Sample Chapter

MANNING

Meteor IN ACTION

Stephan Hochhaus
Manuel Schoebel

Foreword by Matt DeBergalis

Meteor in Action

by Stephan Hochhaus
Manuel Christoph Schoebel

Chapter 1

brief contents

1 A better way to build apps

This chapter covers

- The story behind Meteor
- The Meteor stack
- Full-stack JavaScript, reactivity, and distributed platforms
- Core components of the Meteor platform
- The pros and cons of using Meteor
- The anatomy of Meteor applications

Meteors have a reputation of changing life as we know it. They're capable of making dinosaurs extinct or forcing Bruce Willis to sacrifice his life for humankind. This book is about a Meteor that impacts web development, but it doesn't 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 new libraries, standards, and services; and bundles them to provide an entire ecosystem for developing web and mobile applications that are a delight to use.

Meteor is an open source, MEAN[1] stack–based app development platform designed to have a consistent JavaScript API across client and server with a focus on real-time, reactive applications, rapid prototyping, and code reuse.

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 websites. The smoothness of Google Maps and the directness of Facebook's Messenger led to users having much higher expectations toward 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.

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

1.1 Introducing Meteor

If you look at the state of web development in recent years, you'll see two clear trends. First, applications become more powerful, often indistinguishable from desktop applications. Frankly, users don't care what the technology is that works behind the scenes; they simply expect a great user experience. This includes instant feedback on clicks, real-time 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 that it's becoming 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 the Y Combinator startup seed accelerator, they planned to build a travel recommendation site. But when they talked to fellow start-up companies, they realized how much they struggled with the same challenges they'd already solved when they worked on Asana, an online platform for cooperative project and task management. So they changed their plans and decided to come up with an open source platform to provide a sound foundation for web applications that are just as smooth to use as desktop applications.

[1] 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.

On December 1, 2011 the Meteor Development Group (MDG) announced the 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 Rod 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 to build pipelines to minimize and compile code and lets developers focus on business functionality. With over $11 million in funding, investors find Meteor very appealing. Similar to Xen, the free hypervisor for server virtualization, or JBoss, the Java application server, the MDG will eventually offer additional tools targeted at larger enterprises.

The MDG divides its 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 the node package manager `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 that 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 don't 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. It bundles and provides all required pieces under one umbrella. It consists of Node.js, MongoDB, the actual application code, and a powerful CLI tool that combines the power of `npm` and `make`. As such, it's 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 goes beyond what other web frameworks offer, at its core it still relies on a stack to run applications.

The Meteor stack (see figure 1.1) is a member of the MEAN family, which means it's powered by Node.js on the server side. Node.js is an event-driven, highly scalable

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

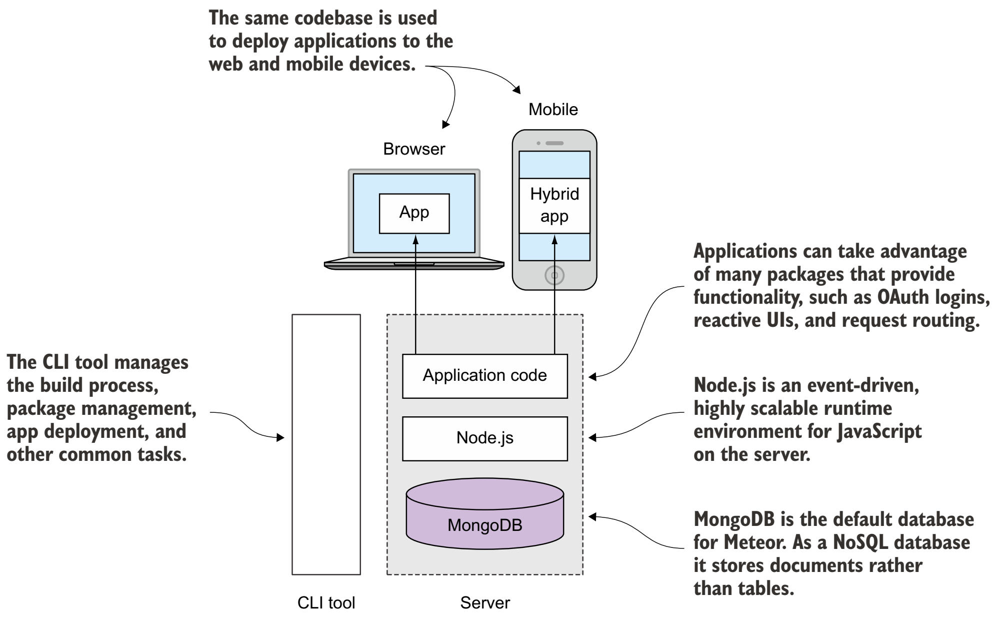

The same codebase is used to deploy applications to the web and mobile devices.
Mobile
Browser
App
Hybrid app
Applications can take advantage of many packages that provide functionality, such as OAuth logins, reactive UIs, and request routing.
The CLI tool manages the build process, package management, app deployment, and other common tasks.
Application code
Node.js
Node.js is an event-driven, highly scalable runtime environment for JavaScript on the server.
MongoDB
MongoDB is the default database for Meteor. As a NoSQL database it stores documents rather than tables.
CLI tool
Server

Figure 1.1 The Meteor stack runs applications powered by smart packages on top of Node.js and MongoDB.

runtime for JavaScript on the server. It serves the same purpose as an Apache web server 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 database is Mongo. It provides a JavaScript API that gives access to all stored content in the form of documents or objects. The same language used inside the browser can be used to access data, which Meteor takes advantage of to implement true full-stack development.

All 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 reactive UI library (Blaze), user account management (accounts), and a library for transparent reactive programming (Tracker).

The Meteor CLI tool allows developers to quickly set up an entire development environment. There's no need to know how to install or configure any server software; Meteor takes care of the infrastructure aspect entirely. It's also both a build tool, comparable to `make` or `grunt`, and a package manager, such as `apt` or `npm`. For example, it can compile preprocessor languages such as 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's 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 use only the server components and use, for example, Angular.js on the client side, or use a Java back end that uses Meteor on the front end to provide real-time updates to all clients.

1.1.3 Isomorphic frameworks: full-stack 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.

Although many frameworks use the same language on both client and server, most of the time they can't share code between the two instances because the frameworks aren't tightly integrated—for example, they use Angular on the front end and Express.js on the back end. Meteor is truly full-stack 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 database. To get started you don't have to learn multiple frameworks, and it results in much better reusability of the code than only using the same language.

To allow you 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 uses only a single thread. In multithreaded 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 (that is, two operations trying 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'll be processed in an asynchronous way. Node.js doesn't wait until these finish, but it attaches a callback and revisits the results of an operation once it finishes, 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 the 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, preheat the oven, put the pizza in, and set an alarm. That's the point when we actually branch off a subprocess. At its core, cooking the pizza in the oven is a long-running I/O process. We set the alarm to be notified when it's done so we can

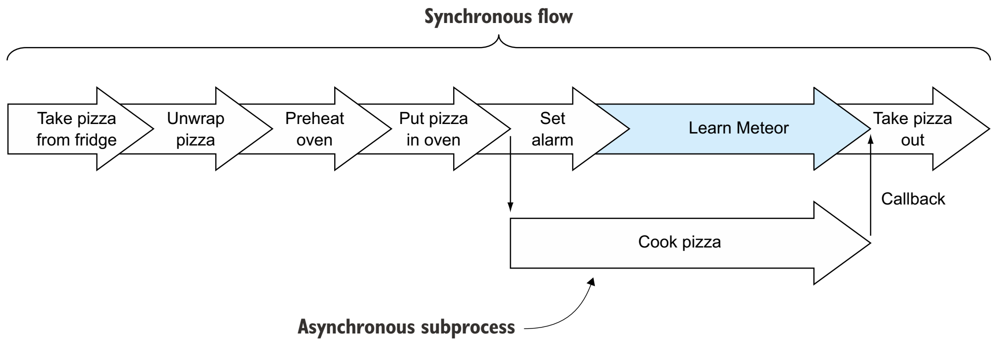


Figure 1.2 Synchronous and asynchronous events when heating up a pizza

attend to more important matters, like learning Meteor. When the alarm goes off, it calls our attention and puts the result of the subprocess back in our synchronous flow. We can take the pizza out and move on.

As you can see from this example, cooking the pizza doesn't block your flow. But if your colleague also wants a pizza and you have room for only one inside the oven, he'll need 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's 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. But 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.

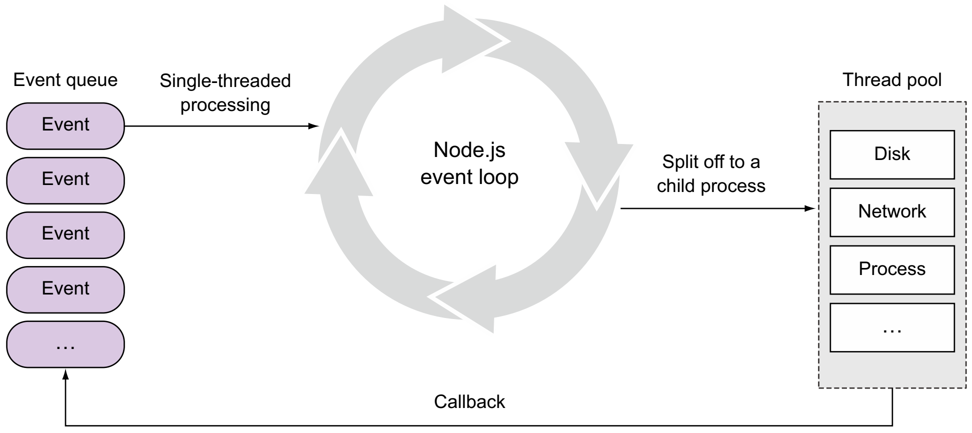


Figure 1.3 The Node.js event loop

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

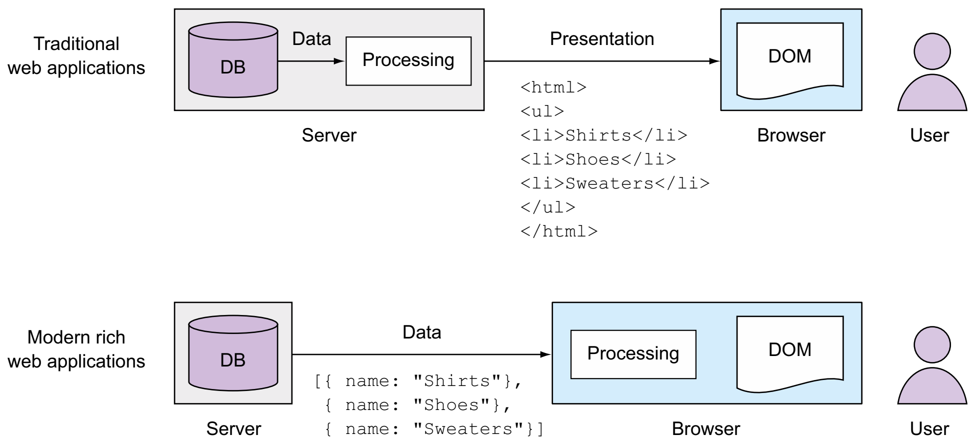
Fortunately, Meteor leverages the full power of the event loop, but it makes it easy because you don't have to worry so much about writing asynchronous code. It uses a concept called *fibers* behind the scenes. Fibers provide an abstraction layer for the event loop that executes asynchronous functions (tasks) in sequence. It removes the need for explicit callbacks so that a familiar synchronous style can be used.

1.1.4 Processing in the browser: running on distributed platforms

When using a back end 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 client. The more clients request the same information, the more caching can be done 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 bottleneck. 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 supercomputers 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 Document Object Model (DOM) only in the browser or mobile device (see figure 1.4).



```
<html>
<ul>
<li>Shirts</li>
<li>Shoes</li>
<li>Sweaters</li>
</ul>
</html>
```

```
[{ name: "Shirts"},
 { name: "Shoes"},
 { name: "Sweaters"}]
```

Figure 1.4 The difference between traditional and modern rich web applications

This client-centric approach brings two significant advantages:

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

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 won't see the updates but an historic snapshot of the site. There's no feedback channel from the server to the client to push out updated content.

Imagine you open your local movie theater's site and see only two seats are left to the new Joss Whedon movie premiere. While you debate whether you should go, someone else buys these tickets. Your browser keeps telling you two seats are available 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 be sent in both directions. In Meteor, the browser is a smart client. Connections aren't stateless anymore; the server may send data to the client whenever there are updates to subscribed content. Figure 1.5 shows the various architectures. To allow bidirectional 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.

As a consequence of moving applications to the browser, all clients essentially become nodes of an application cluster. This introduces new challenges already

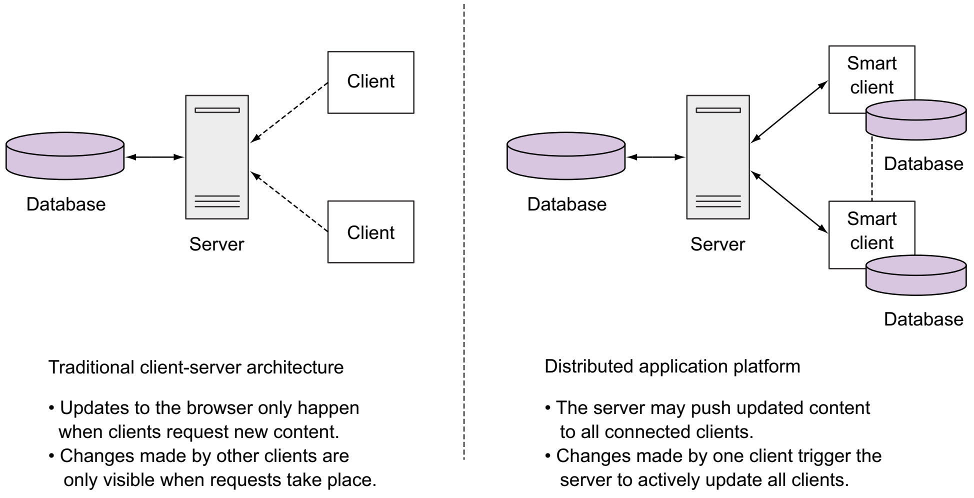

Database
Server
Client
Client
Traditional client-server architecture
• Updates to the browser only happen when clients request new content.
• Changes made by other clients are only visible when requests take place.
Database
Server
Smart client
Database
Smart client
Database
Distributed application platform
• The server may push updated content to all connected clients.
• Changes made by one client trigger the server to actively update all clients.

Figure 1.5 Traditional client-server architectures compared to distributed application platforms

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[1] you sent off with a plan. No matter what happens, the golem will keep walking and following directions. As its creator, you must be diligent about each and every step you command. For example, in an application you must define that you want to listen to changes to a drop-down element and what actions to take when a new value is selected. Also, you need to define what the application should do if another user has meanwhile deleted the associated entry while the first wants to display its contents. 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. Especially on the web, a lot of events happen and the more complex usage scenarios get, the harder it is to foresee in which sequence events will occur.

In a desktop environment reactivity is the norm. When you use a Microsoft 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 *refresh*. An event, such as changing a cell, triggers reactions in related parts of the sheet. All cells are reactive.

To illustrate how reactivity differs from procedural programming, let's look at a simple example. We have two variables: `a` and `b`. We'll 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` won't change unless we explicitly call the addition code again. A developer therefore needs to include a checking method to observe whether `a` or `b` has 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's no need to periodically check that neither `a` nor `b` has changed or even explicitly initiating recalculations. 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

[1] A golem is a mythical creature. Usually made from clay, it's magically brought to life and carries out its master's wishes to the letter. Terry Pratchett's *Feet of Clay* is a good first introduction if you're a fan of the fantastic.

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.[1] To do so, they must be nonblocking and asynchronous. Remember when we talked about full-stack JavaScript? Then you'll notice that reactivity and JavaScript are a natural fit. Also we discussed that Meteor applications run distributed and that the server isn't the only instance 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 doesn't bring down the entire application.

Although 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're 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 real time. There's 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 most 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 How Meteor works

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 CLI 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.

[1] The reactive manifesto defines how reactive systems should be designed and behave in production environments; see www.reactivemanifesto.org.

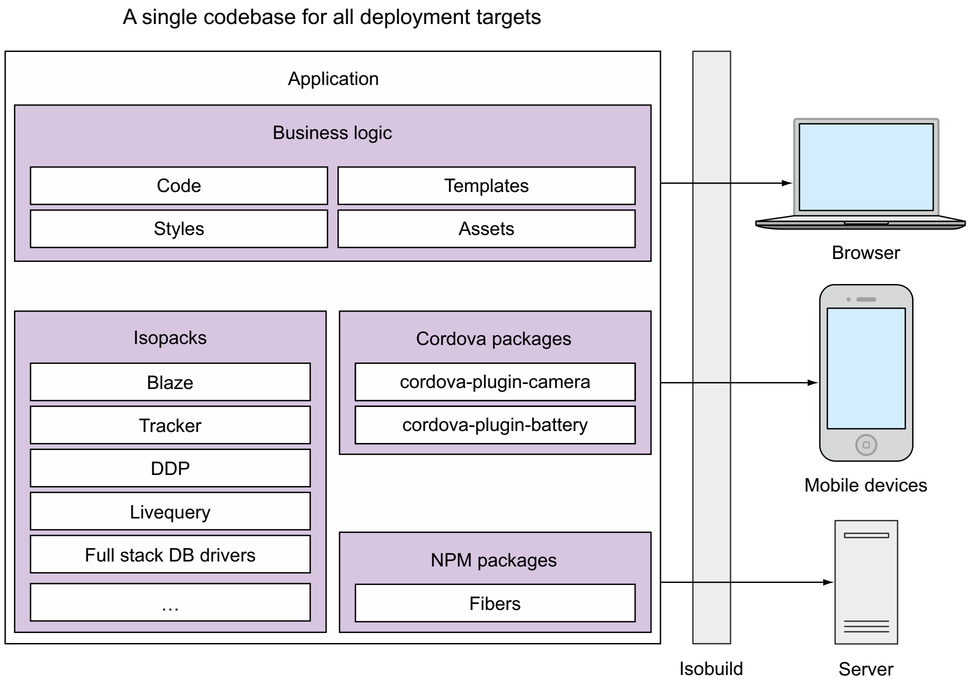


Figure 1.6 Applications consist of business logic and various packages, which are compiled for the target platform using Isobuild.

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` and Cordova 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 also contain 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.

Meteor applications communicate over both HTTP and WebSockets (see figure 1.7). The initial page request and all static files such as images, fonts, styles, and JavaScript files are transferred over HTTP. The applications running on client and server rely on the DDP protocol to exchange data. SockJS provides the necessary infrastructure. The client calls methods on the server using remote procedure calls. The client is calling a function over the network. The server sends back its responses as JavaScript Object Notation (JSON) objects. Furthermore, each client may subscribe to certain data publications. The Livequery component takes care of pushing out any changes to a subscribed dataset over DDP as well. The reactive Tracker library watches for those changes and triggers DOM updates in the UI layer via Blaze.

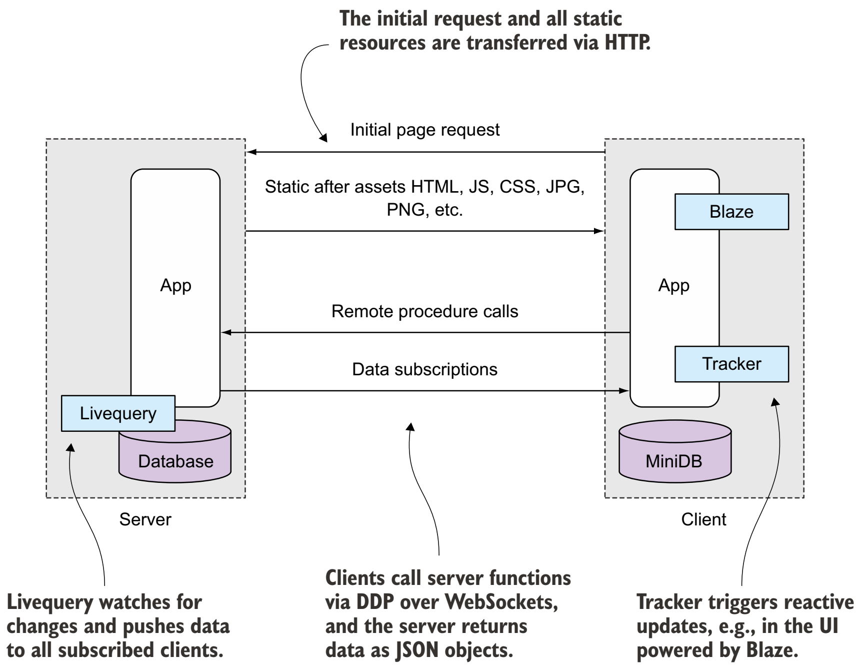

The initial request and all static resources are transferred via HTTP.
Initial page request
Static after assets HTML, JS, CSS, JPG, PNG, etc.
App
Remote procedure calls
Data subscriptions
Livequery
Database
Server
Blaze
App
Tracker
MiniDB
Client
Livequery watches for changes and pushes data to all subscribed clients.
Clients call server functions via DDP over WebSockets, and the server returns data as JSON objects.
Tracker triggers reactive updates, e.g., in the UI powered by Blaze.

Figure 1.7 Communication between the server and client

1.2.1 Core projects

Meteor ships with a number of packages that provide commonly used functionality for web-based applications. A CLI 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

Blaze is a reactive UI library, and one of its parts is the templating language Spacebars. Because developers usually (only) interact with the front-end lib via the templating language and Spacebars is relatively easy to use (in comparison to other templating languages), Blaze is simpler to use than React, Angular, or Ember.

The official documentation describes Blaze as a "reactive jQuery," a powerful library to update the DOM. But it doesn't 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 re-renders only small fragments inside a template and not the entire page. 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 data sources, 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, and `c` is the consumer. A change to either `a` or `b` triggers a recomputation 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 reexecuting functions.

DDP

Accessing web applications is usually done over HTTP, which by definition is a protocol for exchanging documents. Although it does have advantages for transferring documents, HTTP has several shortcomings when passing data only, so Meteor uses a dedicated protocol based on JSON called DDP. DDP is a standard way to pass data over WebSockets bidirectionally, 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. DDP implementations are available in most major languages like Java, Python, or Objective-C. This means you can use Meteor just as a front-end component for an application and use a Java back end 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.

FULL-STACK 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 reuse 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, nonpersistent implementation of MongoDB in pure JavaScript. It doesn't rely on HTML5 Local Storage because 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's 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. Nobody likes to wait, so 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 sending requests to the server.

The client doesn't have to wait for the remote database to finish writing, but the application assumes it'll 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.

A typical flow of events is shown in figure 1.8. Once a user publishes a comment, it'll be validated first and then immediately 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. In the background, though, the action is still ongoing.

The comment gets sent to the server, where again validations take place, and afterward the comment is stored in the database. A notification is sent to the browser indicating whether the storing operation was successful. At this point, at least one full server round-trip together with some disk I/O took place, but those have no impact on the user experience. From a user's perspective there's 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.

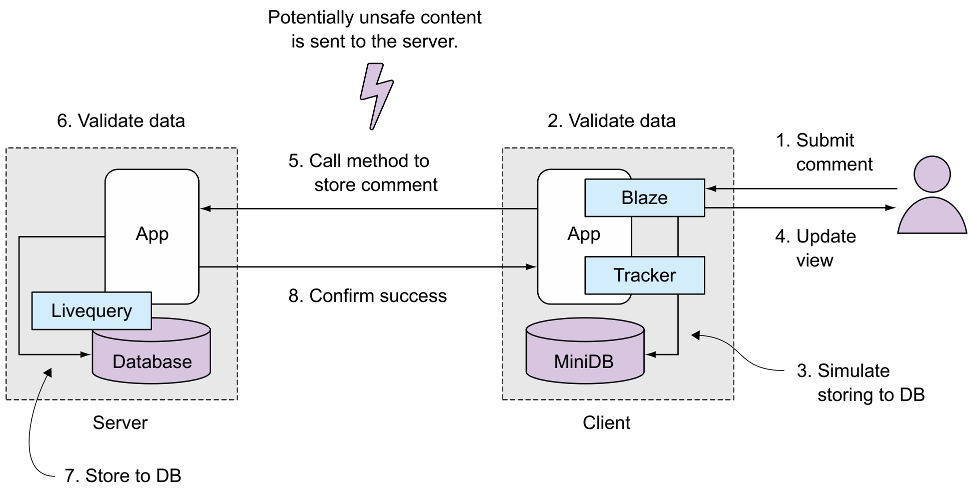


Figure 1.8 Data flow using latency compensation

ADDITIONAL PACKAGES

Besides the core packages many more packages are available as part of Meteor and are 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 CLI tool*

On a computer with a Meteor installation, entering `meteor` on the command line will bring up the CLI tool. This tool 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 the meteor.com infrastructure

Creating a new project is a single command upon which the CLI 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 in the 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, creates source maps, and takes care of any package dependencies. As such, it combines the power of `grunt` with `npm`.

There's 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 compiling from CoffeeScript to JavaScript.

1.2.3 *Client code vs. server code*

When you begin working with Meteor, you'll find that knowing which code should be executed in which environment is essential to writing applications. Theoretically all code can run anywhere in the stack, but some limitations exist. API keys should never be sent to the client—event maps that handle mouse clicks aren't useful on the server. To let Meteor know where to execute specific code, you can organize files in dedicated folders or use a check to verify in which context they're running.

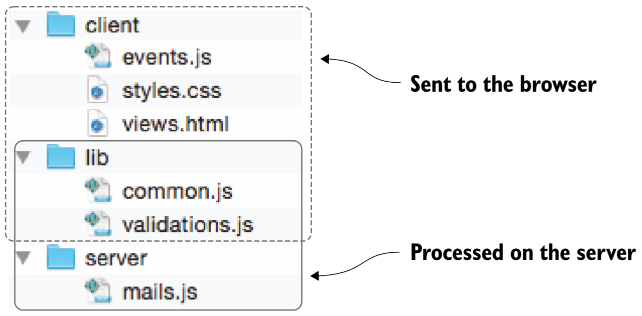


Figure 1.9 The file structure for a simple application

As an example, all code that handles mouse events might be placed inside a folder named client. Also, all HTML and CSS files won't be needed on the server side, which is why they'll 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'll be kept exclusively on the server (see figure 1.9).

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. You can easily use front-end libraries like jQuery on the server as well.

Sharing code between both instances is especially helpful when it comes to input validation. The same method to validate that 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'd have to define one method in JavaScript for validation in the browser and another method in your server context because everything coming from the browser must be validated before working with it to establish a certain level of security. If your back end is written in a language like Ruby, PHP, or Java, not only is there redundant code, but also the same task needs to be done twice. Even if using JavaScript on the server side in other frameworks you'd need to copy and paste the validation section to a file on the server and a second file on the client. Meteor removes this need by processing the same file on both ends.

During the initial page load all JavaScript, styles, and static assets like images or fonts are transferred to the client.[1] As figure 1.10 shows, all files are available on the server but aren't executed as part of the application. Similarly, not all files are sent to the client, so developers can have better control over which code runs in which environment. Transferring files is done via HTTP, which is also used as a fallback for browsers that don't support WebSockets. After the initial page load, only data is exchanged via DDP.

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

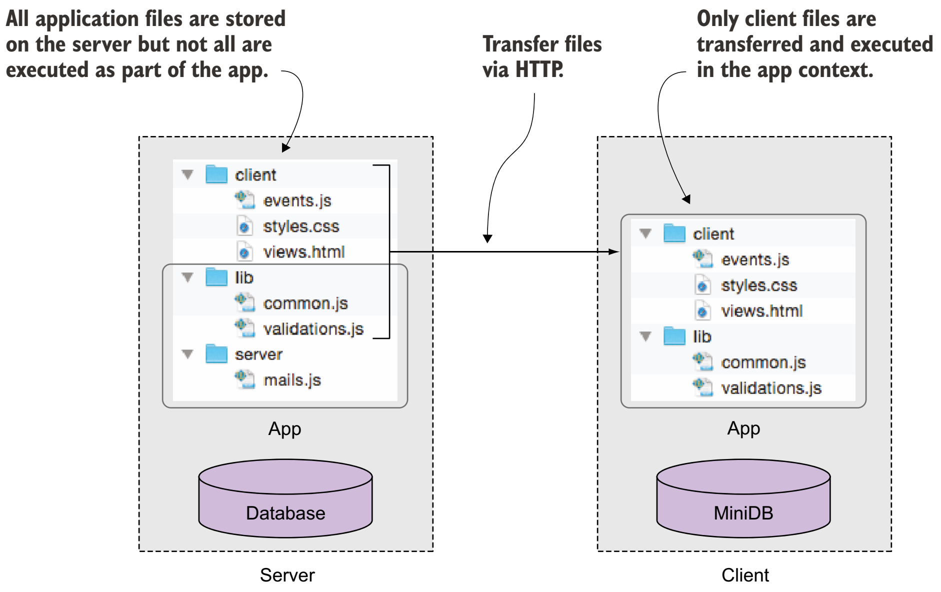


Figure 1.10 Data exchange between server and client via HTTP and DDP

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 on the principles of distributed application platforms will greatly benefit from using it, whereas the more static a site is, the less you'll gain from using Meteor.

1.3.1 Where Meteor shines

The Meteor platform offers all the tools required to build applications for different platforms—the web or mobile. It's a one-stop shop for developers and makes it much simpler to get started with than most other frameworks. The main advantages of Meteor are a single language across the entire stack, built-in reactivity, and a thriving ecosystem of packages to extend existing functionality. In summary, this translates to development speed.

Having one language across the entire application stack, a protocol that's 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 back ends. That makes Meteor extremely well suited for projects that require fast results while still meeting high user expectations.

Easy to learn

Quickly achieving visible results is one of the best motivators for learners. Meteor leverages the power of the MEAN stack, which may be very powerful but also rather complex to learn. To increase developer productivity, Meteor exposes this power behind one common JavaScript API. New developers don't have to take a deep dive into the specifics of loosely coupled front-end libraries and back-end frameworks before they can achieve results. Knowing the fundamentals of JavaScript is sufficient to get started.

Meteor's common API also makes it easier to work with the Node.js event loop by allowing developers to write synchronous code instead of worrying about nested callback structures. Existing knowledge can be reused, because familiar libraries like jQuery or Underscore are part of the stack.

CLIENT-SIDE APPLICATIONS

With increasingly powerful clients, much of the application can be executed on the client instead of the server. This gives us two main benefits that are also valid for Meteor:

- Less load on the server as clients perform some of the processing
- Better responsiveness of actions in the user interface

To efficiently promote browsers to smart clients, it's important to provide a two-way communication infrastructure so that the server may push changes out to the client. With DDP, Meteor provides not only a transport layer but a full solution for communicating in both directions. These stateless connections are a core feature of the platform, and developers can take advantage of them without worrying about message formats.

INSTANT UPDATES USING REACTIVE PROGRAMMING

Much of an application's code is about handling events. Users clicking certain elements may trigger a function that updates documents inside the database and updates the current view. When using reactive programming, the code you need to write for handling events is reduced. Massive collaboration that consists of hundreds of events becomes much more manageable. For that reason, Meteor is especially suited for real-time chats and online games or even to power the Internet of Things.

HIGH CODE REUSE

Meteor delivers on the old Java promise: write once, run anywhere. Because of the isomorphic nature of Meteor, the same code may run inside the browser, on the server, or even on a mobile device.

For example, in REST architectures the back end must talk to the database in SQL while the clients expect JSON. Taking advantage of in-browser mini-databases, the server can publish a handful of records to a client, which in turn accesses this data as if it were in a real database. That enables powerful latency compensation with minimal coding requirements.

POWERFUL BUILD TOOLS

Out of the box, Meteor offers a CLI tool that acts as a package and build manager. It covers the entire build process, from gathering and compiling of source files to minification, source mapping, and resolving of dependencies. This Isobuild tool optimizes an application for the web or packages it as a mobile Android or iOS app.

1.3.2 Challenges when using Meteor

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

Processing intense applications

Especially when your application relies on heavy processing such as data-crunching extract, transform, and load (ETL) jobs, Meteor won't be able to handle the load well. By nature, any Node.js process is single-threaded, so it's much harder to take advantage of fast multiprocessor capabilities. In a multitier 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. But 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 is relatively young and still has to prove itself in production environments in regard to scaling or search engine rankings. Scaling applications in particular requires a lot of knowledge about the components involved and possible bottlenecks.

Although Node.js has proven that it's 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's 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 isn't written with scalability and performance in mind.

Even if the Meteor community is friendly and helpful, it is in no way comparable with the huge resources available for PHP or Java. The same goes for hosting options; there aren't 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, several solutions are available.

As with all young projects, the number of tools available around the framework itself is rather limited with Meteor as of now. Velocity is a community-driven effort to create a testing framework, which has active developers but isn't part of the core Meteor projects. Also, debugging tools aren't 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 who can quickly hack on code, but it requires good coordination between team members when applications grow in size. It's up to developers' preference 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 the only officially supported database is MongoDB. To use additional systems like MySQL or PostgreSQL, community packages must be used. Although several community members have successfully launched applications backed by SQL databases, no full-stack support exists for latency compensation and transparent client-to-server updates. If you need a rock-solid and fully supported stack with relational data high on your priority list, then Meteor is not yet for you.

Serving static content

Some sites like newspapers and magazines rely heavily on static content. Those are the sites that profit most from server-rendered HTML and can use 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 won't be able to leverage all the advantages of Meteor. In fact, you'll need to find ways to work around its standard behavior to optimize for your use case and you therefore might want to use a more traditional framework to build your site.

Who is using Meteor? (From the horse's 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. The main criterion for picking a technology was speed—both in real-time 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 two 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

With lookback.io, it's 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 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 Meteor's event-driven nature a perfect fit for enhancing sales using dynamic merchandising and real-time promotions and pricing. Having a single codebase for the web and mobile devices was a big plus. Meteor isn't 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, he narrowed his choices 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 in 2012 despite its still limited feature set at that time. 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've discussed a lot of theory; now it's time to look at the code. Before you proceed, make sure you've installed Meteor on your machine. Refer to appendix A to guide you through the process.

Because Meteor is also a CLI tool, we'll 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.

1.4.1 Setting up a new project

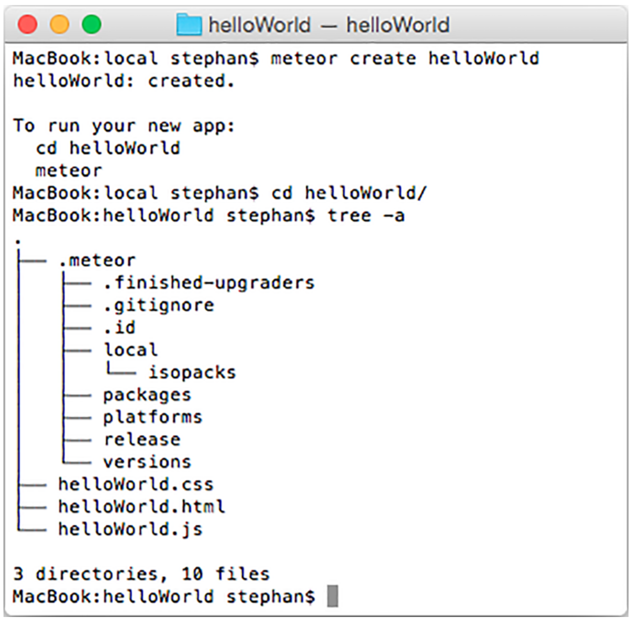
When Meteor is installed, the CLI tool is used to create a new project. Navigate to the folder you want to contain your application and type the following into the terminal (see figure 1.11):

```
$ meteor create helloWorld
```

Meteor creates 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 1.11), 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.



```
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
│   ├── local
│   │   └── isopacks
│   ├── packages
│   ├── platforms
│   ├── release
│   └── versions
├── helloWorld.css
├── helloWorld.html
└── helloWorld.js

3 directories, 10 files
MacBook:helloWorld stephan$ 
```

Figure 1.11 A basic Meteor application created with the Meteor CLI tool

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's even a little more complex, it's better to create folders and split your code into separate files to maintain a better overview. We'll take a close look at how to structure your projects in the next chapter.

1.4.2 Starting the application

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

```
$ meteor run
```

You can also start a Meteor server by calling the `meteor` command 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 1.12).

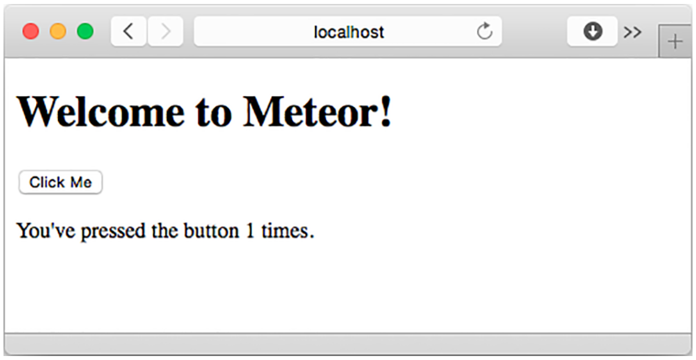
Should you need to change the port Meteor is listening on, you can do so by passing the argument `--port` to the `meteor` command. The following command starts Meteor on port 8080:

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

As you 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've clicked it since you loaded the web page. This is because the application already has an event binding included. Let's take a closer look at the file contents and find out how that binding works.

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're all executed on the server and sent to the client. Let's see what each file does.



Welcome to Meteor!

You've pressed the button 1 times.

Figure 1.12 Every new Meteor project is a simple application with a single button.

1.5.1 *helloWorld.css*

By default, this file is empty. Because it's 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'll see the background of your application in a beautiful red color.

1.5.2 *helloWorld.html*

The file shown in listing 1.1 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 isn't fully valid HTML as you'd expect.

Listing 1.1 `helloWorld` template

```
<head>                                           <- The HTML head
  <title>helloWorld</title>
</head>

<body>                                           <- The page's body, which prints
  <h1>Welcome to Meteor!</h1>                       a heading and imports a
  {{> hello}}                                       template named "hello"
</body>
                                                 The actual template
<template name="hello">                       <- named "hello"
  <button>Click Me</button>
  <p>You've pressed the button {{counter}} times.</p>   <- counter, a helper
</template>                                                 that's filled
                                                            dynamically
```

First, three different elements appear 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 you don't need to worry about it.

The `body` consists of only an `h1` heading and a placeholder using Handlebars syntax. 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's followed by the name of the template. Therefore, the placeholder inserts a template named `hello` into the `body` element:

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

When starting the server, 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 is case sensitive and 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 you'll see in the helloWorld.js file in the next section. Again, the name of the template is used to make that connection.

Finally, you need a way to inject data from the JavaScript code into the template. That's the purpose of `{{ counter }}`, a so-called *helper*. Helpers are JavaScript methods with a return value that's available to the template. If you look at your browser, you'll 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 show you 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. Sometimes, though, you need to specify whether code should run only on the server or only on the client. By checking for these two attributes of the global `Meteor` object, you can always find out where you are.

In any project, the code block in the first `if` statement will run only if the context is the client and the code block in the second `if` statement will run only 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 because doing so may send it directly to the client as well. Anyone opening the source view inside a browser could simply read the lines of code and with those any security-related information, and you definitely don't want this to happen.

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

Of course, there are simple and standard ways to handle sensitive code, and we'll cover this topic in the upcoming chapters when we discuss how to structure a project.

For now, we'll 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 you see two global objects in use: `Session` and `Template`. `Session` allows you to store key-value pairs in memory. The `Template` object enables you to access all templates, which you defined in the HTML file, from your JavaScript files. Because both objects are only available on the client, they can't be called on the server. That would lead to reference errors, which is why this code is wrapped inside the `isClient` context.

As long as `Session` variables aren't 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 you 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 you see why the `hello` template is different from what you actually see in the browser. The template helper in the `hello` template `{{ counter }}` is in fact a function that returns the string that you see in the browser.

On the one hand, you have templates to define the HTML that should be rendered, and on the other hand, you 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) {                    <-- Handling mouse clicks is only useful on the client.
  Template.hello.events({
    'click button': function () {         <-- Define a function to call when the input button is clicked.
      // increment the counter when button is clicked
      Session.set("counter", Session.get("counter") + 1);   <-- Increases the Session variable by 1
    }
  });}
```

Every template has the function `events()` and with that you can define event handling for a specific template. The object to pass to the `events()` function is called an event map, which 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's called if the event is fired.

To specify the event, always use a string in the form `'event target'`, where the target is defined by standard CSS selectors. You can easily change the previous example 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 wouldn't call the function specified here.

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

Notice that there's no code involved that updates the template directly; you rely on the reactive nature of `Session`. The template helper `counter` is rerun whenever the value inside the `Session` object changes. The event simply changes the data source, and Meteor takes care that all places that use this value are recomputed instantly.

The last snippet we'll 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 several different functions at the startup of the application. `Meteor.startup` can also be used on the client side to run functions at the start of the client-side application. This sample application doesn't use any server-side code, so this block and the startup function remain empty.

Now that you've looked at the `helloWorld` example code and you have a solid understanding of the basic concepts, you'll extend these files to develop your own first Meteor application.

1.6 Summary

In this chapter, you've learned that

- Meteor is a full-stack 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 bidirectional message exchange.

- Meteor uses the principle of reactive programming to minimize the need for infrastructure code.
- Development productivity is enforced by reusable packages called 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.

JAVASCRIPT

Meteor IN ACTION

Hochhaus • Schoebel

Free eBook
SEE INSERT

You might call Meteor a reactive, isomorphic, full-stack web development framework. Or, like most developers who have tried it, you might just call it awesome. Meteor is a JavaScript-based framework for both client and server web and mobile applications. Meteor applications react to changes in data instantly, so you get impossibly responsive user experiences, and the consistent build process, unified front- and back-end package system, and one-command deploys save you time at every step from design to release.

Meteor in Action teaches you full-stack web development with Meteor. It starts by revealing the unique nature of Meteor's end-to-end application model. Through real-world scenarios, you'll dive into the Blaze templating engine, discover Meteor's reactive data sources model, learn routing techniques, and practice managing users, permissions, and roles. Finally, you'll learn how to deploy Meteor on your server and scale efficiently.

What's Inside

- Building your first real-time application
- Using MongoDB and other reactive data sources
- Creating applications with Iron Router
- Deploying and scaling your applications

Readers need to know the basics of JavaScript and understand general web application design.

Stephan Hochhaus and **Manuel Schoebel** are veteran web developers who have worked with Meteor since its infancy.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/meteor-in-action

"An enjoyable and approachable book."
—From the Foreword by Matt DeBergalis, Founder Meteor Development Group

"An invaluable guide for any developer, from beginner to expert."
—John Griffiths, UXGent.co

"The only source you need to develop reactive, commercial-grade apps."
—David DiMaria Collective Sessions

"The definitive resource on Meteor. The book's depth is unparalleled and the examples are real-world and comprehensive."
—Subhasis Ghosh, ISACA

ISBN 13: 978-1-617292-47-7
ISBN 10: 1-617292-47-8
54499
9 781617 292477

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]