundamentals of working with oAuth providers are the same for each provider we will not need to discuss each.

6.2.1 An introduction to oAuth

The Open Authentication – or short: *oAuth* – mechanism has become very popular among web applications since its beginnings in 2007. The main idea behind it is to use a service providers, e.g. Facebook, to authenticate a user and allow a third-party application to access specific information from the authenticated user (access authorization). This could be simply the username or more sensitive information such as friends or a permission to post to the user's wall.

Figure 6-6: oAuth flow using Facebook as the Service Provider

As Figure 6-6 shows, there are three main ingredients to every oAuth scenario:

1. A Service Provider – e.g. Facebook or Twitter
2. A Consumer – e.g. your Meteor application
3. The user – e.g. an existing Facebook user wanting to log into your Meteor application

Many sites on the web can act as a service provider for oAuth. We are going to use Facebook as an example to illustrate the process. Our Meteor application must be connected to Facebook. This is done by creating a new application on Facebook's developer site. To verify our application is not a malicious script, it will the be able to identify itself using the corresponding *application ID* (App ID) and a *secret key (App Secret)*. These are basically the username and password for our Meteor server process. Once both are connected we can allow users to sign in with their Facebook account.

Instead of entering any credentials in our Meteor app, users can now click a button to log them in via Facebook. Assuming they are already logged in on Facebook they will now see a dialog asking them whether they want to share information with the Meteor application. Behind the scenes the Meteor application forwarded the login request to Facebook as the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

service provider. If the user agrees to share their login information with the Meteor application, then Facebook generates an access token. This token lets the Meteor app know that the user has been authenticated properly and it grants the permissions provided by the user. In the simplest case Meteor may only have read access to the user's e-mail address. Depending on the configuration settings we could also request more permissions such as posting to the user's wall.

Not all oAuth providers support the same set of permissions, so they must all be individually configured. The advantage of using oAuth is that the consumer application can talk directly with the service provider to exchange data, if permissions exist. That way all Facebook friends, recent Tweets or number of private repos on Github can easily be accessed and added to the user's profile.

6.2.2 Integrating Facebook authentication

To integrate oAuth authentication via Facebook in a Meteor app we need to perform the following steps:

1. Add the `accounts-facebook` package
2. Create a new Facebook application
3. Configure the Facebook integration

ADDING ACCOUNTS-FACEBOOK TO AN APPLICATION

The first step is to add Facebook as the authentication provider for our application. If the application already support username/password authentication as in section 6.1, it is sufficient to add a single package

```
$ meteor add accounts-facebook
```

This package will not add any templates to the application. Therefore, if `accounts-facebook` is the only package available in the project you would need to manually call all functionality within your templates. Or you can add the `accounts-ui` package, which not only provides a login box for use with password authentication, but also for many oAuth services.

All oAuth packages require configuration. Just like users, this configuration is stored inside a MongoDB collection. A collection named `meteor_accounts_loginServiceConfiguration` will be created as soon as the service is configured. Additionally, pending credentials will be stored temporarily as well, which is done in a dedicated collection. This collection is created at server startup already and is called `meteor_oauth_pendingCredentials`.

There is no need to manually access either of these two collections. Meteor will use these internally only and there is no benefit in querying data from them directly.

CREATING A FACEBOOK APPLICATION

If Meteor cannot find a configuration for integration with Facebook the UI will show a red button labeled "Configure Facebook Login" instead of the regular log in button. Clicking on it

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

will bring up a short configuration guide as well as two fields to provide the application ID and secret.

You will need to register yourself as a Facebook developer, which is free of charge but requires you to have a Facebook account. A new Facebook application can be created at https://developers.facebook.com. Under the Apps tab you can add a new application of type Web/WWW. Next you need to assign an application ID. This can be any name that helps you and your users to identify the application. As your users may eventually see the application name it is good practice to use the site name or something that closely describes your application. The actual category for your application and whether it is a test version of another Facebook application do not have any influence on the functionality and can be set to the values that best describe your project.

The site URL for a Facebook application that is used with a local development environment should typically be set to http://localhost:3000. The correct URL setting can be taken from the configuration dialog shown by Meteor.

Once you have made these settings Facebook requires a contact e-mail to be set up for the application before you can activate the application. Navigate to the application dashboard on the Facebook Developer site and enter a contact e-mail in the settings section (see Figure 6-7). Finally you need to activate the application in the Status & Review tab.

Figure 6-7: Settings for a Facebook application to integrate with Meteor

An activated Facebook application can be used to authenticate users. The last step to implement logging in via Facebook is to configure the Meteor application.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

CONFIGURING

Open the Meteor application in the browser and click on the Facebook button to bring up the configuration dialog as shown in Figure 6-8.

Figure 6-8: Configuration dialog for Facebook integration

Besides the basic instructions on how to create a Facebook application the configuration dialog lets you add the application credentials (`App ID` and `App Secret`) as well as a login style. The default is to use a *popup-based* dialog, which means the application window will remain when the user logs into Facebook and a new window with the Facebook dialog is opened. In contrast, the *redirect-based* login style will leave the application, redirect the current browser window to Facebook, and reload the entire application once the authentication was successful. Unless you plan on running your application inside a Cordova container on mobile devices[13] it is preferable to use the popup-based login.

Save the configuration and users can start logging in via Facebook. Should you have misconfigured the application you can either manually delete the configuration information from the `meteor_accounts_loginServiceConfiguration` collection inside the MongoDB or simply issue `meteor reset` to empty all collections.

[13] Refer to chapter 11 on Isobuild to know more about running Meteor applications on mobile devices.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

REMEMBER All oAuth configuration is stored inside the application database. Whenever you issue the `meteor reset` command to empty the database it will also remove all oAuth configuration data from the database.

If you prefer to be able to view your Facebook credentials from within the source code you can insert it into the database just like any other fixtures we used in previous chapters. This requires a package called service-configuration to be available:

```
$ meteor add service-configuration
```

Once the package is available the code shown in Listing 6-8 will take care of setting the correct oAuth credentials for Facebook upon the start of your Meteor application.

Listing 6-8: Inserting Facebook oAuth configuration as fixture in server/server.js

```
if (ServiceConfiguration.configurations.find({service: 'facebook'}).count() === 0) {
          ServiceConfiguration.configurations.insert({
              service: "facebook",
              appId: "oAuth-credentials-from-facebook",
              secret: "oAuth-credentials-from-facebook",
              loginStyle: "popup"
          });
      }
```

ADDING FACEBOOK INFORMATION TO THE USER PROFILE

Every user that logged in via Facebook has a new entry in the `services` field in their user document. It contains the token used for authentication but it also includes information such as the user's first and last name, their gender, and e-mail. If we want to allow a user to edit these it is best to copy them over to the `profile` object inside the user document.

Listing 6-9: The user document when signing up via Facebook

```
{
    "_id" : "nzPMRdhSKx7NJvTGY",
    "createdAt" : ISODate("2014-12-21T21:23:55.475Z"),
    "profile" : {
        "name" : "Stephan Hochhaus" #A
    },
    "services" : {
        "facebook" : {
            "accessToken" : "CAAEkDwbZAj.....",
            "email" : "stephan@meteorinaction.com",
            "expiresAt" : 1421097429424,
            "first_name" : "Stephan",
            "gender" : "male",
            "id" : "1234567890",
            "last_name" : "Hochhaus",
            "link" : "https://www.facebook.com/app_scoped_user_id/123456789/",
            "locale" : "en_US",
            "name" : "Stephan Hochhaus" #A
        },
        "resume" : {
            ...
        }
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
    }
}
```

#A Meteor automatically adds the name property to profile

As you can see from Listing 6-9 Meteor already copied over the name property from Facebook to the user profile. Hooking again into `Accounts.onCreateUser` we can copy over data provided by Facebook to the user's profile. We will copy over first and last name and gender from `user.services.facebook` to `user.profile` so that the user may edit this information inside the Meteor application independently from Facebook.

Listing 6-xx shows how to extend the onCreateUser function to copy over fields from the `facebook.service` to the `profile` field, but only if a user has logged in via Facebook. That way it can be combined with the code used from Listing 6-10 for adding `profile` fields for password authentication.

Listing 6-10: Adding Facebook information to the user profile

```
Accounts.onCreateUser(function (options, user) {
    if (user.services.facebook){
        user.profile.first_name = user.services.facebook.first_name;
        user.profile.last_name = user.services.facebook.last_name;
        user.profile.gender = user.services.facebook.gender;
    }
    return user;
});
```

Even if Facebook has updated information for names or gender it will not update the settings inside the user's profile object as the `onCreateUser` function will only be called the first time a user logs into the application using Facebook.

6.2.3 Integrating other 0Auth providers

As mentioned earlier Meteor comes with multiple packages that allow the integration of social networks as authentication providers. The principles always remain the same. Before you can configure the external service provider you need to create an application in Twitter, Google, Github, or whatever service you plan to integrate. Some of these services require you to set a value for authenticated or callback URLs. As long as an application is under development this is typically `http://localhost:3000`. Therefore it is good practice to create two applications on the service provider, one for the local development environment and another for the live instance of your Meteor application.

USING ALTERNATIVE AUTHENTICATION METHODS IN ONE APPLICATION

Meteor makes it simple to add multiple authentication providers to a single application. However these providers do not share the same data with the consumer, which makes it complicated in many cases to associate alternative login methods to the same user.

Imagine an application is set up to allow authentication via username and password as well as Twitter and Facebook. A user may decide to log in with Twitter one day and with Facebook the next. How could the application tell it is the same user? Possibly if the e-mail address

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

associated with their Twitter account matches the one for the Facebook account. Unfortunately Twitter does not expose a user's e-mail address over the authentication API. As a consequence Meteor cannot connect a Twitter login to a Facebook account. It will assume both are different users. In a worst case scenario users will log in through each of the login methods the application provides and end up having as many accounts as there are login methods.

Of course it is entirely possible to use multiple authentication providers in the same application and allow users to use them all to identify a single user. In order to do so we need the user to manually connect their profiles to the same account. Unfortunately this functionality is not part of the core packages that provide account functionality.

There are some community packages available that save you from creating your own templates and code to allow users to connect multiple social networks to the same account. If you wish to include multiple authentication providers in a single application have a look at `mondora:connect-with` or `bozhao:link-accounts`.

Before adding too many authentication providers to an application you should consider which are actually needed. Due to the fact that working with oAuth is so simple with Meteor may lead to overly complicated applications that have little user benefit.

6.3 Managing user permissions, roles, and groups

Authenticating users is not enough, we also need to authorize their actions. Permissions can be used to define what data a user can access and what functions they are allowed to use. Adminstrators should be able to perform all possible actions while regular users may only edit or delete their own data.

To better illustrate how to manage user permissions we are going to use a simple messaging application. You can find the source code in this chapter's sample code. It offers limited functionality, which makes it easier to illustrate the main points of Meteor's permission system.

Users may sign up with a username and password. Once they are logged in, the application shows a list of users, each of which may be selected by clicking on them. Once a user is clicked, their ID will be assigned to a `Session` variable and a second template will display basic profile data and allow users to view and leave messages. Only the owner of the message board should be allowed to delete messages.

All functionality implemented in this sample application is based on the contents of chapter 3 (templates) and 4 (working with data) with some references to the beginning of this chapter in regards to user accounts.

As highlighted in previous chapters already, Meteor uses a package called `insecure` for every new application. This allows the client and every user, authenticated or not, to save or remove data from the database. The first step in adding security to an application is to remove the package using

```
$ meteor remove insecure
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

The result of removing the `insecure` package is that no user is able to write to the database anymore. Viewing data is possible because the `autopublish` package is still available. We will talk about removing it in the next chapter. First we will concentrate on restricting access for users.

The simplest form of managing user permissions in Meteor is by using the allow/deny function for any collection.

6.3.1 Managing permissions with allow/deny

Meteor trusts all code executed on the server by default. All client code on the other hand is not considered safe to use, especially with the `insecure` package removed. Any `inserts`, `updates`, or `removes` to a database collection from the browser result in an *Access denied* message.

> **NOTE** Allow and deny only affect write operations to the database. Reading from a collection can be controlled using Publications and Subscriptions.

Each collection exposes an `allow` and a deny function that can limit or grant permissions to users. Listing 6-11 shows the syntax, which is the same for both allow and deny. The code must run in the server environment but it is safe to also run it inside the client. Therefore it can be put in the same file where collections are defined, because it removes redundancy. If you prefer, you can also put it in a server-only file. Either way, a client may not change the server code and work around permissions, even if you send this code to the browser.

Listing 6-11: Using allow for a Messages collection

```
// collections.js
MessagesCollection = new Mongo.Collection("messages");

MessagesCollection.allow({
    insert: function (userId, doc) {
        return true;
    },
    update: function (userId, doc) {
        return true;
    },
    remove: function (userId, doc) {
        return true;
    }
});
```

Multiple `allow` and deny rules may be defined, sometimes they will overlap. Meteor first executes all deny callbacks to determine if an action is forbidden. If any of the deny callbacks returns `true`, then the user will not be able to carry out the associated action, even if there is an `allow` rule that returns true. If there are multiple `allow` rules only one has to match and return true to allow a user to carry out an action.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

NOTE The `allow` and `deny` callbacks do not affect Meteor method calls, only direct writes to the database. Methods will be discussed in full detail in the next chapter.

We can use `MessagesCollection.allow` to enable sending messages without the insecure package and also restrict the deletion of messages to a user by the recipient only.

SENDING MESSAGES TO A USER

By selecting a user it is possible for every logged in user to write a message to their site. Each message document contains five fields:

1. `_id` – a unique ID for the message document
2. `sender` – user ID from the author of a message
3. `recipient` – user ID of the recipient of a message
4. `message` – the actual message text
5. `timestamp` – the creation date of the message

It is impossible to create a new message document from the browser unless an `allow` callback returns true. Only logged in users should be able to send a message, so we need to return `true` when the user trying to insert a new document has a user ID. Guests do not have a user ID and therefore their value for `userId` would return `false`.

Listing 6-12: Allowing logged in users to insert new messages

```
MessagesCollection.allow({
    insert: function (userId, doc) {
        return userId;
    }
});
```

Beware that this grants permission to a user to insert whatever they like into the database. The only restriction is, that the user is logged in and writes to the `messages` collection. New fields are added on the fly, for example in case a user wants to add an additional field `messageSubtext`. To prevent users from adding new fields to a document we can use a deny rule that checks whether all defined fields are provided and none is missing. Listing 6-13 shows how the Underscore.js library that is part of Meteor can be used to extract all document fields to an array. Using a second array that contains all valid field names we can compare that no mandatory fields are missing and no additional fields are coming from the client. The `_id` field will be added by the database and is therefore not part the `validFields` array.

Listing 6-13: Deny inserts with missing or additional fields

```
MessagesCollection.deny({
    insert: function (userId, doc) {
        var fieldsInDoc = _.keys(doc); #A
        var validFields = ['sender', 'recipient', 'timestamp', 'message', '_id'];
        if (_.difference(fieldsInDoc, validFields).length > 0) { #B
            console.log("additional fields found");
            return true;
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
        }
        else {
            console.log("all fields good");
            return false
        }
    }
});
```

#A using Underscore.js to put all document field names into an array
#B Check for additional or missing fields

Although we have effectively ensured, that only known fields are provided for a new user document, we have no control over the contents of these fields. A user might send a string as the sender, an object or array, or even a binary blob.

Using allow/deny rules is not too complicated, but it should only be used for relatively simple tasks, otherwise it will become very complicated to maintain the growing complexity. The more rules you add to an application, the harder it becomes to tell which cases will be denied and which will be allowed.

> **SECURITY ADVICE** For better control over database operations consider Meteor methods instead of allow/deny rules. Chapter 7 will provide a thorough introduction.

REMOVING MESSAGES FROM A WHITEBOARD

In case a recipient does not like the contents of a message they should be able to delete the message. But not everybody should be allowed to remove messages, only the recipient should be able to do so.

We cannot set a deny rule that globally denies all `remove` operations and add one `allow` rule for the recipient. Remember that it only takes a single deny rule to return `true` to prevent any other `allow` rule from execution. Therefore we will work with a single deny rule that checks whether the user requesting to `remove` a message is the same as the recipient. The deny rule in Listing 6-14 returns `true` if the `userId` for the currently logged in user is different from the user ID in the `recipient` field of the message document.

Listing 6-14: Deny message removal for everyone but the recipient

```
MessagesCollection.deny({
    remove: function (userId, doc) {
        return doc.recipient !== userId;
    }
});
```

The deny code eventually runs on the server side of the application, even if you placed the code inside a file that can be accessed on both client and server. The `userId` parameter is provided by the `accounts` packages directly and it is determined and passed to the `remove` function on the server. It is not possible to change this value from the browser console to fake another user's ID.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

DELETING A USER ACCOUNT

The users collection is special in several ways. For once, by default users may only edit the content of the profile field. Even if no dedicated `allow` rules are available, new users may still sign up and therefore creating a new user document. Deleting a user, even if it is their own account, is not possible once the `insecure` package is removed. However, using a simple `allow` rule on the `users` collection enables the removal of accounts.

Listing 6-15: Enabling users to remove their account

```
Meteor.users.allow({
    remove: function (userId, doc) {
        return doc._id === userId;
    }
});
```

6.4 Summary

- Meteor ships with several accounts package that enable user registration and login
- Connecting to an SMTP server can be done via environment variable or inside the code
- System e-mail messages can be adjusted via their own Templates objects
- oAuth integration is a core feature that requires little effort to implement
- Simple database permissions can be implemented using `allow` and deny
- For more complex permission settings allow and deny should be replaced by Meteor methods

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

7

Exchanging Data

This chapter covers

- Publishing Collections without the `autopublish` package
- Limiting client data with parameterized subscriptions
- Creating a new aggregation data source
- Making a custom data source reactive
- Securing an application with server side methods

In the early stages of development it may often be helpful to have the contents of the server database available on the client as well. The increased convenience comes at the price of both performance and security, though. If the idea is to build low-latency and high-performance applications for the web we must refrain from replicating the entire database on each client. Additionally, sharing all data probably includes sensitive information that should only be viewed by its owner. In consequence we must get rid of the automatic data publication and regain control over all database content.

This chapter introduces two key concepts to working with Meteor: Publications and Methods (see Figure 7-1).

Figure 7-1: Publications and Methods increase the efficiency of CRUD in applications

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Using Meteor's publications and subscriptions we can fine-tune not only how much data is sent to each client, but also control which fields are available to which user. In this chapter you will learn how to set up data publications on the server that will allow your application to easily hold thousands of database documents while only sending small subsets to the clients. That way we can effectively address many performance issues that might arise.

In chapter 4 we discussed `allow/deny` rules to secure write operations to the database. `Methods`, Meteor's remote procedure calls, are a powerful alternative to these simple rules. `Methods` may run on the server or the client. We will use the to secure all write operations to the database by properly validating all content received from the client. However, their usage is not limited to database operations. They can also be used for other actions like sending e-mails.

Throughout this chapter we will be enhancing an application so it becomes robust enough to be deployed to the internet. The application we will use will be used to store workout data to cover the following aspects

- Manually defining publications and subscriptions
- Limiting data sent to a client by using parameterized subscriptions
- Aggregating data
- Restricting data only to a particular user
- Securing database writes with `Methods`

The workout tracker is very simple, all code is put into 5 files. We use a `fixtures.js` file to populate the collection with random workout data – look at the sample code for this chapter to see how it works.

```
├── client
│   ├── workoutTracker.html
│   └── workoutTracker.js
├── collections
│   └── Workouts.js
└── server
    ├── fixtures.js
    └── publications.js
```

7.1 Publications and Subscriptions

Up until this point Meteor automatically published all collection data to all clients using the `autopublish` package. This package is not suitable for production environments because it does not limit the amount of data sent to a client. What works well for a few database entries during development does not scale well with hundreds or thousands of documents. Also it does not provide any access restrictions, every client may access all data. In this section we will look at ways to sent data to the client in a more efficient and secure way.

We are going to use an application that stores workouts such as running or biking and presents them to a user in a simple table. All workouts will be stored inside a single collection and contain the date when they happened and their distance as well. We will create a lot of sample workout documents at startup and show all of them in one table at first. Since we do

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

not want the client to load all data at once we will then limit the number of workouts loaded to the client and also add a button to fetch more data on each click. Eventually we will add an aggregated view on the whole data. For that we will sum up the distance of the workouts of each month. The aggregated data in the client will also be updated reactively. When a new document is added to the workouts collection the count for the affected month will reactively show the updated sum.

7.1.1 publish() and subscribe()

Publications and subscriptions always come in pairs. While `Collections` are typically declared on both the server and the client, publications exists only on the server. They can retrieve data from a database using the `Collection.find()` method just like a template helper. In Figure 7-x you can see an example where the publication retrieves 3 documents from the database. Those documents are then published under the same name as the collection: `workouts`.

On the client a `Meteor.subscribe()` call initiates a request to the server to send data for the `workouts` collection. Notice that it does not request the content from the database, but from the server, more precisely the result for `Collection.find()`. In this case the result is just 3 documents. The client receives these documents and puts them into a local collection of the same name. Although the `Collection` object has the same name on the server and client, it may hold different data, depending on context and publication settings.

Figure 7-2: An overview of publications and subscriptions

REMOVING AUTOPUBLISH

Because we will manually decide what content to send to clients, we must remove the `autopublish` package. Adding and removing packages with Meteor is done via the command line tool. Getting rid of the `autopublish` package is done via

```
$ meteor remove autopublish
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Once the package is removed and the meteor server is started, the client will have no data available anymore. Even though the client still knows about all available collections, no data is transferred from the server-side MongoDB to the browser's Minimongo instance. If you were to query for documents with any collection you would not get any results back.

SETTING UP A PUBLICATION

In order to display the data we need from the `workouts` collection inside the MongoDB, we need to provide a pub/sub pair. First we will set up a simple publication that sends all workout documents to all clients that subscribe to that publication. Since all publications live in the server scope we put them a new `publications.js` file into the server folder. Listing 7-1 shows how to set up the actual publication.

Listing 7-1: A simple server-side publication

```
Meteor.publish('workouts', function () {#A
    return WorkoutsCollection.find({});#B
});
```

#A You name a publication in order to subscribe to it as you will see later on
#B Publications can return data just like template helpers

NOTE Publications are one-way streets that send data from the server to the client. In order to send data from the client back to the server we must provide a secure method to insert and update data. We will look at this later in the chapter.

Setting up a publication will not have any effect on the client at this point. It must specifically ask for the data via a subscription.

SETTING UP A SUBSCRIPTION

On the client we need to add a subscription. Inside the `client.js` file add the following line at the top of the file:

```
Meteor.subscribe("workouts");
```

Once you did this you will see that there is finally all the data that is available on the server-side MongoDB, available through the Minimongo on the client, again. Calling `Meteor.subscribe()` returns an object with a `stop()` and a `ready()` method. The first can be used to terminate a subscription while `ready()` is true if the server has marked the publication as ready. It is a reactive data source just like `Collection` or `Session`.

This is basically what the `autopublish` package gave us for free for all collections. Next we will start to control the data that is published to the client by limiting the number of documents going to the client.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Using servers as clients

There are scenarios, when two Meteor servers should exchange information. Technically, one server becomes the client of the other. The `subscribe()` command only works inside the client context, but there is a way one server can subscribe to the data of another using a DDP connection.

You can connect to another server using `DDP.connect()`. It takes the URL of the remote server as the only argument. Once successfully connected, it will return an object that allows you to use `subscribe()` (to access published data), `call()` (to invoke methods), and `methods()` (to define client-side methods), and a few more functions.

In order to connect one server to another and act as a client it only takes three lines of code. First, the server-to-server connection is defined, a connection to http://192.168.2.201:3000 will be established. In order to receive the published data we need to declare a collection. This time it will not only take a name as its argument, but also how to connect to the master. Hence, `server2` will be the second argument. Finally, the server may subscribe to `remoteData`. Again there is a slight variation because we need to call the `subscribe()` method on the remote server rather than the local Meteor instance.

```
var server2 = DDP.connect('http://191.168.2.201:3000/');
var RemoteCollection = new Mongo.Collection('remoteData', server2);
server2.subscribe('remoteData');
```

7.1.2 Parameterizing subscriptions

For performance reasons we never want the entire database content to be sent over the network. Besides taking a long time to transfer, too much information may be confusing to users. Therefore we will only publish the 10 newest documents of the `workouts` collection initially. The user can choose to request more documents if they want to see older records as well. Obviously we need to adjust the existing publication code. It needs to support both limits and take an argument to dynamically determine an offset to allow sending a second or third set of 10 documents. Let's take it step by step.

The first thing we need to do is to tell the publication the limit we want to set for the workout query. We can add parameters to the subscription call that will be available as parameters insider the server-side publication function. This way we can set options for the publication, the client tells the server what to do.

> **SECURITY** Whenever we are dealing with data coming from the client we must validate it before using it.

The publication takes an `options` argument that first needs to be checked. Instead of adding our own validation we can use Meteor's `check()`.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

VALIDATING DATA VIA CHECK()

With the `check()` function you can match input values against known patterns. In order to limit the subscription we expect the user to provide a number, nothing else. `check()` uses a simply syntax with two arguments, the value itself and the pattern it should be checked against:

```
check(value, pattern);
```

In order to ensure that the provided parameter only contains a number we use `check(options, Number);` inside the publication. In our example we are actually dealing with an object, so we need to check a pattern for every parameter of the object.

```
check(options,
  {
    limit: Number
  }
);
```

We will use `check()` again when we discuss Methods.

DYNAMIC SUBSCRIPTIONS

Listing 7-2 shows the code from the server's `publications.js` file.

Listing 7-2: Adding parameters to a publication

```
Meteor.publish('workouts', function(options){ #A
        check(options,
          {
            limit: Number
          }
        );

        var qry = {};
        var qryOptions = {
          limit: options.limit,
          sort: {workoutAt: 1} #B
        }

        return WorkoutsCollection.find(qry, qryOptions); #C
      });
```

#A The publication takes an options object as a parameter
#B Sort all database entries by timestamp to ensure the limit starts from the newest
#C Using the limit query options from MongoDB only returns a limited number of documents

Every client can now subscribe to this publication and set a limit. The check function expects an option object and will throw an error unless one is passed by the subscription. We need to create the subscription for the client that subscribes to the data provided by this publication. We do this inside the client folder's `client.js` file since the subscription is only available in the browser. We will use `Session` to keep track of the currently used limit (see Listing 7-3).

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Listing 7-3: Subscribing to a publication with parameters

```
Session.setDefault('limit', 10); #A

// Subscriptions
Tracker.autorun(function(computation){ #B
  Meteor.subscribe('workouts', {
    limit: Session.get('limit') #C
  });
});
```

#A For the limit parameter of the subscription we want to use a Session variable with a default value of 10
#B Autorun creates a reactive context that renews the subscription if the limit changes
#C The Session object is passed as the value for limit

When the application first starts the `Session` variable `limit` is set to 10 since it does not have any other value. This is what `setDefault` does and ensures that `limit` will always have a value.

The reason why we need to put the subscription inside a `Tracker.autorun` is to create a reactive context. Once the `Session` variable `limit` changes, the subscription to `workouts` gets rerun with the updated `limit` value. This means whenever the `limit` value changes, triggered by an event or directly from the JavaScript console, the subscription is updated automatically. Then the new data from the publication is added to the clients Minimongo and also rendered in the template.

To allow for a more convenient way to increase the number of documents to show, we can add a button with a click handler that adds 10 to the current `limit` value in the `Session` object (Listing 7-4).

Listing 7-4: Adding an event handler to increase the limit by 10

```
Template.workoutList.events({
  'click button.show-more': function(evt, tpl){
    var newLimit = Session.get('limit') + 10;
    Session.set('limit', newLimit); #A
  }
});
```

#A Changing the limit in the reactive Session variable updates the subscription automatically

As we can see it is not too difficult to remove the `autopublish` package and take control over the data that is available on the client. With reactive variables it also very easy to modify a subscription and the data that is available. The same approach we have used for limiting the amount of documents can easily be used for filtering and sorting as well. You could add a second button to provide a value of 0 or 1 to sort all document ascending or descending.

7.1.3 Publishing aggregated data to a client-only collection

Imagine you run 3 times a week, 5 weeks a month. Then you are not only incredibly fit, but you also need to look at 15 different entries to understand how many miles you went in a single month. That makes quickly comparing stats a pain. This is when data aggregation

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

comes in. Instead of showing all the fine details we sometimes need a summary over large amounts of data to make sense of it. Let's extend the application so we can say for sure that we ran further in June than in January.

If we had workout documents available on the client aggregating could be an easy task. By iterating over each document and adding the distance for each month on the client directly we could be all set. Unfortunately this approach has multiple downsides. One is you would need to pass a lot of data over the wire. If you would want to aggregate the data of the last 10 years you would end up sending thousands of documents down the wire. Another downside is that the computation takes quite some time and this would slow down the users UI, resulting in poor user experience.

If you are familiar with SQL you are probably already thinking about doing a `SELECT` with a `COUNT(*)` and a `GROUP BY`, but Meteor uses a NoSQL database. MongoDB itself is built to handle large datasets and give analytical insight, so it also offers way to aggregate data, just not using `GROUP BY`. We are going to use the *aggregation pipeline* to compare the distances between all months of the year. Minimongo, the client implementation in the browser, does not support using the aggregation pipeline, but this is okay as we will see.

The approach we will take involves creating a publication that does not send data directly from a `Collection`, but creates the aggregated data inside itself and returns this to all subscribers. This data is stored inside a collection that only exists on the client. It is not persistent in the server side MongoDB as it would leave us with redundant data.

At first we create a new publication named `distanceByMonth`. It does not have a corresponding collection inside the database. Where we previously put a `find()` operation on a database collection, the second argument to `publish()`will now hold the aggregation.

> **NOTE** This aggregation operates on-the-fly, meaning that every subscription will trigger the aggregation on database contents. If you notice that the processing takes a long time it may be a better option better to write the aggregated data to a dedicated collection.

The aggregation framework for MongoDB is not supported out of the box. There are several community packages available that provide aggregation capabilities we can easily add it ourselves. We are going to fall back to the core MongoDB driver, then define the actual pipeline, run the aggregation so that it does not block any other processes, and finally mark the subscription as `ready`.

The publication may send status messages to the subscribers, indicating that content has changed or that it is done with sending all available content. When publishing a database collection, these status messages are taken care of by Meteor. When using a custom publication they can be called directly.

- `added(collection, docId, fields)`

  When a new document is created the first argument is the collection, followed by the document ID. The third argument contains all fields of the document (excluding the `_id` field).

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

- `changed(collection, docId, fields)`

  For changed documents again the collection name and ID are passed as the first argument, followed by an object that contains all updated fields (fields with a value of undefined have been deleted from the document).

- `removed(collection, docId)`

  Takes two arguments: The collection name and the document ID that was removed.

- `ready()`

  Takes no arguments and informs the client that all available data has been sent.

Listing 7-5 gives you the full code which we will go through bit by bit. In order to utilize the MongoDB core driver we need to use `MongoInternals`, which is defined as part of the `mongo` package included with each new Meteor project. A reference to the default database used by Meteor is stored inside `db`. Since we are using the core MongoDB driver we may use all functions, including `aggregate()`. The `pipeline` variable contains an array with details about the actual aggregation. The MongoDB aggregation pipeline consists of stages. Each stage transforms the documents as they pass through the pipeline. First, all matching documents are determined. In this scenario all documents match as we have not defined any restrictions. Next all resulting documents, or rather they specified field contents, are grouped together. All workouts are grouped by month and are given a new `_id` that represents the month (1 = Jan, 2 = Feb, etc).

MongoDB is not reactive by itself and calling it would result in a synchronous call that would block all other server requests until the aggregation is finished. This is why we need a way to un-block the aggregation and receive a callback once it is done while maintaining the full Meteor context. Asynchronous calls to external components should always be wrapped inside `Meteor.bindEnvironment()`.

Using the Underscore library all month's results are added to the subscription distanceByMonth. Finally the `publish()` function signals the client that the subscription is ready.

Listing 7-5: Aggregation inside a publication

```
Meteor.publish('distanceByMonth', function(){
  var subscription = this;

  var db = MongoInternals.defaultRemoteCollectionDriver().mongo.db;#A

  var pipeline = [ #B
    {
      $group: {
        _id: { $month: "$workoutAt" },
        distance: { $sum: "$distance" }
      }
    }
  ];

  db.collection("workouts").aggregate( #C
    pipeline,
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
    Meteor.bindEnvironment( #D
      function(err, result){
        console.log('result', result);
        _.each(result, function(r){
          subscription.added('distanceByMonth', r._id, {distance: r.distance}); #E
        })
      }
    )
  )

  subscription.ready(); #F
});
```

#A Since there is no official support for aggregation from Meteor we use the core mongo driver
#B The aggregation settings create documents with the _id field equal to the month of the workoutAt field and the sum of all distances of this month
#C Creating the aggregation
#D Since we cannot use asynchronous code in a publication we use Meteor.bindEnvironment
#E We add the data to this subscription this way our selves
#F The subscription is ready to send the data to the client

On the client side we create a collection only available on the client that takes this data. We create this collection inside the client folder in the `client.js` file:

```
DistanceByMonth = new Mongo.Collection('distanceByMonth');
```

This looks and behaves exactly like you any other collection, but the data inside comes from our custom publication and not the server side MongoDB. We can use the data inside of this collection as we would normally do. You can create a new template and helper to display the data from the aggregation publication. Refer to the sample code for more details.

One downside of this approach is that this data is not reactive since the aggregation framework is just a dumb data source. That means if someone added a new workout for the month April with an 8 mile distance, the aggregated data on the client for April would not be increased by 8 automatically. When the page is reloaded the subscription would be initialized again, so that the screen would update correctly. This is definitely not how things are supposed to be inside a Meteor application where the client should be reactive. Next we will see how we can improve the publication to make this aggregated publication reactive again.

7.1.4 Turning an aggregation publication into a reactive data source

Unlike a normal `Collection.find()` the aggregation publication is not reactive. Nevertheless we want to update a client's aggregated data to updated reactively when data changes. Just like adding a workout would update the limited list automatically. What is missing to turn the aggregation into a reactive data source is an *observer* that monitors the workout collection and performs an action if a new workout is added.

Every collection cursor available – for example the returned cursor from calling `Workouts.find()` – has the ability to observe the documents that were added, deleted, or changed inside the collection. Which documents are observed depends on the query that is used by the `Collection.find()` method.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

By limiting the query for a `find()` we can keep an eye only on workouts with a type of `jogging` and react if a new document is added, changed or removed. At a later point we could add a different action for workouts of type `chess` or `aerobics`. The function we will use for monitoring updates to a data source is `observeChanges()`.

There are three cases that we can observe, each has an associated callback with one or more attributes. The associated callbacks are similar to those used for setting the publication status, but they do not require to pass a collection name as an argument:

- `added(docId, fields)`

  When a new document is created the first argument is the document ID, the second contains all fields of the document (excluding the `_id` field).

- `changed(docId, fields)`

  For changed documents again the ID is passed as the first argument, the second contains only the updated fields (fields with a value of `undefined` have been deleted from the document).

- `removed(docId)`

  Takes a single argument: The ID of the document that was removed from the collection.

NOTE Although they have the same names, the `added`, `changed`, and `removed` functions in a publications use a slightly different syntax.

Listing 7-6 shows the syntax of using all three callbacks on a watched `Workouts` query.

Listing 7-6: Observing changes in a collection

```
Workouts
        .find( query )
        .observeChanges({
          added: function(id, fields){
            // do something if a document was added matching the query
          },
          changed: function(id, fields){
            // do something if a document of the query changed
          },
          removed: function(id){
      // do something if a document of the query was removed
          }
        });
```

We know how to aggregate data in a publication and also how to create a callback if a document is added to a collection. This is all we need to make the aggregation publication reactive. The trick is to create an object within the publication that keeps track of all the aggregated data. This is created by the aggregation. Then we watch the collection and observe if a new document is added. If so, we can update the total distance of the month in

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

the object that keeps track about the aggregated data. The newly updated data is then sent down to the client by telling that the subscription changed. Let's look at the code:

Listing 7-7: Using observeChanges to update aggregated data

```
var subscription = this;
      var initiated = false; #A
      var distances = {}; #B

      //...

      var workoutHandle = WorkoutsCollection
          .find()
          .observeChanges({
            added: function(id, fields){
              if(!initiated) return; #A

              idByMonth = new Date(fields.workoutAt).getMonth() + 1 #C

              distances[idByMonth] += fields.distance; #D

              subscription.changed('distanceByMonth', #E
                idByMonth,
                { distance: distances[idByMonth] }
              )
            }
          });
```

#A We need this because the very first documents from the initial subscription should not affect the added callback
#B This object keeps track of all distances for each month
#C Create the id of the document. The +1 comes from the month starting at index 0
#D Update the distance for the month since a new workout was added
#E Inform the client that the subscription was changed

Now whenever a new workout is added to `Workouts` the object that tracks the aggregated data (`workoutHandle`) is updated and sends the change to the client. Go to your browser and add a new workout through the console. You will see that the aggregated data updates accordingly.

One last but very important thing to do is to clean up the publication accordingly. The observe methods would run endlessly if we do not stop them. The right moment to stop the observation of the collection is when the client subscription stops.

```
subscription.onStop(function(){ #A
        workoutHandle.stop(); #B
      });
```

#A The subscription (or this) has an onStop callback that is fired whenever the clients subscription is closed.
#B The handle that is returned from the observerChanges() function is used to stop observing.

Whenever the client stops the subscription, observing will be stopped as well.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

NOTE If you want to publish a single document you still have to use `collection.find({_id: options._id})` and not `findOne()`. This is because a publication must return a cursor while `findOne()` returns the actual result as an object.

7.1.5 Limit data visibility by user ID

We can now control the data that is send to the client and not every workout is send to it. Still we need to find a way how we can decide not only how many documents a user can see but we also want that he can only see his own workouts and not the once from other users. For this we add the accounts packages we looked at in the chapter before. Inside of a publication we can access the `userId` of the currently logged in user by `this.userId`, which is null if the user is not logged in.

Listing 7-8: Send only the data a user is allowed to see

```
Meteor.publish('workouts', function(options){
        check(options,
          {
            limit: Number
          }
        );

        var qry = {
          userId: this.userId #A
        };
        var qryOptions = {
          limit: options.limit || 10,
          sort: {workoutAt: 1}
        };

        return WorkoutsCollection.find(qry, qryOptions);
      });
```

#A Query for all workouts that belong to the currently logged in user

It is important to store the reference to the user inside of the workout document. If we do this, it is as simple as adding `{ userId: this.userId }` to the query in the `Workouts.find(qry…)` function. You will also note that if you log in or out the data changes reactively.

The aggregated data is a little bit more complicated to adjust since we need to match the aggregation itself and also the query that we want to observe.

Listing 7-9: Aggregation of a users documents

```
Meteor.publish('distanceByMonth', function(){
        var subscription = this;
        var initiated = false;
        var distances = {};
        var userId = this.userId;
        var db = MongoInternals.defaultRemoteCollectionDriver().mongo.db;
        var pipeline = [
          {
            $match: { userId: userId } #A
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
    },
    {
      $group: {
        _id: { $month: "$workoutAt" },
        distance: { $sum: "$distance" }
      }
    }
  ];

  db.collection("workouts").aggregate(
    pipeline,
    Meteor.bindEnvironment(
      function(err, result){
        console.log('result', result);
        _.each(result, function(r){
          distances[r._id] = r.distance;
          subscription.added('distanceByMonth', r._id, {distance: r.distance});
        })
      }
    )
  )

  var workoutHandle = WorkoutsCollection
    .find({userId: userId}) #B
    .observeChanges({
      added: function(id, fields){
        if(!initiated) return;

        idByMonth = new Date(fields.workoutAt).getMonth() + 1

        distances[idByMonth] += fields.distance;

        subscription.changed('distanceByMonth',
          idByMonth,
          { distance: distances[idByMonth] }
        )
      }
    });

  initiated = true;
  subscription.onStop(function(){
    workoutHandle.stop();
  });
  subscription.ready();
});
```

#A The aggregation should only be done over documents that matches the userId.
#B In this publication there should only the documents with the logged in userId observed.

There is not much to do to update the aggregation publication so that it depends on the user. Only the aggregated documents with the correct `userId` should be counted in and lastly the documents that should be observed have to be found by the query `{userId: this.userId}`.

7.2 Meteor Methods

Meteor makes sending data from clients to the server very easy. But on the web you can never trust the data that comes from a client. You can never be certain that there is not a malicious hacker on the other end trying to access or modify sensitive data. Therefore

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

everything coming from a client must be validated before processing. Using the browser's console every validation could be bypassed. This applies to all web applications, regardless whether they are written in Java or JavaScript.

The solution is to implement a safe-guard on the server-side that deals with all write actions. Meteor uses a concept similar to remote procedure call (RPCs) that can be called from the client and are executed on the client and afterwards on the server, too. These are known as `methods`. They not only help secure applications but they are capable of making them more user-friendly using latency compensation.

Storing data inside a database takes relatively long, depending on network connections and write speeds. Figure 7-x illustrates how Methods are used to apply security and increase the perceived speed of write operations. First, a user submits a new workout. An event handler receives the data and passes it to a client method. This method will perform data validation to check whether the distance entered is valid. If all checks are passed it simulates storing data to the database by adding it to the local Minimongo instance, and updates the screen for the user. This happens in the fraction of a second as all events take place in the memory of the local computer or mobile device. Only now the data is sent to the server, where the same method will be executed. Some server-specific checks might be added as well, like making sure the user ID is correct. If all validations pass, data is actually stored into the database and success is confirmed to the client.

Methods can be used for database operations but also for anything else that needs to take place on the server, like sending emails or tiggering processes.

Figure 7-3: User-generated content must always be validated on the server as it may be possible to bypass the code executed on the client

In this section we will replace the default `insecure` package that lets anybody access every database document with a secure usage of methods. Users will be able to add their own workouts to the collection. For this we will use a method call to send the write actions instead of direct inserts and updates.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

7.2.1 Removing the insecure package

Just like the `autopublish` functionality was provided in form of a package, there is a package called `insecure` that allows a client to initiate writes to the server-side database. As the name suggests it is not intended for production environments but rather to speed up the development process. To remove the package from your application stop the Meteor server and issue the command

```
$ meteor remove insecure
```

Having the possibility to update and insert data from the browser console is very useful during development, so it depends on your personal preference whether it is better to remove the `insecure` package early on in a project or leverage its power for quicker development. Either way, a production application should never be deployed with `insecure` still active.

Once the `insecure` package is removed and the server is started again any attempt to update, insert, or remove a document from the client will result in an error message in the console (see Figure 7-x).

Figure 7-4: When the insecure package is removed access is denied in the browser for any write operation

REMEMBER Read access to data is defined using publications, so removing the `insecure` package does not protect any sensitive data. It simply prohibits the client from writing to the server-side database directly.

7.2.2 Using methods to write data to collections

The place to put methods depends a little bit on what you want to do and there are two places you can put them. If you put a method inside of the server folder you can still call them from the client. But the method is only processed on the server and not on the client. If the method is shared between the server and the client the process is similar to the Collection.insert() functionality. This means a method call is processed on the client immediately. Then if all goes well it is processed on the server as well and if something goes wrong here the function is reverted on the client. This way you gain the latency compensation with methods as well.

We now want the user to be able to add workouts himself by adding a simple form. If the form is submitted we want to extract the data from the form and use it inside of a method

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

call. In the method we want to make sure that the data is allowed, valid and comes from a logged in user, since a guest should not be able to add workouts. Finally we create a new workout inside the method. `Meteor.call()` takes one mandatory argument: the method name. Additionally we can add as many arguments as we like and thos will be available inside the method. The last argument we provide is a callback function to deal with the results returned by the method. The callback itself takes two arguments: `error` and `result`. The value for `error` remains undefined as long as the method finishes as expected. `result` contains the return value of the method, in this case the document ID for the newly inserted workout.

Listing 7-10: A method call from the client

```
Template.addWorkout.events({
        'submit form': function(evt, tpl){ #A
          evt.preventDefault(); #B

          var distance = parseInt(tpl.$('input[name="distance"]').val()); #C

          Meteor.call('CreateWorkout', {distance: distance}, function(error, result){
      #D
            if(error) return alert('Error: ' + error.error);#E
          });
        }
      });
```

#A We listen to the submit form event as usual
#B The default behaviout of a form submit should be prevented since it would reload the page
#C We use jQuery to extract the data from the distance input field and make it an integer
#D The method is called by its name and additional parameters
#E The method has a callback that is called if an error happens or a result was returned from the server

A method always has a name and can have as much parameters as you like. This way you can send data from the client to the server using a method. Next we need to define the method and since its purpose is to create a workout I tend to put it into the file of the Workouts collection. As another benefit it has its place where it is available on the client and the server as well. Since we create a new document and we want to have the benefits of the latency compensation it is a good spot.

Listing 7-11: Using a method to create a new document

```
Meteor.methods({ #A
        'CreateWorkout': function(data) { #B
          check(data, { #C
            distance: Number
          });

          var distance = data.distance;
          if(distance <= 0 || distance > 45){ #D
            throw new Meteor.Error('Invalid distance'); #E
          }

          if(!this.userId){ #F
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
            throw new Meteor.Error('You have to login');
          }

          data.workoutAt = new Date(); #G
          data.type = 'jogging'; #G
          data.userId = this.userId; #G

          return Workouts.insert(data); #J
        }
      });
```

#A Creating a method follows the scheme of e.g. helpers. A method function that takes a key, value object as a parameter
#B The key of the object is the name of the method
#C We use check to make sure that only data is used in a method that we allow
#D We do a vaildation on the distance
#E If the validation fails we throw a new Meteor.Error. It is like a normal JavaScript Error but is automatically populated to the client
#F The userId of the currently logged in user is accessed via this in a method
#G We add some data to the document we want to create
#H We can be sure now that the data that goes into the collection is save

If you look at the method you see that the data parameter is what is finally passed to the `Workouts.insert` method. Because of this it is really important to make sure that you know exactly what is inside of the data object that is coming from the client. If you would not make any security checks a user could add any data inside your `Workouts` collection. We will again use the `check()` function and look at it in a little more detail.

USING AUDIT-ARGUMENT-CHECKS TO VALIDATE ALL USER INPUT

Each argument sent to a method should be checked before actually processing it. The more form fields are used, the harder it is to keep track whether every user input was checked yet. Meteor ships with a package called `audit-argument-checks` which checks that every argument was indeed checked before it was used. Add it to your project via the command line:

```
$ meteor add audit-argument-checks
```

Everytime the client sends an argument to the server for processing `audit-argument-checks` will ensure that it is checked first. Therefore we need to add checks to all methods. If there is no check in place a method will still execute but you will see an exception on the server as in Listing 7-12:

Listing 7-12: Console message for unchecked values in methods

```
Exception while invoking method 'CreateWorkout' Error: Did not check() all arguments
      during call to 'CreateWorkout'
    at _.extend.throwUnlessAllArgumentsHaveBeenChecked (packages/check/match.js:352)
    at Object.Match._failIfArgumentsAreNotAllChecked (packages/check/match.js:108)
    at maybeAuditArgumentChecks (packages/ddp/livedata_server.js:1596)
    at packages/ddp/livedata_server.js:648
    at _.extend.withValue (packages/meteor/dynamics_nodejs.js:56)
    at packages/ddp/livedata_server.js:647
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
    at _.extend.withValue (packages/meteor/dynamics_nodejs.js:56)
    at _.extend.protocol_handlers.method (packages/ddp/livedata_server.js:646)
    at packages/ddp/livedata_server.js:546
```

Table 7-1 Match patterns for checking variable content

Pattern	matches
`Match.Any`	Matches any value.
`String, Number, Boolean, undefined, null`	Matches a primitive of the given type.
`Match.Integer`	Matches a signed 32-bit integer. Doesn't match Infinity, -Infinity, or NaN.
`[pattern]`	A one-element array matches an array of elements, each of which match pattern. For example, `[Number]` matches a (possibly empty) array of numbers; `[Match.Any]` matches any array.
`{key1: pattern1, key2: pattern2, ...}`	Matches an Object with the given keys, with values matching the given patterns. If any pattern is a Match.Optional, that key does not need to exist in the object. The value may not contain any keys not listed in the pattern. The value must be a plain Object with no special prototype.
`Match.ObjectIncluding({key1: pattern1, key2: pattern2, ...})`	Matches an Object with the given keys; the value may also have other keys with arbitrary values.
`Object`	Matches any plain Object with any keys; equivalent to `Match.ObjectIncluding({})`.
`Match.Optional(pattern)`	Matches either `undefined` or something that matches pattern. If used in an object this matches only if the key is not set as opposed to the value being set to undefined.
`Any constructor function (eg, Date)`	Matches any element that is an instance of that type.
`Match.Where(condition)`	Calls the function `condition` with the value as the argument. If `condition` returns `true`, this matches. If `condition` throws a `Match.Error` or returns `false`, this fails. If `condition` throws any other error, that error is thrown from the call to check.

7.3 Summary

- Before going into production `autopublish` and `insecure` should be removed,

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Publications and Methods should take their place

- Pub/sub is Meteor's way to get data from the server to a client
- Publications may return data from a database or publish customized data
- Publications can securely limit the published data by document fields such as a user/owner ID
- Writing to the database via server-side `Methods` is secure and more flexible than using allow/deny patterns
- The `audit-argument-checks` package helps to ensure all data provided by a client is validated before it is used

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

8

Routing

This chapter covers

- Adding routing capabilities to your Meteor application
- Create layouts
- How to improve the code structure using the Iron-Router
- Creating Server Side routes

If your application is getting bigger and more complex you will have lots of subscriptions and publications, more and more collections but also many different templates that have to be rendered at a specific moment. You need a way to organize what subscriptions you need when, what to render and also what data context should be available in the rendered templates.

One good approach that comes with even more benefits is to handle this complexity with routes. This means you decide what to subscribe to, what to render with what data context depending on unique URLs. The router handles all this and the most commonly used router with Meteor is the Iron-Router.

The Iron-Router is a community package maintained by Chris Mather and Tom Coleman. Tom Coleman wrote one of the first router for Meteor called `meteor-router` and Chris Mather also started a routing project called `meteor-mini-pages`. Luckily for the Meteor community they decided to focus on developing one router together and the result is the Iron-Router package.

The MDG had a router on its own roadmap once but decided that it is not necessary for them to build one, even though routing is a crucial aspect of every web framework. The reason simply was that the efforts in the community building a router were so good, that a router by MDG itself was not needed.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

8.1 Routing in client side web applications

If you click a link on a normal website the URL changes in the browser. The browser then requests the resource from the server that belongs to the new URL. The first thing the webserver does after it receives a request with a given route is going through a dictionary of all routes it knows. If the route of the request matches a known route in the dictionary the defined action is performed. At the end of each action the response is created and send back to the browser who renders the html it received for the new route. The router typically handles all of this functionality.

Figure 8-1: The client's HTTP request is handled by a server side router which responds with HTML

Let's say you are on a community website and you see a list of profiles, one of them is from Manuel. If you click on the link 'Manuel' the URL of the browser changes and a request is send to the server. The server performs the actions defined for the route to generate the HTML needed for Manuel's profile. At the end the response is send back.

With Meteor you create client side web applications. This means if you click a link, there is no request and a completely new html document in response. For a web application if you click a link the DOM is changed directly without the need of a new http request to the server. This means you basically do not need any routes because you can specify the functions that changes the DOM directly on the event handler of a click event on an anchor element.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Figure 8-2: A client side web application can handle the DOM manipulation in an event handler

In the same example where links to profiles are listed on a website, the process is quite different to the one of a classic website. If you click on the link for Manuel's profile the URL does not change at all but instead the event is handled by a javascript function directly in the browser. The DOM could be changed directly on the click and show a loading indicator, for example a simple string `Loading …` . In parallel the app fetches some data from the server that is needed in order to render the profile. In Meteor you do this by updating or creating a new subscription. If the new data is available on the client the DOM is changed again and the new profile is rendered.

If you change the current HTML based on click events like this, without changing the browser's URL you have two major disadvantages.

One disadvantage is the maintainability of your application because the URL combined with a dictionary of routes that your application understands is a very good starting point if you want to figure out where to look in your code. Think of a complex application you have to join as a developer that is completely new to you. If you click on a profile of the example above and the URL changes to `/profiles/manuel` you could start looking at the defined routes and you see what action is performed. You can use the URL as a very good starting point where to look for relevant code, which is very important.

The main reason why you should always use URL's even for a client side application is the architecture of the web itself. Its URLs define every resource you can reach in the web and this is very important for several things. URLs enable you to share content to your friends. If your community application would consist only of one URL you could never share a concrete and interesting profile to anyone. But also if you perform actions like filtering or sorting of tables it would be good to reflect this in the URL as well. Think for example about a special and very important filter and sorting combination of a large data set that you need to access very often. If you can access this exact data set with an URL you could easily bookmark it and

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

reach it much faster as if you would have to setup the configuration every time you want to access it.

URLs are not only important for humans browsing through the web but for other applications as well. If a search engine crawls a website it always tries to understand the content of the given document that relates to a concrete URL. If a user then types in some search phrases into the search bar, the search engine will try to present the best matching URL as a response. If your application would have only one URL for any content it contains, the search engine could not represent anything to the concrete search phrase of the searching user.

Since routes are so important for Meteor applications as well, the Iron-Router allows us to use a router as well. The router is available on the client side and the server side as well. On the client side the router helps to setup new subscriptions but also end old ones based on the current URL. Also it takes care about rendering the specified template based on the current URL. As you will see during this chapter, the Iron-Router has some more capabilities.

Figure 8-3: The Iron-Router listens for URL changes and performs the actions defined by you

On the server side you can use the Iron-Router to react as a normal server side router as well. This means you can create normal REST interfaces with a Meteor application as well. The main use case of the Iron-Router is the client side routing and this is what will be focused in this chapter. But we will have a look at the server side routing at the end of this chapter as well.

8.2 Implementing a multi site application

In this chapter we want to build a small example for a multi site web application. The scope of the application will be a community where you can see users profiles and also comment on their profile page. An important aspect of an application like this, but for nearly any web

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

application is to have URL's that are shareable. Think of your profile page on our new community. Without your unique URL you could not share it to anyone or even access it yourself.

At the end of this chapter we will have build an application that could contain an unlimited amount of profile pages each with a unique and sharable URL.

Figure 8-4: A simple profile page of a multi page community application

Since we know how important routing is and also what the best router for Meteor is we will start to develop the community application. The application will have multiple routes for static pages but also for dynamic pages that require data in order to render the templates.

8.2.1 Adding the Iron-Router

Meteor does not come with a router as a core feature itself but instead there was a community effort to build a router very early on. The Iron-Router is such a high quality package and well maintained that it became the de facto standard in routing for Meteor very fast. This was also the reason for Meteor to not build a router itself, since it was just not necessary. For your Meteor project that means you have to add the Iron-Router as a package, first:

```
$ meteor add iron:router
```

After adding the Iron-Router you get access to the Router object in your application in the client scope but also in the server scope. The reason that the Router object is available in the client and the server scope is that you can perform server side routing as well. We will learn about server side routing later on, too. For this reason you should create a router folder just at the root of you application folder. Inside this folder you can put all router related files beginning with the `routes.js` file which contains all route definitions.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Figure 8-5: The Iron-Router works on the client and server scope, so putting a routes.js file somewhere outside the client or server folder is a good idea.

The routes.js file will be the place where we define all the routes the application should contain. It is a good practice to have every route of your application in one file so you can have a very quick overview about the application.

8.2.2 Creating your first routes

The first goal we have after we added the Iron-Router is to setup two really basic routes. One is the standard home route that should be rendered if you are on the root of your application. This route relates to the path "/". The second route is a simple about page that should be rendered if you access the path "/about".

Figure 8-6: By clicking on the About link in the top navigation the URL changes to /about

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

NOTE To make the relevant code we are looking at less complex we will not show any of the bootstrap markup. The code for this chapter will contain all the relevant bootstrap code in order to achieve a little bit more beautiful result. To add the bootstrap CSS-Framework you have to add the package: `$ meteor add mizzao:bootstrap-3`

The file structure we use for the first step should look like this:

```
Manuel-Schoebels-iMac:ch06 Manuel$ tree
.
├── client
│   └── views
│       ├── home
│       │   └── home.html
│       ├── index.html
│       └── staticPages
│           └── about.html
└── router
    └── routes.js

5 directories, 4 files
Manuel-Schoebels-iMac:ch06 Manuel$
```

Figure 8-7: To create two simple routes we need to define the routes and also the templates that should be rendered for a given route

The `home.html` file contains the template that should be rendered if a user is on the root path "/". If the user navigates to the "/about" page there should be rendered some static content with further information about the application. For this we use the about template that in this case is in the `staticPages` folder. The `index.html` file contains some general templates and also the <head> element for the application.

Listing 8-1: Initial templates for the community application

```
// index.html
<head>
  <title>ch06</title>
</head>

<template name="header"> #A
  <nav>
    <ul class="nav navbar-nav">
      <li><a href="/">My Little Community</a></li>
      <li><a href="/about">About</a></li>
    </ul>
  </nav>
</template>

// home.html
<template name="home">
  {{> header}} #B

  <h1>Home</h1>
</template>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
// about.html
<template name="about">
  {{> header}} #B

  <h1>About</h1>
</template>
```

#A The header template contains the navigation so it can be included in other templates.
#B The templates include the header template so the navigation is on top of every view

There is nothing too fancy about the templates, yet. The navigation inside the header template contains two anchor elements. One links to the root path `<a href="/">My Little Community</a>` and the other one links to the about page `<a href="/about">About</a>`. Next we want to add those routes to the Iron-Router and render the appropriate template.

Listing 8-2: Setting up different routes

```
// routes.js
Router.route('/', function(){ #A
  this.render('home'); #B
});

Router.route('/about', function(){ #A
  this.render('about'); #B
});
```

#A Defines a path and associates it with a function to call if the URL matches this path
#B The specified template is rendered

The Router object has a route function that takes two parameters, the path and the associated function, which is called if the URL changes and the path matches the specified one. Inside the scope of the function that is called, you have access to the current instance of the so called `RouteController` object via `this`. With the help of the `RouteController` you can e.g. render a template to a specific location into the DOM. In this case, since you have nothing else defined, the template specified with the string parameter of the `this.render('templateName')` function will be rendered simply inside the `<body>` element.

8.2.3 Define a layout depending on a route

SINGLE LAYOUT

In the example that was already implemented we included the navigation at the top of the application inside every rendered template.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Figure 8-8: Reusing templates for each route without using a layout

A more effective way to reuse the header template is using a layout templates for each route and alter only a part of the layout based on the current route. This is especially useful if your layout gets more complex or you want to use multiple layouts through out your application.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Figure 8-9: A layout template for reusability with a dynamic area that is replaced with the template that should be rendered for the current route using {{> yield}}.

As you can see for both routes the `masterLayout` template should be rendered which always has the header template on top. The dynamic part is changed depending on the current route. If the current path is "/" the dynamic part of the layout should be replaced with the template called `home` and if the path changes to "/about" the dynamic part of the layout has to be exchanged with the template called `about`.

In the layout template you use the `{{> yield}}` template helper that comes with the Iron-Router package. With `{{> yield}}` you can specify where the template for the route should be rendered exactly.

In order to specify a layout that should be used for each route you can use the `Router.configure()` function.

Listing 8-3: Defining a global layout template for each route

```
// router/config.js
Router.configure({
  layoutTemplate: 'masterLayout'
});
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

USING MULTIPLE LAYOUTS

We need to have not only one layout template for any route but instead we want to have multiple layouts since the profile page should have its own layout template. First we have a look at the old `masterLayout` and the new `profileLayout`.

Figure 8-10: The second layout for the profile pages should be used on routes for profiles

As we can see the profile layout has two columns. The one on the left renders the profile picture and on the right side contains some profile information. We still have one main content area that is specified with the `{{> yield}}` template helper. The second content area on the left got a name so it could be referenced to later in the route function. For that you can use a named yield like this `{{> yield "name"}}`.

In the route function you can specify which layout should be used. If you do not specify any layout the one set by the `configure` function is used. And if there isn't even one set with the `configure` function the template is just rendered into the `<body>`.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Listing 8-4: Set a layout inside the route function

```
Router.route('/profiles/manuel', function(){
  this.layout('profileLayout'); #A
  this.render('profileDetail'); #B
});
```

#A sets a layout
#B the layout template is used to render the specified profileDetail template

DEFINE THE CONTENT TEMPLATE FOR NAMED YIELDS

If you have a named yield like we have in the example from above {{> yield "left"}} you want to define what template should be rendered there. For this you have some options.

First you can do this in the template you render itself with a template helper:

Listing 8-5: Render a template inside a named yield with template helpers

```
<template name="profileDetail">
  {{#contentFor 'left'}} #A
    <img src="…">
  {{/contentFor}}

  <h1>Manuel Schoebel</h1> #B
  <p>I like to eat good food and cooking it myself!</p> #B
</template>

// Another variant is using the template helper contentFor and specify the template
      to render

<template name="profileDetail">
  {{> contentFor region='left' template="profileImage"}} #C

  <h1>Manuel Schoebel</h1> #B
  <p>I like to eat good food and cooking it myself!</p> #B
</template>

<template name="profileImage">
  <img src="…">
</template>
```

#A The html inside this block helper is rendered in the yield named 'left'
#B This html is rendered into the main yield
#C Using a template helper you can set the template and region as well

The second way to render a template into a named yield is using the render function:

Listing 8-6: Render a template inside a named yield using JavaScript

```
Router.route('/profiles/manuel', function(){
  this.layout('profileLayout');
  this.render('profileImage', {to: 'left'}); #A
  this.render('profileDetail');
});
```

#A the option "to" specifies where to render the given template

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

We have still defined the route for a specific profile in a static way since we use the route `/profiles/manuel`. Of course we want to have only one route for any profile detail pages, at which we will look next.

8.2.4 Set the data context depending on a route

On the home site of our application we want to have multiple profiles that links to the profiles detail page. The profile detail page of course should have a template that renders the data of the exact profile that is specified through the URL. That means the route `/profiles/stephan` should render the profile detail template but with Stephan's profile data and the route `/profiles/manuel` should render the profile detail template but with Manuel's profile data.

Figure 8-11: With the Iron-Router you can define the rendered template, the layout and also the data context

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

As you can see on the image above we have a list of profiles on the home site and if you click on a `more…` link inside of a profile-list-element on the home site you will be routed to the profile's detail page.

The data context of the home site has to be a set of profiles you want to render. On the profile's detail page you only want to have the specific data of one user as a data context of the profileDetail template. Since all of this is depending on the URL you can define this with the help of the Iron-Router as well.

We will assume that the autopublish package is still active so that all data is available on the client. We will change that in the next chapter.

Listing 8-7: Setting the data context with the Iron-Router

```
Router.route('/', function(){
  this.render('home', {
    data: function(){ #A
      return {profiles: ProfilesCollection.find()}; #B
    }
  });
});

Router.route('/about', function(){
  this.render('about');
});

Router.route('/profiles/:_id', function(){ #C
  profile = ProfilesCollection.findOne({_id: this.params._id}); #D
  this.layout('profileLayout');
  this.render('profileDetailLeft', {
    to: 'left',
    data: function(){ #B
      return profile; #E
    }
  });
  this.render('profileDetail', {
    data: function(){ #B
      return profile; #E
    }
  });
});
```

#A The profiles are accessible via {{profiles}} in the home template
#B The data property of the options object for a route sets the data context of the rendered template
#C A variable route part is notated with a leading ":key"
#D You can access the variable part of the path via this.params.key
#E The data is directly accessible e.g. via {{name}} in the profile templates

The home route now sets the data context of the template home equal to an object that contains all profiles available on the client and makes them accessible via `profiles`. This way you could easily iterate over all single profiles with `{{#each profiles}}…{{/each}}` within the home template.

In case of the profile detail page we chose the path to be `/profiles/:_id`. The part with the leading `:` means that this is a variable inside the path. This variable is accessible through

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

the `params` attribute of the current route controller instance. That means you can access the current value in the URL with `this.params._id`. This way we can relate to a special document from the database easily, like we did with the profile collection. The data option of the RouteController's render function does not have to be a function. It could also be a static value like an array as well.

Not only the data context is depending on the current route but also which data should be subscribed to. We will look at that next.

8.2.5 Define subscriptions with Iron-Router

We have already seen that a lot of stuff depends on the current route. The layout that should be used and also the templates that have to be rendered into the layout. Also we can set the data context of the templates through the Iron-Router, too. Since we normally do not have the autopublish package added to the project, we also have to choose which subscriptions should be active depending on the current route as well.

On the home path of the social community we are building in this chapter we would like to show some random profiles, but only 10 of them as the maximum. This means we have to subscribe to the data. But instead of being subscribed to this data all the time, we only want to be subscribed to it for the home route. If we go to the profile's detail page we do not need to have all the data of the 10 profiles available anymore.

At first we will change the route for the home template a little bit. Instead of passing a route function in which we call `this.render()` we will pass some options instead but the result will be the same.

Listing 8-8: Define a route's behavior by options only

```
Router.route('/', {
  template: 'home', #A
  data: function(){ #B
    return {profiles: ProfilesCollection.find()};
  }
});
```

#A instead of calling this.render('home') you can specify the template to render with the template option
#B The data context can be specified with the data option

As you can see using options instead of the route function you can save some lines of code and it works perfectly fine if you do not need some special logic.

Our goal is to subscribe to some data and render a loading indicator while we are waiting for the data to be received by the client. For this the Iron-Router comes with the `waitOn` option where we can define all subscriptions we need. The Iron-Router automatically shows a loading template if you use the `waitOn` option. Of course you could easily change the default loading template using the `loadingTemplate` option in the route's options or the global router configuration.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Listing 8-9: Subscribing based on a route

```
Router.route('/', {
  waitOn: function(){
    return Meteor.subscribe('profiles'); #A
  },
  template: 'home',
  data: function(){
    return {profiles: ProfilesCollection.find()};
  }
});

// Publications on the server
Meteor.publish('profiles', function(){
  profiles = Meteor.wrapAsync(function(cb){ #B
    Meteor.setTimeout(function(){
      cb(null, ProfilesCollection.find({}, {limit: 10})); #C
    }, 2000);
  })();

  return profiles; #D
});
```

#A You could also wait on multiple subscriptions by returning an array of subscription handles
#B This snippet allows you to simulate some waiting time. Do not worry if it looks a bit complicated
#C Here the actual query to the MongoDB happens and the result is stored in the profiles variable
#D The collection cursor from the MongoDB query is returned from the publication

If the home site is loaded now, we will see a loading indicator and once the subscription is ready, the home template will be rendered with the correct data.

Figure 8-12: The Iron-Router automatically renders a loading indicator if you use waitOn

This is basically all you need to get started with an Meteor application using the Iron-Router to create a multi-site application. The Iron-Router not only helps to organize the code but also lets you define very accurate which templates should be rendered, to what data the application needs to subscribe to and also what data should be available in the templates data context.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

8.3 Advanced Iron Router Capabilities

We have learned the basics of the Iron-Router in order to create multi site applications. But there are techniques that are really useful and commonly used in more advanced applications that we want to learn next.

8.3.1 Active class for a navigation link

If you have a navigation for your application it is a good practice to highlight the link that leads to the side a user is currently on. This way a user can see directly where he is on the application right now.

Figure 8-13: An active navigation item has a css-class active and with this a highlighted UI

For this functionality we need a global template helper so it is possible to use it for any navigational link. The concept for the global helper is to check whether or not the currently active route matches the route of the link.

Listing 8-10: A global template helper to highlight an active link

```
// Defining the global template helper
Template.registerHelper("isActiveRoute", function(routeName){
  return Router.current().route.getName() === routeName; #A
});

// Usage of the isActiveRoute template helper
<nav>
  <ul>
    <li class="{{#if isActiveRoute 'about'}}active{{/if}}"> #B
      {{#linkTo route="about"}}About{{/linkTo}}
    </li>
  </ul>
</nav>

// It is important to set the name property of a route
Router.route("/about", {name: "about"});
```

#A Returns true if the name of the currently active route equals the parameter
#B The template helper takes a string of the route name to check as a parameter

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

You can use this template helper on every navigational link where you want to set a class depending on the current route name. You can of course use this helper for anything else that needs to check the name of the currently active route.

8.3.2 Wait for external libraries to load

Meteor loads every JavaScript to the client at the very beginning. If your application contains a lot of external JavaScript libraries it is good idea to not put it into the main application folder but to load it as an external library only if it is needed. If you are using the Iron-Router it is a good idea to load the external library based on the route. Also often times it is a good idea to wait until the library is loaded before you render the templates.

For this you can use a package called wait-on-lib:

```
$ meteor add manuelschoebel:wait-on-lib
```

The package enables you to use an object called `IRLibloader` which can be used in a `waitOn` function of the iron-Router just like you do with a normal Meteor.subscription. Let's say you want to load a special library only for the `/profiles/:_id` url. The library lies in our public folder and is accessible via `/my-library.js`, then you can simply use the Iron-Router `waitOn` function to wait on the lib to load before rendering the page:

Listing 8-11: Wait for external libraries to load

```
// inside the ProfileController.js
…
waitOn: function(){
  return [
    Meteor.subscribe(…), #A
    IRLibLoader.load("/my-library.js") #B
  ]
}
…
```

#A You can subscribe to publications as you would normally do
#B The IRLibLoader acts exactly like a subscription meaning that you will also see e.g. the loading indicator until the library is loaded

Even though Meteor is loaded as one complete application it still is possible to load libraries you do not need for every visit of the application separately in order to reduce the data that has to be transferred on an initial page load.

8.3.3 Using Controllers for routes

Until now we added all functionality of a route directly to the route itself. If you would continue doing this in a large application the routes.js file would be getting blown very much. Also the idea to put all routes in one routes.js file is to be able to quickly get a good understanding what the application is capable of and where to look for functionality based on the routes.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Luckily we can specify a controller for a route and put that into its own file. This way we can take out all logic from the routes.js file and separate it. In order to specify a controller for a route you have multiple options, one would be to set it explicitly as a string or an controller object itself.

Listing 8-12: Using a Iron-Router controller

```
// router/routes.js
Router.route('/', { controller: 'HomeController' }); #A

// controller/homeController.js
HomeController = RouteController.extend({ #B
  waitOn: function(){
    return Meteor.subscribe('profiles');
  },
  template: 'home',
  data: function(){
    return {profiles: ProfilesCollection.find()};
  }
});
```

#A Using controllers makes the routes.js file much more readable
#B Every controller extends the basic RouteController object

A RouteController can have the same attributes like the route's options. This means you can also create your own custom action function, specify a `layoutTemplate` and so on. After some cleanup our route.js file can look much cleaner:

```
Router.route('/', { controller: 'HomeController' });
Router.route('/about', 'about'); #A
Router.route('/profiles/:_id', { controller: 'ProfileController' });
```

#A The most basic route can also only specify a template to render as the second parameter

8.3.4 Named routes and link helper

A very common practice is to not hard code any links of your app into e.g. an href attribute of an anchor element. The reason for this is that if a route changes you would have to edit all hard coded routes yourself. In order to solve this problem you can give routes names and use a helper to generate the link path by a helper using the routes' names. The name of a route is simply set in the options of the route's definition. In order to use the name of the route you use the `pathFor` template helper.

Another advantage of naming routes is that you do not have to specify the controller anymore. If your route is named `home` then the Iron-Router automatically looks for a controller called `homeController` or `HomeController`.

Listing 8-13: Using named routes

```
// router/routes.js
Router.route('/', { name: 'home' }); #A
Router.route('/about', 'about', { name: 'about' }); #B
Router.route('/profiles/:_id', { name: 'profile' }); #C
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
<template name="header">
  <nav>
        ...
    <ul>
      <li><a href="{{pathFor 'home'}}">My Little Community</a></li> #D
      <li><a href="{{pathFor 'about'}}">About</a></li> #E
   </ul>
  </nav>
</template>

<template name="profilePreview">
  <img src="{{profileImg}}">
  <div>
    <h3>{{name}}</h3>
    <p>{{profileText}}</p>
    <a href="{{pathFor 'profile'}}">more...</a> #F
  </div>
</template>
```

#A The route name home relates to the HomeController or homeController
#B This still renders the about template
#C The route name profile relates to the ProfileController or profileController
#D {{pathFor 'home'}} renders as /
#E {{pathFor 'about'}} renders as /about
#F The route named profile contains a dynamic part ":_id" and since the data context of the profilePreview template contains the _id the link is rendered correctly

If a named route contains a variable like `:_id` it is important that the `{{pathFor}}` template helper has access to this attribute. You could achieve this by setting the data context through the Iron-Router or you could also use the `{{#with _id="1234"}}…{{/with}}` block helper.

If you do need a link that contains the URL's protocol, hostname and port also you can use the `{{urlFor 'routeName'}}` helper instead.

Once you named your routes you can also use the `{{#linkTo}}` block helper that renders the anchor element for you.

Listing 8-14: Using the linkTo block helper to render anchor elements

```
{{#linkTo route='about'}}About{{/linkTo}}

// renders to
<a href="/about">About</a>

{{#linkTo route='home' class='navbar-brand'}}
   My Little Community
{{/linkTo}}

// renders to
<a class="navbar-brand" href="/">
   My Little Community
</a>
```

Any attribute you add to the `{{#linkTo …}}` block helper will be rendered to the anchor element, too. This way you can add classes, data-, id attributes and so on.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

8.3.5 Using Hooks

Hooks are especially practical to create some functionality you need to use for multiple routes. In our community application there are typically lots of routes you can only access if you are a logged in user. Or you may want to track some statistics and count views, for example the views we count for each profile. In order to track each profile view we will use the onRun hook. This hook runs exactly once even though a computation invalidates and a rerun finds place. So this is the perfect hook to use in order to increase the view count.

Iron-Router Hooks

For every hook you can create one function or an array of multiple functions that all will be called.

onRun: Is called when the route runs for the first time. It runs only once!

onRerun: Is called every time if a computation is invalidated

onBeforeAction: This function is called before your action function or route function runs. If there are more than one function, you have to make sure that the next function is called since this does not happen automatically with onBeforeActions. If you want the next onBeforeAction to be called you have to call this.next()

onAfterAction: This function is called after the action or route function ran.

onStop: If a route stops, e.g. a new route is run, this hooks are called.

To implement the incrementing of the views statistic of a profile we add the onRun hook to the ProfileController:

Listing 8-15: Adding a hook to a RouteController

```
// controllers/ProfileController.js
ProfileController = RouteController.extend({
  layoutTemplate: 'profileLayout',
  template: 'profileDetail',
  yieldTemplates: {
    'profileDetailLeft': {to: 'left'}
  },
  onRun: function(){
    ProfilesCollection.update({_id: this.params._id}, {$inc: {views: 1}}) #A
  },
...
});
```

#A On every run of this route the views attribute is increased by 1

Now every view of a profile is counted.

8.3.6 Creating an Iron-Router Plugin

If you want to create hooks that you either want to use for multiple applications or share with the community, creating Iron-Router plugins is the way to go. This plugins allow you to create a functionality that is very easy to share and use.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

For our community we want to be able to make a profile private so it is not accessible from an unwanted user. For this we want to create a plugin that could be reused more often. It should basically take a reactive condition and a template that should be rendered if the condition is false. This way you could use this plugin in a lot of situations.

Listing 8-16: Creating a reusable Iron-Router plugin

```
// router/plugins.js
Iron.Router.plugins.bouncer = function(router, options){ #A
  router.onBeforeAction(function(){ #B
    if(!this.lookupOption('condition').apply(this)){ #C
      this.render(this.lookupOption('bouncerTemplate')); #D
    } else {
      this.next(); #E
    }
  }, options);
}

// Usage of the bouncer
Router.plugin('bouncer', { #F
  condition: function(){ #G
    var profile = this.data().profile();
    if(!profile) return true; #H
    return profile.isPublic;
  },
  bouncerTemplate: 'profileIsPrivate', #I
  only: ['profile'] #J
});
```

#A Set the plugins name to 'bouncer'
#B We decide here that the bouncer should run as an onBeforeHook
#C You do not access the options directly but use the this.lookupOption function. This way you could also access options set in the global Router.configure() settings.
#D The bouncerTemplate is passed directly in the options but could as well be set in the global route configuration
#E If the condition is true only then we continue
#F The plugin is activated
#G In the condition we look if the attribute isPublic is true if not a special template should be rendered
#H If the profile is not yet available the subscription might not be ready so we can render without worries
#I Define the template that should be rendered in case the bouncer says no
#J Hooks as well as plugins have special options like only and except where you can specify where exactly the hook or plugin should run

The bouncer now checks if the profile is public or not and if this property changes the bouncer would reactively change his mind, too. In this concrete scenario it would be more secure to check if the profile is public e.g. in the publication since the profile's data is already on the client and accessible but just not rendered.

8.3.7 Server Side REST API

One very important aspect of every web application is the capability to react on requests directly to the server. This typically happens if other services want to interact with your application. Think about using Instagrams subscription API that would create a POST request

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

to a given URL each time a new image was created with a specified tag. Also payment service providers needs a way to communicate with your server directly. Since the Iron-Router runs on the client and on the server as well, we can create server site routes, too.

To create a server side route you only have to add the option `where: 'server'` and the route will not run on the client but on the server.

Listing 8-17: Simple Server Side Route

```
Router.route('/server/say', function(){
  var request = this.request; #A
  var response = this.response; #B

  response.end('You said: ' + request.query.message);
}, {where: 'server'}) #C
```

#A The node.js request object
#B The node.js response object
#C Definition that this route should only run on the server and not on the client

If you make a request with a query string and message as the key the server will respond with the value.

Figure 8-14: The server parses the query string and makes it accessible through the request object

You can see if the request is an HTTP GET, POST ... in the route function through the request object. But there is a better solution for that:

Listing 8-18: RESTful Routes

```
Router.use(Router.bodyParser.json()); #A


// EXTERNAL SERVICES
Router.route('/webhooks/instagram', {where: 'server'})
  .get(function(){ #B
    this.request.end('Hi Instagram!');
  })
  .post(function(){ #C
    ...
  })
  .put(function(){ #D
    ...
  })
```

#A Adds a connect middleware that parses json data
#B HTTP GET on the route /webhooks/instagram
#C HTTP POST on the route /webhooks/instagram
#D HTTP PUT on the route /webhooks/instagram

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

The Iron-Router is quite a special router that fits the special needs of the client and server side platform Meteor very well. It enables your application to correctly react on the route that is requested and highly improves your code structure.

8.4 Summary

- URL's are very important and enables the application to be accessible and shareable
- Iron-Router is the de facto standard for routing in Meteor
- You define templates, subscriptions and the data context based on routes
- Named routes help much for code maintainability
- Routes can be created for the client and for the server as well

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

12

Going into Production

This chapter covers

- Organizing your code to ease deployments
- Estimating and testing load
- Deployment options from simple to highly available
- Using Meteor UP to deploy an application
- Using environment variables to configure the server
- Basics of highly available architectures

When all features are implemented and all bugs ironed out it's time to go into production. This chapter covers all the essentials of deploying your application. We will not go into the specifics of server administration, but rather have a look at typical architectures and possible options in regards to small as well as scalable deployments, so you know what to expect when your app becomes a success.

In an ideal world, you will already have thought about the go-live of your application even before you started writing a single line of code. If you have not, now should be the time to revisit expectations and requirements. You will notice that your deployment options are greatly influenced by the requirements you have. However, because most of the largest applications started small as well, we will start looking at the most simple deployment scenarios and introduce scalability and the complexity it introduces as a second step.

12.1 Preparing for production

At the simplest level putting your project into production means copying it to a remote server and giving users access. From this moment on, you will notice bugs, plan additional features, deploy patches, receive user feedback – basically you will be working in two worlds: the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

production environment and the development environment. Before you copy code to any servers, let us walk through some essential techniques that make running an application in production.

12.1.1 Using version control

From the moment you issue the first `meteor create` command you are preparing for production. Unless you're at a hackathon and will never again look at the code you created any project should be placed under version control. Especially if it consists of more than just a few lines of code or is developed by more than a single person systems such as git, Subversion, or even IBM ClearCase, give you a safety net that also helps with development as well as deployment.

The benefits of source control for deployment may not be as apparent when you first copy your files to a server. But how do you know what revision is available to users and whether a bug raised from a paying customer still exists or has already been fixed before you went on a two-weeks vacation?

Once you are ready to go into production you should have a dedicated master branch that represents production quality code and tag it every time you deploy it into production. That way it becomes very easy to see what code base is actually used for your running app.

Using git we shall walk through the steps required. Many people do all work on the *master* branch, which makes merging changes much more complicated than it could be. Instead, use a branch for all development or error-fixing and only let working code make it into the *master* branch.

> **TIP** When using git make sure the *master* branch always contains the most recent and stable code only. No development work should be done on *master* but on dedicated development or feature branches.

In a git repository a simple project may look similar to Figure 12-1:

Graph	Description	Commit	Author	Date
	origin/devel, devel preparing for deletion of tasks	0e98155	Stephan Hochhaus <stepha...	04.11.2014 11:56
	v2.0, origin/master, origin/HEAD, master Merge branch 'devel'	8b7d660	Stephan Hochhaus <stephan...	04.11.2014 11:54
	feature complete	e42787a	Stephan Hochhaus <stephan...	04.11.2014 11:52
	task fixtures for database	c1cc7cc	Stephan Hochhaus <stephan...	04.11.2014 11:47
	new feature: tasks from database	fb2c72d	Stephan Hochhaus <stephan...	04.11.2014 11:44
	v1.1 Merge branch 'hotfix-v1.0'	f04f077	Stephan Hochhaus <stephan...	04.11.2014 11:49
	hotfix-v1.0 hotfix for task descriptions	e147ae3	Stephan Hochhaus <stephan...	04.11.2014 11:49
	v1.0 Merge branch 'devel'	07b8d7d	Stephan Hochhaus <stephan...	04.11.2014 11:41
	prettier styling	4aeb273	Stephan Hochhaus <stephan...	04.11.2014 11:39
	added feature: todo list	5f8becd	Stephan Hochhaus <stephan...	04.11.2014 11:38
	create new app	53bc27d	Stephan Hochhaus <stephan...	04.11.2014 11:36
	updated readme	78e4d6e	Stephan Hochhaus <stephan...	04.11.2014 11:34
	Initial commit	0e9ff6c	Stephan Hochhaus <stephan...	04.11.2014 11:33

Figure 12-1: Using multiple branches for development and hotfixes in git

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

All development takes place in a dedicated branch called *development*. Once development is finished and all tests are passed, the code is merged over to *master* and is tagged with a version number.

The advantage of using multiple branches is that you can already add features to your application that you do not yet want to deploy, but still be able to simply fix critical issues in your production code without risking mixing untested development code with a live environment.

In our example we successfully deployed stable code into production and tagged it v1.0. While working on the next version 2.0 a critical issue is reported and requires a hotfix. In order to work on the hotfix, a new branch called *hotfix-v1.0* is created. Once the code is corrected and tested, it gets merged back into the *master* branch. All this is unaffected by anything that might happen in the `development` context.

The more people work on a project or the more complex your application becomes, the more branches might be used. Just make sure that you can easily keep track of the code used in production at all times.

Some providers such as Heroku make it the default to deploy your application from source control. Just like in the example above they also assume that the *master* branch contains stable code.

12.1.2 Testing functionality

All code that you merge into the master branch should be sufficiently tested. Tagging your code makes it easier to keep track of testing candidates as well, especially if the person testing your application is different from the actual developer.

Velocity is the official testing framework for Meteor. Written by a team of developers in the Meteor community, Velocity lets you define tests using any mix of established testing libraries, including Mocha, Jasmine, Cucumber, and Selenium. Velocity makes it easy to use automated testing[14]. You can find out more about Velocity at their github repository https://github.com/meteor-velocity/velocity.

While it is certainly good to have unit tests in place the most important tests should ensure that users are able to perform all actions in your application without errors. Remember when testing your application to not forget to include negative tests. Test for what happens if users do unexpected things such as entering letters into a digit-only field. Any issues discovered should then be managed in an issue tracking system. Again, it does not matter what system you use, a blackboard may work just perfectly for your team, while other teams require distributed systems such as Jira.

[14] If you plan on doing regular updates to your production environment it pays off to set up a Continuous Integration (CI) or Continuous Delivery (CD) environment to perform all tests at the push of a button and optionally deploy to your target systems. Have a look at Travis or Jenkins if you want to start using the benefits of a CI environment.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

At the end of the day, testing your code and tracking issues is all about transparency. You need to know whether the application is actually ready for launch. If there are still open issues you need to decide whether they will prevent going into production or if they are acceptable. Only code that is sufficiently tested and free of any bugs that prevent it from going into production should be tagged for deployment.

12.1.3 Estimating and and testing load

If you have not thought about the expected number of users for your application, now is the time to come back to this matter. Certainly you have a good understanding of the users of your application in terms of functionality, but this time you will have to focus on scale by asking three questions:

1. How many (concurrent) users do you have?
2. How much data (and load) does a single user generate?
3. Where are your users?

Worst, Best, and Realistic Scenarios

Even though it may be hard to guess the exact number of users, chances are you can at least give somewhat reasonable numbers for at least the first months.

A common approach to estimating the visitors of a site is to assume three scenarios: best case, worst case, and realistic case. In a best case you come up with the maximum number of users you can expect if your marketing campaign is extremely successful. You should not consider Slashdotting[15] unless you explicitly aim for it. The worst case marks the low end of the spectrum, where the least number of people accesses your site, while the realistic scenario will be somewhere in the middle. If you already have a similar web presence you can use your existing server logs as a starting point.

To illustrate how to estimate the number of users let us consider an online game where users play a game of Scrabble against each other. Our worst case scenario expects 100 users per day (3000/month), realistically 1000 a day (30000/month) will be playing, and at best there might be 10000 per day (300000/month).

What do these numbers tell us? At first we will use them to calculate possible storage requirements. If every user can upload an avatar image and we expect them to have detailed statistics about each game played as well as the words they used we shall use 1 MB per user as our baseline. That tells us that we need between 3 and 300 GB of storage for our application.

[15] The Slashdot effect, also known as slashdotting, occurs when a popular website links to a smaller site, causing a massive increase in traffic. See: http://en.wikipedia.org/wiki/Slashdot_effect

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Table 12-1: User estimates for a deployment

	Worst Case	Realistic Case	Best Case
Users/Month	3000	30000	300000
Users/Day	100	1000	10000
Storage/User	1 MB	1 MB	1 MB
Overall storage requirements	3 GB	30 GB	300 GB

Unless we figure out the number of concurrent users we cannot come up with any sensible server requirements when it comes to load or memory. Making assumptions and using simple math it is possible to come up with a number of concurrent users, which is an essential figure when designing your server setup.

ESTIMATING CONCURRENT USERS

In order to determine the required resources we need to know a bit more about our user's behavior. The numbers alone do not tell us whether all players are evenly distributed or whether they are only active each Saturday night. This information is important because we must also design our deployment to handle any spike loads, if we can know about them in advance.

Our Scrabble application will be a casual game, so we expect users to mostly play during their non-working hours. That means more traffic between 12am and 1pm (for a quick game during lunch break) and of course in the evening, from 6pm to 11pm. That leaves us with a 6-hour window in which most games will be played. Since we are still estimating only we can safely ignore all times outside of this window and pretend all users stay inside of it.

For the number of concurrent users we will need to make one more assumption: the average time a user spends playing.

In the worst case scenario we have now 100 users that are somehow distributed across 6 hours. From the beta tests we know that players typically spend ten minutes playing the game, on average four players take part per game. Six hours translate into 360 minutes. This number will be used to calculate the maximum concurrency by taking the number of users within our window and dividing it by the duration of the number of games that fit into the time window:

```
Concurrent Users = Expected number of users / ( Time Window / Average Game Length )
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Table 12-2: Calculating concurrent users

	Worst Case	Realistic Case	Best Case
Users	100	1000	10000
Time Window (mins)	360	360	360
Average players per game	4	4	4
Average Game Length (mins)	10	10	10
Concurrent Games (rounded)	1	7	70
Concurrent Users (rounded)	3	28	278

Now we know that our servers should be capable of handling 3, 28, or 278 concurrent users, depending on the scenario.

The next step would be to conduct some sort of load testing to figure out the amount of server resources needed to serve that many concurrent users.

Load Testing

Load testing is quite a complex task, especially if some of the processing is outsourced to a client, as it is the case with Meteor. Traditional tools that generate http load such as Apache Bench (ab) or Siege cannot be used to test JavaScript applications reliably.

The solution is to either use some sort of load simulation by generating DDP messages that would be sent from the clients. An alternative approach would simulate the clients directly, e.g. using PhantomJS. Meteor load test[16] is a load testing tool for Meteor that sends DDP messages to stress an application. An example for using PhantomJS in combination with CasperJS is the Meteor Parties Stresstest[17]. The most important part in load testing is to monitor the results and know how to interpret them.

As a result of a load test you can get to know important information about your application. It will become clear where possible bottlenecks sit and whether your servers should have a bigger CPU or more RAM. Because this topic can become overwhelming very quickly a simple way to perform a load test is to find out how the application scales.

Try to find out whether your application scale in a linear fashion. Let five users run concurrently against you app and see how much CPU and memory the server consumes. Now increase by five users and check again. Perform this at least two more times, the more values you have the more accurate your predictions will be. Increase the number of users until you max out either RAM or CPU on the machine under load.

[16] https://github.com/alanning/meteor-load-test
[17] https://github.com/yauh/meteor-parties-stresstest

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Eventually you should have values for 5, 10, 15, 20, and perhaps more concurrent users. Now you can draw a simple diagram that shows how the server behaves under load. This will tell you what to look for in a server and give you an idea of the cost to expect.

LOCATION

Last but not least you should take into consideration where your users are located. If you deploy an application where the user base comes from a small geographic region, then you should take care that your hosting is not thousands of miles away. Reducing network times is still an important factor to improve user experience, so make sure your servers stay close to your users.

If your users come from all over the world, you should take a look at hosting that supports multiple servers in different geographical locations. Alternatively, a Content Delivery Network can help to reduce network traffic.

> **TIP** Network latency is a huge factor that impacts user experience. Make sure to not only use sufficiently dimensioned servers but also place them as close to your target users as possible.

12.1.4 Server Administration

Most software developers are not interested in running and managing servers, and they should not. Thankfully you do not have to be a server expert as there are many Platform-as-a-Service (PaaS) offerings where you rent a Meteor or Mongo instance rather than managing your own server where you need to set things up yourself. You do need to know enough about the overall architecture, though, so you will be able to estimate the effort and cost involved in running your application.

If you do not choose a full-service PaaS provider looking at your deployment options you must answer the question of who shall manage the application servers, monitor the load and possible outages, apply security updates or renew SSL certificates. Cloud providers often take care of these things, but you will have to pay for the comfort.

BACKUP

What happens when your servers crash and all data is lost? Your application source code is safe in your version control system, but all user contributed data is gone unless you have some sort of backup strategy in place. Typically there are two types of data that need to be backed up with a Meteor project:

- MongoDB/your database
- File uploads

Backing up MongoDB is fairly straight forward. You could create a replica set and run another MongoDB instance off-site to achieve pretty much instant backups. Alternatively your perform a traditional backup. For a typical backup of MongoDB you can use `mongodump` and `mongorestore` to backup and restore the database.

Backing up files is not different from any other web project. Just make sure you include all relevant configuration files as well, if these are not under version control (where they should

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

be!). If your hosting does not come with a backup solution it is best to create your own rolling backup, e.g. using rsnapshot[18]. Be aware that even though cloud providers claim to be secure, they too may lose data and it is always safer to have your own backup, especially if you cannot afford to lose important customer data. Running off a RAID-1 drive is also not a substitute for regular (daily or at least weekly) backups.

12.1.5 Checklist

You can use this simple checklist to make sure you are ready to go into production. Of course, you should be able to answer all questions with yes.

1. Are you using a version control system for your code?
2. Are there no issues with your software that prevent a go-live?
3. Do you know how many (concurrent) users to expect?
4. Did you conduct load testing to determine how much CPU/memory a single user requires?
5. Did you calculate the number of required disk space?
6. Do you know where your users are located?
7. Do users only exchange data via the database or do you need to share data between server instances as well?
8. Do you have dedicated and trained staff to manage the servers?
9. Do you have a backup strategy planned?
10. Did you test your backup strategy by performing a successful restore?

12.2 Set up and deployment

Now it is time to put your application on a live server. Depending on the criticality of your application and the number of users you expect you can choose between three main deployment options. To help you decide for an approach let's look at the pros and cons of each.

Table 12-3: Pros and cons of Meteor server options

Deployment to	Advantages	Disadvantages
Meteor.com	- Easiest deployment - No hosting cost	- No guaranteed availability - No custom URL

[18] http://www.rsnapshot.org/

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Cloud Provider (e.g. Modulus, Heroku, Nodejutsu)	- Low administration overhead - Quick and easy scalability - Pay per actual usage	- No direct control over configuration options
Manual Setup (e.g. Amazon EC2, Rackspace, or own hardware)	- Full flexibility - Re-use existing resources across different applications	- Requires administration knowledge - Scaling requires more time

For quick prototypes the Meteor.com infrastructure is the easiest way to go, although it is not a good choice for actual production usage as you get no guaranteed availability and scaling is rather complicated. Your application may be offline or unresponsive at anytime. On the other hand, since it is a free option most developers will use it at some point.

If you do not want to spend significant overhead on system administration one of the existing cloud providers might be a good fit. Besides not having to work on system configurations they often offer push-button-scalability that allow you to add more instances almost instantly. Of course you will not be able to control each and all settings of the underlying infrastructure as it is often shared between multiple applications. Some unconventional requirements may come at a hefty price, if they are possible at all.

When you are familiar with running servers and have dedicated people who can fix things on short notice a manual setup is the most flexible. Especially for hosting multiple projects it can become the best option, even if there are no exotic configuration settings required. You will have to put some more thoughts into what resources you actually require, though, as you pay for the entire setup, not just your actual usage. Adding more instances also takes significantly more time than clicking a single button.

12.2.1 The simplest deployment - Meteor.com

In order to deploy to meteor.com all you need is the meteor command line tool. When your application is ready the following command will send it to the Meteor.com infrastructure:

```
$ meteor deploy <subdomain>
```

Replace <subdomain> by the name of the subdomain that is going to be used to access your application. Meteor.com uses developer accounts to make sure that once you have deployed to a free subdomain nobody else will be able to overwrite your application. The subdomain will be connected to your Meteor.com developer account. If you do not have one yet – which is generally the case for a first deployment – then you will be asked to automatically set one up during the deployment process. During the deployment process all you need to do is provide your email address. You will then be able to set a username and password on the meteor.com website (see Figure 12-2).

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Figure 12-2: Setting up your Meteor developer account

If you have multiple developers that should be able to deploy to a meteor.com subdomain, you can add them to an organization. Every member of the organization will have the same permissions on your project.

When trying to deploy to an existing subdomain `meteor deploy` will prompt you to enter a username and password to authenticate. Once successfully authenticated Meteor will remember your account on the local system. You can see a list of all subdomains you are authorized for by entering

```
$ meteor list-sites
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
myApp — bash — 110×25
MacBookPro:myApp stephan$ meteor deploy mysampleappon
To instantly deploy your app on a free testing server, just enter your
email address!

Email: mail@meteorinaction.com
Deploying to http://mysampleappon.meteor.com.
Now serving at http://mysampleappon.meteor.com
                                                  -
You can set a password on your account or change your email address at:
https://www.meteor.com/setPassword?5CwrvpAfMg

MacBookPro:myApp stephan$ meteor logs mysampleappon
[Tue Nov 04 2014 13:36:11 GMT+0000 (UTC)] INFO STATUS null -> starting
[Tue Nov 04 2014 13:36:11 GMT+0000 (UTC)] NOTICE Starting application on port 13188
[Tue Nov 04 2014 13:36:11 GMT+0000 (UTC)] INFO STATUS starting -> running
[Tue Nov 04 2014 13:40:31 GMT+0000 (UTC)] INFO HIT / 91.52.215.18
[Tue Nov 04 2014 13:40:32 GMT+0000 (UTC)] INFO HIT /20ae2c8d51b2507244e598844414ecdec2615ce3.css 91.52.215.18
[Tue Nov 04 2014 13:40:32 GMT+0000 (UTC)] INFO HIT /819f96ffbb57aa2351efdbf57c2ea8b6cb0e89bd.js 91.52.215.18
[Tue Nov 04 2014 13:40:33 GMT+0000 (UTC)] INFO HIT /favicon.ico 91.52.215.18
[Tue Nov 04 2014 13:40:47 GMT+0000 (UTC)] INFO HIT / 91.52.215.18
[Tue Nov 04 2014 13:40:47 GMT+0000 (UTC)] INFO HIT /20ae2c8d51b2507244e598844414ecdec2615ce3.css 91.52.215.18
[Tue Nov 04 2014 13:40:47 GMT+0000 (UTC)] INFO HIT /819f96ffbb57aa2351efdbf57c2ea8b6cb0e89bd.js 91.52.215.18
[Tue Nov 04 2014 13:40:48 GMT+0000 (UTC)] INFO HIT /favicon.ico 91.52.215.18
[Tue Nov 04 2014 13:41:06 GMT+0000 (UTC)] INFO HIT /_GALAXY_ 91.52.215.18
MacBookPro:myApp stephan$
```

Figure 12-3: Deploying and accessing the logs from the command line

In order to manage an application on the meteor.com infrastructure you can use the command line tool to access the server log files and even the database shell. This is done with the commands

```
$ meteor logs <subdomain>
```

and

```
$ meteor mongo <subdomain>
```

NOTE Although it is very convenient to use the meteor.com infrastructure it is not yet ready for production-grade deployments.

12.2.2 All-inclusive hosting – Cloud Providers

Although dedicated Meteor hosting is not widely available yet, there are already many platforms to run Node.js applications. Because Meteor applications can be quickly converted to run on a plain Node.js server you can use any provider that is capable of hosting Node.js.

As you are using the infrastructure of a cloud provider there is no need to set up any components and processes on your own. Oftentimes providers will also offer MongoDB in their portfolio, which allows you to host all components with a single company.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Table 12-4: Node.js-as-a-Service providers

Provider	URL
Modulus	https://modulus.io/
Heroku	https://www.heroku.com/
Nodejitsu	https://www.nodejitsu.com/

In order to get Meteor up and running with a Node.js provider you need to prepare your application for the deployment process. The actual preparation depends on your provider of choice, some can directly host a Meteor app, most of them require you to convert your project to a regular Node.js application first.

BUILDING METEOR PROJECTS

Meteor allows you to bundle an application with the `meteor build` command. The output is a fully-contained Node.js application in the form of a tarball. It is very useful if the platform used to create the bundle and the deployment target run the exact same operating system and Node.js version.

Creating the bundle is as simple as navigating to the application root directory and calling the command

```
$ cd myMeteorProject
$ meteor build deploymentDir
```

You will notice that the entire directory structure has changed significantly. Instead of a `client`, `server`, and `public` folder you will now see two main folders: `programs` and `server`, where all application code, assets, and modules are contained.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
deploymentDir
Bochum:deploymentDir stephan$ tar -zxf simple-todos.tar.gz
Bochum:deploymentDir stephan$ tree -d -L 5
.
└── bundle
    ├── programs
    │   ├── server
    │   │   ├── app
    │   │   ├── assets
    │   │   │   └── packages
    │   │   ├── npm
    │   │   │   ├── ddp
    │   │   │   ├── logging
    │   │   │   ├── minifiers
    │   │   │   ├── mongo
    │   │   │   └── webapp
    │   │   └── packages
    │   └── web.browser
    └── server

15 directories
Bochum:deploymentDir stephan$
```

Figure 12-4: meteor build on the server

Although `meteor build` is very simple to use it does have some limitations in regards to portability. For creating truly portable Node.js applications from a Meteor project `demeteorizer` is the best tool.

HIGHLY PORTABLE NODE.JS BUNDLES WITH DEMETEORIZER

The `demeteorizer` project was started by the engineers behind the cloud provider Modulus. Demeteorizer wraps and extends the `build` command by creating something that more closely resembles a standard looking Node.js application. The main difference between the two is that projects processed with `demeteorizer` contain a `package.json` file for dependency management and do not include node modules in a dedicated folder.

In order to create highly portable Node.js bundles with `demeteorizer`, you first need to install it on your development system. Because it comes in form of a node module it can be installed via `npm`:

```
$ npm install -g demeteorizer
```

Once demetorizer is available on your system it works similar to the `build` command. Navigate to your project root folder and call it from the command line:

```
$ cd myMeteorProject
$ demeteorizer -t myApp.tar.gz
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

In contrast to the `build` command `demeteorizer` creates a directory by default, which is why the `-t` switch is needed to create a tarball. By default it will create a new directory called `.demeteorized` where it creates a structure as in Figure 12-5.

```
.demeteorized
MacBookPro:.demeteorized stephan$ ls -l
total 32
-r--r--r--  1 stephan  staff   543  4 Nov 15:09 README
-r--r--r--  1 stephan  staff   485  4 Nov 15:09 main.js
-rw-r--r--  1 stephan  staff  2102  4 Nov 15:09 package.json
drwxr-xr-x  4 stephan  staff   136  4 Nov 15:09 programs
drwxr-xr-x  3 stephan  staff   102  4 Nov 15:09 server
-r--r--r--  1 stephan  staff  1137  4 Nov 15:09 star.json
MacBookPro:.demeteorized stephan$ tree -d -L 4
.
├── programs
│   ├── server
│   │   ├── app
│   │   ├── assets
│   │   │   └── packages
│   │   ├── npm
│   │   │   ├── ddp
│   │   │   ├── logging
│   │   │   ├── minifiers
│   │   │   ├── mongo
│   │   │   └── webapp
│   │   └── packages
│   └── web.browser
└── server

14 directories
MacBookPro:.demeteorized stephan$
```

Figure 12-5: Resulting folder structure of demeteorizer

The resulting archive can be uploaded and extracted to the server. Some providers such as Modulus.io will allow you to directly upload the archive via a web interface. Due to the `package.json` file in the root of the project all node modules will be installed either automatically – if your provider supports it – or via navigating to the root folder of the project on the deployment server and issuing the install command:

```
$ cd /var/www/myDemeteorizedApp
$ npm install
```

Now that we have the application running, we set up the database. Because the exact procedure varies between providers we will not cover it in detail here. You can use any MongoDB instance you like with your application, it does not even have to be hosted with the same provider.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

MongoDB as a Service

Sometimes it is necessary to split the hosting of Meteor and MongoDB. One of the main reasons is that a Node.js provider might not offer a feature for MongoDB that you would like to use (e.g. Oplog tailing) or they are too expensive compared to the competition. In that case there are many MongoDB-as-a-service providers that might just be for you. Some even start with a completely free plan which makes them an ideal starting point for small projects as you can easily scale at a later stage with usually a single click.

Table 12-5: MongoDB-as-a-Service providers

Provider	URL
MongoLab	https://mongolab.com/
Compose.io (formerly known as MongoHQ)	https://www.compose.io/
MongoSoup	https://www.mongosoup.de/
ObjectRocket	http://www.objectrocket.com/
Elastx	http://elastx.com/

Advantages

As you do not need to know a lot about MongoDB and its internals using a provider will get you started very quickly. There is minimum overhead for administration involved. You get all the features of high-availability and load balancing out of the box, so you can focus on your application alone. Also switching to the next bigger database size is easy.

Oftentimes less commonly used features are cheaper with a specialized database provider and for low traffic sites there are even free tiers with some of them.

Downsides

Network latency between your application servers and the MongoDB instances will be much higher compared to hosting your own DB on the next machine. However, since many providers use Amazon or Rackspace infrastructure there may be no noticeable effect at all if you are using the same infrastructure for your own servers as well.

The comfort of not having to administer the infrastructure will cost you, especially if you go beyond the sizes available on the price list. For optimum performance you will want to include Oplog Tailing for your Meteor application, but some providers charge a hefty extra for it because it requires a dedicated replica set and they cannot use a shared shard for your data.

12.2.3 Full flexibility – Manual Setup

Setting up Meteor manually on a server is very straight forward. If you are running Ubuntu, Debian, or OpenSolaris on your server it gets even easier with Meteor Up – or short mup. If you cannot use mup or need more flexibility all it takes to start running your project on your

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

own servers is a regular Node.js server in combination with the bundling functionality we discussed in the previous section.

METEOR UP

Meteor Up is a community project that aims to both setup servers and deploy Meteor. First you initialize a new project, configure the environment, initiate the server installation, and finally deploy your project. Additional information can be found at github: https://github.com/arunoda/meteor-up.

Meteor Up is installed via the Node Package Manager npm:

```
$ npm install -g mup
```

Use a terminal to navigate to your Meteor application folder and initialize a new project with

```
$ mup init
```

Now you have two json files in the current directory:

- `mup.json`

  This file is used to define the servers that will be used as deployment targets and what components to install.

- `settings.json`

  Used to define deployment-specific configuration options available in `Meteor.settings`. This can be used for e.g. API keys or server credentials.

While it is not required to use `settings.json`, you must adjust the contents of `mup.json` to reflect your own server setup. We will use two hosts in this example, but we do not want to set up MongoDB on either of them.

Listing 12-1: mup.json configuration

```
{
  // Server authentication info
  "servers": [#A
    {
      "host": "host1.meteorinaction.com",
      "username": "stephan",
      //"password": "password" #B
      // prefer pem file (ssh based authentication)
      "pem": "~/.ssh/id_rsa" #B
    },
    {
      "host": "host2.meteorinaction.com",
      "username": "stephan",
      "password": "password"
    }
  ],

  // Install MongoDB in the server, does not destroy local MongoDB on future setup
  "setupMongo": false, #C
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
  // WARNING: Node.js is required! Only skip if you already have Node.js installed on
      server.
  "setupNode": true,

  // WARNING: If nodeVersion omitted will setup 0.10.28 by default. Do not use v,
      only version number.
  "nodeVersion": "0.10.28",

  // Install PhantomJS in the server
  "setupPhantom": true, #D

  // Application name (No spaces)
  "appName": "meteorinaction", #E

  // Location of app (local directory)
  "app": "/Users/stephan/Code/meteorinaction", #F

  // Configure environment
  "env": {#G
    "ROOT_URL": "http://www.meteorinaction.com"
    "PORT": "3000" #H
    "MONGO_URL": "mongodb://user:password@192.168.2.210/meteor" #I
    "MONGO_OPLOG_URL":
      "mongodb://oploguser:password@192.168.2.210/local?authSource=admin" #I
  },

  // Meteor Up checks if the app comes online just after the deployment
  // before mup checks that, it will wait for no. of seconds configured below
  "deployCheckWaitTime": 15#J
}
```

#A You can define one or multiple servers that will be used for deployment
#B ssh keys or passwords may be used for authentication
#C if set to true mup will also set up a MongoDB but without replication
#D PhantomJS is not required but is used in conjunction with some packages
#E If you deploy multiple Meteor instances to the same machine you can differentiate between them using different names
#F Path to your application source code on the local machine, not the server
#G Environment variables to be used for this application
#H Optionally specify the port Meteor should use
#I When using an external MongoDB define the connection string as an environment variable, if used also the oplog must be declared
#J Number of seconds to wait whether a deployment was successful

With the settings you see in Listing 12-1 mup will deploy to the servers *host1.meteorinaction.com* and *host2.meteorinaction.com* using a password for the user "stephan".

> **NOTE** While it is tempting to use passwords for test deployments you should consider switching to ssh keys instead of passwords for actual production deployment.

This example will not set up MongoDB, but Node.js in version 0.10.28. It will also install PhantomJS, which is used by some packages like `spiderable` to improve visbility for search engines. The application name is used to identify the node process on the server. You can use

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

mup to deploy multiple node processes to the same machine from different `mup.json` configurations and the application name is how you can differentiate between them. The location of your app relates to the project you want to deploy from your local workstation or laptop. Using the environment variables you can fine-tune the Meteor runtime environment.

In case you have a multi-core server you may want to deploy multiple Meteor instances on the same machine so that all cores are utilized. In that case you need to use a different port and application name for each instance which in turn requires multiple `mup.json` files. In order to avoid any conflicts the way to work in this case is to use a dedicated directory for each core. For a 2-core system the structure might look like this:

```
.
├── client
├── core1
│   └── mup.json
├── core2
│   └── mup.json
├── public
├── server
└── settings.json
```

The content of both `mup.json` files would be exactly the same, only the application name and the setting for the `PORT` environment variable would be different.

Without a local MongoDB instance we need to specify the URL for the database as an environment variable. If you are using the oplog you can define it here as well.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
1. mup (bash)
bochum:meteorinaction stephan$ mup init

Meteor Up: Production Quality Meteor Deployments
------------------------------------------------

Empty Project Initialized!
bochum:meteorinaction stephan$ mup setup

Meteor Up: Production Quality Meteor Deployments
------------------------------------------------


Started TaskList: Setup (linux)
[192.168.2.221] - Installing Node.js
[192.168.2.221] ✔ Installing Node.js: SUCCESS
[192.168.2.221] - Installing PhantomJS
[192.168.2.221] ✔ Installing PhantomJS: SUCCESS
[192.168.2.221] - Setting up Environment
[192.168.2.221] ✔ Setting up Environment: SUCCESS
[192.168.2.221] - Copying MongoDB configuration
[192.168.2.221] ✔ Copying MongoDB configuration: SUCCESS
[192.168.2.221] - Installing MongoDB
[192.168.2.221] ✔ Installing MongoDB: SUCCESS
[192.168.2.221] - Configuring upstart
[192.168.2.221] ✔ Configuring upstart: SUCCESS
Completed TaskList: Setup (linux)
bochum:meteorinaction stephan$
```

Figure 12-6: Setting up a server with Meteor Up

Once the configuration is finished you can set up the environment using

```
$ mup setup
```

This will take care of all server configurations and installation for you. Meteor Up also takes care that the all server processes are started upon boot. Additionally it uses `forever` to restart node in case it crashes. It does not yet copy your application over to the server. Bundling and deploying the application is the final step and is done with

```
$ mup deploy
```

12.3 Connecting the pieces

There are several aspects that can be configured when running an application server. You may need to change the port a server listens on, define the root URL, or include connection details to the database server or mail transports. All settings that need to take effect upon server start already are passed to Meteor in form of environment variables.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

12.3.1 Environment variables

Depending on which packages you use with your application, there are several environment variables you can set that will influence the was your Meteor application will behave. The most common environment variables are listed in.

Table 12-6: Common environment variables

Variable Name	Description
PORT	Network port to bind to (default: 3000)
ROOT_URL	Base URL for the application
MONGO_URL	Connection string for MongoDB
MONGO_OPLOG_URL	Connection string for MongoDB oplog
MAIL_URL	SMPT connection string for the mail server (default: STDOUT)

Most cloud providers have a web interface to supply names and values of environment variable (see Figure 12-7).

Figure 12-7: Defining environment variables with Modulus

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

12.3.2 Connecting Meteor to MongoDB

Once we have a running Node.js server and a MongoDB instance we are going to configure Meteor so it knows how to access the database. We can specify both the regular database and oplog access individually by using the variables `MONGO_URL` and `MONGO_OPLOG_URL`. Both use the same syntax of a standard mongodb connection string:

```
mongodb://<username>:<password>@<host>/<database>?<options>
```

`mongodb://` is the required prefix to identify that this is a string in the standard connection format

`username:password@` are optional. If present, these will be used to login to a specific database

`host` is the only required part. If not port is specified, the default will be used (`27017`). You can define multiple hosts, separated by a comma

`/database` is required in combination wit `username:password` and specifies the database to log into after successfully connecting to the server. If not specified the "`admin`" database will be used by default.

`options` are connection options in form of `name=value` pairs separated by "&"

Let's assume the following environment:

Table 12-7: Example MongoDB Connection Information

Key	Value
MongoDB server address	mongo.local.lan
Database name	meteordb
Database user	meteoruser
Database password	drowssap
Oplog user	oploguser
Oplog password	drowssap

Based on these values the connection strings look like this:

```
$ export MONGO_URL=mongodb://meteoruser:drowssap@mongo.local.lan/meteordb
$ export
      MONGO_OPLOG_URL=mongodb://oploguser:drowssap@mongo.local.lan/local?authSour
      ce=admin
```

Notice that in the `MONGO_URL` definition we specify the actual database Meteor is going to use, which is `meteordb`. In contrast the name of the application database is irrelevant for the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

`MONGO_OPLOG_URL`, it always uses `local`. This is because the oplog is always kept in the `local` database. Since users cannot authenticate against the `local` database, we need to pass another option `authSource` to the connection string, to use the `admin` database as the authentication source instead.

12.4 Scaling Strategies

System architecture is as complex as writing software. Without going into too much detail this section aims to introduce you to the main concepts of scalability. Simplify put there are two different aspects:

- Reliability – no single component failure will break the system
- Availability – every request can be processed

At first glance it may look as though both are the same, but they serve quite different purposes. Both can be translated to high availability, but reliability focuses on redundant components and availability is often achieved by means of load balancing.

ACTIVE-PASSIVE HIGH AVAILABILITY WITH REDUNDANCY

When an application is running in production it is expected to be available 24 hours a day and 7 days a week. Depending on the nature of your application you will start losing users or even money when your application is not available. That is why most production deployments will be architected to be highly available. It boils down to your setup not having any single points of failure. Each single component may go down at any point in time without any significant impact for the user. Operations will be able to continue even without an admin having to fix things immediately. This is not only helpful in a case of failure, but also regular updates can be applied without taking the entire system offline.

You often find active-passive combinations in HA environments: one server does the actual work and another stands idly by, ready to take over should the first server ever go down. If you find yourself maxing out server resources, meaning a single server is not able to handle all requests anymore, you will need to scale vertically (also known as scaling up), which means you add more server resources such as CPU or memory. If you were running on Amazon EC2 you would upgrade your server from a medium to a large instance. However, you cannot scale up infinitely, if you are very successful you will reach a point where no single server will be able to handle the load by itself.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Figure 12-8: From single instances to high availability

ACTIVE-ACTIVE HIGH AVAILABILITY WITH LOAD BALANCING

If only a single instance can ever be active at the same time, it does not leave you with any real options to handle more load. Exchanging a small server for a bigger one usually takes time which is why you should consider scaling horizontally (scale out) as well.

In load balanced environments you find servers running as active-active pairs, meaning both servers actively get and process requests. Unfortunately managing dependencies gets more complicated if both active servers may influence the data or processing that is currently going on with the other server. In the Scrabble application introduced earlier in this chapter we need to consider how two players on two different servers could be able to play the same game.

12.4.1 Single component deployments

When running Meteor locally there is a single Meteor application and a single MongoDB instance. This brings the advantage of having a fairly simple setup that does not share any session data or users between multiple machines. It also makes an ideal setup for development environments, as you know exactly where to look in log files and locate potential bugs. There is no need to first analyze on which of your servers errors might have happened or whether any balancing components may have caused their own bugs.

The downside of having single instances is that this architecture is neither failsafe nor does it scale. If your MongoDB goes down, the entire application will stop working properly. The

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

same goes for the Meteor server. Also as the number of (concurrent) users increases you only have one real scaling solution: Buy a bigger server.

12.4.2 Redundancy and load balancing

The first step to better scalability and high availability is to make sure the database is always available. If you use a PaaS provider for your database they will take care that you can always access your data for you. Should you need to set up a highly available database yourself or if you want to know more about what goes on inside a MongoDB cluster you can have a look at the appendix chapter.

The second step in achieving high availability is making sure the application itself is always accessible. This requires scaling out Meteor servers. Because the Meteor application server is a Node.js server, all of the principles for hosting Node.js applications apply to Meteor as well. Depending on the expected number of users two or more server instances should be set up. Because we cannot ask users to either access server 1 or server 2 we are going to add an automatic dispatcher or load balancer that distributes all requests to a single URL across all application servers.

BALANCING LOAD

It is a common best practice to run web applications behind load balancers. A large range of established tools is available, reaching from open-source software such as haproxy or nginx to dedicated hardware boxes from F5 or Cisco. The principle is the same as with query routers in MongoDB[19], any number of actual worker processes (or applications) run as a logical group and all requests from clients are send to either of these workers.

Meteor is different from the traditional way of serving web pages. In the traditional approach connections are stateless, it makes no difference which application server responds to any request in a series. With Meteor however, each connection maintains state and therefore switching between servers for different requests of the same client will break things as any information regarding the context is lost. No matter which load balancing approach we choose to implement, we must assure that a client does not move across servers unless we have a way to exchange state information between all nodes.

The simplest form of keeping a user on the same server is to remember a user in the load balancer, e.g. by associating an IP address with an application server. All requests from IP 192.168.2.201 will go to server #A only, no matter how much load it currently has. Alternatively the load balancer can set a cookie to remember which server a client should stick to.

COMMON LOAD BALANCING ALGORITHMS

Knowing the most important algorithms for distributing requests will help you choose the best for your application. Many people are familiar with *round-robin*, which means that all requests

[19] See the appendix chapter for more information on query routers in Mongo DB

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

are evenly distributed between servers by count: Request #1 goes to server #A, the second request to server #B, the third to server #A again, 4 to #B, and so on. That works well if all your servers have the same specs, but if one of them is more powerful, a *weighted round-robin* might be the better choice. You might configure server #A to retrieve twice as many requests because it has double the memory and CPU.

Unfortunately these algorithms do not take into account any load a server might currently have, for example because users have ended their session already. Typically a load balancer will not know about the status of the nodes it redirects requests to. Some can, however, determine how long a server takes to respond and avoid sending any more users to an already unresponsive server. A load balancer may also be configured to distribute users based on the actual number of connections. That brings the *least connection* algorithm into play. Here the load balancer actively checks how many users are currently connected to a server and distributes users evenly.

All algorithms have their specific uses. If in doubt, the best way is to start out with round-robin and enable session stickiness. Consult the appendix chapter on nginx for configuration examples.

Single Threads

In order to avoid dealing with the complexity of multithreaded architectures Node.js was designed to run in a single thread. That means it will only ever run on a single core and a single thread of the CPU, regardless of how many cores exist. That is a major blocker for scaling up – you cannot simply use a bigger machine with more CPUs if your application is becoming slow, you need to scale horizontally. Although there is a Node.js cluster package it is not yet production ready[20]. If you happen to have a multi-core server you can take advantage of all cores by running multiple Meteor processes on the same server. Using different ports the load balancer will be able to distribute requests between multiple instances on the same machine just like it would between different physical server. Usually the operating system will take care of running the different processes on different cores, but if you want direct control over which Meteor instance runs on which core you can use the `taskset` package on Linux.

> **TIP** Running multiple instances of Meteor on the same server will allow you to utilize more than a single CPU core but it will not provide true high availability unless you use two or more separate servers.

Secure Connections – SSL

Although Node.js supports SSL connections, Meteor itself does not. To provide secure connections SSL can still be used on the load balancer facing the clients. All traffic between

[20] At the time of writing the Node.js cluster package was marked as „Stability: 1 – Experimental“

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

the load balancer and the application server will then be unencrypted, which is referred to as SSL Offloading. All messages between a user and the datacenter will still be fully secure, only inside the datacenter you need to trust the provider of the infrastructure.

You need to take care when load balancing across datacenters. If the load balancer is located in the US, but it redirects to a Meteor server in Japan, it must not terminate SSL in the US but instead redirect the SSL request to Japan and use a local load balancer to offload the SSL. Otherwise the users may see an HTTPS connection, but in truth it will go halfway across the world just like any other unsecured request.

In many environments nginx is used for ssl offloading, so it shields the Meteor application from handling any concerns around infrastructure and being able to concentrate on a single purpose: running the application.

SESSION STATE

Coming back to our Scrabble game we assume four players are logging in on two load-balanced servers. Two of the players are now on server #A, two are on server #B, but they should all join the same game. The simplest way to share information between all instances is to use the database as the central place of information exchange. While from the application perspective it may seem like an ideal approach, it adds a lot of load to the database servers, requiring many disk operations for reading and writing which can eventually lead to a noticeable lag for the user. Afterall, it does not make sense to persist data that is only short-lived like the position of your opponent's mouse cursor or whether they are currently typing.

A better solution to exchanging volatile data between application servers is to use Redis, an in-memory database. In a nutshell, Redis is a simple key-value storage. It is similar to MongoDB as it supports sharding and provides a means to failover to another server if the primary process is unavailable. The key difference is that all data is kept in memory which removes all disk I/O and makes Redis a very fast and efficient way to store all session relevant data.

PROXY SERVERS

Although technically not part of high availability a good way to increase performance and eventually enable your application to run with less resource usage is to introduce proxy servers.

While Node.js is great for serving dynamic content, it does not work equally efficient for static files. All resources that are the same for all users should be served by a proxy server instead. That means all image files, fonts, and ideally also CSS and JavaScript files should be served by a proxy server. If the requirements are not too demanding both load balancing and serving static content can be accomplished by a single process such as nginx.

12.4.3 Bullet-Proof availability

In comparison, it is relatively easy to design a system to be 99% available. That gives you actually 3.65 days a year or over 7 hours a month that your application may be offline. Going from two nines to five nines (which means 99,999%) reduces possible unavailability of your application to less than 5 and a half minutes in an entire year. Achieving the last 1 percent is

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

exponentially more expensive and rarely worth the effort, unless your application is of vital importance to health or business.

Probability is a driving factor for any decision in system architecture. The more components are highly available the less likely it is that the entire system will shut down. Of course, network routers can fail as well, brownouts may happen, even entire datacenters have been flooded in the past. If you absolutely cannot live with any outages then you should go the extra mile and make sure your servers are in separate datacenters and all cables are connected redundantly. For most use cases however, it will be sufficient to worry about server processes and leave the infrastructure to your provider.

12.5 Summary

- Use version control and a dedicated branch for managing production-ready code
- Make assumptions to estimate the expected load and determine the required architecture
- Decide whether to use a PaaS provider or set up your own infrastructure
- Using environment variables you can determine what components the application server connects to
- Achieving an availability of 99% is relatively easy, but going beyond requires a lot of effort

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

A

Installing Meteor

This appendix covers

- Understanding the prerequisites for installing Meteor
- How to install Meteor on your development machine

In this appendix we will highlight the prerequisites of Meteor and walk you through installing it. Unless you are on Windows you can issue a single command to get started with Meteor, but we will cover all major platforms.

A.1 Prerequisites

In contrast to many other web development tools Meteor is a self-contained installation and does not require any particular software to be present. The installer will put Node.js as well as MongoDB to your home directory, so they will not conflict with any other instances installed by package managers such as `brew` or `apt-get`. To ensure a fully-working environment Meteor will always use the binaries it installed.

Currently supported platforms are

- Mac OS X 10.6 and later
- Linux (x86 and x86_64 systems)

Windows support is scheduled for version 1.1 of Meteor. BSD and other operating systems are not supported, Using virtualization e.g. by running a Vagrant box it is possible to install Meteor and start developing on unsupported systems as well.

If you cannot install Meteor or prefer to run it on the cloud you can use nitrous.io. They offer an integrated development environment in the cloud without the need to install anything locally.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

A.2 Installing Meteor on Linux and Mac OS X

Meteor supports Mac OS X 10.6 and above as well as Linux x86 and x86_64 systems. The installation for the supported systems is just one line to enter in the terminal:

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

This downloads and installs the entire Meteor platform onto the system and makes the command line tool globally available.

Figure A-1 Installation of Meteor on Mac OS X

For the installation you should not need to have administrator privileges. However, it is possible that you must provide your password for Meteor to be able to create a symbolic link to `/usr/local/bin/meteor` so all users on your computer may use the `meteor` command.

UNINSTALLING METEOR Generally you should never run into a need to uninstall Meteor. However, if you have to you can do so by deleting the file `/usr/local/bin/meteor` as well as the `.meteor/` directory inside your user home.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

A.3 Installing Meteor on Windows

The Meteor Development Group will eventually include Windows support in Meteor, but for now there is no official Windows support yet. Still there are some common ways to use Meteor on Windows machines.

The simplest way is to use the unofficial installer for Meteor made by the community which is available at http://win.meteor.com/. You will need to download and run LaunchMeteor.exe on your machine. That way you can use Meteor just like you would on Linux.

A.3.1 Running Meteor using Vagrant

If you are using Windows and want to work with the official releases you can use a virtual machine running Linux as your development environment. One possible way is to use *Vagrant*. It allows you to use portable development environments in form of virtual machines that are tightly integrated into your host system. That way you can easily exchange files and run commands such as `meteor`.

You can download the Vagrant installer from their website https://www.vagrantup.com. You will also need to install VirtualBox, which can be downloaded from https://www.virtualbox.org. Furthermore you should have an SSH client installed. This can either be PuTTY or ssh provided by Cygwin.

Once you have installed all of the above you can add a linux box to your system with the command

```
C:\Users\stephan\> vagrant init hashicorp/trusty32
```

This will create a configuration file named `Vagrantfile` inside the current directory. This file is a Ruby program with most of the lines commented out. These are the configuration settings for your machine. Besides some general configuration settings it includes the blueprint that should be used for setting up the virtual machine. In this case a 32-bit image of Ubuntu 14.04 (Codename: Trusty Tahr), which is provided by Hashicorp, the creator of Vagrant.

HINT Even if you are running a 64-bit OS you can still use a 32-bit guest OS with Vagrant, but not vice versa. In some cases it will be more efficient to use 32-bits, especially if you only assign a small amount of RAM to the guest.

The default value for the guest's RAM is 512MB and it uses a single core of your CPU, which is usually sufficient for small development environment. However, if you need to increase the memory size or core count you can adjust the `Vagrantfile` and add the following just before the last end (resulting in the two last lines both having end in them):

```
config.vm.provider "virtualbox" do |v|
  v.memory = 1024 #A
  v.cpus = 2 #B
end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

#A RAM size in MB
#B CPU cores to assign to the guest

To open your Meteor application in your Windows browser you need to set up port forwarding. For this uncomment (remove the # symbol) the line starting with `config.vm.network` and adjust the settings to

```
config.vm.network "forwarded_port", guest: 3000, host: 3000
```

When the vagrant box is running this will forward all requests to your local machine (host) on port 3000 to the virtual box (guest) on port 3000. Just navigate Internet Explorer or any other browser to http://localhost:3000 when you have started Meteor inside the virtual box.

To start your virtual box issue the command

```
C:\Users\stephan\> vagrant up
```

Vagrant will now fetch the box blueprint defined in the `Vagrantfile` from the internet and store it as a re-usable image (which will speed up all future executions of the `vagrant up` command). You will only need to be connected to the internet for the initial download of the box image. Vegrant will set up your virtual machine and allow you to access it via ssh (default password for most boxes is `vagrant`):

```
C:\Users\stephan\> vagrant ssh
```

From here on, you are basically inside a real, virtualized linux system. The beauty of using Vagrant is that files are instantly shared, so you can use your editor of choice on Windows, and also use your local browser, such as Internet Explorer. By default your user's home directory is shared with the virtual box, so that all files you store in C:\Users\<your usernamer>\ are accessible inside the guest system inside `/vagrant`.

You can now install Meteor in the same way as described in the previous section.

Of course you could also use Vagrant on a Mac OS X or Linux computer if you want to clearly separate the Meteor environment from the rest of your system.

A.4 Summary

- Official support includes Mac OS X, and Linux (both 32 and 64 Bits)
- Vagrant can be used to install Meteor on Windows

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

B

The Anatomy of MongoDB

This appendix covers

- Scalable MongoDB architectures
- How to set up MongoDB with oplog tailing

Out of the box Meteor works hand in hand with MongoDB. If you consider hosting your own database server you should first know the basic components of a MongoDB installation, how it scales, and how it integrates with Meteor.

The following two sections introduce you to more advanced topics. They assume you have a basic understanding of system architecture and administration. In the end you will be familiar with the most important aspects in setting up and running your own MongoDB.

B.1 The MongoDB components

From the outside, a MongoDB is fairly simple. You access it via an endpoint, authenticate with a user and password, and query for what looks like JSON objects. The simplest way to deploy MongoDB is to use a single instance (see Figure B-1).

In a production environment, where you want to make sure the database is always available we need to look behind the scenes to identify the requirements for high availability. Although the database itself is oftentimes referred to in singular, there are in fact multiple processes running. With MongoDB we differentiate between:

- `mongod`
- `mongos`
- `mongoc`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

Figure B-1: The simples way to use MongoDB with Meteor

MONGOD – DATABASES AND SHARDS

The mongod process is the actual database process and takes care of data replication. In the most simple setup it is the only process required for running MongoDB. Once a database becomes too big for a single instance it needs to be scaled out. MongoDB does that by splitting the data between multiple servers, the so called shards. Shards typically contain only a part of the overall database, a subset of all available documents.

Imagine we are going to store a world wide phone book in our database. Because there are a lot of entries we will partition the data to be stored across three servers. The entries for names starting with A-J will be placed on shard #1, K-S on shard #2, and T-Z on shard #3. Each of the shards contains roughly the same amount of data to evenly balance load.

MONGOS - QUERY ROUTING

Because there are now multiple database processes a routing component will be needed to direct requests to the appropriate mongod instance. The Meteor application should only have a single connection and be unaware of any database internal. In a sharded cluster the way an application can access the database is through a process called “MongoDB Shard”, or simply mongos. From an application point of view it behaves just like the mongod process and takes care of distributing data to the correct shard. The application will not know whether it was redirected to any other mongod instance.

If we decide to add a new entry in the phone book for a “Malcolm Reynolds” our application needs to access the database through mongos and write a new entry. But how does mongos know where to redirect the request and store the data?

MONGOC - CONFIGURATION SERVERS

Besides the actual data storing a sharded cluster needs an instance that knows which data resides in which shard. This is where mongoc comes in. This process is often mentioned as the configuration server and technically it is a special kind of mongod instance. It is crucial that the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

config servers are available at all times, although they do not handle a lot of load because the routing instances cache all relevant data for performance purposes. When the routing servers are started they contact the config servers to retrieve cluster metadata. Sometimes a MongoDB may use balancing techniques to split or migrate some data to another shard, which is when data gets written to the configuration servers. The creators of MongoDB recommend using three `mongoc` instances in a production environment. If you decide you do not want to use sharding, then you do not need to use any configuration servers.

In our example the `mongoc` process keeps track that all documents starting with R are stored on shard #2, so the `mongos` instance knows to redirect the application's write request to the corresponding machine.

Replica Sets

As we are designing for high availability we cannot risk to lose a single shard. To prevent the unavailability of data when any single process dies *Replica Sets* are used. Replica Sets can have members of three types:

- Primary

  All write operations are performed on the primary member only. It also maintains the oplog, which is the foundation of all replication action and is used by Meteor as a better alternative to the poll-and-diff approach.

- Secondary

  Secondary members maintain an identical data set from the primary. In case the primary member becomes unavailable they can take over as a new primary. Failover requires the remaining members to vote on the new primary.

- Arbiter

  Although technically not a real member of a replica set, the arbiter has a vote when electing a new primary member. They do not maintain a copy and cannot become a new primary member. An arbiter is a special `mongod` process.

Each replica set requires a dedicated `mongod` process. Multiple processes may run on the same physical or virtual machine, but they must use different ports.

In the phonebook example we have three `mongod` instances running on each of the shards. The primary replica set is the one that is writeable, depending on the exact configuration of the cluster all others will be used as backups for automatic failovers or even as read-only instances for optimum load balancing.

When a primary replica set becomes unavailable it can automatically be replaced by a secondary member. The remaining nodes will vote which secondary member shall be the new primary. Because a member may not vote for themselves an uneven number of members is required in a replica set. When you have three replica sets and one goes down the remaining two will elect the new primary. If for whatever reasons you need to use an even number of replica sets, for example when you two are sufficient and you don't want the additional

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

network and disk i/o overhead of another instance, then using an arbiter will help reach an uneven number. Otherwise, if there is an uneven number of sets already, you most likely do not need an arbiter. Consequently, there should never be more than one arbiter associated with one replica set. Arbiters vote, but they do not put additional load for replication processes on either of the machines. There is no need for an arbiter in Figure B-2 because there is an uneven number of replica set members (3).

Figure B-2: A production-ready setup for MongoDB

OPLOG

Replica Sets are not limited to sharded deployments. Even in single shard deployments - like when running the `meteor` command line tool - replica sets are useful because they enable the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

oplog (operations log) which is an important way to enhance the performance of a Meteor application running on multiple servers.

In our phone book example an administrator might want to import a whole batch of people directly into the database without going through the web application. Also, two instances of the application could be running in parallel, so both can change data at the same time. In both cases the application will only know about any changes if and when it makes a dedicated request (e.g. “find all entries”). The standard behavior for Meteor is to poll the database every ten seconds.

Doing such an operation regularly puts unnecessary load on the database as well as the Meteor server and adds noticeable lag, which is why Meteor is capable of utilizing a much more clever approach, which is to hook directly into the replication stream by subscribing to the oplog. The oplog is a special collection that keeps a rolling record of all operations that modify the data stored in your databases.

COMPONENT DISTRIBUTION

A production grade MongoDB consists of multiple servers, physical or virtual. A minimum of three servers will be required to run the config servers. The actual `mongod` instances could potentially also run on the same servers, although typically you should use a dedicated machine. For the query routing `mongos` you can decide between two best practices:

- Dedicated routing servers, at least two on different servers
- One `mongos` instance deployed on each Meteor server

When running on dedicated routing servers all instances must listen to the same address, because each Meteor server should only ever use one dedicated connection string. Therefore, a load balancer such as haproxy or nginx should be used in front of the mongos. As this introduces another single-point-of-failure the load balancer itself must also be highly-available.

To avoid an overly complex scenario you can simply decide to install `mongos` on every Meteor server you are going to use. Using system tools you can ensure to restart the `mongos` process if it ever crashes, which makes this approach much simpler to manage.

B.2 Setting up MongoDB

While we cannot cover all the details to deploy a fully production-ready cluster for MongoDB with sharding and query routers in this book we will limit this section to the specifics for Meteor. Therefore we will focus on setting up a single instance with a replica set, so that we can take advantage of oplog tailing. Further instructions on how to set up MongoDB in full detail can be found in the official documentation or in MongoDB in Action by Kyle Banker.

SETTING UP OPLOG TAILING

The oplog is stored in a system database called `local`. If you do not have any replica sets defined, then you will not have a collection called oplog.rs. In order to initialize the collection

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

we must define a replica set, but we do not need to add any members. It can be a single primary member only.

Figure B-3: Accessing the oplog with RoboMongo

Each mongod instance has its own configuration[1]. First open the mongodb configuration file `/etc/mongodb.conf`. At the very end of the file we will add two lines:

```
replSet=rs0
oplogSize=100
```

The first parameter, `replSet`, defines the name of the replica set to be used by this mongodb instance. `oplogSize` defines the amount of disk space to be used for the collection, in this case 100MB. If you do not specify `oplogSize` it defaults to 5% of your free disk space.

Next restart the mongod process and open a mongo shell. You can either use a tool such as RoboMongo or use the command line. Once you are connected switch to using the `local` database:

[1] If you are running multiple mongod processes on the same machine, you must take care that you are editing the correct config file.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
> use local
```

Next we can initialize the replica set and thereby enabling the oplog with

```
> rs.initialize()
```

Figure B-4: Initializing a replica set from the mongo shell

As you can see in Figure B-4 two additional collections will be created: oplog.rs and system.replset. Also the shell prompt will change to reflect the replica set name and its member status (`rs0:PRIMARY`).

MongoDB now automatically track all write operations in the oplog.rs collection. Once the specified size is reached it will purge old entries.

SETTING UP AN OPLOG USER

By default MongoDB does not require user to authenticate. In such an environment you can also access the oplog without credentials, so you could skip this step. However, in production environments you should add users to provide a means of access control.

For the purpose of tailing the oplog we shall create a dedicated user that is allowed to access local database and, which is where the `oplog.rs` collection is.

We create an oplog user with the following command:

```
db.addUser( #A
{
    user:'oplog', #B
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
    pwd:'password', #C
    roles:[], #D
    otherDBRoles:{
        local:["read"] #E
    }
})
```

#A In Mongo 2.6 you must use db.createUser instead
#B Set the desired username
#C Set a password
#D No roles are required for this user
#E On the database local this user will have read privileges

B.3 Summary

- MongoDB uses three kinds of processes
- Sharding is used to partition data across servers
- Replica Sets are used to clone data between servers
- Meteor can use oplog tailing to enhance database performance

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

C

Setting up nginx

This appendix covers

- Setting up nginx as a load balancer
- Serving static content from nginx
- Enabling SSL for Meteor applications

While NodeJS – Meteor's underlying server technology - is great for handling events it is not optimized to handle static content such as images well. Even though it is possible to use SSL with a NodeJS application, it is not yet possible with Meteor. Finally, the fact that NodeJS is a single-threaded application that will not take advantage of the full power of multi-core processors might be enough to convince you that scaling Meteor will be complicated.

Thankfully it does not require much to build a production environment for running a Meteor application that takes care of all these shortcomings. In this appendix we are going to discuss how to use the light-weight web server nginx to accomplish all that is necessary to run a rock-solid Meteor project.

C.1 Load Balancing with nginx

Some of the most popular choices for running software load balancers are nginx and haproxy. Both are available as free open source packages, but because haproxy is built as a load

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

balancer and nginx is in fact a webserver which is also capable of load balancing, haproxy provides some more advanced should you need them[2].
We will be using nginx because it can do everything we need to ensure our Meteor application is always available and is more versatile, so we can reduce overall complexity of our setup by just adding a single application to address all our production challenges.

C.1.1 Installing nginx on Ubuntu

On all major Linux distributions nginx can be installed using a package manager. On Ubuntu and Debian the command is:

```
$ sudo apt-get install nginx
```

As Meteor uses WebSockets and nginx only supports them from version 1.3 you should take care that you are using a recent version:

Figure C-1: Installing nginx

[2] Note that the commercial variant nginx plus offers more advanced functionality, but it is not available as open source. More details on the differences between the free and the paid versions of nginx can be found at http://nginx.com/products/feature-matrix/.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

C.1.2 Installing on Debian 7 (Wheezy)

On Debian 7 (Wheezy) the default nginx version is too old to support WebSockets, so you should install it from the Dotdeb repository instead. Simply add these two lines add the end of your `/etc/apt/sources.list`

```
deb http://packages.dotdeb.org wheezy all deb-src http://packages.dotdeb.org
      wheezy all
```

Then issue these two commands to fetch and install the dotdeb GPG key:

```
$ wget http://www.dotdeb.org/dotdeb.gpg
$ sudo apt-key add dotdeb.gpg
```

Finally, after you ran apt-get update will be able to install the most recent stable version of nginx with a simple

```
apt-get install nginx
```

Next we will configure nginx for the meteorinaction.com site that listens to requests, define backend servers running Meteor, and forwarding requests to them. Also nginx must take care to not send any requests to a backend server that is not available.

C.2 Configuring as a load balancer

Similar to Apache nginx uses a general server configuration file and every virtual host should be inside an individual file. We will not touch the general configuration, only create an additional configuration file for our Meteor application.

C.2.1 Creating a site configuration file

First, we create a new file inside the `/etc/nginx/sites-available` directory named *meteorinaction.com*. In order to listen to any requests we just need to define the name under which nginx should be accessed and the port.

Also we want to redirect all incoming requests to have a www-prefix[3]. The corresponding file will look like this:

Listing C-2: nginx site configuration

```
server {
    listen       80;#A
    server_name  meteorinaction.com;#B
    return       301 http://www.meteorinaction.com$request_uri;#C
}
```

[3] While it is convenient for users to drop the www from an address any site running on a „naked“ top level domain creates scalability issues. If you do not use a subdomain, the DNS system will lock you into assigning a single IP address to your domain, which is why we rewrite all traffic to use a subdomain. Users can still access your site without it, they will be automatically redirected. Find out more at http://www.yes-www.org/why-use-www/.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
server {
  listen 80;
  server_name www.meteorinaction.com;#D
}
```

#A Listen on port 80
#B Apply this configuration only when meteorinaction.com is requested
#C Redirect the request to the www-subdomain, maintain the URI string and use a 301 status
#D Apply this configuration to www.meteorinaction.com and all other requests

C.2.2 Defining Meteor Servers

In order to let nginx know which servers to forward requests to we use a module called `upstream`. Because any server configuration may use any `upstream` group, it must not be inside a `server {}` block. At the very beginning of the configuration file place the following block:

```
upstream meteor_servers {#A
    server 192.168.2.221:3000;#B
    server 192.168.2.222:3000;
    ip_hash;#C
}
```

#A You may choose the name for your upstream group as you wish
#B Each line corresponds to one Meteor server instance
#C Further specifies how to distribute requests

You can think of `upstream` as an array of servers. Each line starting with server defines a new instance. The first parameter is the actual address of the server. The machine running nginx must be able to access it, but it is not required that they can be accessed from the internet. In this example both instances are only accessible from a private network in the `192.168.2.0/24` range.

All incoming requests will now be distributed equally between both backend servers. You can further specify parameters such as weight to fine-tune the settings for your environment. Across all requests we need to keep in mind that they are stateful and moving between servers may break a user session. The easiest way to make sure users are not moving back and forth between servers from one request to another is to use the `ip_hash` directive. By adding it to our configuration block we instruct nginx to always send requests from the same IP to the same upstream server.

NOTE If you expect many requests from the same IP using the `ip_hash` directive may result in uneven distribution of users across your servers. In that case you should include the sticky module in your nginx configuration in combination with the `least_conn` directive instead. You can find out more at https://bitbucket.org/nginx-goodies/nginx-sticky-module-ng/overview. Alternatively haproxy might be the better solution for you.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

C.2.3 Forwarding requests to the backend servers

Once nginx listens to requests and knows about the upstream servers we can define how to forward requests. Inside the configuration block for www.meteorinaction.com we add a new location for root (/):

Listing C-3: Location for request forwarding in nginx

```
server {
…
location / {
    proxy_pass http://meteor_server;
    proxy_redirect off;
    proxy_http_version 1.1;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
  }
}
```

Let's go through this configuration line by line:

- `proxy_pass` is used to tell nginx that it should forward requests to this location. It does not use a real URL but the name of the upstream group (`meteor_server`).
- `proxy_redirect` can be used to rewrite the URL request in more complex scenarios. It is not needed in our setup so we switch it off.
- `proxy_http_version` sets the HTTP version to 1.1 (default is 1.0) which is required for WebSockets functionality.
- `proxy_set_header` allows us to add or modify some headers that will be sent to the Meteor server. `X-Forwarded-For` contains the IP address of the user making the actual request. Especially when nginx is on the same host as your Meteor server you need to set this. `Host` passes the hostname for the actual request to the Meteor server. Both `Upgrade` and `Connection` are used to allow forwarding of WebSocket connections.

C.2.4 Activating the nginx site

The final step in configuring the load balancer is to activate the site. First we will create a symbolic link of the configuration that will be placed in the `/etc/nginx/sites-enabled/` directory:

```
$ sudo ln -s /etc/nginx/sites-available/meteorinaction.com /etc/nginx/sites-
      enabled/meteorinaction.com
```

Next we test if the configuration is good to go:

```
$ sudo nginx –t
```

If there are no errors, we can reload the configuration without having the restart nginx:

```
$ sudo nginx –s reload
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
1. stephan@inara: /etc/nginx/sites-enabled (ssh)
stephan@inara:/etc/nginx/sites-enabled$ ls -l
total 0
lrwxrwxrwx 1 root root 33 Jul  9 10:14 meteor -> /etc/nginx/sites-available/meteor
lrwxrwxrwx 1 root root 45 Jul 26 19:19 meteorinaction.com -> /etc/nginx/sites-available/meteorinaction.com
lrwxrwxrwx 1 root root 36 Jul  6 17:45 proxysite -> /etc/nginx/sites-available/proxysite
stephan@inara:/etc/nginx/sites-enabled$ sudo nginx -t
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
stephan@inara:/etc/nginx/sites-enabled$ sudo nginx -s reload
stephan@inara:/etc/nginx/sites-enabled$
```

Figure C-2: Activating the load balancer configuration in nginx

C.3 Serving static content with nginx

Even if you only expect a small number of users, serving content from a Content Delivery Network or using a reverse proxy to serve static files can greatly decrease wait times for the users of your application. If your application is already using an nginx load balancer it requires just some lines of configuration to enable it as a reverse proxy.

Meteor should not have to serve any static files, so we will configure nginx to process all requests for media files and images as well as css and js files. Additionally we will enable gzip compression.

Serving CSS and JavaScript

The `meteor bundle` command automatically minifies and compiles all CSS and JavaScript files and places them in a folder `bundle/programs/client`. If you used `demetorizer` instead you need to replace bundle by the name of your output directory. If nginx is to serve these files they must be accessible from the nginx server. Should Meteor be deployed to a different server you can either copy the files to the nginx machine or configure a folder share using NFS. If you copy the files remember you need to repeat this every time you deploy the application.

To configure serving static application files and styles we need to define a new location block inside our nginx configuration file:

```
server {
…
  location ~* "^/[a-z0-9]{40}\.(css|js)$" {#A
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

```
      root /home/meteor/app/bundle/programs/client;#B
      access_log off;#C
      expires 30d;#D
      add_header Pragma public;#E
      add_header Cache-Control "public";#F
    }
  }
```

#A This applies to all files with names that are 40 characters long and end with js or css
#B The files can be found in this directory
#C No logging of these files to reduce disk I/O
#D Clients may cache these files for 30 days
#E Adds a header that sets Pragma to public
#F Adds a header for Cache-Control to be set to public

During the bundling process all CSS and JS files get a new and unique name consisting of 40 characters made up of letters and digits. Only those will be served from the proxy. Adjust the value for `root` to the directory where nginx can find these files. Logging will be deactivated for these static files and clients may cache these files for 30 days (`expires` in combination with the added headers for Pragma and Cache-Control are responsible for this). The next bundling process will result in new filenames, so we will not run into clients having stale cached files even if we deploy a new version of our app before the 30 days of caching are over.

> **NOTE** nginx must have direct access to the files created by Meteor in order to be able to serve them as a proxy. If nginx cannot access these files locally you need to leave out this configuration block.

SERVING MEDIA FILES AND IMAGES

The content of the public folder should also be served from nginx. Because it is accessible at the root of your application we will use file extensions to determine whether a request will be served statically or from Meteor. The configuration is very similar to the block we saw above:

```
location ~ \.(jpg|jpeg|png|gif|mp3|ico|pdf) {
   root /home/meteor/app/bundle/programs/client/app;
   access_log off;
   expires 30d;
   add_header Pragma public;
   add_header Cache-Control "public";

}
```

You may add all file extensions to the regular expression in the location line. During the bundling process all content from the public folder goes into `bundle/programs/client/app` so you must use this as root.

Again, if nginx has no access to these files locally you should either copy them over manually, use a shared storage such as an NFS export, or leave out this configuration part.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

ENABLING GZIP COMPRESSION

The last optimization for serving static files from an nginx reverse proxy is enabling GZIP compression. Even when minified, text files can be compressed very efficiently and especially users with less bandwidth (e.g. mobile users) benefit greatly from using compression. Configuration is done outside of any location blocks but inside a server block:

```
server{
  …
  gzip on;
  gzip_disable "msie6";
  gzip_vary on;
  gzip_proxied any;
  gzip_comp_level 6;
  gzip_types text/plain text/css application/json application/x-javascript
      text/xml application/xml application/xml+rss text/javascript;
 }
```

The first line activates gzip compresson while the second will disable it for Internet Explorer 6. Enabling vary and proxied ensures that even requests going through proxy servers that may be used along the way to the client will be handled correctly. We set the compression level to 6 on a scale from 1 to 9. Lastly we define which MIME types should be compressed.

C.4 Setting up SSL with nginx

Because Meteor itself does not support SSL we will configure nginx for SSL offloading, meaning the SSL connection will terminate at the load balancing proxy server. It does not matter whether you use a self-signed certificate or one that was issued by Thawte, StartSSL, or any other certification authority. You must however copy both the certificate file (.crt) and the key file (.key) to you nginx server.

Your Meteor application and server will run exactly the same as before, there are no changes required. The only configuration we will need to make is inside the nginx site file.

Inside the server configuration block we will switch the port to 443, turn SSL on, and configure the certificate files to be used:

Listing C-4: SSL setup for nginx

```
server {
  listen 443;#A
  server_name www.meteorinaction.com;
  ssl on; #B
  ssl_certificate /path/to/my.crt; #C
  ssl_certificate_key /path/to/my.key; #D
  ssl_verify_depth 3;  #E

 …
}
```

#A listen on default ssl port 443
#B enable SSL

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action
Licensed to Thang Dang <dangxuanthang@gmail.com>

#C path to the certificate file
#D path to the keyfile
#E when using concatenated certificates the depth must be adjusted

Just like in the non-SSL configuration both port and server name must be defined. When `ssl` is set to on we must also provide a certificate and a key file. Sometimes the instructions for installation a certificate issues by an authority asks you to combine multiple files into one. In order for nginx to accept and verify this combined file you should adjust the `ssl_verify_depth` parameter. This defines the maximum number of intermediate certificate issuers, i.e. the number of CA certificates which are max allowed to be followed while verifying the client certificate. When combining your own with a primary and a secondary intermediate certificate the depth should be 3, self-signed certificates have a depth of 0.

Additionally you should add a server that listens on port 80 and forwards all non-SSL traffic to the SSL port:

```
server {
    listen       80;
    server_name  meteorinaction.com www.meteorinaction.com; #A
    return       301 https://www.meteorinaction.com$request_uri;#B
}
```

#A We added both variants with and without www
#B All requests are forwarded to the URL containing www

Now your configuration file should have two server blocks: one that listens on port 443 and the other listening on port 80 which forwards all requests to the SSL server. Test and reload the configuration and you are good to go.

NOTE This SSL configuration is very limited, there are ways to make connections even more secure. Have a look at the code example coming with this book for a full example that will provide maximum security while being compatible with most browsers.

C.5 Summary

- How to install nginx on Ubuntu and Debian
- Setting up load balancing with sticky IPs to multiple Meteor servers using nginx
- Enabling static files to be served by nginx
- Using SSL offloading for HTTPS connections to Meteor applications

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/meteor-in-action