Rapid Prototyping and Scalable Deployment

Building

Node Applications

with MongoDB and Backbone

O'REILLY®

Mike Wilson

Building Node Applications with MongoDB and Backbone

Build an application from backend to browser with Node.js, and kick open the doors to real-time event programming. With this hands-on book, you'll learn how to create a social network application similar to LinkedIn and Facebook, but with a real-time twist. And you'll build it with just one programming language: JavaScript.

If you're an experienced web developer unfamiliar with JavaScript, the book's first section introduces you to the project's core technologies: Node.js, Backbone.js, and the MongoDB data store. You'll then launch into the project—a highly responsive, highly scalable application—guided by clear explanations and lots of code examples.

- Learn about key modules in Node.js for building real-time apps
- Use the Backbone.js framework to write clean browser code, and maintain better data integration with MongoDB
- Structure project files as a foundation for code that will arrive later
- Create user accounts and learn how to secure the data
- Use Backbone.js templates to build the application's UIs, and integrate access control with Node.js
- Develop a contact list to help users link to and track other accounts
- Use Socket.io to create real-time chat functionality
- Extend your UIs to give users up-to-the-minute information

Mike Wilson is an experienced software architect and web developer who has designed and built everything from government portals and small business sites to MMO server clusters hosting millions of players. He has worked with some of the world's most influential brands, including Disney, Microsoft, and McDonalds.

"This book will not only help you learn Node.js, but Backbone.js and MongoDB as well. Each of these is great all by itself, but this book brings them together to build an incredible, real-time social network."

—Jamie Munro
author of 20 Recipes for Programming PhoneGap *(O'Reilly)*

US $19.99 CAN $20.99
ISBN: 978-1-449-33739-1

Twitter: @oreillymedia
facebook.com/oreilly

O'REILLY®
oreilly.com

Building Node Applications with MongoDB and Backbone

Mike Wilson

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Building Node Applications with MongoDB and Backbone

by Mike Wilson

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Simon St. Laurent and Meghan Blanchette
Production Editor: Kara Ebrahim

Proofreader: Kara Ebrahim
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

December 2012: First Edition

Revision History for the First Edition:

2012-12-07 First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449337391* for release details.

ISBN: 978-1-449-33739-1

[LSI]

Table of Contents

Preface

When Google released the first version of their V8 JavaScript engine in 2008, it felt like a hushed wave of excitement was rippling through the developer community. For the first time (the promise went), we would be able to program with JavaScript on both the client and the server: one language to rule them all. Web applications were already starting to become more desktop-like and ballooning in complexity, so the idea of reducing the number of language dependencies in favor of an open and transparent technology was seen as a way to allow for even more exciting and boundary-pushing applications.

Ryan Dahl was one of the developers who saw the new opportunity and wasted no time converting the non-blocking socket library he had written to the new V8 engine, resulting in the birth of Node.js. The technology he released has turned that original ripple of excitement into a major paradigm shift at a time when interest in responsive real-time applications is reaching a peak. Node.js is more than just a collection of socket functions; it provides a framework for asynchronous I/O that position it as the foundation of a whole new class of event-driven programming patterns.

The online landscape has changed rapidly in the past few years and doesn't show any signs of slowing down. The explosion of the "social" web has meant one big thing for us: more people are online now than ever before, and the demographic has forever shifted away from technical users. The Internet is for all of us, and the winners in this new space will be those companies that can figure out how to make the online experience warm and human by truly connecting individuals to each other.

Using JavaScript to connect your systems puts you at an advantage because you can quickly move from the front of the web stack dealing with human users to the backend

data storage, and all of the network plumbing in between. You will be able to think of your systems as truly modular; each piece can be plugged in and deployed wherever the resources are best suited to it. You will be able to create applications that grow and breathe with your userbase unlike ever before.

Audience and Assumptions

Readers of this book should have an understanding of how websites and web applications are put together. In an effort to stay focused on the core technology, this book brushes past "why" web applications are built in a certain way in favor of the "how."

Some knowledge of JavaScript would come in handy to fully understand the examples in this book. The examples will be thoroughly explained, but prior knowledge will help readers comprehend the back history for programming decisions made during the writing process.

Many developers approach NoSQL data stores as part of a transition from relational database systems. This book makes no assumptions about the reader's proficiency in database design; I will go through the details of why I chose to make various decisions throughout the database architecting phase. MongoDB is friendly to SQL concepts, which is a major motivation for choosing it as the datastore for this project.

In the final section of the book I will discuss a selection of supporting tools and technologies that step outside of the pure JavaScript environment built in the first two sections. Readers are not expected to have a deep understanding of any of those extra languages (like Scala, Java, PHP, or Bash Scripting), but because deep exploration of these concepts are outside the scope of this book, I encourage using these examples as a launching pad for further research.

Organization

This book is broadly organized into two sections, the first providing an overview of Node.js, MongoDB, and Backbone.js (the core technology discussed in this book), and the second detailing how you can go about building a website styled as a social network using these tools. If you are new to any of these I recommend starting with the Part I section to gain a bit of background before diving into the application in the second section. If you are already familiar with JavaScript you will probably be able to skip the first section and find yourself comfortable enough to get through the examples in the second section.

Here's how the book is organized:

Part I: Introduction

Chapter 1, Introduction and Overview

This chapter introduces JavaScript and the core concepts that will be explored throughout the book.

Chapter 2, Node.js

This chapter introduces Node.js and guides you through getting started with your first standalone applications. Here you will become acquainted with the key modules you will later use to build a complete real-time application.

Chapter 3, Backbone.js

Next you will explore how Backbone.js is making programming in the web browser with JavaScript more like building traditional applications and less like building websites. We'll look into some of the more troubling aspects of maintaining JavaScript-based projects, and introduce templating as a way to separate your visual HTML layout from your functional JavaScript application code.

Chapter 4, MongoDB

I love MongoDB because it is fast and easy to set up, easy to interface with, and speaks the same language as my Node.js applications. In this chapter we'll look at how to do basic querying and data manipulation as well as some more complex use cases to think about as your MongoDB usage grows.

Part II: Building a Social Network

Chapter 5, Setting Up the Project

The lack of information about how to structure and put together files in your project is one of the biggest problems facing texts that explain how to build websites. In this chapter we'll set up the Node.js and Backbone project files that will form the website, and lay the foundation for the rest of the code that will be coming.

Chapter 6, Authentication

Before you can do anything with your application, you need a way to create accounts and sign in. This chapter explains how to get users into your database and how to secure their data once you have it.

Chapter 7, The User Interface

Now that the barebone structure and login functionality have been built, this chapter will take you through setting up the web page harness that will contain all of the content presented to your users. This is where we will go into detail on using templates with Backbone.js and integrating access control with Node.js.

Chapter 8, Making Friends
: The contact list is the social aspect behind this website. In this chapter you will learn how to add and remove contacts from your list, denormalizing the data into MongoDB as you go. This will be a departure for anyone coming from a relational database environment; it's recommended reading!

Chapter 9, Chat
: This chapter builds upon the contact list created in Chapter 8 by adding real-time chat functionality using Socket.io. Talk to your friends and receive messages back right away without needing to reload your page.

Chapter 10, Activities in Real Time
: Finally, the user interfaces built throughout the book will be revisited in this chapter and extended with Socket.io just like the chat list. This will add life to the site by giving your users up-to-the- minute information about the comings and goings of their contacts, and turn all of the shared message spaces into interactive rooms.

Conventions Used in This Book

The following typographical conventions are used in this book:

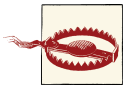
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, if this book includes code examples, you may use the code in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Building Node Applications with MongoDB and Backbone* by Mike Wilson (O'Reilly). Copyright 2013 Mike Wilson, 978-1-449-33739-1."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online (*www.safaribooksonline.com*) is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/mongodb-backbone*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

PART I

Introducing Node.js, Backbone.js, and MongoDB

CHAPTER 1

Introduction and Overview

The Web, already one of the fastest developing areas in technology, is accelerating. This is both good news and bad news for those of us planning to draw income from writing software. Today, good developers have the rare opportunity to do what they love, grow their horizons, and continually evolve and derive even greater satisfaction from their work, as long as they're willing to put in the hard work necessary to understand a huge back catalog of rapidly-expanding knowledge.

Terrific careers come at a price. As a software developer, you must continually search for the next great tool that will help you achieve more, better, faster. What you work with 10 years from now is going to be a major departure from what you are working with today—in essence, you will be retraining yourself multiple times to keep sharp.

In his 2008 book *Outliers* (Back Bay Books), Malcolm Gladwell presents evidence that it takes 10,000 hours of effort to achieve mastery at a professional level. Even prodigies need to put in their time to achieve success; the difference between an average performer and a superb performer comes down to the amount of practice put in by the individual. Picking up a book like this puts you into the latter category; right now you are putting in the extra time to gain more exposure to the leading edge of this craft. The future is arriving, and you will be among the first positioned to take advantage of it.

Node.js has introduced an army of programmers to event-oriented programming. Regardless of what your technology background is, if you come to Node with an open mind and drop any preconceived notions you might have about JavaScript, you will come away with a greater appreciation of how powerful single-threaded programming can be in a world that has gone crazy for multi-threaded applications. What's more, you will have a greater appreciation of event handling that will help you when you do need to tackle multi-threaded problems in other programming languages.

JavaScript is a unique and sometimes misunderstood programming language that has finally taken its deserved place in the development toolbox. As the toolsets for developing JavaScript applications continue to improve and mature, you can look forward to seeing this language's importance continue to grow in organizations worldwide.

Building a Social Network

The project in this book examines how you would go about constructing a social network in a similar vein to LinkedIn, MySpace, or Facebook, with a real-time networking twist. Using Node.js, Backbone.js, and MongoDB you will learn how to create a highly responsive application that can be adapted to scale to millions of users.

By way of example many of the components described throughout the text will take some shortcuts to use a built-in method provided by Node or MongoDB in order to demonstrate certain functionality that wouldn't be practical in "real" large deployments. When one of those shortcuts is presented, I will point it out with a special note and discuss how to begin moving to a more scalable or modifiable construct. The challenge throughout will be trying to balance the need for clarity with the task of building a real and useful application.

What is a social network? "Social network" is a simple phrase that seems to communicate a lot of meaning—and in behavioral science, it does—but let's look more carefully at the individual words and apply them to the Internet. A "network" is an interconnected group of systems, which could be anything from a series of roads crisscrossing the country to a row of computers in a school lab to a Rolodex filled with professional contacts. The word "social" refers to the interaction of organisms—such as animals or people—and to their existence as an entire group. So a social network in this context means an interconnected, interactive group of people.

The human component is important above all else. When building any kind of software you are remiss to develop toward a particular goal or functionality without first (and constantly) thinking about the person who is expected to use your finished code at the end of the day, whether it is a customer, a professor, or even yourself. Unless you can visualize the end purpose for your work, resist the urge to continue down the programming road for technology's sake.

When we speak of building a social network, of course it's impossible to build a social network as defined here. What you will be creating is the forum, the raw pathways, upon which a social network can take root and grow. Every feature of the system is intended to deliver upon that goal by getting out of the users' way and by providing just enough of a feature set to promote, encourage, and facilitate communication without any extra frills. It's a difficult line to walk, but one that ultimately separates a mediocre product from a great one.

Model-View-Controller (MVC)

This book makes frequent reference to, and use of, the Model-View-Controller (MVC) design pattern for both server-side and frontend programming. While MVC was arguably popularized on the web by the growth of Ruby on Rails, it was first developed for the Smalltalk platform in the 1970s.

MVC as it is practiced today promotes decoupling your system into three components:

Model
: A structure containing the data that is being read or acted upon

View
: An interface through which the user interacts with the model

Controller
: Delegates user actions from the view to the underlying model

Models and controllers are typically paired; in this book, the controller's job will be to act as a contract for what a user is able to do to a model, and to pass information back and forth. While it is possible to have a controller perform actions on more than one model, doing so should be considered poor form: one model, one controller.

Views are a different story; just as in real life, in software there is often multiple ways to perceive the same information. For example, a textual transcription of an audio recording contains the same information as the original, but presents its contents in a way that is more accessible to some users or convenient for others. The Internet is full of great examples of this: many web services display data in both JSON and XML format, two different formats that provide the same information in different ways.

Pure JavaScript

Using Node, Backbone, and MongoDB will allow you to focus on your application logic in a single programming language, ultimately reducing the number of connections between each part of your system. As you will see, this is a compelling way to program because the boundaries between client-facing UI, backend server logic, and database persistence will blur into almost a living system. The picture becomes clearer as real-time networking is gradually added; your data will dance across the application and even across multiple users almost as if everything were happening in concert in a single process.

There are pitfalls to watch out for. Although the connectors are strong and speak the same dialect, under the covers your program code is still going across a wire between web browsers, servers, and databases. Some of the JavaScript paradigms change slightly depending on whether their primary goal is to serve UI (as in the case of Backbone in the web browser), authentication (as in the case of Node on the web server), or

persistence (as in the case MongoDB). You need to be ever vigilant about where your data is going, whether or not you are blocking any of your own processes, and how to listen for and react to incoming and outgoing events. It can be a challenge, but as with any other system with lots of moving parts, there are a ton of interesting lessons to glean from the experimentation.

CHAPTER 2

Node.js

The Internet of today is different from the Internet of 1990 and 2000. In the "old days," the interaction between a user and a website was very much oriented toward consumption. The web server would generate largely static pages and the user would navigate between them. There were of course dynamic elements but the interfaction flow was largely limited to request and reply. Years of research have gone into optimizing that client-server flow—it's safe to say that it's well understood at this point in time.

Around the time Internet Explorer 6 started to appear, a subtle but fundamental shift was beginning to take hold. Internet users were becoming more comfortable and savvy online, computers were becoming far more powerful, and broadband connections were starting to become the norm. Instead of using the Internet primarily for information and transactions, people were spending more time online for socializing and entertainment. The Internet is now a media channel, but unlike the television, radio, and newspapers before it.

Instead of consuming data, web users are now producing it in volumes never imagined. The traditional notion of web servers and browsers as consumers is still present, but understanding it provides only a glimpse into what publishers are able to accomplish. The focus now is on putting people in control of their experience, and leveraging the data they create to change, improve, and enhance that experience in real time. This is a new world where the web server and programmer are no longer the sources of experience; rather, they are the facilitators.

Node.js is one of a new breed of technologies geared toward the Internet-as-experience paradigm.

Installing Node.js

The first thing to do before you begin is install Node.js, if you haven't done so already. Node can be downloaded from the project home page where you will be presented with a package installer for your operating system. Binary installers are available for Unix, Mac, and Windows. If you're feeling really adventurous, you can follow the links to Node's GitHub repository and install one of the development snapshots. The stable release version of Node at the time of writing is 0.8.12.

Node provides a JavaScript runtime environment, which you can access at any time by going into your command prompt or terminal window and typing **node**. For the purposes of the book you will be running JavaScript source files rather than typing code directly into Node. To run a JavaScript file, you run the `node` command with a parameter like this: `node filename.js`.

Node ships with a package management utility called `npm`, which enables you to import third-party libraries into your workspace to use in your code. Package management is an important aspect of working with Node; without it, you would have to program all of your applications from the ground up, reinventing solutions to common problems that have already been solved (and shared) by hundreds of other developers.

To install a library using npm, run the `npm` command with the library name as its parameter, like this: `npm install async`.

npm is useful for a lot more than simple installation: it helps you bundle your applications, control library versions, and even share projects with your friends. We'll get into more detail about npm in Part II; suffice it to say for now that you and npm are going to become close friends.

Express

The application in this book makes heavy use of the Express framework. Built upon the Connect HTTP server framework, Express provides view rendering and a language for describing routes.

To install Express, use npm from the directory where you will be working: `npm install express`.

Example 2-1 demonstrates a simple and very practical server built using Express. To begin, the variable `app` is initialized by calling the `express()` function from the `express` library. The `require` command instructs Node to import the library `express` and assign it to a local variable (also called `express`), thus exposing its functionality to the current namespace.

Example 2-1. A small Express application

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

app.get('/stooges/:name?', function(req, res, next) {
    var name = req.params.name;

    switch ( name ? name.toLowerCase() : '' ) {
        case 'larry':
        case 'curly':
        case 'moe':
          res.send(name + ' is my favorite stooge.');
          break;

        default:
          next();
    }
});

app.get('/stooges/*?', function(req, res){
  res.send('no stooges listed');
});

app.get('/?', function(req, res){
  res.send('hello world');
});

var port = 8080;
app.listen(port);
console.log('Listening on port ' + port);
```

With the app initialized, three routes are defined:

/stooges/[name]
: Expecting the name of one of the stooges as input

/stooges/
: A fallback from the previous route, in case the name provided was not found

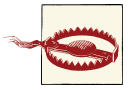
/
: A default route used to access the application's home page

In the first route, Express is instructed to compare the name of the stooge provided and print a message if the name is `'larry'`, `'curly'`, or `'moe'`. In the next route, Express displays a message stating simply that no stooges are listed, and in the third route, a default `'hello world'` message is displayed.

Wait a minute—where did the parameter `next` come from in that first route and what is it? In fact, `next` refers to a function. The `next` command instructs Express to try processing the next route matching the current request. In this example, entering the

URL `/stooges/` is intercepted by the first defined route (`/stooges/:name?`) rather than the second (`/stooges`). Since there was no name supplied, the logic will fall through to the `default` case in the switch statement at which point the `next()` function will be called. The next route (`/stooges`) contains the expected response.

The question mark after the `:name` parameter indicates that the name is an optional input—this route will load even if no name is provided.



A lot of online resources you will come across instantiate their app using `express.createServer()` instead of `express()`. The command `createServer` has been deprecated; although it will probably be supported for quite some time, it is a good idea to avoid using it in order to keep your application as future-proofed as possible.

Inside the switch statement I used a ternary operator as shorthand for an if/else statement: `name ? name.toLowerCase() :: ''`. This is equivalent to a much longer block of code that checks whether the name variable exists, and if so, returns it as a lowercase string, or otherwise sets it to an empty string for comparison:

```
if ( null != name ) {
  name = name.toLowerCase();
} else {
  name = '';
}
```

Templates

Templates allow you to split your presentation information out from your program code, making it easier to arrange your project files and render out web pages with complicated structure. Although this book will focus heavily upon creating and displaying views in pure JavaScript, there are times when you will want to use Express's templating capabilities to render out web pages or simply to make it easier to bootstrap all of your JavaScript files into a web page container. Jade is my engine of choice because it is conceptually similar to CSS and produces clear and succinct code.

To install the Jade templating engine, use the NPM command from your working directory:

```
npm install jade
```

Example 2-2 repeats Example 2-1 but adds support for Jade templates. Instead of printing text to the screen using `res.send` in each of the three routes, Express is instructed to `render` the contents of files with a *.jade* extension: either the *stooges.jade* template for the listing (or lack thereof) of stooge names, or a default *index.jade* for the root of the site.

Example 2-2. Using Jade templates within an Express application

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

app.set('view engine', 'jade');
app.set('view options', { layout: true });
app.set('views', __dirname + '/views');

app.get('/stooges/:name?', function(req, res, next) {
    var name = req.params.name;

    switch ( name ? name.toLowerCase() : '' ) {
        case 'larry':
        case 'curly':
        case 'moe':
          res.render('stooges', {stooge: name});
          break;

        default:
          next();
    }
});

app.get('/stooges/*?', function(req, res){
  res.render('stooges', {stooge: null});
});

app.get('/?', function(req, res){
  res.render('index');
});

var port = 8080;
app.listen(port);
console.log('Listening on port ' + port);
```

The other new addition is the inclusion of the third line of code: `app.set('views'...)`. This command is telling Express that all of its views should be drawn from the folder named *views*, which is subordinate to the directory from where the code file is being run (`__dirname`).

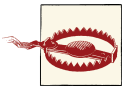
Although Jade is used to render the views, you may have noticed that there is no mention or instantialization of Jade anywhere in the code. This is because the `render` command in Express takes care of loading any of the required template modules—in this case, Jade. If you don't have Jade installed, your application will dump a stack trace and stop working when it comes time to render the page.

Example 2-3 shows the entirety of a page layout using Jade. This is the bit that will surround the "actual" web page—that is, all header information, opening tags, and metadata shared between every page on the site. The body tag will contain the rendered layout for the individual views, which will be included with this layout when Express renders the template.

Example 2-3. The Jade layout (layout.jade)

```
!!! 5
html(lang='en')
  head
    title My Web Site

    block scripts
  block content
```



Indentation carries important meaning within Jade templates; if you are copying and pasting these examples from a PDF, depending on your software the spacing may not be properly maintained.

The first line `!!! 5` translates to the document type for HTML5. It could have been written as `doctype html`, but who wants to type that much?

The template contained in Example 2-4 should be saved to *views/index.jade*. It contains the contents of the website root: nothing more than a simple "hello world" message to anyone who happens by. The first line of this template extends the *layout.jade* template created in Example 2-3, which will cause the layout contents to render in the user's web browser. Because the `h1` tag contents are nested under the block content element in the template, Jade understands that it should be rendered when the `block` content is defined in the layout.

Example 2-4. The new index page in Jade style (index.jade)

```
extends layout

block content
  h1 hello world
```

You might notice that the layout in Example 2-3 also contained a block element called `scripts`, which is not used by the index template. This will cause the script block to remain empty when displayed in the web browser. It is possible to include or omit any block content in your templates, allowing for a lot of flexibility in page layout.

Example 2-5 introduces conditional logic inside the Jade template. If a stooge variable was passed into the template it will display that stooge's name, otherwise it will display a generic "not found" message.

Example 2-5. The new stooges page in Jade style (stooges.jade)

```
extends layout

block content
  if (stooge)
    p #{stooge} is my favorite stooge.
  else
    p no stooges listed
```

Events

Events are the lifeblood of Node.js and indeed of JavaScript itself. While other languages handle workflows in multiple concurrent threads, with each thread spending most of its time waiting for blocking I/O operations like reading from a disk, manipulating a database, or fetching information over the network, JavaScript was always conceived of as an event-based programming model. In the early days, an event was as simple as a mouse click, a page load, or a form submission. More advanced usages include events such as the completion of a database write or the contents of a file after it has been read from a disk.

JavaScript uses callbacks to approach the problem from the opposite side; instead of managing long-running processes, programmers hook into specific events and write special functions called callbacks, which are executed when the event criteria is reached. The routing examples in Example 2-1 and Example 2-2 demonstrate this: Node.js sets a particular response to each given URL and then executes the code only when a web browser accesses those routes.

Node.js includes a specialized `events` library you can use to provide your own custom events. Imagine you have a web application that you have made available to other developers. If they want to extend your work, they have two choices: create function prototypes that derive your code and reimplement any function they want to behave differently, or modify your code directly to build the features they need. Events give a third option: at specific points in your code you can emit an event that will let any observers know that an action has taken place and inject their own behavior.

For example, if your application included a login feature, you might have an `OnLoggedIn` event. A subsequent developer can add a listener for your event to provide extra features, such as connecting to social networking sites to gather any news related to the logged in user.

Example 2-6. Creating and handling application events

```
var events = require('events');
var eventEmitter = new events.EventEmitter();

function mainLoop() {
  console.log('Starting application');
```

```
  eventEmitter.emit('ApplicationStart');

  console.log('Running applicatin');
  eventEmitter.emit('ApplicationRun');

  console.log('Stopping application');
  eventEmitter.emit('ApplicationStop');
}

function onApplicationStart() {
  console.log('Handling Application Start Event');
}

function onApplicationRun() {
  console.log('Handling Application Run Event');
}

function onApplicationStop() {
  console.log('Handling Application Stop Event');
}

eventEmitter.on('ApplicationStart', onApplicationStart);
eventEmitter.on('ApplicationRun', onApplicationRun);
eventEmitter.on('ApplicationStop', onApplicationStop);

mainLoop();
```

Example 2-6 demonstrates how three unrelated functions `onApplicationStart`, `onApplicationRun`, and `onApplicationStop` can be strung together to produce this output:

```
Starting application
Handling Application Start Event
Running applicatin
Handling Application Run Event
Stopping application
Handling Application Stop Event
```

The `ApplicationStart`, `ApplicationRun`, and `ApplicationStop` events are registered using the `eventEmitter`'s `on` method before the `mainLoop` function is executed. This adds an event listener for each of these events—whenever any event is raised from now on, it will be checked against these listeners to determine if a match is available, in which case the callback function from that match is executed.

The screen output highlights an important trait of Node.js: all of its work is done on a single thread. When an event is raised and answered by a callback, the calling method is paused while the callback executes. This is important because if something happens during the callback and consumes a lot of processing time, the original function will not continue running until all of the work is completed. So the execution in this example follows the path:

1. Run `mainLoop`, trigger `ApplicationStartEvent`.
2. Run `onApplicationStart` callback.
3. Continue `mainLoop` execution, trigger `ApplicationRun`.
4. Run `onApplicationRun` callback.
5. Continue `mainLoop` execution, trigger `ApplicationStop`.
6. Run `onApplicationStop` callback.
7. Return to `mainLoop` execution, there's nothing left to do; stop.

Socket.io

Socket.io is your friend—it will take the drudgery out of making real-time web applications by dealing with all of the cross-browser compatibility issues and leaving you with a clean, simple JavaScript interface shared between your backend Node server and frontend JavaScript client. This is an exciting library because it lets you as a programmer focus on your program code in a single scripting language with no network barriers between your data and end user.

To install Socket.io, use npm:

```
npm install socket.io
```

Example 2-7 adds real-time chat capability to the stooges website by creating a Socket.io object and attaching it to the `http.Server` in front of Express. Upon receiving a socket connection from the web browser, Socket.io triggers a callback function in the application, which dispatches an arbitrary welcome message to the connecting user. The `sendChat` function was created for convenience; given a title and content, it uses Socket.io's `emit` command to send a JSON payload to the connected socket. Because it lives inside the callback function, it is available to any of the socket-level events but is invisible to the rest of the application.

Example 2-7. Adding chat capability to the Express server

```
var express = require('express');
var http = require('http');
var app = express();
var server = http.createServer(app);
var io = require('socket.io').listen(server);
var catchPhrases = ['Why I oughta...',
  'Nyuk Nyuk Nyuk!', 'Poifect!', 'Spread out!',
  'Say a few syllables!', 'Soitenly!'];

app.set('view engine', 'jade');
app.set('view options', { layout: true });
app.set('views', __dirname + '/views');
```

```
app.get('/stooges/chat', function(req, res, next) {
  res.render('chat');
});

io.sockets.on('connection', function(socket) {
  var sendChat = function( title, text ) {
    socket.emit('chat', {
      title: title,
      contents: text
    });
  };
  setInterval(function() {
    var randomIndex = Math.floor(Math.random()*catchPhrases.length)
    sendChat('Stooge', catchPhrases[randomIndex]);
  }, 5000);
  sendChat('Welcome to Stooge Chat', 'The Stooges are on the line');
  socket.on('chat', function(data){
    sendChat('You', data.text);
  });
});

app.get('/?', function(req, res){
  res.render('index');
});

var port = 8080;
server.listen(port);
console.log('Listening on port ' + port);
```

The `socket.on('chat')` line creates an event callback that executes whenever the connected user sends a message over the socket. There isn't a lot of functionality at this point; the server will respond to these events by echoing the users' message back to them.

To add a bit of life to this example I've included a reference to JavaScript's `setInterval` function with a callback who sends a random *Three Stooges* catch phrase to the connected client every 5 seconds.

Express server is instantiated differently in Example 2-7 than it was in earlier examples. Instead of having the Express object listen directly for incoming connections, it is first attached to an `http.Server` using `http.createServer(app)` and the resulting server object listens for incoming connections. Under the covers, Express's `listen` command does the same thing without making the `http.Server` available; you need to expose the `http.Server` in this manner in order to connect Socket.io to it.

The chat layout in Example 2-8 introduces content in the scripts block, which places Socket.io's functionality up above your HTML content when rendered in the web browser. Socket.io makes certain files available for download, which is why you are able

to include a script reference to `/socket.io/socket.io.js`. The JavaScript file *socket.io.js* includes all of the functionality the web browser will need to connect to your socket server, including fallback mechanisms that will provide socket-like functionality on web browsers that are behind the times and don't yet have web socket support.

Example 2-8. The chat page's Jade template (chat.jade)

```
extends layout

block scripts
  script(type='text/javascript', src='/socket.io/socket.io.js')
  script(type='text/javascript')
    var socket = io.connect('http://localhost:8080');
    socket.on('chat', function(data) {
      document.getElementById('chat').innerHTML =
        '<p><b>' + data.title + '</b>: ' + data.contents + '</p>';
    });
    var submitChat = function(form) {
      socket.emit('chat', {text: form.chat.value});
      return false;
    };

block content
  div#chat

  form(onsubmit='return submitChat(this);')
    input#chat(name='chat', type='text')
    input(type='submit', value='Send Chat')
```

Whenever the submit button is activated, Socket.io will emit the contents of the chat textbox. Upon receiving chat events from the web server, Socket.io will replace the contents of the chat display window (`div#chat`) with the incoming message. It's important to note that the socket events and functions used on the client are the exact same as those used on the web server. This allows you to provide a clean communications contact across the entire application without worrying about converting data for transmission or reception.

Modules and CommonJS

Node has sparked huge interest in server-side JavaScript programming, not to mention JavaScript in general. But it is not the first tool that has made JavaScript available outside of the browser; in fact, Netscape released a web server that incorporated the language shortly after it made its debut in their web browser software. Less than a year later Microsoft's Internet Information Services (IIS) server software also supported server-side JavaScript (calling their dialect JScript). Around the same time, Netscape planned to rewrite their flagship web browser using the Java programming language—a project that ultimately spun off Mozilla's Rhino JavaScript engine.

If this sounds like a quickly-fragmenting market, imagine trying to write JavaScript code for one of the incumbent server products and then using that same code elsewhere. If your application was small enough, you could probably get by with some modifications. Any decently-sized application quickly finds itself needing to call in external libraries and modularize its components, otherwise you end up with an unmaintainable mess with thousands of lines of script in a single file. Each of the servers had ways of breaking apart application code, but there was no standard—once you picked one, you were locked in.

The situation isn't a whole lot better today although major strides have been made toward standards for things like code development, namespace protection, object creation, and modules. CommonJS is a movement that aims to provide a standard set of specifications for JavaScript outside of the web browser, many of which have been adopted by Node. If you're just starting out developing with Node and your application has grown beyond what can reasonably exist in a single file, what you need to know is variables declared in external files are not available to your application unless you explicitly make them visible using the `exports` keyword.

Example 2-9 demonstrates a simple Node.js module that exposes a function `getFlagWidth` used for calculating the regulation fly (width) of an American flag. The fly should be 1.9 times the length of the hoist (height) of the flag; this ratio is stored in the variable `FLAG_WIDTH`. The function `getFlagWidth` takes the hoist height and multiplies it by the width ratio, yielding the width of the flag appropriate for any given height.

Example 2-9. Module for calculating the wdth of government flags

```
var FLAG_WIDTH = 1.9;

exports.getFlagWidth = function(h) {
  return h * FLAG_WIDTH;
};
```

When you include this file in your application, you will be able to access the `getFlagWidth` function, which is exposed using the exports keyword, but not the `FLAG_WIDTH` variable. `FLAG_WIDTH` can be considered a "private" variable accessible only within the context of the module, and not outside in the greater application.

CHAPTER 3

Backbone.js

Backbone.js is a Model-View-Controller (MVC) framework for client-facing JavaScript. Anyone who has spent time working with JavaScript projects larger than trivial in size has seen how quickly the language spirals into a web of callbacks and pyramid code. When writing code for the web browser, it is almost inevitable to find display-specific code leaking its way into your application logic. Over time, the code mix becomes heavier and harder to maintain. Changes to the domain logic affect the view and vice versa.

Backbone aims to solve the code coupling problem by providing a model-view framework with templates that separate programming concerns in a way that should feel familiar to developers coming from either a desktop application or server side programming background.

It isn't possible to talk about Backbone without also discussing Underscore.js, Backbone's prerequisite helper library. Underscore provides functional programming support in the form of utility functions like map/reduce, array iteration and filtering, and advanced object binding and chaining. jQuery or Zepto, although not strictly required, are supported by Backbone. jQuery in particular will play a role in the application developed over the course of this book.

Model

Models form the nucleus of your Backbone application. Although models may be transient app-only creations, in most cases the model will represent an object stored in a database.

Backbone's philosophy has models responsible for storing, retrieving, and transforming data. Some frameworks have distorted this intention of MVC and made the controller

responsible for data transformation. In reality the controller's only business is in interpreting requests made by the user through the view and accessing the correct parts of the model. The model itself needs to understand how to handle the data it receives, and how to fetch itself from the data store.

Example 3-1 illustrates how a model type is declared and later initialized using Backbone. The `extend` function sets up a prototype chain for the `Stooge` class, so you can access all of the properties of `Model` whenever you work with a `Stooge`; all of the subclass-specific functionality is declared inside `extend`'s properties object. This is an important concept in Backbone—although other libraries include their own `extend` methods, which typically copy content from one class to another, the one used in Backbone also creates a constructor function so you can instantiate your class, and copies itself into the new class so you can extend many levels.

Example 3-1. Initializing a Backbone model

```
Stooge = Backbone.Model.extend({
    defaults: {
        'name': 'Guy Incognito',
        'power': 'Classified',

        'friends': [],
    },

    initialize: function() {
        // Do initialization
    }
});

var account = new Stooge({ name: 'Larry', power: 'Baldness',
                           friends: ['Curly', 'Moe']});
```

View

Views represent a display of data within a model, usually providing different information depending upon the context needed. For example, in Canada citizens have the option of ordering either a short form or long form version of their birth certificate. Both documents (views) represent the same vital information (model) but with different levels of detail. Views in Backbone provide a window into a model's data and give you as a developer the ability to listen for user interaction or changes in the underlying model to trigger an update to what is displayed in the web browser.

Example 3-2 contains a web page that can be run directly in a web browser. Before any coding can begin, the jQuery, Underscore, and Backbone libraries are included within the page's `head`; when you run this page the web browser will stop here and wait until all of the JavaScript has been downloaded and loaded into memory before continuing.

Example 3-2. Initializing a Backbone view

```
<html>
<head>
<script type="text/javascript"
  src="http://cdnjs.cloudflare.com/ajax/libs/jquery/1.7.2/jquery.min.js"></script>
<script type="text/javascript"
  src="http://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.3.3/underscore-min.js">
</script>
<script type="text/javascript"
  src="http://cdnjs.cloudflare.com/ajax/libs/backbone.js/0.9.2/backbone-min.js">
</script>
</head>

<body>
<div id="certificate"></div>

<script type="text/javascript">
  CertificateView = Backbone.View.extend({
    initialize: function() {
      this.render();
    },

    render: function() {
      $(this.el).html("<h1>Guy Incognito</h1><p>DOB: March 2, 1967</p>");
    }
  });

  var certificate_view = new CertificateView({ el: $("#certificate")});
</script>
</body>
</html>
```

You should try to avoid placing your script tags in the head of a web page intended for use by the public. The web browser will pause to download these scripts before continuing on to render the page, so you should only include scripts in the head if they contain functionality that is required by the rendering process.

A `CertificateView` class, which is a placeholder for a birth certificate, uses Backbone's `extend` method to create a prototype chain for this custom view the same way Backbone's Model prototype was extended in Example 3-1. The view will have two custom functions:

initialize
: Executed when a new instance of the view is created; in this case, the desire is for the view to render its contents immediately

render
: Draws the contents of the view to the target element as described next

Finally, every view in backbone is associated with an element in the HTML DOM tree. In Example 3-2, you specifically instruct the view to render itself inside the `div` whose ID is `certificate`, but in the absence of an explicitly declared element, Backbone will use a generic `div`. So when we reach the `new CertificateView({ ... })` instruction, the view will first initialize and then render its contents into the `div` with ID `certificate`.

View Template

Templating is a critically important part of working with Backbone, so much so that it is better to discuss it early on than introduce it later because templates will fill a major role throughout this book.

Example 3-2 carries a potent antipattern: it is an HTML script containing JavaScript code whose role is to generate objectified HTML.

In Example 3-3, the HTML needed to generate the page has been moved out of the view's `render` function and placed into a script tag by itself. The template's script is given a type of `text/template` in order to prevent the web browser from trying to render the code to the screen as JavaScript, which would fail because the template is not executable JavaScript.

Example 3-3. Rendering a template in a Backbone view

```
<html>
<head>
<script type="text/javascript"
  src="http://cdnjs.cloudflare.com/ajax/libs/jquery/1.7.2/jquery.min.js"></script>
<script type="text/javascript"
  src="http://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.3.3/underscore-min.js">
</script>
<script type="text/javascript"
  src="http://cdnjs.cloudflare.com/ajax/libs/backbone.js/0.9.2/backbone-min.js">
</script>
</head>

<body>
<div id="certificate"></div>

<script type="text/template" id="tpl-certificate">
  <h1><%= name %></h1>

  <p>DOB: <%= dob %></p>
</script>

<script type="text/javascript">
  CertificateView = Backbone.View.extend({
```

```
    template:_.template($('#tpl-certificate').html()),

    initialize: function() {
      this.render();
    },

    render: function() {
      var templateArgs = {
        name: "Guy Incognito",
        dob: "March 2, 1967"
      };
      $(this.el).html(this.template(templateArgs));
    }
  });

  var certificate_view = new CertificateView({ el: $("#certificate")});
</script>
</body>
</html>
```

I made the decision to treat the certificate owner's name and date of birth as variable inputs, so when it comes time to display them in the HTML they get specified as the variable names `name` and `dob`, respectively. Placing the variables inside the script tags `<%` and `%>` causes the JavaScript engine to act upon their contents; rather than printing out "name" and "dob", the values they contain will be acted upon instead. Placing an equality operator (=) next to the opening tag acts as shorthand to display the contents of the variable on screen. So if `name` contained the value "Steve Smith", that is what would appear inside the `<h1>` tags.

Two steps are required to use the template inside the view: first, the view must know which template it will be using, and second, the template must be given a set of parameters to render. Using a bit of jQuery goodness, the contents of the template—remember, they're stored in the script tag with the ID of `tpl-certificate`—are extracted using the `.html()` function, which returns the raw HTML content inside a tag. Underscore's `.template()` function accepts the HTML contents and creates a compiled function that can be used later on.

In JavaScript, functions are considered "first-class citizens," meaning they can be assigned to variables just like numbers or strings, then reassigned or passed between functions. This is an important trait because it enables the functional-style programming that Underscore provides helpers for.

To render the template now, all you have to do is call the newly-created `template` function along with a list of expected variables, and the resulting HTML can be printed

to the screen just like the hard-coded HTML in Example 3-2. The data is still static at this point; the `templateArgs` object contains a dictionary of names and values corresponding to the '`name`' and '`dob`' variables expected in the template. In typical usage you would use a separate data model instead of a stricly-defined object as done here.

Collection

In Backbone, collections provide statefulness to your models by brokering reads and writes of data to and from the backend server, and notifying any listeners when models are read or changed. Collections are responsible for storing, retrieving, and updating families of models; because most applications use some type of list or menu, collections are perfect for storing this kind of data.

Why would you want to take on the overhead of a collection rather than using a more basic and straightforward JavaScript array? For one, writing objects that extend collections helps document your code; in Example 3-4, the `Team` collection defines `Stooge` as its model type, giving you the context that Backbone will be looking for models of type `Stooge` when it iterates or changes an instance of `Team`. Of course, the more compelling reason to use collections is because they expose a suite of powerful commands that enable you to manipulate their contents and even retrieval methods much more easily than performing the same tasks in your own custom code.

Example 3-4. Building a collection in Backbone.js

```
var Stooge = Backbone.Model.extend({
  defaults: {
    'name': '',
    'power': ''
  }
});

var Team = BackBone.Collection.extend({
  model: Stooge
});

var larry = new Stooge({ name: 'Larry', power: 'Baldness' });
var moe = new Stooge({ name: 'Moe', power: 'All Powers' });
var curly = new Stooge({ name: 'Curly', power: 'Hair' });

var threeStooges = new Team([ larry, curly, moe ]);
```

The collection raises events when any of the models it contains are updated, which can be useful in launching display updates in other areas of your application.

Sync

The `Sync` class is used every time Backbone needs to read or save a model from the server. By default, this uses jQuery's `.ajax` method to send and receive JSON data, but depending on your needs you may want to override this to use a different storage mechanism.

Later in this book you will use the Backbone.ioBind project as a drop-in replacement for Backbone's default sync command. This will move all of your models' CRUD (Create, Read, Update, and Delete) operations to Socket.io, enabling full real-time socket operations over JavaScript without requiring new web connections.

Router and History

The router allows Backbone to respond to hash tag (#) changes by displaying a new resource. This provides deep linking capability to your application by creating links that can be bookmarked and shared by visitors. Combined with the history component, routes can hook into the web browser's history buttons and support back and forward navigation.

It may seem odd that Backbone lacks a base `Controller` type. In fact, what we now know as the router originated as a controller, with each route declared when the `Controller` object was extended. That setup is problematic because it tightly couples the concept of a controller, which is intended to control a model based upon input from a view, with routing, which is intended to navigate the end user between views. By moving `Router` to its own prototype, the makers of Backbone have elegantly separated the interface mechanics from the manipulation and display of models.

So why no dedicated controller prototype? A controller is the purest implementation of your application's logic. Because all of the application's maintenance details have been moved away from the controller, you are free to write a controller in plain JavaScript that deals exclusively with the input and output from your model. At this level there is ideally very little repeatable logic in your code that would warrant the need for a base controller type.

This example shows how to wire up a router that listens for certificate requests such as `#/certificates/123` and `#/certificates/mycertificatename`, then creates a view for the given certificate using the `CertificateView` object created in Example 3-5. In this simplified example, `CertificateView` doesn't actually accept any parameters, but if it did, `id` would contain `123` or `mycertificatename` if the sample URLs were used.

Example 3-5. Demonstration of Backbone routing

```
var MyRouter = Backbone.Router.extend({
  routes: {
    "/certificates/:id": "getCertificate",
  },
  getCertificate: function( id ) {
    new CertificateView({ el: $("#certificate")});
  }
});

var router = new MyRouter;

Backbone.history.start();
```

CHAPTER 4

MongoDB

When it comes to NoSQL databases, it is hard to beat the ease of use offered by MongoDB. Not only is it well documented and supported by a large and helpful community, but it is friendly to developers coming from an SQL background—many queries and a great deal of relational thinking can be directly applied from SQL to MongoDB—making it an especially attractive system for newcomers to the NoSQL world.

In relational databases, a single entity is stored in a row with a series of columns. Because entities are defined in a strict schema, every row will have the same columns. Working with entities involves comparing columns with very little overhead: all of the data is the same by design. In MongoDB there is no strictly defined schema and there are no rows containing columns—instead, every entity is stored in a document with any number of fields.

Documents provide a lot of power; you can store much more related information about each entity inside the document, even putting lists of documents *inside* other documents. Instead of making multiple queries to the database to get a complete set of information (as you would have to do with an SQL database), you can load entire datasets in a single operation.

Accessing Data

Not only is MongoDB friendly to developers coming from an SQL background, but its website goes out of its way to show how many SQL statements can be converted to MongoDB queries. In any database system, the end goal is always writing data—usually, persisting it to a disk—and reading it back out again.

Example 4-1 demonstrates a simple session in MongoDB. No special setup is required at any step of the way—all that was needed here was to install MongoDB and run it, then connect using the `mongo` client. Once connected to MongoDB, I immediately started using a database that I called `newdb`, but I didn't have to do anything to set it up: all I needed to do was to start writing data.

Example 4-1. Basic MongoDB usage

```
> use newdb;
switched to db newdb
> book = { author: 'Jamie Munro', title: '20 Recipes for Programming PhoneGap',
published: new Date('04/03/2012') };
{
  "author" : "Jamie Munro",
  "title" : "20 Recipes for Programming PhoneGap",
  "published" : ISODate("2012-04-03T07:00:00Z")
}
> db.books.insert(book);
> db.books.find();
{ "_id" : ObjectId("5063d1d89e302eaf24b259a0"), "author" : "Jamie Munro",
  "title" : "20 Recipes for Programming PhoneGap",
  "published" : ISODate("2012-04-03T07:00:00Z") }
```

I created a variable named `book` to store information about a programming book I've been reading, including the author, title, and publication date. Then I created a books collection and inserted it using the `db.books.insert command`. Along the way I didn't stop once to define a schema; at no point do I tell MongoDB what a book is, what data it should contain, or even what a collection of books is. MongoDB takes care of creating documents, maintaining lists, and even—as we will discuss later in this chapter—indexing and constraints.

Writing

As you've seen, writing data to MongoDB is extremely free form. You have a lot of flexibility because each record stored in the database is basically a JSON document and therefore parsable and usable in both a free and structured manner. You are not bound to a rigid set of columns per table as you would be in a traditional RDBMS. Building upon Example 4-1, you can add an additional book to the database without being bound to follow the structure that came before.

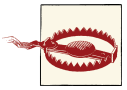
When I added a book in Example 4-2, I included a new field called `keywords` that was not present in the book added in Example 4-1. That doesn't matter because when I later queried the list of books, both books were returned even though they didn't have identical field names. MongoDB happily fetches "20 Recipes for Programming PhoneGap" along with "50 Tips and Tricks for MongoDB Developers" even though they aren't structured exactly the same.

Example 4-2. Inserting a document

```
> book = { title: '50 Tips and Tricks for MongoDB Developers',
author: 'Kristina Chodorow',
published: new Date('05/06/2011'),
keywords: ['design', 'implementation',
'optimization'] };
{
  "title" : "50 Tips and Tricks for MongoDB Developers",
  "author" : "Kristina Chodorow",
  "published" : ISODate("2011-05-06T07:00:00Z"),
  "keywords" : [
    "design",
    "implementation",
    "optimization"
  ]
}
> db.books.insert(book);
> db.books.find();
{ "_id" : ObjectId("5063d1d89e302eaf24b259a0"), "author" : "Jamie Munro",
  "title" : "20 Recipes for Programming PhoneGap",
  "published" : ISODate("2012-04-03T07:00:00Z") }
{ "_id" : ObjectId("5063d6909e302eaf24b259a1"),
  "title" : "50 Tips and Tricks for MongoDB Developers",
  "author" : "Kristina Chodorow",
  "published" : ISODate("2011-05-06T07:00:00Z"),
  "keywords" : [ "design", "implementation", "optimization" ] }
```

As you can well imagine, there is a lot of power in being able to insert records in such a free form manner. When you are building an application against MongoDB, your program code is in control of the structure of the data you will be using. Any time you need to add a new field, record type, or even database, you will be able to do so by doing nothing more than declaring it and using it. But this flexibility and power comes with a management underside: your application will need to be able to handle old data formats as well as new ones after your application has grown for a period of time. That means you must either be very guarded about making changes at all or your application must be crafted to be resilient to data changes.

The simple scenario demonstrated in Example 4-2 is a perfect example of this. You've just added a `keywords` field to your book documents—every new book entered into the system from here on out will contain a keywords field that is exposed to a library terminal somewhere down the line. What happens when that terminal tries to read a book that is missing the `keywords` field? Hopefully the developer who built the interface thought of that and is able to display an empty list—or a special message—when no keywords are found.



You should always build your application logic to check for the presence of database fields before using them, otherwise you could end up with a broken application even though your database is behaving exactly as expected.

If you want to make sure all your documents have a field called `keywords`, you could trigger an update across the entire collection, as shown in Example 4-3.

Example 4-3. Adding a feld to all documents in a collection

```
> db.books.update({},{$set:{"keywords":[]}},false,true);
```

Example 4-3 demonstrates the use of MongoDB's update command with all four of its parameters:

Search criteria
: This parameter contains all of the search criteria that MongoDB should use to determine which records need to be modified. In this case, no criteria is given, meaning any record is a fair match for this function.

Update object
: During normal operation, this parameter will contain an entire record just like the `insert` command in Examples 4-1 and 4-2. When presented with an object, MongoDB will save its contents over any document it found matching the search criteria from the first parameter. MongoDB also supports a number of special functions including the `$set` function shown here, which allows manipulation of part of a document, leaving the rest of the data intact.

Upsert
: In most cases you would want to update an existing document using the `update` command, but there are many times when you will want to update a piece of information or create a new document if that information does not already exist in the database. The upsert pattern means "update if possible, insert otherwise." In Example 4-3, the goal is to set a keyword field for all of the documents but not create any new records; therefore, upsert is set to `false`.

Multiple update
: MongoDB expects you to update one record at a time, so when you need to update more than one you need to set this variable to `true`; otherwise, the database will stop updating after it operates on the first match. This is a useful safety valve to prevent you from accidentally trashing an entire collection because of a poorly thought out wildcard search pattern.

Example 4-4 demonstrates another useful operator: the `$push` command. This command allows you to add a new item to the end of an array without modifying anything else in your document. As shown here, the keyword `developer` is added to the "50 Tips and Tricks for MongoDB Developers" book.

Example 4-4. Updating part of a document

```
> db.books.update( { author: "Kristina Chodorow" },
{ "$push": { "keywords": "developer" } } );
> db.books.find();
{ "_id" : ObjectId("5063d1d89e302eaf24b259a0"), "author" : "Jamie Munro",
  "title" : "20 Recipes for Programming PhoneGap",
"published" : ISODate("2012-04-03T07:00:00Z") }
{ "_id" : ObjectId("5063d6909e302eaf24b259a1"), "author" : "Kristina Chodorow",
  "keywords" : [ "design", "implementation", "optimization", "developer" ],
  "published" : ISODate("2011-05-06T07:00:00Z"),
  "title" : "50 Tips and Tricks for MongoDB Developers" }
```

Querying

Querying in MongoDB is analogous to `SELECT` in SQL. You can not only query across fields in collections of documents, you can also use custom JavaScript functions to perform more complicated filtration on your result sets.

The earliest example in this chapter, Example 4-1, contains an extremely basic query:

```
db.books.find();
```

The `find` command with no parameters instructs MongoDB to find documents in the books collection without applying conditions to the search. When there are no conditions to apply to the search, MongoDB responds by returning all of the documents in the collection. In this sense, the `find` command with no parameters is the same as saying "find all" to the database.

If we were to express this in SQL, the query would look like this:

```
SELECT * FROM books;
```

Example 4-5. A simple field search in MongoDB

```
> db.books.find({author: "Jamie Munro"});
{ "_id" : ObjectId("5063d1d89e302eaf24b259a0"),
  "author" : "Jamie Munro", "keywords" : [ ],
  "published" : ISODate("2012-04-03T07:00:00Z"),
  "title" : "20 Recipes for Programming PhoneGap" }
```

Example 4-5 uses the `find` command again, but this time a specific author is specified in the criteria. This time MongoDB will search through the books collection and return all of the records whose author field matches the author field in the `find` command. If we were to express this in SQL, the query would look like this:

```
SELECT * FROM books WHERE author = 'Jamie Munro';
```

Imagine for a moment that the average document in your collection contained dozens, or even hundreds, of rows. During regular use you would not always want to get every field from the database, especially if you're interested in only one or two bits of information at a time.

The `find` command in Example 4-6 has two parameters: the search criteria (empty in this case) and a desired field map. Because no arguments are supplied in the search criteria, MongoDB will once again return all of the documents in the books collection. The desired field map includes the `title` field, and will cause MongoDB to return only the title field.

Example 4-6. Finding specific fields from a collection

```
> db.books.find({}, {title:1});
{ "_id" : ObjectId("5063d6909e302eaf24b259a1"),
  "title" : "50 Tips and Tricks for MongoDB Developers" }
{ "_id" : ObjectId("5063d1d89e302eaf24b259a0"),
  "title" : "20 Recipes for Programming PhoneGap" }
```

Wait a minute! Why is the `_id` field being returned? MongoDB assumes you will need the record's ID field in most cases, else you would not be able to uniquely name a document from your application code. If you wanted to show only the titles without the `_id` field, you could explicitly hide the `_id` field like this, using `0` to mean `false`:

```
db.books.find({}, {title:1, _id:0});
```

The `find` function as shown in Example 4-6 is analogous to this SQL query:

```
SELECT _id, title FROM books;
```

Indexes

It's easy to be fooled into thinking your code is fast when you're working on small datasets and querying against databases running on your own computer. Performance can suffer tremendously once your code hits a production workload and needs to serve a growing dataset to a large number of users. Although MongoDB is smart about where it looks for data, unless you set up indexes to keep search fields in memory, your database will be doing a lot more work than it needs to.

While going deeply into detail on the `explain` command would (and does!) fill an entire book, the first piece of information you should be looking for is in the `cursor` field. MongoDB uses either a BasicCursor or BtreeCursor when scanning collections of documents; for a heavily-used query, you want to avoid using a BasicCursor because it scans through every document in the collection to find a result.

The books collection queries in Example 4-7 is tiny, having only two records. But because a BasicCursor is used to perform the search, MongoDB has to examine both records before it can return a result set. You can see the number of scanned objects in the `nscannedObjects` field from the `explain` function's output.

Example 4-7. Diagnosing slow queries

```
> db.books.find({author: "Jamie Munro"}).explain();
{
  "cursor" : "BasicCursor",
  "nscanned" : 2,
  "nscannedObjects" : 2,
  "n" : 1,
  "millis" : 0,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "isMultiKey" : false,
  "indexOnly" : false,
  "indexBounds" : {

  }
}
```

You use the `ensureIndex` function to add an index to your collection. Indexed fields are tracked in memory and can be retrieved much more rapidly by MongoDB. More importantly, they act as a filter on queried data; the database only needs to look at records that it knows match your search criteria based upon its knowledge of the table as held in the indexes.

Notice how `nscannedObjects` dropped to just `1` in Example 4-8 after an index was added to the author field of the books collection. Because the author is now stored in memory and known to the database, MongoDB knows it only needs to look more closely at a single document. If you try to search for an author who doesn't exist, MongoDB will not have to check any records at all.

Example 4-8. Adding an index to a collection

```
> db.books.ensureIndex({author:1});
> db.books.find({author: "Jamie Munro"}).explain();
{
  "cursor" : "BtreeCursor author_1",
  "nscanned" : 1,
  "nscannedObjects" : 1,
  "n" : 1,
  "millis" : 0,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "isMultiKey" : false,
  "indexOnly" : false,
  "indexBounds" : {
  "author" : [
```

```
    [
      "Jamie Munro",
      "Jamie Munro"
    ]
  ]
  }
}
```

It can take some time to make sure you have all of the right indexes set up in your database. If done properly, it can mean the difference between queries that take minutes versus queries that take seconds or less.

MapReduce

MapReduce is used to batch process huge amounts of data often across clusters of database servers. If you were to compare MongoDB to a traditional SQL-based server, MapReduce would fit in the space where you would normally use `GROUP BY` to collect aggregate results.

A MapReduce operation involves two phases: the map phase, which plucks out the relevant data into key/value pairs for aggregation, and the reduce phase, which collects all of the keys and performs math on their values. Take a counting operation for example: if you wanted to know how many books each author in the books collection has written, you would need to go through each document and count the number of times each author appeared in the collection.

Since there have been only two records in the books collection so far, the first thing that needs to happen in Example 4-9 is the creation of more data, so that's what happens.

Example 4-9. Performing a MapReduce query on the books collection

```
> db.books.insert({"author": "Kristina Chodorow", "title": "Scaling MongoDB",
... "published": new Date("03/02/2011")});
> db.books.insert({"author": "Stoyan Stefanov", "title": "JavaScript Patterns",
... "published": new Date("09/28/2010")});
> db.books.insert({"author": "Stoyan Stefanov",
... "title": "JavaScript for PHP Developers",
... "published": new Date("10/22/2012")});
> db.books.insert({"author": "Stoyan Stefanov", "title": "Web Performance Daybook",
... "published": new Date("06/27/2012")});
> db.books.insert({"author": "Jamie Munro",
... "title": "20 Recipes for Programming MVC 3",
... "published": new Date("10/11/2011")});


> map = function() { emit( this.author, 1 ); };
function () {
    emit(this.author, 1);
}
> reduce = function( key, values ) {
```

```
...   var total = 0;
...   values.forEach(function(value) {
...     total+=value;
...   });
...   return total;
... }
function (key, values) {
    var total = 0;
    values.forEach(function (value) {total += value;});
    return total;
}
> db.books.mapReduce(map,reduce, { out: "bookoutput"});
{
  "result" : "bookoutput",
  "timeMillis" : 3,
  "counts" : {
    "input" : 7,
    "emit" : 7,
    "reduce" : 3,
    "output" : 3
  },
  "ok" : 1,
}
> db.bookoutput.find();
{ "_id" : "Jamie Munro", "value" : 2 }
{ "_id" : "Kristina Chodorow", "value" : 2 }
{ "_id" : "Stoyan Stefanov", "value" : 3 }
```

With some data in the collection, create a `map` function that will emit an author name and the number 1 when given a document. `emit` is a MongoDB helper function that groups objects by keys; in this case, the key is the author name found in each document. I used the value 1 for convenience: each time MongoDB reads a book object, it will count as 1 book credit toward that author. This could be simplified to emit an empty value, but it's being left this way for convenience because most of the MapReduce functions you create will start with this format and become more complex.

Once all of the keys have been emitted, MongoDB collects the results and reduces them; so while any particular key may have been found multiple times if any author wrote more than one book, the final result should contain only one value—the sum of written books—per author. The `reduce` function in Example 4-9 adds all of the values found for each author and returns a total count of each books.

When the `mapReduce` function is performed on the books collection, it is given three parameters: the user-defined map function, the user-defined reduce function, and the name of a new collection to contain the results. After the `mapReduce` function executes, it will save the author book counts into a new collection called `bookoutput`, which can then be queried just like any other collection. Example 4-9 concludes by querying it to reveal the number of books next to each author's name.

Working with Node.js

The primary MongoDB driver supported for Node.js is the Node MongoDB Native Project, a pure-JavaScript driver that provides asynchronous I/O to MongoDB from Node. Because the driver can save your JavaScript objects directly into MongoDB, by all rights you could build out the full application with it.

The Mongoose project extends the native drivers by providing a means to define the database schema. If this seems to go against the NoSQL "schema-less" philosophy, don't worry. Changes to the JavaScript schema definitions do not require special processing by MongoDB—the schema only exists to make your life easier as a developer trying to make a consistent application.

Mongoose also provides a powerful set of middleware designed to ease the process of working with serial and parallel requests in Node's asynchronous environment:

```
npm install mongoose
```

Concurrent Access

Picture this: Adam and Greg access the same document and begin making changes. Since they are each working on their own computers, their changes are not being saved directly back to the database and refreshed in each other's work—they can be said to be editing in "offline" mode. Adam finishes his work first and saves his complete changes back to the database. At some later point, Greg finishes his own edits and saves those into the database. Because Greg started working on the document before Adam's changes went into effect, his edits do not include Adam's; so when he saves his work back into the database, Adam's work is effectively erased, as demonstrated in Figure 4-1.

What to do? While most of the examples in this book involve write-only transactions (meaning we will write but never modify certain data), there will inevitably be occasions where you will need to work on a shared document at the same time as someone else and want to prevent your users from losing their changes if they were unfortunate enough to post first.

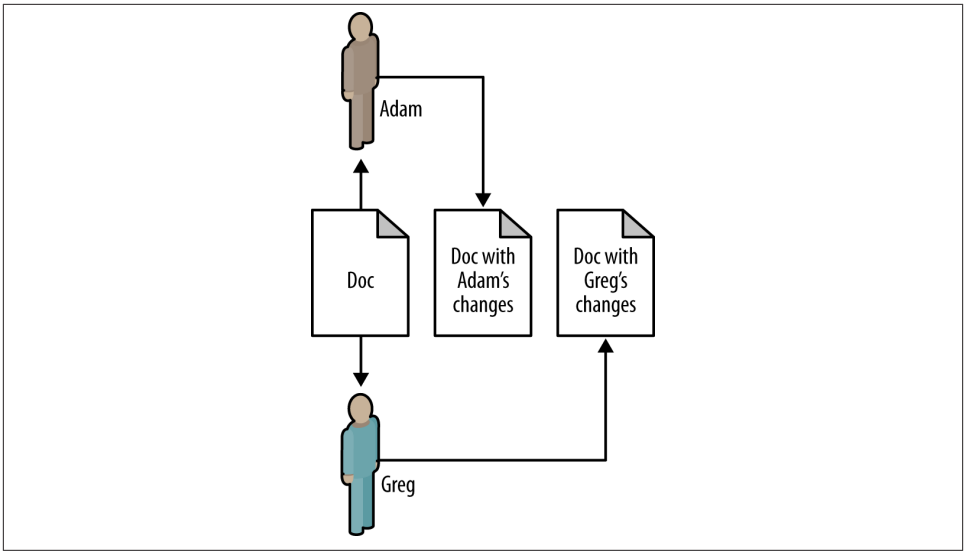


Figure 4-1. What happens when two users change the same document

One way to accomplish this is by assigning a signature to every record and updating that signature every time you write to the database. If updating an existing document, only update the document whose ID and signature both match the values observed when the document was first read. This way, when two users try to save the same data, the first user's update will cause the signature to change and the second user's update will fail because the signature does not match.

In Figure 4-2, users Greg and Adam both begin editing the same document, but when Adam saves his changes he updates the document's version number from a5 to b7. Now, when Greg saves his changes, he specifies that he is updating version a5, which no longer exists in the database; instead of writing his changes, the update fails. From here, Greg can update his copy of the document using the changes submitted by Adam, or the software he is using can do it intelligently in the background and resubmit on his behalf.

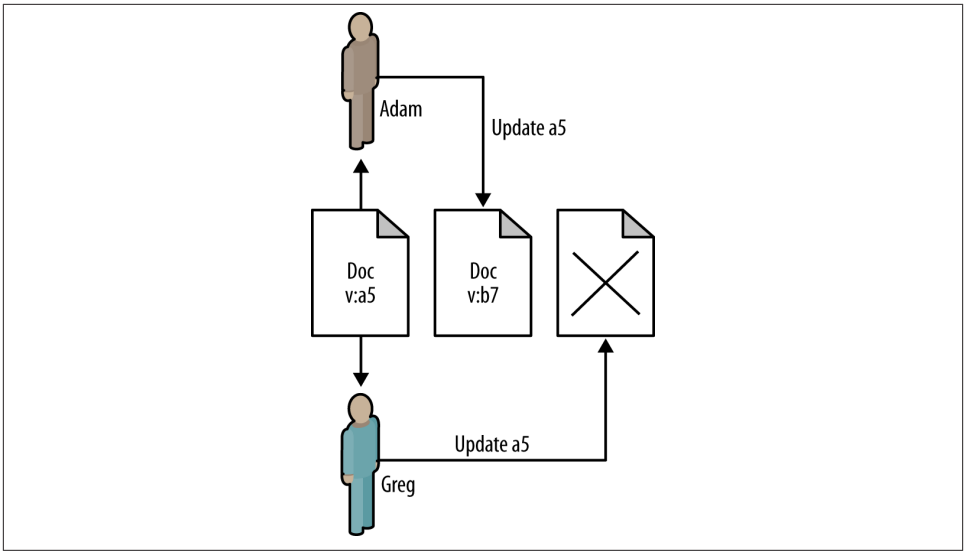


Figure 4-2. Updating documents using findAndModify

MongoDB includes a function called `findAndModify`, which handles searching and updating in a single function. This adds an extra layer of security because it ensures the update happens immediately when a record is found, rather than adding round-trip time to process the record on the client side and save it back to the database, during which time someone else might change the record and cause the concurrent access problem described earlier.

Example 4-10 demonstrates how an article can be created and updated using the `findAndModify` method. Notice how the first time `findAndModify` is executed, it returns the contents of the article without the updated revision number—that is, it indicates the revision is `a5` instead of the new value `b7` provided in the command. If you wanted to display the new, updated version of the article, you would include the keyword `new` in the `findAndModify` command. In either case, when the `find` command is later issued, the new revision value is shown. Next, when the `findAndModify` command is rerun, Mongo is unable to find the record because the revision version is no longer a5, therefore it returns `null`.

Example 4-10. Using findAndModify from the console

```
> db.articles.save( {
... title: "Jolly Roger",
... published: "September 12, 2007",
... description: "A riveting tale of suspense and drama.",
... revision: "a5"
... } );

> db.articles.find();
```

```
{
  "_id" : ObjectId("505a9b4fd5f42989fe6d8015"),
  "title" : "Jolly Roger",
  "published" : "September 12, 2007",
  "description" : "A riveting tale of suspense and drama.",
  "revision" : "a5"
}

> db.articles.findAndModify({
... query: {"_id": ObjectId("505a9b4fd5f42989fe6d8015"), revision: "a5"},
... update: {$set: {revision: "b7"} }
... });
{
  "_id" : ObjectId("505a9b4fd5f42989fe6d8015"),
  "title" : "Jolly Roger",
  "published" : "September 12, 2007",
  "description" : "A riveting tale of suspense and drama.",
  "revision" : "a5"
}

> db.articles.find();
{
  "_id" : ObjectId("505a9b4fd5f42989fe6d8015"),
  "title" : "Jolly Roger",
  "published" : "September 12, 2007",
  "description" :
  "A riveting tale of suspense and drama.",
  "revision" : "b7"
}

> db.articles.findAndModify({
... query: {"_id": ObjectId("505a9b4fd5f42989fe6d8015"), revision: "a5"},
... update: {$set: {revision: "90jasv"} }
... });
null

> db.articles.find();
{
  "_id" : ObjectId("505a9b4fd5f42989fe6d8015"),
  "title" : "Jolly Roger",
  "published" : "September 12, 2007",
  "description" : "A riveting tale of suspense and drama.",
  "revision" : "b7"
}

>
```

PART II

Building a Social Network

CHAPTER 5

Setting Up the Project

Over the course of this section the Node, Backbone, and MongoDB concepts from Part I will be combined to demonstrate how you can build a communication application in the style of a social network. This basic but functional site will enable its users to authenticate securely, manage a list of contacts (or "friends"), chat, and view updates in real time.

Before starting any project it is a good idea to take some time to understand what your goals are and decide how to organize your work. It is far easier to make changes at this early stage than later once many files have been added to the application.

Chapter 1 introduced the social network application that will be built over the course of this book. As you build out the functionality you will find yourself ending up with two complementing MVC systems: one for the frontend that appears in the web browser, and the other for the backend Node services running on the web server. The directory structure will be set up in a way that makes it easy for you to come back and understand (not to mention makes it simple for newcomers to come onboard), and facilitate code reuse.

Additionally, a package definition file will be created to document the dependencies needed by the application. All of the third-party libraries and supporting software will be controlled by the package file, and later on this will define the entry point for submodules you will add to the application. The package file can later be used to deploy your code to a web server or share with another developer without needing to bundle it with all its modules.

Directory Structure

The directory structure for this project will boil down to two broad categories:

1. Node.js program files that reside and operate on the server; hese will not be visible to anyone using the application
2. Backbone.js models, collections view templates, and controllers that are downloaded by the end user and run inside the web browser

Making a clear separation between the Node.js files and the Backbone.js may seem like a subtle distinction, but it is an enormously important decision that will affect performance at the end of the development cycle. It isn't possible to put enough emphasis on how profoundly developers can be affected by early design choices long after the initial programming has been done. As the application is built out throughout the coming chapters, decisions designed to improve maintainability and scalability of the application in production will frequently be addressed.

File Listing

With the directory structure's philosophy well in hand, it's time to create actual files that will contain the basic application framework. This is the bare minimum needed to provide a foundation upon which the entire application will be built:

app.js
: The entry point for the Node.js application; you will be able to execute the server by running this in the command line: `node app.js`

public/
: The root folder for all client-downloadable files pertaining to the Backbone.js portion of the application

public/js/
: The root folder for all of the JavaScript files that will be rendered in the web browser

public/js/SocialNet.js
: The main application class for the social network; this class handles the messaging between views, controllers, and models

public/js/boot.js
: The bootstrapper object instantiates the global configuration and establishes module dependencies. This is instantiated by RequireJS when the page is initially loaded.

public/js/libs
: This folder contains third-party libraries used by the application.

public/js/lib/backbone.js
: Backbone.js available here.

public/js/lib/jquery.js
: The jQuery library is available here.

public/js/lib/require.js
: The RequireJS library is available here.

public/js/lib/text.js
: The RequireJS text plug-in; you will use it to load textual content from the templates folder for the view rendering

public/js/views
: This folder contains the view objects used by the Backbone application.

public/js/views/index.js
: The default template shown to users when they arrive at the application; this view renders the contents of the file located at *public/templates/index.html*

public/styles/styles.css
: The stylesheet used to control how HTML elements are laid out in your user's web browser

public/templates
: This folder contains the HTML templates, which will be rendered by the views into web pages displayed in the browser.

public/templates/index.html
: The default template shown to users when they arrive at the application; the contents of this file are rendered by *public/js/views/index.js*

views/
: This folder contains the Jade templates, which are rendered by the Express server and sent to the client.

views/index.jade
: This file is displayed to the user when they load the ropot (/) of the website *http://localhost:8080/*. Its purpose is to trigger the browser into bootstrapping the application.

Package Definition

When building a Node application, you should always create a package file to provide details about the operating conditions and configuration expected by your code. Doing this helps prevent future changes to third-party modules from breaking your logic and can define the runtime environment in the case of multi-platform development.

The file in Example 5-1, saved to disk as *package.json*, is used to synchronize your application with its dependencies. This is important to lock your code to a specific version—in this case, Express version 3.0.0—or define a minimum version, such as Mongoose version 2.6.5 or later. Later in this book we will discuss strategies for working in teams, and when we do, we'll see how the *package.json* file helps your friends get up and running in a single command.

Example 5-1. The application's package file

```
{
    "name": "my-social-network",
    "version": "0.0.1",
    "private": true,
    "dependencies": {
      "express": "~3.0.0",
      "jade": ">= 0.0.1",
      "mongoose": ">= 2.6.5"
  }
}
```

Once the *package.json* file has been set up, use npm to install any needed dependencies:

```
npm install
```

When you run `npm install` without supplying a package name, npm attempts to parse the *package.json* file in the current directory. Since you have a package file, npm will determine which dependencies you have specified and will download the necessary versions.

Web Server

Many developers who come from a "traditional" server-based background are familiar with setting up web server software—whether it's Apache, nginx, or IIS—to act as a communication channel between the web browser and the backend code. Newer technologies such as Ruby on Rails, Play! Framework, and PHP 5.4 have mechanisms for booting a local "development" server so you can begin programming without being bogged down by implementation details. (Play! Framework's built-in Netty HTTP server is intended to be used in production as well as development. It's listed here as an example of a framework that is designed to get developers up and running in as little time as possible.)

Node is interesting because the program code you write for it is also the server implementation. You have a greater expectation that the application will perform and behave similar in production as in development because there aren't any additional libraries,

brokers, or daemons getting in the way (apart from proxy servers, which will be discussed later on). Because Node exposes the nuts and bolts of the network to the programmer, it is very straightforward to create a fairly feature-rich application in very few lines of code.

Example 5-2 creates a functional and capable application in a short amount of code. Despite its small size, this program handles routing for incoming HTTP requests, provides a view engine to render server-side views into browser-friendly HTML5 markup, and provides downloadable access to static file resources on the local filesystem. All of this functionality is provided by the connect middleware, which utilizes Node's base network libraries, all exposed by the Express routes defined in the program code.

Example 5-2. app.js, the web server entry point

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

app.configure(function(){
    app.set('view engine', 'jade');
    app.use(express.static(__dirname + '/public'));
});

app.get('/', function(req, res){
    res.render("index.jade", {layout:false});
});

app.listen(8080);
```

The first two lines initialize the Express library, first by `require`-ing the Express library, and then by calling `express()` to build a new instance of Express and assigning it to the variable `app`. From here on, we will use `app` as a shortcut to refer to our application.

The variable `express` was used as a function in Example 5-2. This works because functions are first-class objects in JavaScript, meaning you can treat variables as functions and call them with parameters anywhere in your code.

Express's `configure` command is useful for controlling settings that change between environments (for example, paths to static files, cache settings, and render optimizations). The expected behavior is for `configure` to read the contents of the `NODE_ENV` environment variable, which should be set to `production` when your application is deployed. When no parameter string is provided to `configure`, as is the case in Example 5-2, the configuration settings in the provided function are applied to all runtime environments. This can be combined with environment-specific settings as described during the later chapters of this book.

Inside the `configure` command, the view engine is set to `jade`. Behind the scenes, this command causes Express to attempt to load the Jade library so it is able to handle Jade-based views when rendering responses to browser requests.

After the view engine has been installed, `express static` is invoked with a path of `__dirname + '/public'`). `__dirname` is one of several global objects available to all modules within Node; it contains the name of the directory that the currently executing script resides in. Because *app.js* will always live at the root of the project structure along with *package.json*, `__dirname` is a safe choice as a base path. Later on, you may want to pass `__dirname` from this file's context to a downstream module whose program code may not be in the same directory.

The last line in the file, `app.listen(8080)`, causes the configured Express application to begin listening for HTTP requests on port 8080. Using a large port number such as 8080 is useful during development because the standard ports including port 80 (the default web server port) are restricted to super users on many Unix-based systems. While you very likely have those privileges when developing locally, it is generally a poor idea to constantly switch privileges between your regular user permissions and super user permissions. For one, if you were to make a mistake affecting the filesystem, your runaway code could do a lot more damage when it has root-level access to the hard drive in your computer.

Index Template

Because Backbone.js will be the presenter for the majority of the action that will happen in the web browser, the Node application will not play a major role in the visual arena for this project.

The first line in Example 5-3 defines the document type, which is needed by the web browser to understand how the rest of the document should be formatted. If this line is missing, the browser is forced to make a guess about how best to display content: this state is called Quirks Mode, and has consequences on the way your visual styles will be rendered. It is best practice to always include a doctype in all of your pages.

Example 5-3. index.jade: the default view

```
!!! 5
html(lang="en")
  head
    title Social Network
    script(data-main='js/boot', type='text/javascript',src='/js/libs/require.js')
  body
    div#content
```

The second line opens the `<html>` container for the document and defines its language as English. Defining a language in the HTML is not strictly necessary but it is a recommended practice because it helps search engines index your site and provides context for spell checkers and speech synthesizers. This line will be rendered in the web browser as `<html lang="en">`.

After the `html` tag is opened, notice the spacing before the `head` tag is entered. Jade uses indentation to control which tags are inside each other. Looking at this example you can see that there are two tags, `title` and `script`, residing within `head`. The next tag, `body`, is indented to the same level as `head`, which Jade understands to mean "close the `head` tag and open the `body` tag."

Notice the `data-main` property inside the `script` tag. This is not typical in `script` tags but serves an important purpose for the bootstrapping of your application. The HTML5 `data` attribute (you can use `data-*` to create any custom property) is used to define non-visual application data in the context of web page tags. Require.js uses `data-main` to trigger loading of the first dependencies; for your application, this will be the bootstrapper that will load the views, router, and models for presentation to the end user.

Bear with me if it seems a bit wasteful that Express is using the Jade-rendering engine to generate the HTML output for this template. This is the only web page in the project and there are no dynamic variables being passed from the application to the page; it does little more than bootstrap the Backbone JavaScript files. Later in this book the Jade templates will be expanded to provide in-browser support for the real-time event programming you will be adding.

Application JavaScript

While Express serves as the centerpiece of the server-side architecture, its client-side counterpart will be a centralized application JavaScript structure called SocialNet. SocialNet will handle bootstrapping the display templates for the application and will serve as the main point of access between the web browser and the rest of the application.

The bootstrapper in Example 5-4 is performing two jobs. First, it is defining the paths to all of the dependencies used by the application, and second, it is initializing and launching the user interface. Although Require.js will load dependencies from the same (or relative) path as the application files, I like to explicitly define all of the libraries used by the application in this central location. Doing this makes it easier to change paths later on (we'll discuss this in more detail when we talk about scaling out) and keeps an easy-to-track record of which external libraries I am using, for easier upgrades and refactoring later in the project's lifespan.

Example 5-4. boot.js: the SocialNet bootstrapper

```
require.config({
  paths: {
    jQuery: '/js/libs/jquery',
    Underscore: '/js/libs/underscore',
    Backbone: '/js/libs/backbone',
    text: '/js/libs/text',
    templates: '../templates'
  },

  shim: {
    'Backbone': ['Underscore', 'jQuery'],
    'SocialNet': ['Backbone']
  }
});

require(['SocialNet'], function(SocialNet) {
  SocialNet.initialize();
});
```

The `shim` section configures dependencies that use traditional browser globals rather than the module export style of JavaScript used by RequireJS. This section ensures that the required dependencies (jQuery and Underscore) are loaded before Backbone initializes, in order to prevent conflicts from parallel loading.

Because the application will contain a lot of user-facing views, it is impractical to embed HTML code into your JavaScript or even Express view pages. RequireJS's text plugin allows you to read text content into your application, provided they reside (due to browser security restrictions) on the same domain as your JavaScript files.

Application class

After RequireJS has loaded all of the dependencies, it calls SocialNet's `initialize` method. Since you're just setting up the project at this stage, the initialization will consist of rendering the view so the web page renders in the web browser.

Example 5-5 packs a lot of punch. First, let's strip out all of the actual code and review the structure of a RequireJS module, seen in Example 5-6.

Example 5-5. SocialNet.js: the application object

```
define(['views/index'], function(indexView) {
  var initialize = function() {
    indexView.render();
  }

  return {
    initialize: initialize
  };
});
```

Example 5-6. A minimal RequireJS module template

```
define([dependency1, dependency2, ...], function(dependency1, dependency2, ...) {
  // Internal program code

  return {
    // Expose externally accessible functions
  }
});
```

So you can see how the SocialNet module fits this pattern perfectly. The only dependency in this module is the index view, which is loaded by RequireJS and passed into the SocialNet module as the variable named `indexView`. The `initialize` function called by the bootstrapper is returned at the end of the module; the function itself would otherwise be accessible only inside the scope of the `define` function.

Index view object

The index view extends a plain Backbone view and renders text into the HTML element tagged with the `content` identifier. This will be wrapped up in the RequireJS `define` properties in order to expose the `view` class but not any of its internal content.

In Example 5-7, the index view is instantiated after *index.html* is loaded. The `text!` prefix instructs RequireJS to load the contents of *templates/index.html* as a string of text and make it available to the module as the variable called `indexTemplate`. Instead of returning a reference to the `indexView` or to a function, the module returns an instantiated objected; that's why you were able to immediately render the index view in the application (in Example 5-5) without having to use the `new` keyword.

Example 5-7. index.js: the JavaScript index view

```
define(['text!templates/index.html'], function(indexTemplate) {
  var indexView = Backbone.View.extend({
    el: $('#content'),

    render: function() {
      this.$el.html(indexTemplate);
    }
  });

  return new indexView;
});
```

CHAPTER 6

Authentication

Because this application will be fully multi-user, the first gateway to build involves registration and identity authentication. Before users can access any other functionality, they must first identify themselves and prove they have authority to perform certain functions.

In this chapter you will create an account model to represent a user who has registered with your system, with the email address being the primary means of accessing the system. The user will also be expected to supply a password, which will be verified against the account with the matching email.

With a working account model, the next task will be creating login and registration views to bring users into and grant them access to the system.

Account

The account model is the main point of contact between Node.js and the MongoDB database.

The account model in Example 6-1 includes database fields for an email address, password, name, photo, description, and biography. This is a CommonJS module, which exports the account and `register`, `forgotPassword`, `changePassword`, and `login` functions.

Example 6-1. The user account: models/Account.js

```
module.exports = function(config, mongoose, nodemailer) {
  var crypto = require('crypto');

  var AccountSchema = new mongoose.Schema({
    email:     { type: String, unique: true },
    password:  { type: String },
    name: {
```

```
    first:    { type: String },
    last:     { type: String }
  },
  birthday: {
    day:      { type: Number, min: 1, max: 31, required: false },
    month:    { type: Number, min: 1, max: 12, required: false },
    year:     { type: Number }
  },
  photoUrl:  { type: String },
  biography: { type: String }
});

var Account = mongoose.model('Account', AccountSchema);

var registerCallback = function(err) {
  if (err) {
    return console.log(err);
  };
  return console.log('Account was created');
};

var changePassword = function(accountId, newpassword) {
  var shaSum = crypto.createHash('sha256');
  shaSum.update(newpassword);
  var hashedPassword = shaSum.digest('hex');
  Account.update({_id:accountId}, {$set: {password:hashedPassword}},{upsert:false},
    function changePasswordCallback(err) {
      console.log('Change password done for account ' + accountId);
  });
};

var forgotPassword = function(email, resetPasswordUrl, callback) {
  var user = Account.findOne({email: email}, function findAccount(err, doc){
    if (err) {
      // Email address is not a valid user
      callback(false);
    } else {
      var smtpTransport = nodemailer.createTransport('SMTP', config.mail);
      resetPasswordUrl += '?account=' + doc._id;
      smtpTransport.sendMail({
        from: 'thisapp@example.com',
        to: doc.email,
        subject: 'SocialNet Password Request',
        text: 'Click here to reset your password: ' + resetPasswordUrl
      }, function forgotPasswordResult(err) {
        if (err) {
          callback(false);
        } else {
          callback(true);
        }
      });
    }
```

```
    });
  };

  var login = function(email, password, callback) {
    var shaSum = crypto.createHash('sha256');
    shaSum.update(password);
    Account.findOne({email:email,password:shaSum.digest('hex')},function(err,doc){
      callback(null!=doc);
    });
  };

  var register = function(email, password, firstName, lastName) {
    var shaSum = crypto.createHash('sha256');
    shaSum.update(password);

    console.log('Registering ' + email);
    var user = new Account({
      email: email,
      name: {
        first: firstName,
        last: lastName
      },
      password: shaSum.digest('hex')
    });
    user.save(registerCallback);
    console.log('Save command was sent');
  }

  return {
    register: register,
    forgotPassword: forgotPassword,
    changePassword: changePassword,
    login: login,
    Account: Account
  }
}
```

The `login` and `register` functions make use of Node's `crypto` library to convert the plain text password supplied into an encrypted hash using the SHA1 algorithm. This performs a one-way scramble on the text in order to prevent anyone who accesses your database from easily reverse-engineering your users' passwords.

Directly hashing a password as shown in Example 6-1 provides a layer of protection from casual attackers, but does not stop dictionary-based attacks where passwords are compared against a dictionary of pregenerated SHA1 hashes. If you were to "salt" the hash by adding a secret key in front of the password before encrypting it, anyone hoping to perform a dictionary-based attack against your database would need to generate a new list of encrypted passwords using your secret key before they could get at your passwords; they would need a copy of your database and your source code to perform this attack.

The login function queries MongoDB and returns a truth flag indicating whether or not it was able to find a user whose email address and encrypted password match the login credentials supplied by Node.js. If no accounts were found in the database, the `doc` variable will be `null`; otherwise, it will be populated from MongoDB.

The `forgotPassword` function sends an email to the account owner, instructing him on how to reset the password. This function will be described in more detail in Example 6-14.

The `changePassword` function supports the `forgotPassword` functionality by updating the account's password with a newly encrypted password, using MongoDB's `$set` command to change a single value in the account record rather than the entire document. Setting `upsert` to `false` means the query will only work on a document that exists in the database: it will not create a new account.

Routing

In Chapter 5, a basic website was set up without much consideration for multiple views and navigation between those views. Backbone.js provides a `Router` class that handles movement between the main views of your application the same way you move between views in a common web application such as Gmail. When faced with a URL like *http://localhost:8080/#register*, the router understands it should display content based upon the content after the hash (#) character—"register," in this case.

The router in Example 6-2 describes how to display four of the five screens that will be built over the course of this chapter. The `routes` object contains a list of patterns being watched by the router, and which function to execute when a match is made. In this case, the router is watching for #index, #login, #register, and #forgotpassword, and in each case executing a function with the same name.

Example 6-2. The Backbone.js router: public/js/router.js

```
define(['views/index', 'views/register', 'views/login', 'views/forgotpassword'],
       function(IndexView, RegisterView, LoginView, ForgotPasswordView) {
  var SocialRouter = Backbone.Router.extend({
```

```
    currentView: null,

    routes: {
      "index": "index",
      "login": "login",
      "register": "register",
      "forgotpassword": "forgotpassword"
    },

    changeView: function(view) {
      if ( null != this.currentView ) {
        this.currentView.undelegateEvents();
      }
      this.currentView = view;
      this.currentView.render();
    },

    index: function() {
      this.changeView(new IndexView());
    },

    login: function() {
      this.changeView(new LoginView());
    },

    forgotpassword: function() {
      this.changeView(new ForgotPasswordView());
    },

    register: function() {
      this.changeView(new RegisterView());
    }
  });

  return new SocialRouter();
});
```

The `changeView` function is important because it does the actual work of displaying each view by calling its `render` function. When a view is changed, the old view (`currentView`) is told to stop listening to web page events through the `undelegateEvents`. If you don't unhook the listeners when changing views, your old view will remain in memory and continue to react to user events, becoming so-called zombie functions.

Checking for Authentication

Users may be in one of two states when they come to the application: authenticated and able to access more content, or non-authenticated and in need of registration. To

determine which category the current user falls into, the first thing the application needs to do is to make an AJAX request to the Node backend server. Node will verify that the current session—identified by a session ID in the request header—is linked with a valid access token.

One thing that happens when you start to really play with Node.js is you think of JavaScript in terms of I/O interactions. The `initialize` function is a shining example of the callback flow; it's easy to try to read the file as a whole and get lost in layers of callbacks and functions, but when we write our code like in Example 6-3, it makes the programmer's intent clear.

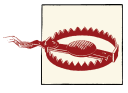
Example 6-3. SocialNet.js enhanced with a login check

```
define(['router'], function(router) {
  var initialize = function() {
    checkLogin(runApplication);
  };

  var checkLogin = function(callback) {
    $.ajax("/account/authenticated", {
      method: "GET",
      success: function() {
        return callback(true);
      },
      error: function(data) {
        return callback(false);
      }
    });
  };

  var runApplication = function(authenticated) {
    if (!authenticated) {
      window.location.hash = 'login';
    } else {
      window.location.hash = 'index';
    }
    Backbone.history.start();
  };

  return {
    initialize: initialize
  };
});
```



Naming is important: because of the load order in the *boot.js* file, the `router` class needs to be saved as */public/js/router.js*. If the router is saved to a different filename, your web browser will report a "Not Found" error such as "Failed to load resource: the server responded with a status of 404 (Not Found)."

The instruction given in the `initialize` function is, very literally, "Check Login, then Run Application".

The `checkLogin` function makes an AJAX request to the Express server running on Node.js and uses the response to determine if the user is allowed to access the deeper content within the application. `checkLogin` expects to execute the `callback` function with a single parameter indicating whether the user has been authenticated (true or false). If the AJAX callback returns success (HTTP code 200) the callback is returned with a `true` status, otherwise it is executed with a `false` flag.

The `runApplication` command is responsible for booting up the router, which will control what appears in the web browser. Based on the results of the `checkLogin` call, the active page will be set to either `#index` (for users who have logged in) or `#login` (for users who have not logged in).

The process isn't bulletproof at this point. For one, it's very easy to click on your address bar and change the path to the index view. This will get revisited later in the project, but right now the focus will be on creating and logging into new accounts. After all, there is nothing to break into at this point. Just be ready to come back and plug this security hole.

Authentication Handler

Server-side code is needed to kick off the Backbone views. The authenticated function checks the session data associated with the user request and returns a status code depending on whether or not that user has been signed in. If the user has not been logged in, the function will return status 401, which is a standard HTTP response code that has a standard meaning of "Unauthorized." Using well-defined standards makes the program code easier to understand; anyone versed in response codes will immediately understand 401 to mean the server responded properly but user authorization was either not provided or failed its security tests.

There's a lot happening in Example 6-4, even though it's such a short function. `app.get('/account/authenticated')` is Express's trigger to run the response callback when someone directs the web browser to *http://localhost:8080/account/authenticated*. When Backbone.js executes jQuery's `$.ajax` function, Express considers it the same as a human going to that URL in the web browser. In both cases, the `get` verb is sent to the Express server via HTTP.

Example 6-4. Authenticating the current user in Express

```
app.get('/account/authenticated', function(req, res) {
  if ( req.session.loggedIn ) {
    res.send(200);
```

```
  } else {
    res.send(401);
  }
});
```

Notice that the `session` variable is attached to the request (`req`) object. We usually think of a request as a message sent from the web browser to the web server, and the response as the message from the server back to the browser. By the time the request comes through to your callback method, it contains more than just the web browser's message—it contains everything the server knows about that message. Express's `session` middleware is part of the server's understanding of the request; the web browser sends an identifier, which the server uses to read the user's session data from the memory store (described later in this chapter). The nice part about this is it's done under the covers so you can just access the results of that read through the `session` variable attached to the request object.

If the `loggedIn` property is set on the `session` object, the server will respond with HTTP status code 200 (OK). Otherwise, it will send HTTP status code 401 (Unauthorized). The Backbone router will interpret the response code to decide which view should be shown to the user, depending on whether or not the user is seen as logged in.

Registration

The registration page accepts the user's name, email address, and password. Some of this information will also be used to prepopulate some of the account details later on, but this is the bare minimum required to get the user past the front door and into the system.

Registration Template

The registration template's job is to collect accurate login information from users so they can authenticate with the system later on. Since the two most critical pieces of information are the email address and password, the user will be asked to enter each piece twice to guard against small mistakes like typos.

Example 6-5 shows a common basic form that demonstrates many principals of good web design. First and foremost, each logical section of the form is grouped together inside field sets. So many sites still use tables and artificial spaces to break up the content on the page but grouping items together using a field set and descriptive legend is a simple way to provide context to the user, visually separate content, and present it in a way that is often similar to controls they will be accustomed to using elsewhere in their operating system.

Example 6-5. The registration template

```
<h1>Register</h1>

<form>
<fieldset>
<legend>Your Name</legend>
  <label>
    First:
    <input type="text" name="firstName" />
  </label>

  <label>
    Last:
    <input type="text" name="lastName" />
  </label>
</fieldset>

<fieldset>
  <legend>Email Address</legend>

  <label>
    Email:
    <input type="text" name="email" />
  </label>

  <br />

  <label>
    Email (confirm):
    <input type="text" name="cemail" />
  </label>
</fieldset>

<fieldset>
  <legend>Password</legend>

  <label>
    Password:
    <input type="password" name="password" />
  </label>

  <br />

  <label>
    Password (confirm):
    <input type="password" name="cpassword" />
  </label>
</fieldset>
```

```
<p>
  <input type="submit" value="Register Now"/>
</p>
</form>
```

The view class in Example 6-6 takes the data submitted by the user and posts it to the Express backend server. The `register` function returns `false` in order to disable the default form functionality, which would trigger a page reload. You don't need to reload the page because you have negotiated the server communication behind the scenes using the `post` command.

Example 6-6. The registration view class

```
define(['text!templates/register.html'], function(registerTemplate) {
  var registerView = Backbone.View.extend({
    el: $('#content'),

    events: {
      "submit form": "register"
    },

    register: function() {
      $.post('/register', {
        firstName: $('input[name=firstName]').val(),
        lastName: $('input[name=lastName]').val(),
        email: $('input[name=email]').val(),
        password: $('input[name=password]').val(),
      }, function(data) {
        console.log(data);
      });
      return false;
    },

    render: function() {
      this.$el.html(registerTemplate);
    }
  });

  return registerView;
});
```

This process could be improved by implementing support for the "confirm email" and "confirm password" input areas. Right now there is no checking done, so if those fields don't match, the user will still be able to register and proceed.

The form could also be improved by providing user feedback when the post command fails. Since the command can fail for many reasons (bad email address, bad password, a user has already registered with the same email class, etc.), this is really not doing your user any favors at the moment.

Registration Handler

Once a working client view has been created for the registration process, you can hook it up to the web server and make interesting things start to happen.

The `register` command in Example 6-7 goes inside the *app.js* file after Express has been configured, and is responsible for setting up the route to handle registration requests. As long as a valid email and password are provided, the registration process is allowed to proceed. Otherwise, error code 400 (Bad Request) is returned.

Example 6-7. Express's registration endpoint

```
app.post('/register', function(req, res) {
  var firstName = req.param('firstName', '');
  var lastName = req.param('lastName', '');
  var email = req.param('email', null);
  var password = req.param('password', null);

  if ( null == email || null == password ) {
    res.send(400);
    return;
  }

  Account.register(email, password, firstName, lastName);
  res.send(200);
});
```

Watch out! The `Account.register` function is called followed by `res.send(200)`. There is no callback in this case, which means the actual registration is going to get fired off and handled even after the user receives an "OK" response from the server. If there is a problem with the registration (for example, the email has already been used in an account), the user will not be notified during his submission.

Login

Returning users can resume their experience by providing their email address and password. The login screen also exposes the "Forgot Password" functionality, so users who don't remember their password can reset them.

Login Template

The login template is the first view users will be presented with when they come to the site if they are not already logged into a valid account. Its purpose is to grant access to the site or lead to the registration template.

Like the register template, the login template in Example 6-8 groups the input controls together with a field set. Since this template is the entry page for our application when users are not authenticated, links to the Forgot Password and Sign Up pages are provided.

Look at the link structure: traditionally hashes (#) are used to denote bookmarks on the same page. The Backbone router handles those bookmarks and uses them to control which view is shown to the user. Clicking either of those links will cause the router to display a new view (Forgot Password or Sign Up) to the user.

Example 6-8. The login template

```
<h1>Login</h1>

<form>
<fieldset>
  <legend>Credentials</legend>

  <p class="error" id="error"></p>

  <label>
    Email:
    <input type="text" name="email">
  </label>

  <br />

  <label>
    Password:
    <input type="password" name="password">
  </label>

  <br />

  <input type="submit" value="Login Now">
</fieldset>
</form>

<ul>
  <li><a href="#forgotpassword">Forgot Password?</a></li>
  <li><a href="#register">Sign Up</a></li>
</ul>
```

As the application is built out more, navigational elements like Forgot Password and Sign Up will be moved to the overall site template instead of handled within individual views. Otherwise, you would have to edit every template whenever a navigational change needs to be made—not very practical.

Finally, the login view in Example 6-9 handles the login form very similarly to the registration form.

Example 6-9. The login view class

```
define(['text!templates/login.html'], function(loginTemplate) {
  var loginView = Backbone.View.extend({
    el: $('#content'),
```

```
    events: {
      "submit form": "login"
    },

    login: function() {
      $.post('/login', {
        email: $('input[name=email]').val(),
        password: $('input[name=password]').val()
      }, function(data) {
        console.log(data);
      }).error(function(){
        $("#error").text('Unable to login.');
        $("#error").slideDown();
      });
      return false;
    },

    render: function() {
      this.$el.html(loginTemplate);
      $("#error").hide();
    }
  });

  return loginView;
});
```

When the user submits the form, the view collects the login details and generates a form post to the Express server backend. If the login is a failure (username and password don't authenticate against any accounts), some error text will slide into view, informing the user about a problem trying to connect. This is intentionally vague: security best practices discourage telling the user much about why the login failed. It's generally a bad idea to even confirm the existence of an account. Otherwise, attackers would know when they hit upon a "real" user and can bombard that account with password requests.

We are only concerned about presenting the registration and login functionality to the user in this chapter, so when the login is a success there is no action performed by the web browser. This will be expanded upon and built out in Chapter 7.

Login Handler

The login form needs a server-side component to actually work. Example 6-10 is the Express function for handling user login requests.

Example 6-10. Express's login endpoint

```
app.post('/login', function(req, res) {
  console.log('login request');
  var email = req.param('email', null);
  var password = req.param('password', null);
```

```
  if ( null == email || email.length < 1
      || null == password || password.length < 1 ) {
    res.send(400);
    return;
  }

  Account.login(email, password, function(success) {
          if ( !success ) {
            res.send(401);
            return;
          }
      console.log('login was successful');
          res.send(200);
  });
});
```

The `login` function makes certain to check whether the email and password are `null` (not supplied at all) or empty. If so, Express returns error code 400 (Bad Request) to the sender rather than wasting time and resources checking a guaranteed fail against the data store. If the inputs are correct, Express will pass control to the `login` function from the account model.

The `login` function triggers a callback with a `success` parameter indicating whether the login was successful. The important point to stress here is the parameters sent to the `login` function: the `email` and `password` parameters, not the `request` and `response` objects. It's tempting to send everything to `Account.login` and let it handle responding to the user and closing the connection, but if you were to do that you would be tightly coupling the account model with the server response logic. This is a bad thing because every time you want to modify this function, you would need to jump back and forth between this file and the account model's source file, and you would not be able to access the `login` function outside the context of a web request (for example, if you wanted to login from a socket instead of from a form post).

Using a callback makes it very clear that you expect to do something with the result of the login request, and puts the server response inline with the original request. Whenever possible, keep your variable list as small as possible by calling functions with as few parameters as you can.

Forgot Password

The Forgot Password feature is needed to provide access to accounts when users are unable to provide their password because they lost or forgot it. Sometimes web applications are launched that do not include this feature at first, but I argue it is a necessary part of a minimum viable product (MVP). People forget passwords; it's better to acknowledge this and not let it become a barrier to using the software.

Forgot Password Template

The HTML for the Forgot Password form will be minimal, as shown in Example 6-11.

Example 6-11. The Forgot Password template

```
<form>
  <label>
    Email Address:
    <input type="text" name="email" />
  </label>

  <p>
    <input type="submit" value="Reset my Password" />
  </p>
</form>
```

The only information you need in order to reset a user's password is his email address. There is no placeholder for error text: the user will receive no error message if the account does not exist. Just like the login form, you do not want to encourage attackers to probe the system for valid accounts they can attempt to brute-force their way into.

The workflow presented here makes the assumption that the user who registered the account is still in control of his email address; if someone else has captured the email account, that attacker would easily be able to take control of your user's SocialNet account, unless the security is tightened.

By now, the view class follows a predictable development pattern, as shown in Example 6-12.

Example 6-12. The Forgot Password view class

```
define(['text!templates/forgotpassword.html'], function(forgotpasswordTemplate) {
  var forgotpasswordView = Backbone.View.extend({
    el: $('#content'),

    events: {
      "submit form": "password"
    },

    password: function() {
      $.post('/forgotpassword', {
        email: $('input[name=email]').val()
      }, function(data) {
        console.log(data);
      });
      return false;
    },
```

```
    render: function() {
      this.$el.html(forgotpasswordTemplate);
    }
  });

  return forgotpasswordView;
});
```

The view classes in this application are built similarly on purpose. Later on when you need to maintain the application, having the classes as similar as possible makes debugging and creating new features simpler. All of the functionality is arranged similarly between views, making it easy to understand where changes should go and even how new elements should be named.

When the user submits the Forgot Password form, Backbone makes a jQuery post to the server with the user's email address and logs the result to the debug console. If the function were to fail, it would do so silently; later on you will come back to this and add some additional user experience improvements such as acknowledging the user's post when the server call has completed and providing an option to return to the login screen.

Forgot Password Handler

The Express route for the Forgot Password handler, like the login handler, passes data to the account model and responds to the user upon callback. This route is a bit more interesting because users will receive an email redirecting them back to the application in order to change their password. The link is generated based upon the request information, as in Example 6-13.

Example 6-13. Express's Forgot Password handler

```
app.post('/forgotpassword', function(req, res) {
  var hostname = req.headers.host;
  var resetPasswordUrl = 'http://' + hostname + '/resetPassword';
  var email = req.param('email', null);
  if ( null == email || email.length < 1 ) {
    res.send(400);
    return;
  }

  Account.forgotPassword(email, resetPasswordUrl, function(success){
    if (success) {
      res.send(200);
    } else {
      // Username or password not found
      res.send(404);
    }
  });
});
```

I try to avoid having environment-specific functionality wherever practical. So when it comes time to generate a URL, instead of going the obvious route and using a development, staging, or production URL, the hostname is retrieved from the request parameters used to activate the route. This is really useful in situations where multiple people are developing the same application because every developer has a preference for their host setup, port usage, and depending on their proxy settings, may not even hit a predictable URL. Pulling the hostname out of the request gives you a universal way to bring users back to the correct instance of your application regardless of your hosting setup.

But why generate the URL at all? The account model's `forgotPassword` function requires this in order to send a reset password link to the user, as shown in Example 6-14.

Example 6-14. Handling forgotten passwords in the account model

```
var forgotPassword = function(email, resetPasswordUrl, callback) {
  var user = Account.findOne({email: email}, function findAccount(err, doc){
    if (err) {
      // Email address is not a valid user
      callback(false);
    } else {
      var smtpTransport = nodemailer.createTransport('SMTP', config.mail);
      resetPasswordUrl += '?account=' + doc._id;
      smtpTransport.sendMail({
        from: 'thisapp@example.com',
        to: doc.email,
        subject: 'SocialNet Password Request',
        text: 'Click here to reset your password: ' + resetPasswordUrl
      }, function forgotPasswordResult(err) {
        if (err) {
          callback(false);
        } else {
          callback(true);
        }
      });
    }
  });
};
```

Example 6-14 demonstrates the use of nodemailer, an email library for Node included in this chapter's *package.json* configuration file. After loading the account data from the database based upon the supplied email address, the user is sent an email prompting him to click on the Reset Password link generated by the calling route in *app.js*.

Three events can trigger the callback:

1. The account cannot be found (`findOne` fails).
2. The email could not be sent.
3. The email was successfully sent.

Reset Password

After users create a Forgot Password request, they must reset their password in order to regain control of their account. You may have seen some websites that send a copy of your password to you, but this is bad for a number of reasons. In the event of a security breach, we do not want our users' passwords to be decryptable by attackers, so passwords should not be stored in the database. As shown, the hashed version of the password—a one-way encryption that cannot be decrypted to the original value—is stored instead. Intruders can still retrieve the original password by creating their own hashes of a dictionary of passwords and comparing against the data they stole, but doing so is more time-intensive. By the time they're able to compromise your users' accounts, you should already be aware that your system was corrupted and send notification to your users to change their passwords, reducing their exposure to harmful activities.

Reset Password Templates

The goal of the Reset Password view is to restore access of the account to the user, who presumably also has control of his email. The password is not reset when the Forgot Password form is used; rather the data is changed after the user completes the transaction by coming to the form in Example 6-15 from his email prompt and supplying a new password.

Example 6-15. The Reset Password form: views/resetPassword.jade

```
extends layout

block content
  form(action='/resetPassword',method='post')
    input(type='hidden',name='accountId',value='#{locals.accountId}')
    p
      label(for='password') New Password:
      input#password(type='password',name='password')
    p
      input(type='submit')
```

The email contains a token identifying the user account to change, so the only information the user needs to supply at this stage is a new password. Upon clicking Submit, the form is posted back to the Express server where the account is updated and the user is brought back to the success screen, as shown in Example 6-16.

Example 6-16. The Reset Password success screen: views/resetPasswordSuccess.jade

```
extends layout

block content
```

```
  p Your password has been reset.

  p
    a(href='/') Login
```

Both of these views use Express's Jade template engine rather than Backbone templates. It would have been straightforward enough to parse in the account token and handle a route in Backbone for the password reset, but because this flow is such an important departure from the rest of the application's logic it makes sense to handle it directly from Express. Once the Reset Password transaction has completed, the user can click on the Login link to return to the SocialNet application.

Reset Password Handler

The Express server-side reset password handler is interesting because it handles both the initial GET view as well as the user's POST request.

The first route listens for a GET request at *http://localhost:8080/resetPassword* and responds with the Jade template described in Example 6-17.

Example 6-17. The Express handler for resetting passwords

```
app.get('/resetPassword', function(req, res) {
  var accountId = req.param('account', null);
  res.render('resetPassword.jade', {locals:{accountId:accountId}});
});

app.post('/resetPassword', function(req, res) {
  var accountId = req.param('accountId', null);
  var password = req.param('password', null);
  if ( null != accountId && null != password ) {
    Account.changePassword(accountId, password);
  }
  res.render('resetPasswordSuccess.jade');
});
```

The second route listens for a POST request at the same address, indicating the user has supplied a new password. The account model's `changePassword` function is fired asynchronously and the "success" template is displayed to the user. Behind the scenes, the account's password is changed. The user doesn't need to wait for that to finish before receiving a response from the server, but the operation will have completed by the time he is able to get through to the login page and attempt to enter the site using the new password.

Putting It Together

While the snippets presented in this chapter are useful for understanding how Node and Backbone talk to each other, it isn't always clear how those bits fit into the bigger picture. This section attempts to bring those examples into context with the code written up to this point.

Node.js

In this chapter, Node got a workout as you implemented a custom signup and authentication system. Between sending password reminder emails to collecting and storing user passwords, this chapter laid the foundation for Node's role as a data provider for the entire application.

The nodemailer and connect libraries were added to the packages list during this chapter, as shown in Example 6-18. nodemailer is a library that handles sending email through SMTP. In this chapter it was used to send the password reminder through Sendgrid. connect is a set of middleware used by Express that also provides a series of handy functions you can utilize from your own work, such as the `MemoryStore` session storage object.

Example 6-18. The updated package.json

```
{
    "name": "my-social-network",
    "version": "0.0.1",
    "private": true,
    "dependencies": {
      "express": "~3.0.0",
      "jade": ">= 0.0.1",
      "mongoose": ">= 2.6.5",
      "nodemailer": "0.3.20",
      "connect": ">= 1.9.1"
  }
}
```

In Example 6-19, there are three new lines inside the `app.configure` section.

Example 6-19. The updated app.js

```
var express = require("express");
var app = express();
var nodemailer = require('nodemailer');
var MemoryStore = require('connect').session.MemoryStore;

// Import the data layer
var mongoose = require('mongoose');
var config = {
  mail: require('./config/mail')
};
```

```
// Import the accounts
var Account = require('./models/Account')(config, mongoose, nodemailer);

app.configure(function(){
  app.set('view engine', 'jade');
  app.use(express.static(__dirname + '/public'));
  app.use(express.limit('1mb'));
  app.use(express.bodyParser());
  app.use(express.cookieParser());
  app.use(express.session(
    {secret: "SocialNet secret key", store: new MemoryStore()}));
  mongoose.connect('mongodb://localhost/nodebackbone');
});

app.get('/', function(req, res){
  res.render('index.jade');
});

app.post('/login', function(req, res) {
  console.log('login request');
  var email = req.param('email', null);
  var password = req.param('password', null);

  if ( null == email || email.length < 1
      || null == password || password.length < 1 ) {
    res.send(400);
    return;
  }

  Account.login(email, password, function(success) {
    if ( !success ) {
      res.send(401);
      return;
    }
    console.log('login was successful');
    req.session.loggedIn = true; res.send(200);
  });
});

app.post('/register', function(req, res) {
  var firstName = req.param('firstName', '');
  var lastName = req.param('lastName', '');
  var email = req.param('email', null);
  var password = req.param('password', null);

  if ( null == email || email.length < 1
       || null == password || password.length < 1 ) {
    res.send(400);
    return;
  }
```

```
  Account.register(email, password, firstName, lastName);
  res.send(200);
});

app.get('/account/authenticated', function(req, res) {
  if ( req.session.loggedIn ) {
    res.send(200);
  } else {
    res.send(401);
  }
});

app.post('/forgotpassword', function(req, res) {
  var hostname = req.headers.host;
  var resetPasswordUrl = 'http://' + hostname + '/resetPassword';
  var email = req.param('email', null);
  if ( null == email || email.length < 1 ) {
    res.send(400);
    return;
  }

  Account.forgotPassword(email, resetPasswordUrl, function(success){
    if (success) {
      res.send(200);
    } else {
      // Username or password not found
      res.send(404);
    }
  });
});

app.get('/resetPassword', function(req, res) {
  var accountId = req.param('account', null);
  res.render('resetPassword.jade', {locals:{accountId:accountId}});
});

app.post('/resetPassword', function(req, res) {
  var accountId = req.param('accountId', null);
  var password = req.param('password', null);
  if ( null != accountId && null != password ) {
    Account.changePassword(accountId, password);
  }
  res.render('resetPasswordSuccess.jade');
});

app.listen(8080);
```

The `limit` and `bodyParser` middleware provide parsing service for the form posts. Unlike regular GET requests performed over HTTP that are easily parsed and often limited in size by web browsers, POST requests can contain huge bodies of data that require special processing to yield usable parameters. Using `limit` causes Express to cut

off incoming requests after a certain amount of data—in this case, one megabyte. This will help protect our application from distributed denial of service (DDOS) attacks, which would arise if sessions were allowed to post large requests that would otherwise bog down Express attempting to parse.

CHAPTER 7

The User Interface

User registration and login is important, but if you have spent any amount of time building websites you have probably already built that kind of functionality in several different languages. As the main source of user identification and as the first line of defense security-wise, authentication is always an important piece of the puzzle not to be overlooked.

The real experience begins once your user has successfully created his account and logged into the application. This is the user interface: the boundary between the user and your system where all of the interaction will take place.

The user interface for our social network will consist of three parts:

1. Account details
2. A contact list
3. An activity stream

As you will see, despite having multiple things happening on the page, each component is quite distinct. When the application code is kept cleanly separated it becomes possible to think of each piece of interaction separately from all of the others; this makes it easy to build, change, and share code even between radically different parts of the application.

Account Details

The account details page contains all of the information our user has entered about himself including his name, date of birth, email address, photograph, and biographical information.

You may edit any of the visible contents when you view your own account details page. Those edits will be immediately published and viewable by others next time they visit your profile. Later in the book this will be enhanced so your updated information reflects on your profile page immediately without viewers needing to refresh.

Anyone who views your account details page will have an option to add you to their contact list if you aren't already a contact.

Account Details Template

Let's go! First thing's first, the Backbone project needs to understand what an account is and how to fetch one, as shown in Example 7-1.

Example 7-1. The Backbone.js /models/Account.js

```
define(['models/StatusCollection'], function(StatusCollection) {
  var Account = Backbone.Model.extend({
    urlRoot: '/accounts',

    initialize: function() {
      this.status       = new StatusCollection();
      this.status.url   = '/accounts/' + this.id + '/status';
      this.activity     = new StatusCollection();
      this.activity.url = '/accounts/' + this.id + '/activity';
    }
  });

  return Account;
});
```

Since all of the account data is actually defined server-side by Express and MongoDB, there isn't much left for Backbone to do except read and display the stored information. When you define a model in Backbone, you are really just defining the route to pull the account down from the server by providing a `urlRoot`.

The profile view class will be responsible for displaying a single profile on screen, as shown in Example 7-2. This view will listen for changes to the underlying account model; the first change will happen when the model is first loaded.

Example 7-2. Backbone profile view: views/profile.js

```
define(['SocialNetView', 'text!templates/profile.html',
        'text!templates/status.html', 'models/Status',
        'views/Status'],
function(SocialNetView,  profileTemplate,
         statusTemplate, Status, StatusView)
{
  var profileView = SocialNetView.extend({
    el: $('#content'),

    initialize: function () {
```

```
      this.model.bind('change', this.render, this);
    },

    render: function() {
      this.$el.html(
        _.template(profileTemplate,this.model.toJSON())
      );

      var statusCollection = this.model.get('status');
      if ( null != statusCollection ) {
        _.each(statusCollection, function (statusJson) {
          var statusModel = new Status(statusJson);
          var statusHtml = (new StatusView({ model: statusModel })).render().el;
          $(statusHtml).prependTo('.status_list').hide().fadeIn('slow');
        });
      }
    }
  });

  return profileView;
});
```

When the view is initialized, call the `bind()` function on the model (in this case, we will be expecting an account model). The first parameter, `change`, means the action is to occur whenever any changes occur to the class model. The second parameter, `this.render`, refers to the function that will be executed every time the model changes. The third parameter, `this`, is a reference to the object (the profile view, in this case), which should execute `this.render`. Remember that JavaScript is not a class-based language, so the `this` keyword won't always refer to the object from which it is called. Explicitly binding the view to the model event allows you to reliably reference it using `this` in your callback.

The `render` function in this view is slightly different from the others that have been presented so far. Until now templates have been plain HTML text directly rendered to the browser using jQuery's `html()` function. This time the model is converted to a JSON object using `toJSON` and passed along with the view's HTML template to Underscore's `._template` function.

Example 7-3 is the first view in this book that actually uses a model as the source data for a view. Inside the template, you can access the properties of the model by surrounding your desired property with `<%=` and `%>`. This causes the property value to render in the HTML as if it were coded into the original page.

Example 7-3. Backbone profile view template

```
<h1><%=name.first %> <%= name.last %></h1>

<h2>Status Updates</h2>

<ul class="status_list" />
```

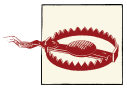
Account Details Handler

The backend component for the account details is fairly minimal, as shown in Example 7-4. When prompted for an account's information, Express queries MongoDB for the correct account and outputs its data in JSON format that can be used directly by Backbone.

Example 7-4. Express's account data listing

```
app.get('/accounts/:id', function(req, res) {
  var accountId = req.params.id == 'me'
                    ? req.session.accountId
                    : req.params.id;
  Account.findOne({_id:accountId}, function(account) {
    res.send(account);
  });
});
```

The account request isn't difficult on the server side. If the client requests "me", Node will get the account's ID from the current session object, otherwise it will retrieve the account from the provided ID.



Do you see the security problem with this handler? Because there is no processing done on the account, the JSON output will include the account's encrypted password. It would be very easy for an attacker to use this information to reverse-engineer your secret key and build a password generator offline without even hitting your site—this type of intrusion would be difficult or impossible to detect.

This is why it's important to be aware of how the code in your application interacts between modules and always think about the security implications of any data passed to and from your application.

Contact List

The contacts view will contain a list of users you have added to your contact list, as shown in Example 7-5. Each line item will display the contact's name, profile photograph, and a link to his account details page.

Example 7-5. contacts.html: the contacts template

```
<div id="contactlist">
</div>

<p>
  <a href="#addcontact">Add a Contact</a>
</p>
```

The contacts page template displays a user's contacts and provides a means to add new contacts (if viewing your own contact list). Notice the template contains a placeholder for something called `contactlist`; this will contain a list of contacts belonging to the profile whose contacts are to be displayed.

The list also contains a link to add a contact; this will be covered in more detail in Chapter 8.

The single contact view is essentially a stripped down profile, as shown in Example 7-6. The contact list provides a quick way to navigate between profiles, and is intended to be lightweight.

Example 7-6. contact.html: a single contact

```
<h1><%=name.first %> <%= name.last %></h1>
```

Activity Stream

The activity stream functions as your home page. Its purpose is to highlight changes made to your contacts' account details.

Activity Stream Template

As you can probably tell by the names of the functions in the updated index view in Example 7-7, this view now fulfills two purposes:

1. Allows users to update their status with `updateStatus`
2. Displays existing and new statuses to the user with `render` and `onStatusAdded`, respectively

Example 7-7. index.js: the new landing page after login

```
define(['SocialNetView', 'text!templates/index.html',
        'views/status', 'models/Status'],
function(SocialNetView, indexTemplate, StatusView, Status) {
  var indexView = SocialNetView.extend({
    el: $('#content'),

    events: {
      "submit form": "updateStatus"
    },

    initialize: function() {
      this.collection.on('add', this.onStatusAdded, this);
      this.collection.on('reset', this.onStatusCollectionReset, this);
    },

    onStatusCollectionReset: function(collection) {
      var that = this;
```

```
      collection.each(function (model) {
        that.onStatusAdded(model);
      });
    },

    onStatusAdded: function(status) {
      var statusHtml = (new StatusView({ model: status })).render().el;
      $(statusHtml).prependTo('.status_list').hide().fadeIn('slow');
    },

    updateStatus: function() {
      var statusText = $('input[name=status]').val();
      var statusCollection = this.collection;
      $.post('/accounts/me/status', {
        status: statusText
      }, function(data) {
        statusCollection.add(new Status({status:statusText}));
      });
      return false;
    },

    render: function() {
      this.$el.html(indexTemplate);
    }
  });

  return indexView;
});
```

Like the login and register templates, this view will accept form input from the user. The `updateStatus` function collects the information supplied by the user, posts it to the Express backend, generates a new `status` object, and adds it to the view's collection object.

The collection object is an instance of `StatusCollection` that extends the `Backbone.Collection` object. During the index view's initialization, the collection's `add` event is bound to the `onStatusAdded` function whose job is to create an HTML representation of the status and prepend it to the list of statuses, which have already been rendered.

Since the page loads with all statuses already in place, why not just create the status HTML and add it right to the page when the user adds a new status, rather than bothering to go through the collection object. The `onStatusAdded` function also looks forward into the future: when the web browser receives asynchronous updates from the server about friend status changes, `onStatusAdded` will cause those updates to animate onto the screen immediately.

Have a look at the HTML markup for the index page and for each status in Example 7-8.

Example 7-8. index.html: the new template container for the landing page

```
<h1>My Social Network</h1>

<h2>Activity Stream</h2>

<form>
  <fieldset>
    <legend>Update My Status</legend>

    <input type="text" name="status" />
    <input type="submit" value="Add" />
  </fieldset>
</form>

<ul class="status_list" />

<p><a href="#profile/me">See My Profile</a></p>
```

This is the HTML markup for the new status page. The form with a textfield for status update is new, as is the list element (`<ul>`) placeholder for the status list. At the bottom of the page is a link to the currently logged-in user's profile, which will eventually be phased out as the overall site layout is built and polished.

It would be pretty difficult to create a more minimal template than for the status update in Example 7-9. Because the `StatusView` class defines the list item (`<li>`) as the placeholder, all you need to include in the template is the contents of the list item, which for now will be the status text, not even a profile image, date, or time. As the application is built out you will definitely return to this template and build it out with more functionality, but for now this is a good example of how little it takes to produce a result using Backbone.

Example 7-9. status.html: a single status

```
<%= status %>
```

Just like the `model` class, the status in Example 7-10 comes equipped with a `urlRoot` so Backbone knows where to look for status information. All of the data properties (status text, username, profile picture) will be filled in dynamically when the application runs, so there is no need to explicitly define them here.

Example 7-10. status.js: a status model

```
define(function(require) {
  var Status = Backbone.Model.extend({
    urlRoot: '/accounts/' + this.accountId + '/status'
  });

  return Status;
});
```

In Backbone collections, the `model` variable is used to specify the class type contained in the collection. So the collection as defined in Example 7-11 will include a range of objects of the `Status` type.

Example 7-11. StatusCollection.js: a collection of status models

```
define(['models/Status'], function(Status) {
  var StatusCollection = Backbone.Collection.extend({
    model: Status
  });

  return StatusCollection;
});
```

Collections do not include a `urlRoot`. When Backbone needs to load a list of collections, it does so based upon the `urlRoot` of the underlying model object.

You might be wondering how Backbone can tell the server which accounts' status list it needs. When you instantiate the collection, you can also set its `urlRoot`, as shown in Example 7-12.

Example 7-12. Instantiating a StatusCollection with a custom urlRoot

```
var statusCollection = new StatusCollection();
statusCollection.url = '/accounts/me/activity';
statusCollection.fetch();
```

In this example, a status collection was created to store all activity status updates for the account with ID of "me". When the `fetch` command is called, Backbone connects to */accounts/me/activity* and uses the results to fill the status collection.

Activity Stream Handler

The activity stream is a hugely important part of the application because it is the first experience users will have when they log into the site and will be the recurrent central hub of interaction between the user and all of their contacts. The backend goal for this section is to make those interactions as fast as possible, generally staying out of the way as much as possible.

Getting the status list for an account is fairly straightforward: load the account and return its status list property, as shown in Example 7-13. Can you spot the bug here? If the account doesn't exist, Node will report an error, although it doesn't crash the application.

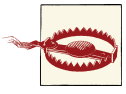
Example 7-13. Getting and setting the status list

```
app.get('/accounts/:id/status', function(req, res) {
  var accountId = req.params.id == 'me'
                    ? req.session.accountId
                    : req.params.id;
  models.Account.findById(accountId, function(account) {
```

```
    res.send(account.status);
  });
});

app.post('/accounts/:id/status', function(req, res) {
  var accountId = req.params.id == 'me'
                    ? req.session.accountId
                    : req.params.id;
  models.Account.findById(accountId, function(account) {
    status = {
      name: account.name,
      status: req.param('status', '')
    };
    account.status.push(status);

    // Push the status to all friends
    account.activity.push(status);
    account.save(function (err) {
      if (err) {
        console.log('Error saving account: ' + err);
      }
    });
  });
  res.send(200);
});
```



There is no authentication for the status handlers, meaning anyone who can guess the ID for an account can post to it willy nilly. In the coming chapters we will discuss how to close security holes like this by taking advantage of connect's middleware.

Posting a status is more interesting. First, the request returns right away, regardless of what happens to the status. This returns control of the frontend experience to Backbone so the status can instantly appear on the user's screen. The backend processing can take longer and longer as the dataset grows, but the end user won't be aware of the lag time and should receive feedback right away that the application has received the status. If the status fails to save for some reason, the user's experience can be reconciled later; the assumption is that the rare case of losing a single status will not have a breaking consequence on the user's interactive experience.

After Mongoose has loaded the account in question, the status is posted to that account's status feed and activity feed. Later on after contact list functionality has been added, the status will push out of each of the accounts' contacts' activity lists.

In Example 7-14, the activity list call duplicates the status call. In both cases, these handlers exist for the sole purpose of filling `StatusCollection` objects in Backbone—the same data is present in the account model and can be pulled from there without making direct requests for each list. We would only want to make these calls when the underlying account object is unknown or not needed for a particular view.

Example 7-14. Getting the activity list

```
app.get('/accounts/:id/activity', function(req, res) {
  var accountId = req.params.id == 'me'
                    ? req.session.accountId
                    : req.params.id;
  models.Account.findById(accountId, function(account) {
    res.send(account.activity);
  });
});
```

Data Model

Now that you know all of the data requirements for the account's functionality, it's time to define the schema for the account. This will be the base from which all of the system will be built, so it's important to have as versatile a model as possible.

Example 7-15 demonstrates this project's first serious schema definition using Mongoose, on a file that should be located at *models/Account.js* in relation to the root of the project. The first line pulls the Mongoose library into the namespace, making the `schema` object available for use.

Example 7-15. The account model

```
module.exports = function(config, mongoose, Status, nodemailer) {
  var crypto = require('crypto');

  var Status = new mongoose.Schema({
    name: {
      first:    { type: String },
      last:     { type: String }
    },
    status:     { type: String }
  });

  var AccountSchema = new mongoose.Schema({
    email:      { type: String, unique: true },
    password:   { type: String },
    name: {
      first:    { type: String },
      last:     { type: String }
    },
    birthday: {
      day:      { type: Number, min: 1, max: 31, required: false },
```

```
    month:   { type: Number, min: 1, max: 12, required: false },
    year:    { type: Number }
  },
  photoUrl:  { type: String },
  biography: { type: String },
  status:    [Status], // My own status updates only
  activity:  [Status]  //  All status updates including friends
});

var Account = mongoose.model('Account', AccountSchema);

var registerCallback = function(err) {
  if (err) {
    return console.log(err);
  };
  return console.log('Account was created');
};

var changePassword = function(accountId, newpassword) {
  var shaSum = crypto.createHash('sha256');
  shaSum.update(newpassword);
  var hashedPassword = shaSum.digest('hex');
  Account.update({_id:accountId}, {$set: {password:hashedPassword}},{upsert:false},
    function changePasswordCallback(err) {
      console.log('Change password done for account ' + accountId);
  });
};

var forgotPassword = function(email, resetPasswordUrl, callback) {
  var user = Account.findOne({email: email}, function findAccount(err, doc){
    if (err) {
      // Email address is not a valid user
      callback(false);
    } else {
      var smtpTransport = nodemailer.createTransport('SMTP', config.mail);
      resetPasswordUrl += '?account=' + doc._id;
      smtpTransport.sendMail({
        from: 'thisapp@example.com',
        to: doc.email,
        subject: 'SocialNet Password Request',
        text: 'Click here to reset your password: ' + resetPasswordUrl
      }, function forgotPasswordResult(err) {
        if (err) {
          callback(false);
        } else {
          callback(true);
        }
      });
    }
  });
};
```

```
  var login = function(email, password, callback) {
    var shaSum = crypto.createHash('sha256');
    shaSum.update(password);
    Account.findOne({email:email,password:shaSum.digest('hex')},function(err,doc){
      callback(doc);
    });
  };

  var findById = function(accountId, callback) {
    Account.findOne({_id:accountId}, function(err,doc) {
      callback(doc);
    });
  }

  var register = function(email, password, firstName, lastName) {
    var shaSum = crypto.createHash('sha256');
    shaSum.update(password);

    console.log('Registering ' + email);
    var user = new Account({
      email: email,
      name: {
        first: firstName,
        last: lastName
      },
      password: shaSum.digest('hex')
    });
    user.save(registerCallback);
    console.log('Save command was sent');
  }

  return {
    findById: findById,
    register: register,
    forgotPassword: forgotPassword,
    changePassword: changePassword,
    login: login,
    Account: Account
  }
}
```

Next, `AccountSchema` is defined—take special notice of the email field that has its index property set to `true`. When a field is indexed, MongoDB creates a special data structure that is used by Mongo to greatly enhance query performance.

The email field also uses the `unique` index property. This will prevent two accounts from sharing the same email address: a second register request for a particular email will fail.

The unique field will also prevent multiple empty email fields, so it won't be possible to accidentally create a bunch of accounts with missing email addresses. This is clearly a good thing for this application, but in some apps you won't want to do this because having `null` values would be desirable. In that case, adding the `sparse` index property will hide empty columns from the unique index.

Putting It Together

Hopefully it has been easy to follow along up to this point. With all the small changes happening throughout the application's code base, it's time to take a look at the big picture and make sure all of the pieces are coming together in a stable and extendable way.

Backbone

While talking about the new views and models, I skipped over the important glue that ties everything together: the router. Since the views and models are largely unaware of each other, it is the router's job to kick off the data population tasks, authenticate the user, and move between the new views.

In Example 7-16, the new routes are `index` and `profile`. `index` is responsible for handling the activity list, so a `StatusCollection` is created and passed into the new `IndexView` object. The status list shown in the profile view isn't handled as a true `StatusCollection` (the statuses are pulled from the `model` class during rendering), so only an account model is passed into the new `ProfileView`. In both views, the `fetch` function is called here in the router—it might make sense to fetch the data during the initialization step in the views themselves, but doing it here keeps control of the data transfer in the same spot where the data container is created. Performing this function in the views would force you as a programmer to constantly remember where the data for your models is initialized—it's far better to simply handle the event when the data load is complete and not worry about how or why the load was done.

Example 7-16. The updated /public/js/router.js

```
define(['views/index', 'views/register', 'views/login',
        'views/forgotpassword', 'views/profile', 'models/Account',
        'models/StatusCollection'],
function(IndexView, RegisterView, LoginView, ForgotPasswordView, ProfileView,
         Account, StatusCollection) {
  var SocialRouter = Backbone.Router.extend({
    currentView: null,

    routes: {
      "index": "index",
      "login": "login",
```

```
      "register": "register",
      "forgotpassword": "forgotpassword",
      "profile/:id": "profile"
    },

    changeView: function(view) {
      if ( null != this.currentView ) {
        this.currentView.undelegateEvents();
      }
      this.currentView = view;
      this.currentView.render();
    },

    index: function() {
      var statusCollection = new StatusCollection();
      statusCollection.url = '/accounts/me/activity';
      this.changeView(new IndexView({
        collection: statusCollection
      }));
      statusCollection.fetch();
    },

    login: function() {
      this.changeView(new LoginView());
    },

    forgotpassword: function() {
      this.changeView(new ForgotPasswordView());
    },

    register: function() {
      this.changeView(new RegisterView());
    },

    profile: function(id) {
      var model = new Account({id:id});
      this.changeView(new ProfileView({model:model}));
      model.fetch();
    }
  });

  return new SocialRouter();
});
```

Node.js

The Express methods have expanded slightly during this chapter, supporting the new data that is sent to and from the account model. Because all of the display and processing is handled on the frontend by Backbone.js, Node's involvement for this stretch has been relegated to the simple getting and setting of account data.

Aside from the GET route for the application data in Example 7-17, there are three new status-related routes in the updated application class. Both `activity` and `status` have GET operations that Backbone uses to populate its `collection` objects, but only `status` has a POST operation. This is because the application's users are only able to post status updates; even when they create a status from the activity view, they are really performing the same action as if they added a status from their profile-only status list. When a status is saved to the database, it is Express's job to ensure it is pushed out to all of the corresponding activity feeds.

Example 7-17. The updated app.js

```
var express     = require("express");
var app         = express();
var nodemailer  = require('nodemailer');
var MemoryStore = require('connect').session.MemoryStore;
var dbPath      = 'mongodb://localhost/nodebackbone';

// Import the data layer
var mongoose = require('mongoose');
var config = {
  mail: require('./config/mail')
};

// Import the models
var models = {
  Account: require('./models/Account')(config, mongoose, nodemailer)
};

app.configure(function(){
  app.set('view engine', 'jade');
  app.use(express.static(__dirname + '/public'));
  app.use(express.limit('1mb'));
  app.use(express.bodyParser());
  app.use(express.cookieParser());
  app.use(express.session({
    secret: "SocialNet secret key",
    store: new MemoryStore()
  }));
  mongoose.connect(dbPath, function onMongooseError(err) {
    if (err) throw err;
  });
});

app.get('/', function(req, res){
  res.render('index.jade');
});

app.post('/login', function(req, res) {
  console.log('login request');
  var email = req.param('email', null);
  var password = req.param('password', null);
```

```
  if ( null == email || email.length < 1
      || null == password || password.length < 1 ) {
    res.send(400);
    return;
  }

  models.Account.login(email, password, function(account) {
    if ( !account ) {
      res.send(401);
      return;
    }
    console.log('login was successful');
    req.session.loggedIn = true;
    req.session.accountId = account._id;
    res.send(200);
  });
});

app.post('/register', function(req, res) {
  var firstName = req.param('firstName', '');
  var lastName = req.param('lastName', '');
  var email = req.param('email', null);
  var password = req.param('password', null);

  if ( null == email || email.length < 1
       || null == password || password.length < 1 ) {
    res.send(400);
    return;
  }

  models.Account.register(email, password, firstName, lastName);
  res.send(200);
});

app.get('/account/authenticated', function(req, res) {
  if ( req.session.loggedIn ) {
    res.send(200);
  } else {
    res.send(401);
  }
});

app.get('/accounts/:id/activity', function(req, res) {
  var accountId = req.params.id == 'me'
                    ? req.session.accountId
                    : req.params.id;
  models.Account.findById(accountId, function(account) {
    res.send(account.activity);
  });
});
```

```
app.get('/accounts/:id/status', function(req, res) {
  var accountId = req.params.id == 'me'
                    ? req.session.accountId
                    : req.params.id;
  models.Account.findById(accountId, function(account) {
    res.send(account.status);
  });
});

app.post('/accounts/:id/status', function(req, res) {
  var accountId = req.params.id == 'me'
                    ? req.session.accountId
                    : req.params.id;
  models.Account.findById(accountId, function(account) {
    status = {
      name: account.name,
      status: req.param('status', '')
    };
    account.status.push(status);

    // Push the status to all friends
    account.activity.push(status);
    account.save(function (err) {
      if (err) {
        console.log('Error saving account: ' + err);
      }
    });
  });
  res.send(200);
});

app.get('/accounts/:id', function(req, res) {
  var accountId = req.params.id == 'me'
                    ? req.session.accountId
                    : req.params.id;
  models.Account.findById(accountId, function(account) {
    res.send(account);
  });
});

app.post('/forgotpassword', function(req, res) {
  var hostname = req.headers.host;
  var resetPasswordUrl = 'http://' + hostname + '/resetPassword';
  var email = req.param('email', null);
  if ( null == email || email.length < 1 ) {
    res.send(400);
    return;
  }

  models.Account.forgotPassword(email, resetPasswordUrl, function(success){
    if (success) {
      res.send(200);
```

```
    } else {
      // Username or password not found
      res.send(404);
    }
  });
});

app.get('/resetPassword', function(req, res) {
  var accountId = req.param('account', null);
  res.render('resetPassword.jade', {locals:{accountId:accountId}});
});

app.post('/resetPassword', function(req, res) {
  var accountId = req.param('accountId', null);
  var password = req.param('password', null);
  if ( null != accountId && null != password ) {
    models.Account.changePassword(accountId, password);
  }
  res.render('resetPasswordSuccess.jade');
});

app.listen(8080);
console.log('Listening on port 8080');
```

CHAPTER 8

Making Friends

Now that users are able to log into the application, retrieve their lost passwords, and create new accounts, they need a way to connect to one another; in other words, it's time to put the *social* aspect into this social network.

In Chapter 7 the contact list was described and scaffolded into the user interface; in this chapter you will add functionality to allow accounts to link to each other, publicize those relationships, and track each other's progress.

Contact List

The contact list's purpose is to aggregate all of an account's contacts into a single view, displaying only the most recent and top-level information about each relationship. The list will also expose a means to search for new contacts.

Contact List Template

The template controlling the contacts view is not very different from the templates you created for status and activity updates. Like the status update controller, the contact list is responsible for generating an overall "container" page as well as each of the "child" elements containing contacts belonging to the account. Unlike the status page in Chapter 7, the contact page will not need to load and display a `model` class; this makes a collection object a perfect choice to store the models.

Unlike the status views, when the contact list is found or updated, the new entry will not be animated to the screen. Instead, the entire contact list will refresh. To accomplish this, bind the `render` function on the collection's `reset` event during the contact list's `initialize` routine, as shown in Example 8-1. The `render` function wipes out any existing HTML and replaces it with a fresh copy of the list template, then proceeds to add a single contact view for each contact in the attached collection.

Example 8-1. contacts.js: the contacts list

```
define(['SocialNetView', 'views/contact', 'text!templates/contacts.html'],
function(SocialNetView, ContactView, contactsTemplate) {
  var contactsView = SocialNetView.extend({
    el: $('#content'),

    initialize: function() {
      this.collection.on('reset', this.renderCollection, this);
    },

    render: function() {
      this.$el.html(contactsTemplate);
    },

    renderCollection: function(collection) {
      collection.each(function(contact) {
        var statusHtml = (new ContactView(
                            { removeButton: true, model: contact }
                          )).render().el;
        $(statusHtml).appendTo('.contacts_list');
      });
    }
  });

  return contactsView;
});
```

The contact list shown in Example 8-2 contains enough information to generate the page view that will contain all of the contacts belonging to your account. As with the status messages screen built in Chapter 7, the contact list view is a placeholder for the eventual page content. All of the action—like actually filling up and displaying the contact list—will happen after the page is rendered by the web browser.

Example 8-2. contacts.html: the contacts view template

```
<div class="contacts_list" />

<p>
  <a href="#addcontact">Add a Contact</a>
</p>
```

The Add a Content link directs the browser to the hash-based URL `#addcontact`. When your user clicks this link, the web browser will attempt to load the same-page bookmark for `#addcontact`, which is what will trigger the router to load and render the add contact view.

Example 8-3 builds the contact model, which Backbone will eventually use to populate the contact list created in Examples 8-1 and 8-2. Looking back at the database model introduced in Chapter 6, you will notice there is no such thing as a distinct `Contact`

entity—each user's contact list is embedded inside his account recorded. This is very useful for querying because you can read the account information and get a list of his contacts in the same request, but it can be difficult to work with; extracting small pieces of a large response and manipulating them requires a lot of thought and precision.

Example 8-3. models/Contact.js: the contact model

```
define(function(require) {
  var Contact = Backbone.Model.extend({
  });

  return Contact;
});
```

The solution here is to use a logical model (`Contact`) to house all of the information contained in the account's contact list. This lets you manipulate all of the contacts as if they were in fact real database objects, while Backbone takes care of the actual reading, writing, and updating work.

The `ContactCollection` model shown in Example 8-4 provides a Backbone collection based on a group of contacts. Remember: the `model` property is a keyword extended from the collection prototype that has special meaning (that is, which model type is contained in a `ContactCollection`). Because `Contact` has been specified as the model type, Backbone understands how individual elements in the collection should behave and how they should be read from and written to the database.

Example 8-4. models/ContactCollection.js

```
define(['models/Contact'], function(Contact) {
  var ContactCollection = Backbone.Collection.extend({
    model: Contact
  });

  return ContactCollection;
});
```

Wherever possible, I have tried not to include any class dependencies between Backbone models in order to preserve as much separation between their concerns as possible. In this case the `ContactCollection` absolutely cannot function unless it has knowledge of the `Contact` model; this is why `models/Contact` is included as a dependency for the `define` function.

Because the `ContactCollection` lives within the context of RequireJS's `define` method, you make the class available to the outside world by exporting it via the `define` function's `return`. If you skip that step, code you write in other files will not be able to "see" the `ContactCollection`, and will not be able to instantiate or use it.

The controller in Example 8-5 implements the `ContactView` object referenced from the controller in Example 8-1. Depending on the context in which the contact is shown,

there may be no particular interactions available to end users, or they may have a chance to add or remove a contact from their list. The contact view delegates all of these interactions from the web browser to the backend server and updates the view inline. This is a great way to show off Backbone's power as a stateful client-side technology.

Example 8-5. views/contact.js: a single contact view controller

```
define(['SocialNetView', 'text!templates/contact.html'],
function(SocialNetView, contactTemplate) {
  var contactView = SocialNetView.extend({
    addButton: false,

    removeButton: false,

    tagName: 'li',

    events: {
      "click .addbutton": "addContact",
      "click .removebutton": "removeContact"
    },

    addContact: function() {
      var $responseArea = this.$('.actionArea');
      $.post('/accounts/me/contact',
        {contactId: this.model.get('_id')},
        function onSuccess() {
          $responseArea.text('Contact Added');
        }, function onError() {
          $responseArea.text('Could not add contact');
        }
      );
    },

    removeContact: function() {
      var $responseArea = this.$('.actionarea');
      $responseArea.text('Removing contact...');
      $.ajax({
        url: '/accounts/me/contact',
        type: 'DELETE',
        data: {
          contactId: this.model.get('accountId')
        }}).done(function onSuccess() {
          $responseArea.text('Contact Removed');
        }).fail(function onError() {
          $responseArea.text('Could not remove contact');
        });
    },

    initialize: function() {
      // Set the addButton variable in case it has been added in the constructor
      if ( this.options.addButton ) {
        this.addButton = this.options.addButton;
```

```
      }

      if ( this.options.removeButton ) {
        this.removeButton = this.options.removeButton;
      }
    },

    render: function() {
      $(this.el).html(_.template(contactTemplate, {
        model: this.model.toJSON(),
        addButton: this.addButton,
        removeButton: this.removeButton
      }));
      return this;
    }
  });

  return contactView;
});
```

Looking back at the Contact List view in Example 8-2, the contacts will live inside an unordered list (`<ul>`) HTML tag. Therefore, each model will be contained inside a list item (`<li>`) tag; by setting this as the `tagName` you are instructing Backbone to wrap the contents of your contact model inside a list item when rendering its HTML.

Finally, Example 8-6 contains the completed contact template. Whenever a contact appears on screen, the user has the option of clicking through to that contact's full profile page or removing the contact from his contact list. In both cases, the application router will pick up on the content of the hash tag (#) and direct the user to a new view template.

Example 8-6. contact.html: a single contact view template

```
<h1><%=model.name.first %> <%= model.name.last %></h1>

<% if ( !addButton ) { %>
  <p><a href="#profile/<%=model.accountId%>">View</p>
<% }%>

<div class="actionarea">
  <% if ( addButton ) { %>
    <p><button class="addbutton">Add Contact</p>
  <% } %>

  <% if ( removeButton ) { %>
    <p><button class="removebutton">Remove Contact</p>
  <% } %>
</div>
```

Contact List Handler

All of the heavy lifting is done by Backbone when it comes to displaying the contact list. Fetching the account model and returning its list of contacts will be the backend server's role in enabling this functionality.

Be careful when working with callbacks in Express (and Node.js in general): the single most common cause of difficult-to-track errors is the inadvertent closing of the response before the request has had a chance to complete. In Example 8-7, the account's contacts are sent to the response object within the callback function of the account model's `findById` method. If a response were to be made outside of the callback, the web browser would always receive an empty contact list even when the account has one or more contacts.

Example 8-7. Express's account contact endpoint

```
app.get('/accounts/:id/contacts', function(req, res) {
  var accountId = req.params.id == 'me'
                    ? req.session.accountId
                    : req.params.id;
  models.Account.findById(accountId, function(account) {
    res.send(account.contacts);
  });
});
```

Add Contact

The Add Friend feature has two major use cases:

1. You want to add a contact who is a mutual acquaintance of both yourself and someone else who you already have in your contact list.
2. You want to add a contact for whom you have either a partial or complete set of identifying information.

Focusing on the second case first, if you know your contact's name or email address, you can search the database for her profile and add her to your list. In cases where the person has a common name, your search will probably turn up multiple people, so the results should support more than one.

Add Contact Template

The Add Contact template controller renders both the initial view as well as the search results. Control is passed away from the Add Contact template when the user navigates.

The Add Contact view's primary role is for handling search queries. The first important part of this object is the `events` list; in Example 8-8, Backbone is being instructed to call the Add Contact view's `search` method whenever it observes a form submission.

Example 8-8. views/addcontact.js: the add friend form controller

```
define(['SocialNetView', 'models/Contact', 'views/Contact',
'text!templates/addcontact.html'],
function(SocialNetView, Contact, ContactView, addcontactTemplate)
{
  var addcontactView = SocialNetView.extend({
    el: $('#content'),

    events: {
      "submit form": "search"
    },

    search: function() {
      var view = this;
      $.post('/contacts/find',
        this.$('form').serialize(), function(data) {
        view.render(data);
      }).error(function(){
        $("#results").text('No contacts found.');
        $("#results").slideDown();
      });
      return false;
    },

    render: function(resultList) {
      var view = this;
      this.$el.html(_.template(addcontactTemplate));
      if ( null != resultList ) {
        _.each(resultList, function (contactJson) {
          var contactModel = new Contact(contactJson);
          var contactHtml = (new ContactView(
                              { addButton: true, model: contactModel }
                            )).render().el;
          $('#results').append(contactHtml);
        });
      }
    }
  });

  return addcontactView;
});
```

The `search` function traps a closure for the current Add Contact view and performs a `find` command against the Node.js backend. The `serialize()` function turns the form fields supplied by the user into a JSON array sent to the server. If Backbone receives a successful response, it will re-render the view along with the search results; otherwise it will display a No Contacts Found error on screen.

The `search` function contains a variable called `view`, which is assigned the value of `this`. Doing this ensures that a reference to the Add Contact form is carried through to the POST callback; otherwise you would not be able to call the view's `render` function.

The HTML for this view contains a simple form with a single text input area, a submit button, and a placeholder for the results, as seen in Example 8-9.

Example 8-9. templates/addcontact.html: the add friend form template

```
<form>
  <p>
    <label>
      Name or Email:
      <input type="text" name="searchStr" />
    </label>
  </p>

  <p>
    <input type="submit" value="Search Now" />
  </p>
</form>

<div id="results"></div>
```

All the frontend pieces are now in place for adding contacts to your account. Now it is time to start building out the backend functionality and using MongoDB to perform a more advanced search based on the single text input from the Add Contact form.

Add Contact Handler

Throughout the search and add contact process, the backend server fills two roles: authentication and data retrieval. Authentication means two things: first, it checks if the user is logged in and allowed to search the contacts list, and second, it checks if the user is allowed to add a particular contact to the list. While most sites will have some kind of filtering system that requires friends to validate each other, or may show reduced amounts of information about contacts depending on their sharing settings, this social network is wide open. In order to add a contact to your list, the contact must be new (not already on the list) and a different individual (someone other than yourself).

The first thing to do when receiving a search request is to make sure it is valid. If the search string was not provided by the client, respond with error status 400 (Invalid Request Arguments), as shown in Example 8-10. Sending a specific response code lets the client handle the error in whatever way is best for the context in which the request was sent—either through a general error message or through more specific error messaging where the user is told which fields he failed to provide.

Example 8-10. The Express route for find contact

```
app.post('/contacts/find', function(req, res) {
  var searchStr = req.param('searchStr', null);
  if ( null == searchStr ) {
    res.send(400);
    return;
  }

  models.Account.findByString(searchStr, function onSearchDone(err,accounts) {
    if (err || accounts.length == 0) {
      res.send(404);
    } else {
      res.send(accounts);
    }
  });
});
```

The goal for the search tool is to provide an exceedingly simple user interface, accomplished through the use of a single text input box on the search form. What could be simpler and more minimal than that? It is up to Node.js, Express, and Mongo to make sense of the user's query and find contacts from what was typed. One way to accomplish this is to use regular expressions to perform case-insensitive searches against the name and email fields in each account.

Example 8-11 demonstrates how to build a regular expression in Node. The `RegExp` constructor is used to generate a regular expression dynamically from a string; here it is using the contents of `searchStr` with the option `i` meaning the regular expression should look anywhere in a string for the contents of `searchStr` and match against both lowercase and uppercase versions.

Example 8-11. The account model's findByString method

```
var findByString = function(searchStr, callback) {
  var searchRegex = new RegExp(searchStr, 'i');
  Account.find({
    $or: [
      { 'name.full': { $regex: searchRegex } },
      { email:       { $regex: searchRegex } }
    ]
  }, callback);
};
```

There is a lot of meaning in the function descriptor in Express's Add Contact route, as shown in Example 8-12. By having the route listen for the POST HTTP verb, you are communicating that submitting a form (using POST) signifies that a contact should be added. Just like any written language, the order of the words is important: you are saying "Post [a] Contact [belonging to] Account [whose ID is] :id."

Example 8-12. The Express route for addContact

```
app.post('/accounts/:id/contact', function(req,res) {
  var accountId = req.params.id == 'me'
                    ? req.session.accountId
                    : req.params.id;
  var contactId = req.param('contactId', null);

  // Missing contactId, don't bother going any further
  if ( null == contactId ) {
    res.send(400);
    return;
  }

  models.Account.findById(accountId, function(account) {
    if ( account ) {
      models.Account.findById(contactId, function(contact) {
        models.Account.addContact(account, contact);

        // Make the reverse link
        models.Account.addContact(contact, account);
        account.save();
      });
    }
  });

  // Note: Not in callback - this endpoint returns immediately and
  // processes in the background
  res.send(200);
});
```

The `addcontact` function in Example 8-13 creates a contact and pushes it to the account's contact list.

Example 8-13. The account model's addContact method

```
var addContact = function(account, addcontact) {
  contact = {
    name: addcontact.name,
    accountId: addcontact._id,
    added: new Date(),
    updated: new Date()
  };
  account.contacts.push(contact);

  account.save(function (err) {
    if (err) {
      console.log('Error saving account: ' + err);
    }
  });
};
```

Remove Contact

Sometimes the relationship just doesn't work out and you will need to remove a contact from your list, as well as any permissions that contact had for interacting with your account.

Remove Contact Template

Earlier in this chapter you added a Remove button to the Contact view. Let's revisit the click handler for that button in Example 8-14.

Example 8-14. The contact.js view's remove contact event listener

```
removeContact: function() {
  var $responseArea = this.$('.actionarea');
  $responseArea.text('Removing contact...');
  $.ajax({
    url: '/accounts/me/contact',
    type: 'DELETE',
    data: {
      contactId: this.model.get('accountId')
    }}).done(function onSuccess() {
      $responseArea.text('Contact Removed');
    }).fail(function onError() {
      $responseArea.text('Could not remove contact');
    });
}
```

Unlike all of the other asynchronous requests sent to the server so far, this event uses jQuery's `.ajax` method directly rather than `.get` or `.post`. Because this function is removing a contact from an account's contact list, we want to communicate this intent to the server using HTTP's DELETE verb. Notice how the URL (*/accounts/me/contact*) is exactly the same as the URL for the `addContact` function—using a different HTTP verb communicates a different meaning, while the actual URL allows you to refer to the same object.

Remove Contact Handler

For the backend server, removing a contact from your account is a process that involves loading your account data, finding the offending contact in your list of contacts, removing the contact from your contact list, and saving your account data back into the database. Although it is a fairly linear workflow, it involves two database scans and two input validations: first, to verify that you are who you say you are and your account exists, and second to find your removal target in your account list.

Like other routes, the verb in Example 8-15 used to set up the contact removal is important; the contact with a matching `contactId` will be removed from your account only if you submit your request using the HTTP DELETE verb. If no `contactId` is provided, the request is immediately rejected with error code 400 (Bad Request).

Example 8-15. Express's remove contact route

```
app.delete('/accounts/:id/contact', function(req,res) {
  var accountId = req.params.id == 'me'
                    ? req.session.accountId
                    : req.params.id;
  var contactId = req.param('contactId', null);

  // Missing contactId, don't bother going any further
  if ( null == contactId ) {
    res.send(400);
    return;
  }

  models.Account.findById(accountId, function(account) {
    if ( !account ) return;
    models.Account.findById(contactId, function(contact,err) {
      if ( !contact ) return;

      models.Account.removeContact(account, contactId);
      // Kill the reverse link
      models.Account.removeContact(contact, accountId);
    });
  });

  // Note: Not in callback - this endpoint returns immediately and
  // processes in the background
  res.send(200);
});
```

Assuming it is happy with the request, Express will then try to load your account from MongoDB. If found, Express will load your contact's account from MongoDB. As long as both accounts are found in the database, you will be removed from your contact's list and he will be removed from yours.

Take note of the response at the end of the function: all of the contact deletion occurs inside the account function callbacks, but the response is outside, within the scope of the route callback. This means the response will not wait for the contact removal to complete before sending a success indicator back to the client. As you have seen several times throughout this chapter, this programming pattern results in extremely fast responses to the client browser, but trades off reliability—if the removal is not successful, the contact will remain in your list next time you refresh your web page. If we can assume the majority of requests will be successful, we are accepting the small but rare inconvenience in order to serve the majority of requests at an accelerated speed.

Actually removing the contact from the account is a fairly straightforward process, as shown in Example 8-16. If the account has no contacts, it is safe to return right away. Otherwise, loop through each element in the contact list using the `forEach` function and compare its ID with the `contactId` provided by the user. If the IDs match, remove the contact from the list of contacts. Once this is all done, save the account back into MongoDB.

Example 8-16. The Mongoose remove contact function

```
var removeContact = function(account, contactId) {
  if ( null == account.contacts ) return;

  account.contacts.forEach(function(contact) {
    if ( contact.accountId == contactId ) {
      account.contacts.remove(contact);
    }
  });
  account.save();
};
```

One optimization to this function would be to save the account and return immediately when a match is found; this would prevent Node from having to check every element in a large list when a match is found early in the search. One reason I did not do this was because if you've been building this application along with the book and testing as you go, you probably have multiples of the same contact on your friend list by now. Looping through everyone on your list effectively removes doubles of the same contact.

Commenting

At last you are able to view, add, and remove contacts. You can already add status updates to your own account. The only thing missing at this stage is the ability to add comments to your contacts' profiles. As you will see, this functionality is little more than an extension of the existing status update with some security and authentication thrown into the mix.

Comment Template

Because the profile page is so similar to the index template you reach immediately after login, adding similar comment functionality won't be too difficult. The major difference between the two templates is that the index page contains the comments and status updates in its own dedicated collection, which can auto-update, whereas the profile page receives comments as a list object nested inside your contact's account data. Your task in this section will be to convert that list into a usable collection, and update it in a way that saves to the database as well as remains responsive to the end user.

The input form in Example 8-17 is lifted right from the *index.html* template. The web browser will display a text box with a single submission button, and the user is expected to enter their comment or message in the space provided.

Example 8-17. The new public/templates/profile.html

```
<h1><%=name.first %> <%= name.last %></h1>

<form>
  <fieldset>
    <legend>Add Status</legend>

    <input type="text" name="status" />
    <input type="submit" value="Add" />
  </fieldset>
</form>

<h2>Status Updates</h2>

<ul class="status_list" />
```

The profile page still contains the unordered list named `status_list`: this already contains all of the account's status updates, but you are about to give it real-time update capabilities.

In the last chapter, the profile page listed an account's activity stream updates and didn't do anything else. Now you can add status updates to your own account or to your contacts' accounts from this view. In Example 8-18 be aware of three new items: the `events` object, the `postStatus` function, and the `prependStatus` function.

Example 8-18. The new profile Backbone.js view: public/js/views/profile.js

```
define(['SocialNetView', 'text!templates/profile.html',
        'text!templates/status.html', 'models/Status',
        'views/Status'],
function(SocialNetView,  profileTemplate,
         statusTemplate, Status, StatusView)
{
  var profileView = SocialNetView.extend({
    el: $('#content'),

    events: {
      "submit form": "postStatus"
    },

    initialize: function () {
      this.model.bind('change', this.render, this);
    },

    postStatus: function() {
      var that = this;
      var statusText = $('input[name=status]').val();
```

```
      var statusCollection = this.collection;
      $.post('/accounts/' + this.model.get('_id') + '/status', {
        status: statusText
      }, function(data) {
        that.prependStatus(new Status({status:statusText}));
      });
      return false;
    },

    prependStatus: function(statusModel) {
      var statusHtml = (new StatusView({ model: statusModel })).render().el;
      $(statusHtml).prependTo('.status_list').hide().fadeIn('slow');
    },

    render: function() {
      var that = this;
      this.$el.html(
        _.template(profileTemplate,this.model.toJSON())
      );

      var statusCollection = this.model.get('status');
      if ( null != statusCollection ) {
        _.each(statusCollection, function (statusJson) {
          var statusModel = new Status(statusJson);
          that.prependStatus(statusModel);
        });
      }
    }
  });

  return profileView;
});
```

The `events` object contains all of the web browser events Backbone should be listening to. At this point in time the view needs to be interested in the form submit action when the user adds a status to the account profile. Notice the underlying data model has a `change` event still being defined in the view's `initialize` function rather than in the `events` object; this is because the `events` object refers to document object model (DOM) events in the web browser, whereas the `model.bind` function works on the `model` object stored within JavaScript's memory—in other words, it isn't a "real" event in a web browser sense, although it is important to the behavior of your application.

The `postStatus` function does the work of sending the user's status updates to the backend server. The function returns `false` in order to prevent the form from performing its default POST behavior, which would otherwise cause the page to refresh. This is not desirable since the app should update the web page in place. Once the status is acknowledged successfully by the web server with an HTTP status code of 200 (Success), the submitted status is added to the top of the account's contact list. As patterned elsewhere, the first thing you want to do when entering the function is to make a

reference to the view object (`this`), in this case using a variable named `that`. Doing this is important because when the form post success callback is reached, the variable named `this` will reference the form rather than the view. Defining a variable named `that` gives you access to the view object and its `prependStatus` function after posting the status to the backend server.

Finally, the `prependStatus` function takes a `Status` model, creates a `StatusView` to contain it, and renders it to the front of the `status_list` described in the template example. The same work was being done directly in the `render` function previously, but moving the code to its own function allows you to access it from both `render` and the `postStatus` callback.

Comment Handler

The backend server portion of the comment handling has, for the most part, already been done. You can already post a comment to your own stream, and you can already view your friends' comments. The only thing left to do is validate whether or not you are allowed to post comments on other people's streams. Doing this involves loading your contacts' account details and checking their relationship to you when it comes time to add something to their account on your behalf.

Because all of the objects in JavaScript are dynamic, you can very easily add new attributes, as shown in Example 8-19. Although the account model does not define a property called `isFriend`, it will appear in the JSON output from Express when anyone requests an account's details.

Example 8-19. Checking if an account has friend permissions in Express

```
app.get('/accounts/:id', function(req, res) {
  var accountId = req.params.id == 'me'
                    ? req.session.accountId
                    : req.params.id;
  models.Account.findById(accountId, function(account) {
    if ( accountId == 'me'
         || models.Account.hasContact(account, req.session.accountId) ) {
      account.isFriend = true;
    }
    res.send(account);
  });
});
```

The `hasContact` function's purpose is to determine whether a given `contactId` exists in an account's contact list. Given that the account has a contact list, Node loops through it using the `forEach` statement and compares each contact's `accountId` with the `contactId` given to the function, as shown in Example 8-20. If a match is found, `hasAccount` returns positive immediately, otherwise it continues to iterate over the contact list. If the full contact list is searched without a match, `hasContact` returns negative, indicating that the `contactId` is not found within the account's contacts.

Example 8-20. The Mongoose model's check contact functionality

```
var hasContact = function(account, contactId) {
  if ( null == account.contacts ) return false;

  account.contacts.forEach(function(contact) {
    if ( contact.accountId == contactId ) {
      return true;
    }
  });
  return false;
};
```

Putting It Together

Over the course of this chapter you learned how to connect two entities in a MongoDB collection, search entire collections for freeform text, loop through database results, and compare groups of objects to each other. Much of this functionality was built upon the code you built in the previous chapters, but some of the objects—particularly some of the views—were totally new.

To help put the preceding code into context with what came before, this section contains the unabridged models, views, templates, and controllers that were modified throughout the discussion.

Backbone

The majority of the Backbone changes in this chapter involved creating new views and templates. Since those new files were already described in detail they won't be repeated here. The two major touchpoints that were affected through all these changes were the router class, which controls the back-and-forth navigation through the application, and the index template, which is—at least for now—the main entry point to all of the application's functionality.

Router

The router now has knowledge of the contacts view and is charged with populating the list of contacts shown for your account.

The profile pages are likewise populated from the router, with the model being read from the backend server at the same time as the profile view for that model is being loaded, as shown in Example 8-21. If you happen to have a very slow connection to the backend server, you might end up with a default profile page—temporarily missing the account information—while the account data is in transit.

Example 8-21. The updated router.js

```
define(['views/index', 'views/register', 'views/login',
        'views/forgotpassword', 'views/profile', 'views/contacts',
        'views/addcontact', 'models/Account', 'models/StatusCollection',
        'models/ContactCollection'],
function(IndexView, RegisterView, LoginView, ForgotPasswordView, ProfileView,
         ContactsView, AddContactView, Account, StatusCollection,
         ContactCollection) {
  var SocialRouter = Backbone.Router.extend({
    currentView: null,

    routes: {
      'addcontact': 'addcontact',
      'index': 'index',
      'login': 'login',
      'register': 'register',
      'forgotpassword': 'forgotpassword',
      'profile/:id': 'profile',
      'contacts/:id': 'contacts'
    },

    changeView: function(view) {
      if ( null != this.currentView ) {
        this.currentView.undelegateEvents();
      }
      this.currentView = view;
      this.currentView.render();
    },

    index: function() {
      var statusCollection = new StatusCollection();
      statusCollection.url = '/accounts/me/activity';
      this.changeView(new IndexView({
        collection: statusCollection
      }));
      statusCollection.fetch();
    },

    addcontact: function() {
      this.changeView(new AddContactView());
    },

    login: function() {
      this.changeView(new LoginView());
    },
```

```
    forgotpassword: function() {
      this.changeView(new ForgotPasswordView());
    },

    register: function() {
      this.changeView(new RegisterView());
    },

    profile: function(id) {
      var model = new Account({id:id});
      this.changeView(new ProfileView({model:model}));
      model.fetch();
    },

    contacts: function(id) {
      var contactId = id ? id : 'me';
      var contactsCollection = new ContactCollection();
      contactsCollection.url = '/accounts/' + contactId + '/contacts';
      this.changeView(new ContactsView({
        collection: contactsCollection
      }));
      contactsCollection.fetch();
    }
  });

  return new SocialRouter();
});
```

Index

The index page, which previously just contained the list of your status updates, now also provides links to your profile and contact pages, as shown in Example 8-22.

Example 8-22. The updated /public/templates/index.html

```
<h1>My Social Network</h1>

<h2>Activity Stream</h2>

<form>
  <fieldset>
    <legend>Update My Status</legend>

    <input type="text" name="status" />
    <input type="submit" value="Add" />
  </fieldset>
</form>

<ul class="status_list" />
```

```
<p><a href="#profile/me">See My Profile</a></p>

<p><a href="#contacts/me">See My Contacts</a></p>
```

Node.js

While Backbone is now fronting the brunt of the work for this application, Node is still running behind the scenes and serving the role as the glue holding everything together.

As a general rule of thumb, you should never trust the client to control any data. So while Backbone is a beautiful stateful architecture and can update and change the information it displays inside the confines of a web browser, it should never be trusted to tell the database which user can post to a particular account's status list. The backend server is responsible for data integrity; that means it always validates whether a user is allowed to do what he is trying to do, and cutting him off if he doesn't have permission.

If someone were to come along and manipulate the JavaScript portion of your application into posting to a non-contact's account (this would be a fairly trivial task using built-in script inspectors with most web browsers), the server would recognize the forgery and refuse to service the update. The result would be that the new status update would appear (maybe) in the attacker's browser, but would be gone the next time he refreshed the page, and would not be shown to any other users.

Main Express application

The Express application is starting to get long; very soon it will need to be refactored into route-specific container files or else the application will become more difficult to extend and improve upon.

At this stage your application has GET, POST, and DELETE methods for most of the resources used by the social network application (accounts, contacts, and statuses, respectively). In this chapter you added security checks to some of the functions to prevent users from making changes to accounts they do not have relationships with, as shown in Example 8-23.

Example 8-23. The updated app.js

```
var express     = require("express");
var app         = express();
var nodemailer  = require('nodemailer');
var MemoryStore = require('connect').session.MemoryStore;
var dbPath      = 'mongodb://localhost/nodebackbone';

// Import the data layer
var mongoose = require('mongoose');
var config = {
  mail: require('./config/mail')
};
```

```
// Import the models
var models = {
  Account: require('./models/Account')(config, mongoose, nodemailer)
};

app.configure(function(){
  app.set('view engine', 'jade');
  app.use(express.static(__dirname + '/public'));
  app.use(express.limit('1mb'));
  app.use(express.bodyParser());
  app.use(express.cookieParser());
  app.use(express.session({
    secret: "SocialNet secret key",
    store: new MemoryStore()
  }));
  mongoose.connect(dbPath, function onMongooseError(err) {
    if (err) throw err;
  });
});

app.get('/', function(req, res){
  res.render('index.jade');
});

app.post('/login', function(req, res) {
  console.log('login request');
  var email = req.param('email', null);
  var password = req.param('password', null);

  if ( null == email || email.length < 1
      || null == password || password.length < 1 ) {
    res.send(400);
    return;
  }

  models.Account.login(email, password, function(account) {
    if ( !account ) {
      res.send(401);
      return;
    }
    console.log('login was successful');
    req.session.loggedIn = true;
    req.session.accountId = account._id;
    res.send(200);
  });
});

app.post('/register', function(req, res) {
  var firstName = req.param('firstName', '');
  var lastName = req.param('lastName', '');
  var email = req.param('email', null);
```

```
  var password = req.param('password', null);

  if ( null == email || email.length < 1
       || null == password || password.length < 1 ) {
    res.send(400);
    return;
  }

  models.Account.register(email, password, firstName, lastName);
  res.send(200);
});

app.get('/account/authenticated', function(req, res) {
  if ( req.session.loggedIn ) {
    res.send(200);
  } else {
    res.send(401);
  }
});

app.get('/accounts/:id/contacts', function(req, res) {
  var accountId = req.params.id == 'me'
                    ? req.session.accountId
                    : req.params.id;
  models.Account.findById(accountId, function(account) {
    res.send(account.contacts);
  });
});

app.get('/accounts/:id/activity', function(req, res) {
  var accountId = req.params.id == 'me'
                    ? req.session.accountId
                    : req.params.id;
  models.Account.findById(accountId, function(account) {
    res.send(account.activity);
  });
});

app.get('/accounts/:id/status', function(req, res) {
  var accountId = req.params.id == 'me'
                    ? req.session.accountId
                    : req.params.id;
  models.Account.findById(accountId, function(account) {
    res.send(account.status);
  });
});

app.post('/accounts/:id/status', function(req, res) {
  var accountId = req.params.id == 'me'
                    ? req.session.accountId
                    : req.params.id;
  models.Account.findById(accountId, function(account) {
```

```
    status = {
      name: account.name,
      status: req.param('status', '')
    };
    account.status.push(status);

    // Push the status to all friends
    account.activity.push(status);
    account.save(function (err) {
      if (err) {
        console.log('Error saving account: ' + err);
      }
    });
  });
  res.send(200);
});

app.delete('/accounts/:id/contact', function(req,res) {
  var accountId = req.params.id == 'me'
                    ? req.session.accountId
                    : req.params.id;
  var contactId = req.param('contactId', null);

  // Missing contactId, don't bother going any further
  if ( null == contactId ) {
    res.send(400);
    return;
  }

  models.Account.findById(accountId, function(account) {
    if ( !account ) return;
    models.Account.findById(contactId, function(contact,err) {
      if ( !contact ) return;

      models.Account.removeContact(account, contactId);
      // Kill the reverse link
      models.Account.removeContact(contact, accountId);
    });
  });

  // Note: Not in callback - this endpoint returns immediately and
  // processes in the background
  res.send(200);
});

app.post('/accounts/:id/contact', function(req,res) {
  var accountId = req.params.id == 'me'
                    ? req.session.accountId
                    : req.params.id;
  var contactId = req.param('contactId', null);

  // Missing contactId, don't bother going any further
```

```
  if ( null == contactId ) {
    res.send(400);
    return;
  }

  models.Account.findById(accountId, function(account) {
    if ( account ) {
      models.Account.findById(contactId, function(contact) {
        models.Account.addContact(account, contact);

        // Make the reverse link
        models.Account.addContact(contact, account);
        account.save();
      });
    }
  });

  // Note: Not in callback - this endpoint returns immediately and
  // processes in the background
  res.send(200);
});

app.get('/accounts/:id', function(req, res) {
  var accountId = req.params.id == 'me'
                    ? req.session.accountId
                    : req.params.id;
  models.Account.findById(accountId, function(account) {
    if ( accountId == 'me'
         || models.Account.hasContact(account, req.session.accountId) ) {
      account.isFriend = true;
    }
    res.send(account);
  });
});

app.post('/forgotpassword', function(req, res) {
  var hostname = req.headers.host;
  var resetPasswordUrl = 'http://' + hostname + '/resetPassword';
  var email = req.param('email', null);
  if ( null == email || email.length < 1 ) {
    res.send(400);
    return;
  }

  models.Account.forgotPassword(email, resetPasswordUrl, function(success){
    if (success) {
      res.send(200);
    } else {
      // Username or password not found
      res.send(404);
    }
  });
```

```
});

app.post('/contacts/find', function(req, res) {
  var searchStr = req.param('searchStr', null);
  if ( null == searchStr ) {
    res.send(400);
    return;
  }

  models.Account.findByString(searchStr, function onSearchDone(err,accounts) {
    if (err || accounts.length == 0) {
      res.send(404);
    } else {
      res.send(accounts);
    }
  });
});

app.get('/resetPassword', function(req, res) {
  var accountId = req.param('account', null);
  res.render('resetPassword.jade', {locals:{accountId:accountId}});
});

app.post('/resetPassword', function(req, res) {
  var accountId = req.param('accountId', null);
  var password = req.param('password', null);
  if ( null != accountId && null != password ) {
    models.Account.changePassword(accountId, password);
  }
  res.render('resetPasswordSuccess.jade');
});

app.listen(8080);
console.log('Listening on port 8080');
```

Account model

The new account model contains methods for adding and removing contacts, validating permissions from one account to another, and searching the entire collection for accounts based on partial matches of the name or email fields. As before, all of the functions are private and internal to the account JavaScript file unless included within the `exports` list, as shown in Example 8-24.

Example 8-24. The updated account.js model

```
module.exports = function(config, mongoose, Status, nodemailer) {
  var crypto = require('crypto');

  var Status = new mongoose.Schema({
    name: {
      first:    { type: String },
      last:     { type: String }
```

```
  },
  status:    { type: String }
});

var Contact = new mongoose.Schema({
  name: {
    first:   { type: String },
    last:    { type: String }
  },
  accountId: { type: mongoose.Schema.ObjectId },
  added:     { type: Date },     // When the contact was added
  updated:   { type: Date }      // When the contact last updated
});

var AccountSchema = new mongoose.Schema({
  email:     { type: String, unique: true },
  password:  { type: String },
  name: {
    first:   { type: String },
    last:    { type: String },
    full:    { type: String }
  },
  birthday: {
    day:     { type: Number, min: 1, max: 31, required: false },
    month:   { type: Number, min: 1, max: 12, required: false },
    year:    { type: Number }
  },
  photoUrl:  { type: String },
  biography: { type: String },
  contacts:  [Contact],
  status:    [Status], // My own status updates only
  activity:  [Status]  //  All status updates including friends
});

var Account = mongoose.model('Account', AccountSchema);

var registerCallback = function(err) {
  if (err) {
    return console.log(err);
  };
  return console.log('Account was created');
};

var changePassword = function(accountId, newpassword) {
  var shaSum = crypto.createHash('sha256');
  shaSum.update(newpassword);
  var hashedPassword = shaSum.digest('hex');
  Account.update({_id:accountId}, {$set: {password:hashedPassword}},{upsert:false},
    function changePasswordCallback(err) {
      console.log('Change password done for account ' + accountId);
  });
};
```

```
var forgotPassword = function(email, resetPasswordUrl, callback) {
  var user = Account.findOne({email: email}, function findAccount(err, doc){
    if (err) {
      // Email address is not a valid user
      callback(false);
    } else {
      var smtpTransport = nodemailer.createTransport('SMTP', config.mail);
      resetPasswordUrl += '?account=' + doc._id;
      smtpTransport.sendMail({
        from: 'thisapp@example.com',
        to: doc.email,
        subject: 'SocialNet Password Request',
        text: 'Click here to reset your password: ' + resetPasswordUrl
      }, function forgotPasswordResult(err) {
        if (err) {
          callback(false);
        } else {
          callback(true);
        }
      });
    }
  });
};

var login = function(email, password, callback) {
  var shaSum = crypto.createHash('sha256');
  shaSum.update(password);
  Account.findOne({email:email,password:shaSum.digest('hex')},function(err,doc){
    callback(doc);
  });
};

var findByString = function(searchStr, callback) {
  var searchRegex = new RegExp(searchStr, 'i');
  Account.find({
    $or: [
      { 'name.full': { $regex: searchRegex } },
      { email:       { $regex: searchRegex } }
    ]
  }, callback);
};

var findById = function(accountId, callback) {
  Account.findOne({_id:accountId}, function(err,doc) {
    callback(doc);
  });
};

var addContact = function(account, addcontact) {
  contact = {
    name: addcontact.name,
```

```
    accountId: addcontact._id,
    added: new Date(),
    updated: new Date()
  };
  account.contacts.push(contact);

  account.save(function (err) {
    if (err) {
      console.log('Error saving account: ' + err);
    }
  });
};

var removeContact = function(account, contactId) {
  if ( null == account.contacts ) return;

  account.contacts.forEach(function(contact) {
    if ( contact.accountId == contactId ) {
      account.contacts.remove(contact);
    }
  });
  account.save();
};

var hasContact = function(account, contactId) {
  if ( null == account.contacts ) return false;

  account.contacts.forEach(function(contact) {
    if ( contact.accountId == contactId ) {
      return true;
    }
  });
  return false;
};

var register = function(email, password, firstName, lastName) {
  var shaSum = crypto.createHash('sha256');
  shaSum.update(password);

  console.log('Registering ' + email);
  var user = new Account({
    email: email,
    name: {
      first: firstName,
      last: lastName,
      full: firstName + ' ' + lastName
    },
    password: shaSum.digest('hex')
  });
  user.save(registerCallback);
  console.log('Save command was sent');
};
```

```
  return {
    findById: findById,
    register: register,
    hasContact: hasContact,
    forgotPassword: forgotPassword,
    changePassword: changePassword,
    findByString: findByString,
    addContact: addContact,
    removeContact: removeContact,
    login: login,
    Account: Account
  }
}
```

CHAPTER 9
Chat

The social networking application is off to a good start. It's possible to share your updates, add and find contacts, and interact with each other all from piecing together some of the useful libraries that are already available for Node.js and Backbone. But if you were to try to use this application in real life, you might find that it feels cold and sterile—it's certainly a networking site, but it isn't very social.

It's time to brighten up the environment by enhancing your features with real-time capabilities. First up: chat!

Online chat presents your users with a means to communicate with one another almost as quickly as if they were having a real conversation. During a chat, the goal of your application should be to deliver messages from one party to the other as quickly as possible, without letting other processes like logging or database retrieval get in the way. A good chat system is really little more than a conduit between two communicators.

Refactoring

On the Node.js side of things, the application is starting to get too large to reasonably handle in a single script, so you're going to pull all the authentication- and account-related routes out of *app.js* and move them into their own files. Then you're going to loop through all of the routes during startup and add them to your application, as shown in Example 9-1.

Example 9-1. Moving routes to their own files

```
fs.readdirSync('routes').forEach(function(file) {
  var routeName = file.substr(0, file.indexOf('.'));
  require('./routes/' + routeName)(app);
});
```

This is a great way to handle external files because now you can add unlimited routes to your application without having to do anything special. Since every route is loaded the same way—that is, by passing the application object to the route during import—you are forced to structure your application in a way that keeps shared information, like models, available at the application level.

In order to share the session across different parts of the application, `key` item is added to the session store created during the application configuration stage, as shown in Example 9-2. This is the cookie key that will be shared between the frontend client and the backend server behind the scenes. It contains a unique ID (separate from the account ID that you use to reference a user in-app) that can be compared against the session store to get the user's stored session data.

Example 9-2. Improving the session store

```
app.use(express.session({
  secret: "SocialNet secret key",
  key: 'express.sid',
  store: app.sessionStore
}));
```

Connecting to the Chat Server

Your user needs to connect to the server before any messages can be passed around. Going online involves two steps: authorization and connecting. So far you have done the authorization step many times (every time any action is performed by Express), so Node.js needs to authorize the incoming request. When connecting to a web socket, the same principle applies; however, once you have successfully authenticated you do not need to verify your users any more. All of their communications can be considered "safe" from that point on. The authentication feature of sockets makes for much lighter-weight applications because you do not need to spend as much energy reading and writing to the database; once your user has logged on, you can persist his information and use it over and over again.

The real-time chat capabilities introduce new library requirements, so `socket.io` and `cookie` is added to the project's *package.json* file, as shown in Example 9-3. Run `npm install` to download these new dependencies before you continue. Socket.io, introduced in Chapter 2, will provide the real-time communication channel you will use to build out your chat. The `cookie` library exposes helper functions that will allow you to access the cookie generated by Express from within the context of a Socket.io connection. Although Socket.io and Express will be sharing the same project space, Express is not built upon the `connect` middleware so you are required to take extra steps in order to share data between the two stacks.

Example 9-3. Adding dependencies to package.json

```
{
    "name": "my-social-network",
    "version": "0.0.1",
    "private": true,
    "dependencies": {
      "express": "~3.0.0",
      "jade": ">= 0.0.1",
      "mongoose": ">= 2.6.5",
      "nodemailer": "0.3.20",
      "connect": ">= 1.9.1",
      "socket.io": "~0.9",
      "cookie": "0.0.4"
  }
}
```

Backbone

The first step in adding chat capability to the Backbone.js part of your application involves identifying where in the workflow a user can go from being an anonymous stranger to an authenticated, logged-on user. Once the user has logged on, he should be connected to the chat server and able to begin participating.

Fortunately, the application was designed to have as few touch points as possible. Your users can only be considered to be logging in when they have a successful login form completion and when they refresh their browser after a previously successful login.

I have chosen to implement a global event dispatcher in the router snippet in Example 9-4. First create a `socketEvents` property owned by the router, and have it extend the `Backbone.Events` object. This will create a standalone event object (as in, not attached to a particular model or view), which can be used independently of views. Although the event object could be totally generic, it is going to be strictly defined as events related to socket communications to make it easier to understand and work with in these examples.

Example 9-4. Adding an event dispatcher to the router

```
define(['views/index', 'views/register', 'views/login',
        'views/forgotpassword', 'views/profile', 'views/contacts',
        'views/addcontact', 'models/Account', 'models/StatusCollection',
        'models/ContactCollection'],
function(IndexView, RegisterView, LoginView, ForgotPasswordView, ProfileView,
         ContactsView, AddContactView, Account, StatusCollection,
         ContactCollection) {
  var SocialRouter = Backbone.Router.extend({
    currentView: null,

    socketEvents: _.extend({}, Backbone.Events),

    ...
```

```
    login: function() {
      this.changeView(new LoginView({socketEvents:this.socketEvents}));
    },


    ...

  });

  return new SocialRouter();
});
```

After creating the `socketEvents` object, it is passed into the login view when the login route is activated. This will cause the `socketEvents` object to appear in the view's initialization objects, which you will now trap inside the view.

Example 9-5 shows how the new `socketEvents` object is consumed within the login view. When the view is first created, `socketEvents` is presented as a key in the `initialize` function's `options` parameter and assigned to the view's own `socketEvents` property. Later, when the user has a successful login attempt, the view will trigger an `app:loggedin` event with `socketEvent`. This will send the text "app:loggedin" to any object that has registered a listener against the `app:loggedin` event using `bind`.

Example 9-5. Dispatching a login event

```
    initialize: function(options) {
      this.socketEvents = options.socketEvents;
    },

    login: function() {
      var socketEvents = this.socketEvents;
      $.post('/login',
        this.$('form').serialize(), function(data) {
          socketEvents.trigger('app:loggedin');
          window.location.hash = 'index';
      }).error(function(){
        $("#error").text('Unable to login.');
        $("#error").slideDown();
      });
      return false;
    }
```

Remember as before when performing a POST callback, the `socketEvents` was stuffed into a local variable because the callback will not have access to the view using the keyword `this`. So the `socketEvents` property is referenced in a variable inside the `login` function so it is available in the callback.

The socket class makes use of the Socket.io JavaScript library. Just like in the Socket.io examples in Chapter 2, the web browser will be loading the required JavaScript files from the express server. When the Socket.io library is enabled, it makes that JavaScript file available. For simplicity's sake, you can add the path to the Socket.io script to the `paths` list in your application's *boot.js* file:

```
Sockets: '/socket.io/socket.io'
```

Events are only useful when they are consumed and used by reactive functions. Because the sockets/chat functionality will be so involved, it will be contained within its own class (I'm calling it *SocialNetSockets.js*), as shown in Example 9-6. The class is intended to be initialized after the application's router, and takes the router's `socketEvents` object as the `initialization` parameter. When the class is initialized, the `socketEvents` object —called `eventDispatcher` here—is told to bind the `app:loggedin` event to the `connectSocket` function.

Example 9-6. A socket mediator class for Backbone

```
define(['Sockets'], function(sio) {
  var socket = null;

  var initialize = function(eventDispatcher) {
    eventDispatcher.bind('app:loggedin', connectSocket);
  };

  var connectSocket = function() {
    socket = io.connect().socket;

    socket
      .on('connect_failed', function(reason) {
        console.error('unable to connect', reason);
      })
      .on('connect', function() {
        console.info('successfully established a connection');
      });
  };

  return {
    initialize: initialize
  };
});
```

From now on when users connect to your application, they will trigger the `app:loggedin` event, which in turn causes the `connectSocket` function to make a connection to Node.js's Socket.io listener. If the connection is successful, an info message will be logged to the console. If the connection fails for some reason an error message will be triggered instead.

Node.js

As always the Node.js Express server will be responsible for authentication and data delivery.

Example 9-7 sets up the authorization portion of the Socket.io handshake. Before the socket is allowed to connect it goes through an authorization process to ensure the user is allowed to access the server. Because you set up a shared key for the Express cookie in Example 9-2, Socket.io is able to access the memory store for the connecting user to determine if the session ID matches one that is known to Node.js. If it is, the connection will be allowed to proceed.

Example 9-7. Setting up routes/chat.js

```
module.exports = function(app, models) {
  var io = require('socket.io');
  var utils = require('connect').utils;
  var cookie = require('cookie');
  var Session = require('connect').middleware.session.Session;

  var sio = io.listen(app.server);

  sio.configure(function() {
    sio.set('authorization', function( data, accept) {
      var signedCookies = cookie.parse(data.headers.cookie);
      var cookies = utils.parseSignedCookies(signedCookies,app.sessionSecret);
      data.sessionID = cookies['express.sid'];
      data.sessionStore = app.sessionStore;
      data.sessionStore.get(data.sessionID, function(err, session) {
        if ( err || !session ) {
          return accept('Invalid session', false);
        } else {
          data.session = new Session(data, session);
          accept(null, true);
        }
      });
    });
  });
}
```

The authorization method takes two parameters; the first (`data`) is the handshake data received by the server. This includes the session cookie where you will search for the user's login information. The second parameter (`accept`) is the callback function that should be triggered at the end of an authorization cycle. The callback requires two arguments: `error` and `success`. If there was a problem with the handshake, `error` will contain information about the problem; in Example 9-7 the handshake failure generates an error message stating "Invalid session," while the `success` parameter will contain `false` indicating that the authorization was a failure.

When the authorization is successful, the callback is executed with nothing in the `error` parameter and the `success` parameter set to `true`.

Example 9-8 contains the `connection` function, which will be executed after a successful handshake. Since the session was added to the handshake data in Example 9-7, you can access it in the `handshake` property of the socket object to find information about the connected account. In this case, the `accountId` is retrieved from the session—remember, this is the Mongo `ObjectID` you stored for the connected user. Once this information has been retrieved, the socket is instructed to join a room with the same name as the `accountId`. What this does is provide a filter on the socket communications that will allow your application to communicate with specified users without sending unnecessary network traffic at anyone else.

Example 9-8. Connecting to the chat server

```
sio.sockets.on('connection', function(socket) {
  var session = socket.handshake.session;
  var accountId = session.accountId;
  socket.join(accountId);
});
```

Sending and Receiving Chat Messages

The `join` command issued to Socket.io when a user connects is critically important to the chat system we will be building. Now that every user is effectively boxed into his own channel, you have a mechanism by which to address them all. Remember, if that filter were not set up then every message emitted through Socket.io would be received by everyone connected to the network. That would take an enormous amount of computing resource, and most of those messages would get discarded by people who aren't the intended recipient.

Example 9-9 adds more sophistication to the `Sockets` mediator. When the Socket.io server sends a chat event using the `chatserver` event name (indicating that the chat originated from the server), the `SocialNetSockets` class will trigger two of its own events in the `socketEvents` dispatcher: `socket:char:start` and `socket:chat:in`. Whenever a chat message is received, all interested observers will know they need to start a session and process an incoming message.

Example 9-9. The updated SocialNetSockets.js

```
define(['Sockets', 'models/contactcollection', 'views/chat'],
function(sio, ContactCollection, ChatView) {
  var SocialNetSockets = function(eventDispatcher) {
    var socket = null;

    var connectSocket = function() {
      socket = io.connect();
```

```
      socket
        .on('connect_failed', function(reason) {
          console.error('unable to connect', reason);
        })
        .on('connect', function() {
          eventDispatcher.bind('socket:chat', sendChat);
          socket.on('chatserver', function(data) {
            eventDispatcher.trigger('socket:chat:start:' + data.from );
            eventDispatcher.trigger('socket:chat:in:' + data.from, data);
          });
          var contactsCollection = new ContactCollection();
          contactsCollection.url = '/accounts/me/contacts';
          new ChatView({collection: contactsCollection,
                        socketEvents: eventDispatcher}).render();
          contactsCollection.fetch();
        });
    };

    var sendChat = function(payload) {
      if ( null != socket ) {
        socket.emit('chatclient', payload);
      }
    };

    eventDispatcher.bind('app:loggedin', connectSocket);
  }

  return {
    initialize: function(eventDispatcher) {
      SocialNetSockets(eventDispatcher);
    }
  };
});
```

Immediately upon connecting, the `eventDispatcher` will bind to an event called `socket:chat`. This will be triggered whenever the user sends a chat message to any of his contacts, as shown later in Example 9-14. When a chat is sent, the socket emits a `chatclient` event to the Socket.io server, indicating that the chat originated from the client.

Backbone

The Backbone.js chat user interface (UI) will have two main components: the list of contacts available for chatting, and a chat session window for each contact that will contain the running chat history. The desired goal for this phase is to have a list of contacts always available—clicking on any of the contacts initiates a chat session that you can use to send a message to your contact. If someone were to send you a message while you were online, a chat session with that person would automatically open on your screen.

The chat view in Example 9-10 will manage all of the UI relevant to the chat functionality. This view will contain a list of all of the connected user's contacts and keep him on screen at all times. During the `renderCollection` function, the list is redrawn, and for each contact the chat view binds the `startChatSession` callback on the `chat:start` event to handle cases when the user clicks on a particular contact's name.

Example 9-10. public/js/views/chat.js

```
define(['SocialNetView', 'views/chatsession', 'views/chatitem',
        'text!templates/chat.html'],
function(SocialNetView, ChatSessionView, ChatItemView, chatItemTemplate) {
  var chatView = SocialNetView.extend({
    el: $('#chat'),

    chatSessions: {},

    initialize: function(options) {
      this.socketEvents = options.socketEvents;
      this.collection.on('reset', this.renderCollection, this);
    },

    render: function() {
      this.$el.html(chatItemTemplate);
    },

    startChatSession: function(model) {
      var accountId = model.get('accountId');
      if ( !this.chatSessions[accountId]) {
        var chatSessionView = new ChatSessionView({
          model:model,
          socketEvents: this.socketEvents
                                                    });
        this.$el.prepend(chatSessionView.render().el);
        this.chatSessions[accountId] = chatSessionView;
      }
    },

    renderCollection: function(collection) {
      var that = this;
      $('.chat_list').empty();
      collection.each(function(contact) {
        var chatItemView = new ChatItemView({ socketEvents: that.socketEvents,
                                              model: contact });
        chatItemView.bind('chat:start', that.startChatSession, that);
        var statusHtml = (chatItemView).render().el;
        $(statusHtml).appendTo('.chat_list');
      });
    }
  });

  return chatView;
});
```

The `startChatSession` function checks to see whether a chat session already exists between the user and the given contact. If a session does not already exist, it will create one and add it to the `chatSessions` object (see Figure 9-1). The `chatSessions` object is a dictionary—because the `accountId` is used as the key for each session, it is kept easily accessible in memory, so checking for the existence of a particular account in this object is a fast operation.

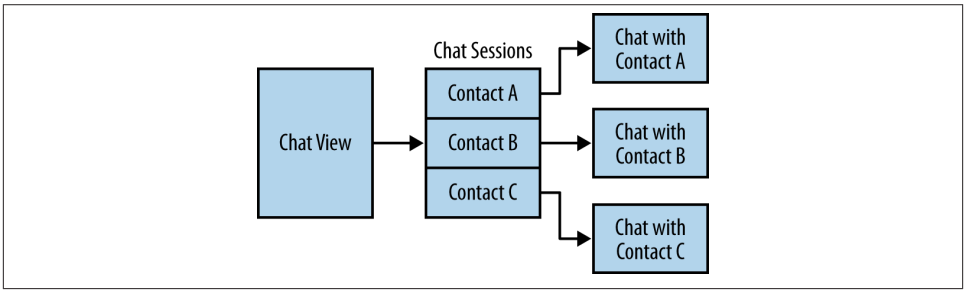


Figure 9-1. The chatSessions object

The `chatSessions` object contains a list of contact IDs. Each contact ID entry contains a reference to the memory location for the chat session with that contact. Using a dictionary like this ensures that only one chat session is open at a time for any given contact.

The entire chat view is shown in Example 9-11. Since the individual contacts will be housed in their own templates (as we'll see in Example 9-13), the chat window only needs to consist of the list item that will contain the list of contacts.

Example 9-11. public/templates/chat.html

```
<ul class="chat_list"></ul>
```

The chat contact view shown in full in Example 9-12 is responsible for determining when a chat session should be created by detecting when the user clicks on one of his contacts or when one of the contacts sends a message to the logged-in user. The only event in the `events` list is `'click': 'startChatSession'`, meaning the function `startChatSession` is to be run whenever the user clicks *anywhere* within the view.

Example 9-12. public/js/views/chatitem.js

```
define(['SocialNetView', 'text!templates/chatitem.html'],
function(SocialNetView, chatItemTemplate) {
  var chatItemView = SocialNetView.extend({
    tagName: 'li',

    $el: $(this.el),

    events: {
      'click': 'startChatSession',
```

```
    },

    initialize: function(options) {
      options.socketEvents.bind(
        'socket:chat:start:' + this.model.get('accountId'),
        this.startChatSession,
        this
      );
    },

    startChatSession: function() {
      this.trigger('chat:start', this.model);
    },

    render: function() {
      this.$el.html(_.template(chatItemTemplate, {
        model: this.model.toJSON()
      }));
      return this;
    }
  });

  return chatItemView;
});
```

When the view is initialized, it binds to the socket's `start` event. This instructs Backbone to go through the motions of starting a chat session whenever a chat is initiated from Socket.io. So whether the user clicks on a contact or the contact initiates a discussion with the user, the same process for starting the chat on screen is put into motion.

Starting a chat session from the chat item view means using Backbone's event `trigger` functionality. As shown in Example 9-10, the chat view spawns off actual chat sessions when triggered by the chat item view. So whenever someone clicks on the chat item or whenever a chat message comes through Socket.io, the chat item is responsible for notifying the chat controller about the appropriate time to bring a chat window on screen.

Because each item of a chat item view is contained in a list item (`<li>`), the template in Example 9-13 for this view needs to contain the template parameters to show the contact's name on screen. For each contact, Backbone will show the first name in a list.

Example 9-13. public/templates/chatitem.html

```
<span class="name"><%= model.name.first %></span>
```

The chat session view in Example 9-14 provides real-time interactivity between two users. Assuming both the sender and receiver are online at the same time, any messages sent by either party will be immediately visible on the screen of the other.

Example 9-14. public/js/views/chatsession.js

```
define(['SocialNetView', 'text!templates/chatsession.html'],
function(SocialNetView, chatItemTemplate) {
  var chatItemView = SocialNetView.extend({
    tagName: 'div',

    className: 'chat_session',

    $el: $(this.el),

    events: {
      'submit form': 'sendChat'
    },

    initialize: function(options) {
      this.socketEvents = options.socketEvents;
      this.socketEvents.on(
        'socket:chat:in:' + this.model.get('accountId'),
        this.receiveChat,
        this
      );
    },

    receiveChat: function(data) {
      var chatLine = this.model.get('name').first + ': ' + data.text;
      this.$el.find('.chat_log').append($('<li>' + chatLine + '</li>'));
    },

    sendChat: function() {
      var chatText = this.$el.find('input[name=chat]').val();
      if ( chatText && /[^\s]+/.test(chatText) ) {
        var chatLine = 'Me: ' + chatText;
        this.$el.find('.chat_log').append($('<li>' + chatLine + '</li>'));
        this.socketEvents.trigger('socket:chat', {
          to: this.model.get('accountId'),
          text: chatText
        });
      }
      return false;
    },

    render: function() {
      this.$el.html(_.template(chatItemTemplate, {
        model: this.model.toJSON()
      }));
      return this;
    }
  });

  return chatItemView;
});
```

The `events` list contains a single entry: `submit form`. Whenever the logged-in user clicks the Send button, this event puts the `sendChat` function into action. In order to send a message to your contact, you first use jQuery to get the value of the chat message form field:

```
var chatText = this.$el.find('input[name=chat]').val();
```

The jQuery express `input[name=chat]` instructs jQuery to look for input fields that contain a property called `name` with a value of `chat`. The `val()` function returns the value that is currently stored in the field. The message could be empty—for example, if the user entered a bunch of spaces or activated the Submit button without entering any test—so to avoid wasting Node's time processing an empty message, you can perform a regular expression to check not only whether or not the string was empty, but whether it was filled with empty space:

```
if ( chatText && /[^\s]+/.test(chatText) ) {
```

The regular expression `/[^\s]+/` translates to "one or more characters that are not white space." The brackets (`[]`) cause the regular expression engine to check for a single character. The code `\s` is a character class that matches white space characters. Putting a caret (`^`) in the first position negates the expression. So `[^\s]` would mean "a character that is NOT a white space." Placing a plus sign (`+`) at the end will cause the regular expression to search for one or more instances of "not a white space."

Now that the message is known to contain text, Backbone proceeds to add the message to the chat session's message list, and emit a socket event `socket:chat` to the contact's `accountId` with the chat message text. The chat message will get processed by the `SocialNetChat` dispatcher and emitted to the server, as shown in Example 9-9.

The HTML for building the chat session view is shown in Example 9-15. The chat session will consist of the contact's full name (so the user knows who she is talking to), a chat log containing all of the chat messages that have been sent or received, and a small form containing a text box and button to send a message to the contact.

Example 9-15. public/templates/chatsession.js

```
<span class="name"><%= model.name.full %></span>

<ul class="chat_log">
</ul>

<form>
  <input type="text" name="chat" />
  <input type="submit" value="Chat">
</form>
```

Node.js

The chat box will be ever-present in the HTML now, so it should be added to the Jade layout in */views/layout.jade*:

```
    div#chat
```

The pattern for dealing with Node.js so far has been to have it handle authentication and the passing of data. Because the chat server is already set up to authorize users—in Examples 9-7 and 9-8—all that is left to do is route chats sent from users to their intended recipients.

Example 9-16 contains the entirety of Node.js logic needed to pass chat messages between contacts. As shown in Example 9-14, a chat message consists of a "to" field containing the `accountID` of the chat recipient, and a "text" field containing the contents of the chat message. Since each user is joined to his own channel when he connects to the server (shown in Example 9-8), sending a message targeted to specific users involves emitting an event inside that channel.

Example 9-16. Routing chat messages in Node.js

```
socket.on('chatclient', function(data) {
  sio.sockets.in(data.to).emit('chatserver', {
    from: accountId,
    text: data.text
  });
});
```

Putting It Together

Although the chat functionality is fairly straightforward, it involves coordinating a lot of small bits. This part of the chapter is intended to help organize the project and discuss what changes were made to the existing code files.

Backbone

In this chapter you added a brand new user interface for chatting with your contacts. This is significant because the new interface is a global view—it doesn't get controlled by the router that displays all of the other views, and so its event handlers aren't handled in the same way. Whereas other views are added and removed, all of the chat views stay in sight at all times. This kind of functionality is what makes Backbone.js compelling; unlike traditional web applications, you as a developer can create persistent experiences on top of the regular flow of information through your application.

The file *boot.js* contains the paths to the global JavaScript libraries required by various parts of the application. In Example 9-17, a new library named Sockets is added to the list. This will be used by the `SocialNetSocket.js` class discussed throughout the chapter.

Example 9-17. public/js/boot.js

```
require.config({
  paths: {
    jQuery: '/js/libs/jquery',
    Underscore: '/js/libs/underscore',
    Backbone: '/js/libs/backbone',
    Sockets: '/socket.io/socket.io',
    models: 'models',
    text: '/js/libs/text',
    templates: '../templates',

    SocialNetView: '/js/SocialNetView'
  },

  shim: {
    'Backbone': ['Underscore', 'jQuery'],
    'SocialNet': ['Backbone']
  }
});

require(['SocialNet'], function(SocialNet) {
  SocialNet.initialize();
});
```

Example 9-18 brings back the trusted `router` class, which controls the client-facing display pages. In Chapter 9 you added `socketEvents`, an object that extends Backbone's `Event` prototype. The `socketEvents` object will be passed to the login so the socket server can be informed when the user successfully logs onto the site.

Example 9-18. public/js/router.js

```
define(['views/index', 'views/register', 'views/login',
        'views/forgotpassword', 'views/profile', 'views/contacts',
        'views/addcontact', 'models/Account', 'models/StatusCollection',
        'models/ContactCollection'],
function(IndexView, RegisterView, LoginView, ForgotPasswordView, ProfileView,
         ContactsView, AddContactView, Account, StatusCollection,
         ContactCollection) {
  var SocialRouter = Backbone.Router.extend({
    currentView: null,

    socketEvents: _.extend({}, Backbone.Events),

    routes: {
      'addcontact': 'addcontact',
      'index': 'index',
```

```
    'login': 'login',
    'register': 'register',
    'forgotpassword': 'forgotpassword',
    'profile/:id': 'profile',
    'contacts/:id': 'contacts'
  },

  changeView: function(view) {
    if ( null != this.currentView ) {
      this.currentView.undelegateEvents();
    }
    this.currentView = view;
    this.currentView.render();
  },

  index: function() {
    var statusCollection = new StatusCollection();
    statusCollection.url = '/accounts/me/activity';
    this.changeView(new IndexView({
      collection: statusCollection
    }));
    statusCollection.fetch();
  },

  addcontact: function() {
    this.changeView(new AddContactView());
  },

  login: function() {
    this.changeView(new LoginView({socketEvents:this.socketEvents}));
  },

  forgotpassword: function() {
    this.changeView(new ForgotPasswordView());
  },

  register: function() {
    this.changeView(new RegisterView());
  },

  profile: function(id) {
    var model = new Account({id:id});
    this.changeView(new ProfileView({model:model}));
    model.fetch();
  },

  contacts: function(id) {
    var contactId = id ? id : 'me';
    var contactsCollection = new ContactCollection();
    contactsCollection.url = '/accounts/' + contactId + '/contacts';
    this.changeView(new ContactsView({
      collection: contactsCollection
```

```
      }));
      contactsCollection.fetch();
    }
  });

  return new SocialRouter();
});
```

In addition, the application will reach into the router to get the `socketEvents` object to use for the overall `socket` mediator (`SocialNetSockets.js`).

Example 9-19 shows how the login view changes to support the new socket functionality. When a login view is initialized, it is given a reference to the router's `socketEvents` property. When the user successfully logs into the site, `socketEvents` `trigger` method is called with the event name `app:loggedin`. This will trigger the `sockets` mediator to connect to the Socket.io server and begin listening for chat events.

Example 9-19. public/js/views/login.js

```
define(['SocialNetView', 'text!templates/login.html'],
function(SocialNetView, loginTemplate) {
  var loginView = SocialNetView.extend({
    requireLogin: false,

    el: $('#content'),

    events: {
      "submit form": "login"
    },

    initialize: function(options) {
      this.socketEvents = options.socketEvents;
    },

    login: function() {
      var socketEvents = this.socketEvents;
      $.post('/login',
        this.$('form').serialize(), function(data) {
          socketEvents.trigger('app:loggedin');
          window.location.hash = 'index';
      }).error(function(){
        $("#error").text('Unable to login.');
        $("#error").slideDown();
      });
      return false;
    },

    render: function() {
      this.$el.html(loginTemplate);
      $("#error").hide();
      $("input[name=email]").focus();
    }
```

```
  });

  return loginView;
});
```

Node.js

Throughout this book I have tried to communicate the importance of Node.js as a gatekeeper to information. When you connect to a server, the goal should be to get in as quickly as possible and then get out, so any authentication has to be fast and transparent. Adding Socket.io to the mix helps with this because it opens up the possibility of performing many requests in a single connection while enduring the overhead of authentication only once. This can open up a lot of bandwidth to the rest of the application because you can pre-load all of your user's information once instead of hitting the database multiple times with more connections.

More advanced usage of sockets will be explored in Chapter 10. Because the application is more or less complete at this point, you can start adding interesting real-time activities without incurring a lot of extra overhead. Even though the website will be more interactive and alive-feeling, the enhanced experience won't cause a huge drain on your server's resources.

When I first refactored the class in Example 9-20, I grouped the route imports with the model imports, and got all kinds of wacky errors. It's important to configure the `express.bodyParser()` middleware before defining your application routes, or else your POST requests will arrive with empty parameter lists since Express will not have evaluated the page request body containing the form fields until after the page request has already responded to the user.

Example 9-20. app.js

```
var express     = require('express');
var http        = require('http');
var nodemailer  = require('nodemailer');
var MemoryStore = require('connect').session.MemoryStore;
var app         = express();
var dbPath      = 'mongodb://localhost/nodebackbone';
var fs          = require('fs');

// Create an http server
app.server      = http.createServer(app);

// Create a session store to share between methods
app.sessionStore = new MemoryStore();

// Import the data layer
var mongoose = require('mongoose');
var config = {
  mail: require('./config/mail')
```

```
};

// Import the models
var models = {
  Account: require('./models/Account')(config, mongoose, nodemailer)
};

app.configure(function(){
  app.sessionSecret = 'SocialNet secret key';
  app.set('view engine', 'jade');
  app.use(express.static(__dirname + '/public'));
  app.use(express.limit('1mb'));
  app.use(express.bodyParser());
  app.use(express.cookieParser());
  app.use(express.session({
    secret: app.sessionSecret,
    key: 'express.sid',
    store: app.sessionStore
  }));
  mongoose.connect(dbPath, function onMongooseError(err) {
    if (err) throw err;
  });
});

// Import the routes
fs.readdirSync('routes').forEach(function(file) {
  if ( file[0] == '.' ) return;
  var routeName = file.substr(0, file.indexOf('.'));
  require('./routes/' + routeName)(app, models);
});

app.get('/', function(req, res){
  res.render('index.jade');
});

app.post('/contacts/find', function(req, res) {
  var searchStr = req.param('searchStr', null);
  if ( null == searchStr ) {
    res.send(400);
    return;
  }

  models.Account.findByString(searchStr, function onSearchDone(err,accounts) {
    if (err || accounts.length == 0) {
      res.send(404);
    } else {
      res.send(accounts);
    }
  });
});
```

```
// New in Chapter 9 - the server listens, instead of the app
app.server.listen(8080);
console.log('Listening on port 8080');
```

The application's usability depends on a well-formatted user interface. In order to keep the chat system as unobtrusive as possible, I've docked the chat window to the lower righthand quadrant of the screen, as shown in Example 9-21. Each chat session has a fixed width and height and is set to float left—as new chat sessions are created, they will automatically appear next to the existing sessions moving left across the screen.

Example 9-21. public/styles/styles.css

```
form {
  width: 400px;
}

#chat form {
  width: auto;
}

#chat {
  position: absolute;
  right: 0;
  bottom: 0;
}

.chat_list {
  float: right;
  border: 1px solid black;
  list-style-type: none;
  overflow: auto;
  width: 120px;
  height: 300px;
  margin: 0;
  padding: 0;
}

.chat_list li {
  width: 100%;
  padding: 10px 0;
  background-color: #0099ff;
}

.chat_list li:nth-child(odd) {
  background-color: #80ccff;
}

.chat_list span {
  margin: 10px;
}
```

```
.chat_session {
  float: left;
  width: 250px;
  height: 300px;
}
```

This is the first time styles have been introduced to this otherwise rather ugly website. Showing off some mind-bending CSS3 skills, let's add a bit of visual flare to the list of contacts shown in the chat window. Every line will have a blue background behind the contact's name, but because of the `:nth-child(odd)` pseudo selector, every other contact will have a brighter blue background.

Example 9-22 shows the final version of *accounts.js*. This is the new routes file that was created to handle all of the account-related endpoints in the Express application. Storing them in a grouped file like this makes managing and changing the application simpler —it's easier to remember where in the file structure you need to look to make updates and add functionality, and it provides a cleaner, self-documenting directory structure.

Example 9-22. routes/accounts.js

```
module.exports = function(app, models) {
  app.get('/accounts/:id/contacts', function(req, res) {
    var accountId = req.params.id == 'me'
                      ? req.session.accountId
                      : req.params.id;
    models.Account.findById(accountId, function(account) {
      res.send(account.contacts);
    });
  });

  app.get('/accounts/:id/activity', function(req, res) {
    var accountId = req.params.id == 'me'
                      ? req.session.accountId
                      : req.params.id;
    models.Account.findById(accountId, function(account) {
      res.send(account.activity);
    });
  });

  app.get('/accounts/:id/status', function(req, res) {
    var accountId = req.params.id == 'me'
                      ? req.session.accountId
                      : req.params.id;
    models.Account.findById(accountId, function(account) {
      res.send(account.status);
    });
  });

  app.post('/accounts/:id/status', function(req, res) {
    var accountId = req.params.id == 'me'
                      ? req.session.accountId
```

```
                    : req.params.id;
  models.Account.findById(accountId, function(account) {
    status = {
      name: account.name,
      status: req.param('status', '')
    };
    account.status.push(status);

    // Push the status to all friends
    account.activity.push(status);
    account.save(function (err) {
      if (err) {
        console.log('Error saving account: ' + err);
      }
    });
  });
  res.send(200);
});

app.delete('/accounts/:id/contact', function(req,res) {
  var accountId = req.params.id == 'me'
                    ? req.session.accountId
                    : req.params.id;
  var contactId = req.param('contactId', null);

  // Missing contactId, don't bother going any further
  if ( null == contactId ) {
    res.send(400);
    return;
  }

  models.Account.findById(accountId, function(account) {
    if ( !account ) return;
    models.Account.findById(contactId, function(contact,err) {
      if ( !contact ) return;

      models.Account.removeContact(account, contactId);
      // Kill the reverse link
      models.Account.removeContact(contact, accountId);
    });
  });

  // Note: Not in callback - this endpoint returns immediately and
  // processes in the background
  res.send(200);
});

app.post('/accounts/:id/contact', function(req,res) {
  var accountId = req.params.id == 'me'
                    ? req.session.accountId
                    : req.params.id;
  var contactId = req.param('contactId', null);
```

```
    // Missing contactId, don't bother going any further
    if ( null == contactId ) {
      res.send(400);
      return;
    }

    models.Account.findById(accountId, function(account) {
      if ( account ) {
        models.Account.findById(contactId, function(contact) {
          models.Account.addContact(account, contact);

          // Make the reverse link
          models.Account.addContact(contact, account);
          account.save();
        });
      }
    });

    // Note: Not in callback - this endpoint returns immediately and
    // processes in the background
    res.send(200);
  });

  app.get('/accounts/:id', function(req, res) {
    var accountId = req.params.id == 'me'
                      ? req.session.accountId
                      : req.params.id;
    models.Account.findById(accountId, function(account) {
      if ( accountId == 'me'
           || models.Account.hasContact(account, req.session.accountId) ) {
        account.isFriend = true;
      }
      res.send(account);
    });
  });
}
```

As with every other route, the accounts route is set up in a CommonJS module format: all of the functionality is contained within this file and the rest of the application is unable to access any of its functions directly because they are not exported. Instead, the export statement acts as a `setup` function—the application loads this file, providing a reference to itself and the list of models, and the routes are set up and added to Express using Express's own `post` and `get` functions to register each route.

Example 9-23 shows how the authentication-related routes were refactored into a standalone routing file, instead of continuing to load directly from the application's *app.js* JavaScript file. Like the accounts route in Example 9-22, the authentication file provides an exported function that handles registration of all of the authentication-related routes during Express's setup.

Example 9-23. routes/authentication.js

```
module.exports = function(app, models) {
  app.post('/login', function(req, res) {
    var email = req.param('email', null);
    var password = req.param('password', null);

    if ( null == email || email.length < 1
        || null == password || password.length < 1 ) {
      res.send(400);
      return;
    }

    models.Account.login(email, password, function(account) {
      if ( !account ) {
        res.send(401);
        return;
      }
      req.session.loggedIn = true;
      req.session.accountId = account._id;
      res.send(200);
    });
  });

  app.post('/register', function(req, res) {
    var firstName = req.param('firstName', '');
    var lastName = req.param('lastName', '');
    var email = req.param('email', null);
    var password = req.param('password', null);

    if ( null == email || email.length < 1
         || null == password || password.length < 1 ) {
      res.send(400);
      return;
    }

    models.Account.register(email, password, firstName, lastName);
    res.send(200);
  });

  app.get('/account/authenticated', function(req, res) {
    if ( req.session && req.session.loggedIn ) {
      res.send(200);
    } else {
      res.send(401);
    }
  });

  app.post('/forgotpassword', function(req, res) {
    var hostname = req.headers.host;
    var resetPasswordUrl = 'http://' + hostname + '/resetPassword';
    var email = req.param('email', null);
```

```
    if ( null == email || email.length < 1 ) {
      res.send(400);
      return;
    }

    models.Account.forgotPassword(email, resetPasswordUrl, function(success){
      if (success) {
        res.send(200);
      } else {
        // Username or password not found
        res.send(404);
      }
    });
  });

  app.get('/resetPassword', function(req, res) {
    var accountId = req.param('account', null);
    res.render('resetPassword.jade', {locals:{accountId:accountId}});
  });

  app.post('/resetPassword', function(req, res) {
    var accountId = req.param('accountId', null);
    var password = req.param('password', null);
    if ( null != accountId && null != password ) {
      models.Account.changePassword(accountId, password);
    }
    res.render('resetPasswordSuccess.jade');
  });
}
```

None of these functions can be directly accessed from outside this file because they are not included in the export list; however, since all the functions are Express `setup` commands, there will be no need to come back to this code in other modules. All interaction will happen against the model objects, which are shared between every route in Express.

Finally, Example 9-24 contains the unabridged *chat.js* functionality: this is the Socket.io server that will provide access to the chat functionality created throughout Chapter 9.

Example 9-24. routes/chat.js

```
module.exports = function(app, models) {
  var io = require('socket.io');
  var utils = require('connect').utils;
  var cookie = require('cookie');
  var Session = require('connect').middleware.session.Session;

  var sio = io.listen(app.server);

  sio.configure(function() {
    sio.set('authorization', function( data, accept) {
      var signedCookies = cookie.parse(data.headers.cookie);
      var cookies = utils.parseSignedCookies(signedCookies,app.sessionSecret);
```

```
      data.sessionID = cookies['express.sid'];
      data.sessionStore = app.sessionStore;
      data.sessionStore.get(data.sessionID, function(err, session) {
        if ( err || !session ) {
          return accept('Invalid session', false);
        } else {
          data.session = new Session(data, session);
          accept(null, true);
        }
      });
    });

    sio.sockets.on('connection', function(socket) {
      var session = socket.handshake.session;
      var accountId = session.accountId;
      socket.join(accountId);

      socket.on('chatclient', function(data) {
        sio.sockets.in(data.to).emit('chatserver', {
          from: accountId,
          text: data.text
        });
      });
    });
  });
}
```

Pay close attention to the order of embedding for the events and functions inside the Socket.io calls. When a chat is sent from the client to the server, the server responds by emitting an ever thorough `sio.sockets`, not through the `socket` object that was created upon connection. This is an important fact: although you should think of every chat as being between two people, in a purely technical sense the chat is performed from one person (the sender) to one or more people (the receivers).

There is no single sign-on concept built into the functionality of this application. If you have two web browsers, if you have a phone, or if you have shared your password, it is entirely possible that your account can be accessed from multiple locations at the same time. If that is the case, any chat sent to your account will be instantly received by everyone logged into your account. So while the communication is from one account to one other account, the actual chat will be sent from one web browser to an unknown number of other web browsers.

CHAPTER 10

Activities in Real Time

We are going to be using Node.js's events library to listen for friend changes from within our Socket.io channel. This gets us around having to subscribe to multiple channels (which isn't possible) and opens an avenue for scaling out the chat function using something like Redis or RabbitMQ.

This command tells you how many sockets are in a room:

```
var clients = io.sockets.clients(nick.room)
```

Adding Custom Events

In Chapter 2 you learned about the events library that ships with Node.js. Events are at the core of JavaScript's power and Node.js makes it easy to create, trigger, and consume everything from I/O progress to user input to custom actions you define yourself in your functions.

Why are custom events so important for real-time notification in the social networking application? After all, you could very easily trigger an event every time someone logs in, updates his status, or comments on someone else's profile, and let the event handler decide which connected sockets should see the event.

Because Node.js is single-threaded, any logic you do in your event handler will effectively block your running code. So while it might be fast to process a login event if you have only two connections open from your development computer, if you have a real running site your server will be spending a lot of time going through all of the connections to figure out who a particular message should be sent to. That would be a lot of wasted time when it is just as easy to filter the message to only those listeners who would be interested in it.

Triggering Events

Every message triggered by a user event will be named `event:[userid]` and contain at least two properties: `from` and `action`. So if user 539 logs into the system, this could trigger an event called `event:539` with an `action` value of `login` and a `from` value of `539`. Since the event will contain a JavaScript object, you can include as many extra parameters you want, but the unofficial contract between Backbone.js and Node.js will be an `AccountID` in the event name and at least an action property so Backbone will know how to react to the event.

In Example 10-1, an `EventEmitter` object is created at the same time as the overall Express `app` instance. Rather than just adding the new dispatcher to the app object so it can be used by the routes, I've added three functions to expose the `eventDispatcher` to the rest of the application. You can access `triggerEvent` from anywhere in the application to raise an event to all the listeners attached to the `eventDispatcher`. Use `addEventListener` to add a function as a listener, and if you're adding a listener that will be going away—such as a connected user who logs off the system and no longer needs notifications—use `removeEventListener` with the same parameters as the corresponding `addEventListener` call.

Example 10-1. Adding an event dispatcher

```
var events      = require('events');

// Create an event dispatcher
var eventDispatcher = new events.EventEmitter();
app.addEventListener = function ( eventName, callback ) {
  eventDispatcher.on(eventName, callback);
};
app.removeEventListener = function( eventName, callback ) {
  eventDispatcher.removeListener( eventName, callback );
};
app.triggerEvent = function( eventName, eventOptions ) {
  eventDispatcher.emit( eventName, eventOptions );
};
```

Adding Listeners

After logging into the application and initiating a Socket.io connection, you need a way to be informed of changes to your contacts' data. To get notified of these changes, Node.js will subscribe to the events sent by each of your friends when you log in.

Example 10-2 shows how to sort through the connected user's contact list and add an event listener for each contact. This work happens immediately after connecting to the socket and has the additional benefit that loading the user's account and storing it in a local variable keeps it primed in memory and means you never have to go back to the database.

Example 10-2. event_login.js

```
var handleContactEvent = function(eventMessage) {
  socket.emit(eventName, eventMessage);
}

var subscribeToAccount = function(accountId) {
  var eventName = 'event:' + accountId;
  app.addEventListener(eventName, handleContactEvent);
  console.log('Subscribing to ' + eventName);
};

models.Account.findById(accountId, function subscribeToFriendFeeds(account) {
  var subscribedAccounts = {};
  sAccount = account;
  account.contacts.forEach(function(contact) {
    if ( !subscribedAccounts[contact.accountId]) {
      subscribeToAccount(contact.accountId);
      subscribedAccounts[contact.accountId] = true;
    }
  });

  if (!subscribedAccounts[accountId]) {
    // Subscribe to my own updates
    subscribeToAccount(accountId);
  }
});
```

The `subscribedAccounts` variable is a dictionary object that provides a map of which accounts the socket has created listeners for. This serves a similar function to the dictionary used to make the chat sessions in Chapter 9: by creating a key for the `accountId` every time an event listener is added and checking for that ID in the next iteration of the `forEach` loop, you avoid problems that would arise if your account listed the same contact more than once. After checking all the accounts, the `subscribedAccounts` list is checked for your current `accountId`, and if you have not yet subscribed to your own events, `subscribeToAccount` is triggered so you will receive updates from your own account.

The `handleContactEvent` function funnels incoming events down to the socket. From that point on it will be up to the Backbone.js UI on the client to figure out what needs to be displayed to the subscribed user.

When the socket is disconnected, loop through the user's contact list again but this time remove the event listeners corresponding to each `accountID`, as shown in Example 10-3. Removing the event listeners lets the callback function get picked up by the garbage collector, and prevents the unnecessary processing that would otherwise continue to happen when those events were raised.

Example 10-3. event_logout.js

```
socket.on('disconnect', function() {
  sAccount.contacts.forEach(function(contact) {
    var eventName = 'event:' + contact.accountId;
    app.removeEventListener(eventName, handleContactEvent);
    console.log('Unsubscribing from ' + eventName);
  });
});
```

Contact Login Notification

In Chapter 9 you added a persistent contact list that stays onscreen as long as your user is connected to the website. Although it's possible to send messages to your contacts in real time, you can never be sure if they are online and able to receive your communication.

Backbone.js

The user interface for contact login notifications will be minimal but clear; when a contact is offline, he will have a red "stop light" circle icon next to his name, and when the contact is online, the circle's color will change to green (see Figure 10-1).

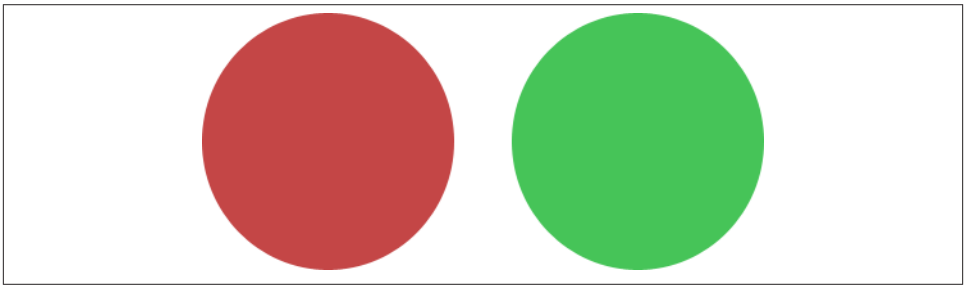


Figure 10-1. Online/offline traffic light indicators

The icon should be small enough to fit comfortably on a line of text—when I created mine, I gave it a maximum height of 15 pixels. Each circle will have a diameter of 15 pixels, so when both are saved side-by-side in one image, the final image size became 30 pixels wide by 15 pixels high.

The online indicator will be a fixed-size HTML element with the traffic light indicator as a background image with CSS, as shown in Example 10-4. Because the image is larger than the HTML element it is contained in, the visual effect will be a red light. When the class `online` is added to the element, the CSS engine will switch the background-position property so the top left of the visible image will start with a 15-pixel offset. The HTML element does not change sizes, so the visual effect will be that of the light changing from

red (offline) to green (online). This effect is known as a *CSS sprite*. Sprites combine multiple images into a single file so they can be retrieved by web browsers in a single request. So when your contact comes online, his indicator circle will switch to green without needing to download a special "green light" icon.

Example 10-4. Online indicator CSS

```
.online_indicator {
  float: left;
  width: 15px;
  height: 16px;
  background-image: url('/images/trafficlight.png');
}

.online_indicator.online {
  background-position: -15px 0;
}
```

The `handleContactEvent` function in Example 10-5 will accept contact events from Node.js and dispatch to them the custom event emitter that was added to Backbone in Chapter 9. When the socket is connected, bind to the `contactEvent` message and point to the `handleContactEvent` function. Remember from earlier in the chapter that each even contains at least an `action` and a `from` property, so you can build a unique event string and emit that to your application. Each view can listen for events from specific contacts and react to changes from them, such as changes to the contact's login status that you are about to implement.

Example 10-5. Adding contact events to the socket mediator

```
socket.on('contactEvent', handleContactEvent);


var handleContactEvent = function(eventObj) {
  var eventName = eventObj.action + ':' + eventObj.from;
  eventDispatcher.trigger(eventName, eventObj);
};
```

Example 10-6 contains the new event bindings for *chatitem.js* and *chatsession.js*. When these classes are initialized they will now listen for events named "login" and "logout" associated with their underlying model. When a contact comes online, the `handleContactLogin` function springs into action and adds the `online` class to the online indicator. This will cause the CSS to switch the red status indicator circle to green onscreen. When a contact goes offline, the `handleContactLogout` function reverses the effect: it removes the `online` class from the indicator icon, which will cause the graphic to switch back to the red offline visual circle.

Example 10-6. Reacting to online status

```
initialize: function(options) {
  var accountId = this.model.get('accountId');
  options.socketEvents.bind(
    'login:' + accountId,
    this.handleContactLogin,
    this
  );
  options.socketEvents.bind(
    'logout:' + accountId,
    this.handleContactLogout,
    this
  );
  options.socketEvents.bind(
    'socket:chat:start:' + accountId,
    this.startChatSession,
    this
  );
},

handleContactLogin: function() {
  this.model.set('online', true);
  this.$el.find('.online_indicator').addClass('online');
},

handleContactLogout: function() {
  this.model.set('online', false);
  $onlineIndicator = this.$el.find('.online_indicator');
  while ( $onlineIndicator.hasClass('online') ) {
    $onlineIndicator.removeClass('online');
  }
}

render: function() {
  this.$el.html(_.template(chatItemTemplate, {
    model: this.model.toJSON()
  }));
  if ( this.model.get('online') ) this.handleContactLogin();
  return this;
}
```

Also new in this example is the model's `online` property. In most cases, the contact list will have been downloaded before the Socket.io connect process completes. When that happens, Backbone will use the new `online` property from the connect list to determine whether or not the contact's visual indicator should show him as online or offline.

Node.js

When someone connects to the Socket.io server, two event types need to be triggered. First, the user's presence is announced to all of his contacts. Second, all of the user's contacts who have already logged in need to be communicated back to the user.

When a user connects to the Socket.io server, he will now trigger an event against his own `accountId`, as shown in Example 10-7. This will notify all of his contacts that he logged in by changing his online indicator icon in his contacts' Backbone.js UI.

Example 10-7. Emitting login events: snippets/login_connect.js

```
app.triggerEvent('event:' + accountId, {
  from: accountId,
  action: 'login'
});
```

Upon the Socket.io disconnection, Node.js will now trigger an account logout event, as shown in Example 10-8. This will be received by everyone who is subscribed to the current user and cause their indicator icons to show the user as offline.

Example 10-8. Emitting logout events: snippets/login_disconnect.js

```
socket.on('disconnect', function() {
  app.triggerEvent('event:' + accountId, {
    from: accountId,
    action: 'logout'
  });
});
```

Example 10-9 introduces a virtual property called `online` to the contact schema in the account model's Mongoose schema definition. A virtual property is temporary, never stored to the database, and by default is not shown in JSON results. The `online` property is given a `get` function, which triggers the application to check the connected sockets to see whether or not the given contact is currently online.

Example 10-9. Adding a virtual property to the account model

```
var schemaOptions = {
  toJSON: {
    virtuals: true
  },
  toObject: {
    virtuals: true
  }
};

var Contact = new mongoose.Schema({
  name: {
    first:    { type: String },
    last:     { type: String }
```

```
  },
  accountId: { type: mongoose.Schema.ObjectId },
  added:     { type: Date },     // When the contact was added
  updated:   { type: Date }      // When the contact last updated
}, schemaOptions);

Contact.virtual('online').get(function(){
  return app.isAccountOnline(this.get('accountId'));
});
```

The `isAccountOnline` function in Example 10-10 counts the number of sockets connected to the room whose name matches the supplied `accountId`. By adding this function to the application, it makes this functionality available to the rest of the application—you can check whether or not a user is currently logged into the specific account from anywhere else in your application code.

Example 10-10. Determining if an account is online

```
app.isAccountOnline = function(accountId) {
  var clients = sio.sockets.clients(accountId);
  return (clients.length > 0);
};
```

Status Updates

There are a few points of contact where you can see a contact's activity stream. The activity stream so far has been a cold thing that required manually updating the page in order to get new updates. Even if you add a status update yourself, your own update will appear on screen but you will not see updates added by other people since you came to the activity list.

Using the lessons and framework built for the chat notifications, it's time to enhance the activity stream with real-time updates.

Backbone.js

Back in Example 10-5 you added a generic handler to listen for all events coming from the server, which can be consumed on a per-contact basis by individual views. Now it will be straightforward to bring that same logic into other parts of the application, starting with the index page you come to when you first log into the site.

First, add a reference to the `socketEvents` object to the router when the index view is instantiated, as shown in Example 10-11.

Example 10-11. Adding the socketEvents object to the index view

```
index: function() {
  var statusCollection = new StatusCollection();
  statusCollection.url = '/accounts/me/activity';
```

```
    this.changeView(new IndexView({
      collection: statusCollection,
      socketEvents:this.socketEvents
    }));
    statusCollection.fetch();
  },
```

The `collection` object was already being passed to the new `IndexView` when it was instantiated, but now the `socketEvents` object is also sent. This will get picked up in the custom `initialization` function and used by the view when it is rendered by the web browser.

The `IndexView`'s `initialize` function in Example 10-12 has been modified to include `options` as a `function` parameter. The `options` parameter is always available to initialization methods in Backbone.js but most of the classes you have used so far haven't needed them. JavaScript does not match function calls to the expected parameter counts like many other languages, so if you include an extra, or too few, parameters in your function declaration, the unused variables will contain `null` values.

Example 10-12. Listening for socket events on the index view

```
  initialize: function(options) {
    options.socketEvents.bind('status:me', this.onSocketStatusAdded, this );
    this.collection.on('add', this.onStatusAdded, this);
    this.collection.on('reset', this.onStatusCollectionReset, this);
  },
```

When the view is initialized it adds a listener to the event named `status:me`. Whenever events with the status type originating from the currently logged-in account come through the dispatcher, the function `onSocketStatusAdded` will take effect and add the new status to the current index view.

The `onSocketStatusAdded` function's purpose in Example 10-13 is to take the incoming data payload coming in from Socket.io and convert it to a `Status` model, then add it to the view's collection object. Remember that the index view has an event `onStatusAdded` that handles the work of rendering the new status and displaying it onscreen, so `onSocketstatusAdded` needs to create a `Status`, throw it at the collection, and not worry about the work of displaying the status to your user.

Example 10-13. Handling socket events on the index view

```
  onSocketStatusAdded: function(data) {
    var newStatus = data.data;
    this.collection.add(new Status({status:newStatus.status,name:newStatus.name}))
  },
```

The actual contents of the status message will come in the `data` property of the Socket.io payload. To make the status a bit easier to work with, I referenced the `data` property into the `newStatus` variable. That way I can access its contents through `newStatus` rather than having to type the full path (`data.data`).

Just like the index view, the `socketEvents` object is added to the profile view in Example 10-14. The creation of the account model object is unaffected, nor is the method by which the account is loaded asynchronously to the activation of the profile View. This is a bit different from the index view, because you will need to consider the lifecycle of the account model in order to properly handle socket events for that account.

Example 10-14. Adding the socketEvents object to the profile view

```
profile: function(id) {
  var model = new Account({id:id});
  this.changeView(new ProfileView({model:model,
                                   socketEvents:this.socketEvents}));
  model.fetch();
},
```

The initialization method in Example 10-15 now includes an `options` parameter and creates a local variable for the `socketEvents`, saving it to the profile's instance properties so it can be used later on. You can't listen directly for changes to the account at this stage because nine times out of ten your account object will not have loaded over the network yet, so you won't have an account ID to bind against. You could always pass this in from the router, but in this case, it's going to be more effective to pull the ID from the account after it finally arrives.

Example 10-15. Getting ready to listen for socket events on the profile view

```
initialize: function (options) {
  this.socketEvents = options.socketEvents;
  this.model.bind('change', this.render, this);
},
```

Example 10-16 demonstrates how you can add a listener for socket events from the account you are viewing when the profile page is rendered. Because changes to the account model trigger the render process, you need to check to see whether the model has a valid `_id` field yet. In most cases, the `render` function will run twice: first, when the view is created and `render` is called by the router, and second, when the account model finishes loading the profile will render again, at which point you will finally have an accountId to listen for.

Example 10-16. Listening for socket events on the profile view

```
render: function() {
  if ( this.model.get('_id')) {
    this.socketEvents.bind('status:' + this.model.get('_id'),
                           this.onSocketStatusAdded, this );
  }
```

Finally, when a new status arrives through Socket.io, add it to the profile list by calling the pre-existing `prependStatus` function that was created earlier, as shown in Example 10-17. This function was previously used in the `render` function as well as on the form postback when you added a status to the profile. Now take it out of the form postback and only include it during the `onSocketStatusAdded`. Whenever you or anyone else adds a status to the profile, the update will appear onscreen for everyone who might be currently looking at this user's profile.

Example 10-17. Handling socket events on the profile view

```
onSocketStatusAdded: function(data) {
  var newStatus = data.data;
  this.prependStatus(new Status({
    status:newStatus.status,name:newStatus.name
  }))
},
```

Node.js

When you first added the status update endpoint to the Express server, the "success" response was sent to the client right away while the server went off and actually processed the status. This allowed your user to see the update appear almost right away—the only prerequisite was acknowledgement from the server that the payload was received. It is safe to assume that the status will eventually end up in the target account, but there is no need for your user to waste time waiting on database I/O.

That doesn't happen anymore; Backbone will now display status updates as they happen regardless of where they originate. Updates will no long appear on screen as a direct result of hitting the Submit button in your web browser.

Even though Express's quick response-then-process flow isn't used to trigger an immediate onscreen update, it is still valuable from the server's point of view. By ending the response immediately, your computer is able to take on more connections even while Node.js is processing the previous requests in the background.

Example 10-18 adds a `triggerEvent` call when the status is successfully saved into the target account. This works out for the best of all worlds because the request finishes right away rather than blocking, everyone connected to the server receives the status

update as soon as it is applied, and the update is sent out only after a successful database insert. This is better than before: when the status was displayed to the sending user, there was a chance that if the status was in fact not saved, his view of the account would be out of sync with what the rest of the world sees.

Example 10-18. snippets/express_status.js

```
app.post('/accounts/:id/status', function(req, res) {
  var accountId = req.params.id == 'me'
                    ? req.session.accountId
                    : req.params.id;
  models.Account.findById(accountId, function(account) {
    status = {
      name: account.name,
      status: req.param('status', '')
    };
    account.status.push(status);

    // Push the status to all friends
    account.activity.push(status);
    account.save(function (err) {
      if (err) {
        console.log('Error saving account: ' + err);
      } else {
          app.triggerEvent('event:' + accountId, {
            from: accountId,
            data: status,
            action: 'status'
          });
      }
    });
  });
  res.send(200);
});
```

Putting It Together

Although this chapter didn't introduce any new classes or components, it enhanced much of the existing functionality that you have built so far. The full versions of the changed source files will be presented in this section along with explanations of what changed and why certain functionality was added to specific areas of code instead of others.

Backbone.js

In this chapter Backbone took on a major role, ensuring that messages are passed to and from the backend server and displayed in real time. At last, the frontend JavaScript is closely married to the backend JavaScript thanks to help from the Socket.io library.

The `socketEvents` object added to the router in Chapter 9 is shared among more of the views in Example 10-19. The `index` and `profile` views now require the `socketEvent`, so it is included in the constructor for those routes.

Example 10-19. public/js/router.js

```
define(['views/index', 'views/register', 'views/login',
        'views/forgotpassword', 'views/profile', 'views/contacts',
        'views/addcontact', 'models/Account', 'models/StatusCollection',
        'models/ContactCollection'],
function(IndexView, RegisterView, LoginView, ForgotPasswordView, ProfileView,
         ContactsView, AddContactView, Account, StatusCollection,
         ContactCollection) {
  var SocialRouter = Backbone.Router.extend({
    currentView: null,

    socketEvents: _.extend({}, Backbone.Events),

    routes: {
      'addcontact': 'addcontact',
      'index': 'index',
      'login': 'login',
      'register': 'register',
      'forgotpassword': 'forgotpassword',
      'profile/:id': 'profile',
      'contacts/:id': 'contacts'
    },

    changeView: function(view) {
      if ( null != this.currentView ) {
        this.currentView.undelegateEvents();
      }
      this.currentView = view;
      this.currentView.render();
    },

    index: function() {
      var statusCollection = new StatusCollection();
      statusCollection.url = '/accounts/me/activity';
      this.changeView(new IndexView({
        collection: statusCollection,
        socketEvents:this.socketEvents
      }));
      statusCollection.fetch();
    },

    addcontact: function() {
      this.changeView(new AddContactView());
    },

    login: function() {
      this.changeView(new LoginView({socketEvents:this.socketEvents}));
```

```
    },

    forgotpassword: function() {
      this.changeView(new ForgotPasswordView());
    },

    register: function() {
      this.changeView(new RegisterView());
    },

    profile: function(id) {
      var model = new Account({id:id});
      this.changeView(new ProfileView({model:model, socketEvents:this.socketEvents}));
      model.fetch();
    },

    contacts: function(id) {
      var contactId = id ? id : 'me';
      var contactsCollection = new ContactCollection();
      contactsCollection.url = '/accounts/' + contactId + '/contacts';
      this.changeView(new ContactsView({
        collection: contactsCollection
      }));
      contactsCollection.fetch();
    }
  });

  return new SocialRouter();
});
```

The only change to the `SocialNet` class in Example 10-20 is the addition of the login results contained in the `data` property send to the `app:loggedin` event. This will contain the logged-in user's `accountId`, which the Socket.io event handler needs to figure out when events are coming in that are related to the current user versus an outside contact.

Example 10-20. public/js/SocialNet.js

```
define(['router', 'SocialNetSockets'], function(router, socket) {
  var initialize = function() {
    socket.initialize(router.socketEvents);
    checkLogin(runApplication);
  };

  var checkLogin = function(callback) {
    $.ajax("/account/authenticated", {
      method: "GET",
      success: function(data) {
        router.socketEvents.trigger('app:loggedin', data);
        return callback(true);
      },
      error: function(data) {
        return callback(false);
```

```
      }
    });
  };

  var runApplication = function(authenticated) {
    if (authenticated) {
      window.location.hash = 'index';
    } else {
      window.location.hash = 'login';
    }
    Backbone.history.start();
  };

  return {
    initialize: initialize
  };
});
```

Before this, the `accountId` was never explicitly shared with Backbone: all user-related functionality was handled by Node.js. Now Backbone is more aware of the account belonging to the user who is logged in.

The `SocialNetSockets` class in Example 10-21 adds awareness of events originating from contacts who are using the system in real time via the `contactEvent` binding inside the socket's connect callback. The new `handleContactEvent` function translated events from your contacts into triggers for consumption by the rest of the client UI. Each event coming from Node.js will contain an `accountId` and an action—event names are derived by concatenating these values. This allows you to precisely control the filters for events coming into your application so individual views can focus on specific user events and not spend time filtering through messages that do not apply to them.

Example 10-21. public/js/SocialNetSockets.js

```
define(['Sockets', 'models/contactcollection', 'views/chat'],
function(sio, ContactCollection, ChatView) {
  var SocialNetSockets = function(eventDispatcher) {
    var accountId = null;

    var socket = null;

    var connectSocket = function(socketAccountId) {
      accountId = socketAccountId;
      socket = io.connect();

      socket
        .on('connect_failed', function(reason) {
          console.error('unable to connect', reason);
        })
        .on('connect', function() {
          eventDispatcher.bind('socket:chat', sendChat);
          socket.on('chatserver', function(data) {
```

```
            eventDispatcher.trigger('socket:chat:start:' + data.from );
            eventDispatcher.trigger('socket:chat:in:' + data.from, data);
          });

          socket.on('contactEvent', handleContactEvent);

          var contactsCollection = new ContactCollection();
          contactsCollection.url = '/accounts/me/contacts';
          new ChatView({collection: contactsCollection,
                        socketEvents: eventDispatcher}).render();
          contactsCollection.fetch();
        });
    };

    var handleContactEvent = function(eventObj) {
      var eventName = eventObj.action + ':' + eventObj.from;
      eventDispatcher.trigger(eventName, eventObj);

      if ( eventObj.from == accountId ) {
        eventName = eventObj.action + ':me';
        eventDispatcher.trigger(eventName, eventObj);
      }
    };

    var sendChat = function(payload) {
      if ( null != socket ) {
        socket.emit('chatclient', payload);
      }
    };

    eventDispatcher.bind('app:loggedin', connectSocket);
  }

  return {
    initialize: function(eventDispatcher) {
      SocialNetSockets(eventDispatcher);
    }
  };
});
```

The `ChatItem` class in Example 10-22 is now aware of logins and logouts by all of the connected contacts. When it initializes, the view attached to the login and logout events compares against the model's `accountId`. When the contact logs in, `handleContactLogin` adds the `online` CSS class to the `online_indicator` graphic, causing the traffic light graphic to switch to green. When the contact logs out, `handleContactLogout` removes all of the `online` class entries from `online_indicator`, which causes the CSS engine to reset the background-position property of the graphic back to the default red position, indicating that the contact is offline.

Example 10-22. public/js/views/chatitem.js

```
define(['SocialNetView', 'text!templates/chatitem.html'],
function(SocialNetView, chatItemTemplate) {
  var chatItemView = SocialNetView.extend({
    tagName: 'li',

    $el: $(this.el),

    events: {
      'click': 'startChatSession',
    },

    initialize: function(options) {
      var accountId = this.model.get('accountId');
      options.socketEvents.bind(
        'login:' + accountId,
        this.handleContactLogin,
        this
      );
      options.socketEvents.bind(
        'logout:' + accountId,
        this.handleContactLogout,
        this
      );
      options.socketEvents.bind(
        'socket:chat:start:' + accountId,
        this.startChatSession,
        this
      );
    },

    handleContactLogin: function() {
      this.model.set('online', true);
      this.$el.find('.online_indicator').addClass('online');
    },

    handleContactLogout: function() {
      this.model.set('online', false);
      $onlineIndicator = this.$el.find('.online_indicator');
      while ( $onlineIndicator.hasClass('online') ) {
        $onlineIndicator.removeClass('online');
      }
    },

    startChatSession: function() {
      this.trigger('chat:start', this.model);
    },

    render: function() {
      this.$el.html(_.template(chatItemTemplate, {
        model: this.model.toJSON()
      }));
```

```
      if ( this.model.get('online') ) this.handleContactLogin();
      return this;
    }
  });

  return chatItemView;
});
```

The chat item in Example 10-23 has been outfitted with `handleContactLogin` and `handleContactLogout` to deal with contacts logging in and out while a user is connected. When the contact's login or logout event is raised, his status indicator will change colors to indicate the new connected state. The actual work of changing the color of the image is handled in CSS, but the jQuery functions `addClass` and `removeClass` called from Backbone.js cause the CSS settings to change dynamically in the web browser.

Example 10-23. public/js/views/chatsession.js

```
define(['SocialNetView', 'text!templates/chatsession.html'],
function(SocialNetView, chatItemTemplate) {
  var chatItemView = SocialNetView.extend({
    tagName: 'div',

    className: 'chat_session',

    $el: $(this.el),

    events: {
      'submit form': 'sendChat'
    },

    initialize: function(options) {
      this.socketEvents = options.socketEvents;
      var accountId = this.model.get('accountId');
      this.socketEvents.on('socket:chat:in:' + accountId, this.receiveChat, this);
      this.socketEvents.bind(
        'login:' + accountId,
        this.handleContactLogin,
        this
      );
      this.socketEvents.bind(
        'logout:' + accountId,
        this.handleContactLogout,
        this
      );
    },

    handleContactLogin: function() {
      this.$el.find('.online_indicator').addClass('online');
      this.model.set('online', true);
    },

    handleContactLogout: function() {
```

```
      this.model.set('online', false);
      $onlineIndicator = this.$el.find('.online_indicator');
      while ( $onlineIndicator.hasClass('online') ) {
        $onlineIndicator.removeClass('online');
      }
    },

    receiveChat: function(data) {
      var chatLine = this.model.get('name').first + ': ' + data.text;
      this.$el.find('.chat_log').append($('<li>' + chatLine + '</li>'));
    },

    sendChat: function() {
      var chatText = this.$el.find('input[name=chat]').val();
      if ( chatText && /[^\s]+/.test(chatText) ) {
        var chatLine = 'Me: ' + chatText;
        this.$el.find('.chat_log').append($('<li>' + chatLine + '</li>'));
        this.socketEvents.trigger('socket:chat', {
          to: this.model.get('accountId'),
          text: chatText
        });
      }
      return false;
    },

    render: function() {
      this.$el.html(_.template(chatItemTemplate, {
        model: this.model.toJSON()
      }));
      if ( this.model.get('online') ) this.handleContactLogin();
      return this;
    }
  });

  return chatItemView;
});
```

The full index view shown in Example 10-24 displays status updates when they come through the Socket.io event listener, rather than immediately when you press the post button to submit a status update to your profile. Now, whenever anyone adds a status to your profile, it will trigger the `status:me` event, which is handled by the `onSocketStatusAdded` function in this view.

Example 10-24. public/js/views/index.js

```
define(['SocialNetView', 'text!templates/index.html',
        'views/status', 'models/Status'],
function(SocialNetView, indexTemplate, StatusView, Status) {
  var indexView = SocialNetView.extend({
    el: $('#content'),

    events: {
```

```
      "submit form": "updateStatus"
    },

    initialize: function(options) {
      options.socketEvents.bind('status:me', this.onSocketStatusAdded, this );
      this.collection.on('add', this.onStatusAdded, this);
      this.collection.on('reset', this.onStatusCollectionReset, this);
    },

    onStatusCollectionReset: function(collection) {
      var that = this;
      collection.each(function (model) {
        that.onStatusAdded(model);
      });
    },

    onSocketStatusAdded: function(data) {
      var newStatus = data.data;
      var found = false;
      this.collection.forEach(function(status) {
        var name = status.get('name');
        if ( name && name.full == newStatus.name.full &&
             status.get('status') == newStatus.status ) {
          found = true;
        }
      });
      if (!found ) {
        this.collection.add(new Status({status:newStatus.status,name:newStatus.name}))
      }
    },

    onStatusAdded: function(status) {
      var statusHtml = (new StatusView({ model: status })).render().el;
      $(statusHtml).prependTo('.status_list').hide().fadeIn('slow');
    },

    updateStatus: function() {
      var statusText = $('input[name=status]').val();
      var statusCollection = this.collection;
      $.post('/accounts/me/status', {
        status: statusText
      }); // New: no longer adding to screen
      return false;
    },

    render: function() {
      this.$el.html(indexTemplate);
    }
  });

  return indexView;
});
```

The login view shown in Example 10-25 now triggers the `app:loggedin` event after a successful login, much like the application loader did in Example 10-20. This will pass your user credentials to the application listener so it can react properly to Socket.io events directed at your account.

Example 10-25. public/js/views/login.js

```
define(['SocialNetView', 'text!templates/login.html'],
function(SocialNetView, loginTemplate) {
  var loginView = SocialNetView.extend({
    requireLogin: false,

    el: $('#content'),

    events: {
      "submit form": "login"
    },

    initialize: function(options) {
      this.socketEvents = options.socketEvents;
    },

    login: function() {
      var socketEvents = this.socketEvents;
      $.post('/login',
        this.$('form').serialize(), function(data) {
          socketEvents.trigger('app:loggedin', data);
          window.location.hash = 'index';
      }).error(function(){
        $("#error").text('Unable to login.');
        $("#error").slideDown();
      });
      return false;
    },

    render: function() {
      this.$el.html(loginTemplate);
      $("#error").hide();
      $("input[name=email]").focus();
    }
  });

  return loginView;
});
```

The profile view shown Example 10-26 adds support for real-time status changes originating from Node.js. When the status event is triggered, `onSocketStatusAdded` takes care of adding the incoming text to the top of the status list.

Example 10-26. public/js/views/profile.js

```
define(['SocialNetView', 'text!templates/profile.html',
        'text!templates/status.html', 'models/Status',
        'views/Status'],
function(SocialNetView,  profileTemplate,
         statusTemplate, Status, StatusView)
{
  var profileView = SocialNetView.extend({
    el: $('#content'),

    events: {
      "submit form": "postStatus"
    },

    initialize: function (options) {
      this.socketEvents = options.socketEvents;
      this.model.bind('change', this.render, this);
    },

    postStatus: function() {
      var that = this;
      var statusText = $('input[name=status]').val();
      var statusCollection = this.collection;
      $.post('/accounts/' + this.model.get('_id') + '/status', {
        status: statusText
      });
      return false;
    },

    onSocketStatusAdded: function(data) {
      var newStatus = data.data;
      this.prependStatus(new Status({status:newStatus.status,name:newStatus.name}))
    },

    prependStatus: function(statusModel) {
      var statusHtml = (new StatusView({ model: statusModel })).render().el;
      $(statusHtml).prependTo('.status_list').hide().fadeIn('slow');
    },

    render: function() {
      if ( this.model.get('_id')) {
        this.socketEvents.bind('status:' + this.model.get('_id'),
                               this.onSocketStatusAdded, this );
      }
      var that = this;
      this.$el.html(
        _.template(profileTemplate,this.model.toJSON())
      );

      var statusCollection = this.model.get('status');
      if ( null != statusCollection ) {
        _.each(statusCollection, function (statusJson) {
```

```
          var statusModel = new Status(statusJson);
          that.prependStatus(statusModel);
        });
      }
    }
  });

  return profileView;
});
```

Node.js

In this chapter you added extra events to Node.js to support the Backbone real-time updates. Let's examine the small changes within the context of the rest of the code they modify to see how the real-time events affect the way the application is written.

The full application in Example 10-27 now contains an event dispatcher and exposes functions to add and remove event listeners from the dispatcher. Because the dispatcher is declared at this level, it is treated as a private variable and is not directly accessible to any other functions outside of the *app.js* source file. This prevents routes and models from accessing the events dispatcher directly, giving you full control over how it is used.

Example 10-27. app.js

```
var express     = require('express');
var http        = require('http');
var nodemailer  = require('nodemailer');
var MemoryStore = require('connect').session.MemoryStore;
var app         = express();
var dbPath      = 'mongodb://localhost/nodebackbone';
var fs          = require('fs');
var events      = require('events');

// Create an http server
app.server      = http.createServer(app);

// Create an event dispatcher
var eventDispatcher = new events.EventEmitter();
app.addEventListener = function ( eventName, callback ) {
  eventDispatcher.on(eventName, callback);
};
app.removeEventListener = function( eventName, callback ) {
  eventDispatcher.removeListener( eventName, callback );
};
app.triggerEvent = function( eventName, eventOptions ) {
  eventDispatcher.emit( eventName, eventOptions );
};

// Create a session store to share between methods
app.sessionStore = new MemoryStore();
```

```
// Import the data layer
var mongoose = require('mongoose');
var config = {
  mail: require('./config/mail')
};

// Import the models
var models = {
  Account: require('./models/Account')(app, config, mongoose, nodemailer)
};

app.configure(function(){
  app.sessionSecret = 'SocialNet secret key';
  app.set('view engine', 'jade');
  app.use(express.static(__dirname + '/public'));
  app.use(express.limit('1mb'));
  app.use(express.bodyParser());
  app.use(express.cookieParser());
  app.use(express.session({
    secret: app.sessionSecret,
    key: 'express.sid',
    store: app.sessionStore
  }));
  mongoose.connect(dbPath, function onMongooseError(err) {
    if (err) throw err;
  });
});

// Import the routes
fs.readdirSync('routes').forEach(function(file) {
  if ( file[0] == '.' ) return;
  var routeName = file.substr(0, file.indexOf('.'));
  require('./routes/' + routeName)(app, models);
});

app.get('/', function(req, res){
  res.render('index.jade');
});

app.post('/contacts/find', function(req, res) {
  var searchStr = req.param('searchStr', null);
  if ( null == searchStr ) {
    res.send(400);
    return;
  }

  models.Account.findByString(searchStr, function onSearchDone(err,accounts) {
    if (err || accounts.length == 0) {
      res.send(404);
    } else {
      res.send(accounts);
    }
```

```
  });
});

// New in Chapter 9 - the server listens, instead of the app
app.server.listen(8080);
console.log('Listening on port 8080');
```

A virtual property `online` is added to the account model's schema in Example 10-28. This is used in the contact list to get the initial online state for each of your contacts, but it is a virtual property because it should not be saved into MongoDB. To determine whether an account is online, check whether or not someone is subscribed to the Socket.io channel. Since you subscribe to your own channel when you log in, and no one else ever subscribes to your channel, having one or more listeners on your channel means your account is logged onto the system.

Example 10-28. models/Account.js

```
module.exports = function(app, config, mongoose, Status, nodemailer) {
  var crypto = require('crypto');

  var Status = new mongoose.Schema({
    name: {
      first:    { type: String },
      last:     { type: String }
    },
    status:     { type: String }
  });

  var schemaOptions = {
    toJSON: {
      virtuals: true
    },
    toObject: {
      virtuals: true
    }
  };

  var Contact = new mongoose.Schema({
    name: {
      first:    { type: String },
      last:     { type: String }
    },
    accountId: { type: mongoose.Schema.ObjectId },
    added:     { type: Date },     // When the contact was added
    updated:   { type: Date }      // When the contact last updated
  }, schemaOptions);

  Contact.virtual('online').get(function(){
    return app.isAccountOnline(this.get('accountId'));
  });
```

```
var AccountSchema = new mongoose.Schema({
  email:     { type: String, unique: true },
  password:  { type: String },
  name: {
    first:   { type: String },
    last:    { type: String },
    full:    { type: String }
  },
  birthday: {
    day:     { type: Number, min: 1, max: 31, required: false },
    month:   { type: Number, min: 1, max: 12, required: false },
    year:    { type: Number }
  },
  photoUrl:  { type: String },
  biography: { type: String },
  contacts:  [Contact],
  status:    [Status], // My own status updates only
  activity:  [Status]  //  All status updates including friends
});

var Account = mongoose.model('Account', AccountSchema);

var registerCallback = function(err) {
  if (err) {
    return console.log(err);
  };
  return console.log('Account was created');
};

var changePassword = function(accountId, newpassword) {
  var shaSum = crypto.createHash('sha256');
  shaSum.update(newpassword);
  var hashedPassword = shaSum.digest('hex');
  Account.update({_id:accountId}, {$set: {password:hashedPassword}},{upsert:false},
    function changePasswordCallback(err) {
      console.log('Change password done for account ' + accountId);
  });
};

var forgotPassword = function(email, resetPasswordUrl, callback) {
  var user = Account.findOne({email: email}, function findAccount(err, doc){
    if (err) {
      // Email address is not a valid user
      callback(false);
    } else {
      var smtpTransport = nodemailer.createTransport('SMTP', config.mail);
      resetPasswordUrl += '?account=' + doc._id;
      smtpTransport.sendMail({
        from: 'thisapp@example.com',
        to: doc.email,
        subject: 'SocialNet Password Request',
        text: 'Click here to reset your password: ' + resetPasswordUrl
```

```
      }, function forgotPasswordResult(err) {
        if (err) {
          callback(false);
        } else {
          callback(true);
        }
      });
    }
  });
};

var login = function(email, password, callback) {
  var shaSum = crypto.createHash('sha256');
  shaSum.update(password);
  Account.findOne({email:email,password:shaSum.digest('hex')},function(err,doc){
    callback(doc);
  });
};

var findByString = function(searchStr, callback) {
  var searchRegex = new RegExp(searchStr, 'i');
  Account.find({
    $or: [
      { 'name.full': { $regex: searchRegex } },
      { email:       { $regex: searchRegex } }
    ]
  }, callback);
};

var findById = function(accountId, callback) {
  Account.findOne({_id:accountId}, function(err,doc) {
    callback(doc);
  });
};

var addContact = function(account, addcontact) {
  contact = {
    name: addcontact.name,
    accountId: addcontact._id,
    added: new Date(),
    updated: new Date()
  };
  account.contacts.push(contact);

  account.save(function (err) {
    if (err) {
      console.log('Error saving account: ' + err);
    }
  });
};

var removeContact = function(account, contactId) {
```

```
    if ( null == account.contacts ) return;

    account.contacts.forEach(function(contact) {
      if ( contact.accountId == contactId ) {
        account.contacts.remove(contact);
      }
    });
    account.save();
  };

  var hasContact = function(account, contactId) {
    if ( null == account.contacts ) return false;

    account.contacts.forEach(function(contact) {
      if ( contact.accountId == contactId ) {
        return true;
      }
    });
    return false;
  };

  var register = function(email, password, firstName, lastName) {
    var shaSum = crypto.createHash('sha256');
    shaSum.update(password);

    console.log('Registering ' + email);
    var user = new Account({
      email: email,
      name: {
        first: firstName,
        last: lastName,
        full: firstName + ' ' + lastName
      },
      password: shaSum.digest('hex')
    });
    user.save(registerCallback);
    console.log('Save command was sent');
  };

  return {
    findById: findById,
    register: register,
    hasContact: hasContact,
    forgotPassword: forgotPassword,
    changePassword: changePassword,
    findByString: findByString,
    addContact: addContact,
    removeContact: removeContact,
    login: login,
    Account: Account
  }
}
```

The account status route has been updated in Example 10-29 so it will now send a `status` event whenever an activity status is added to any account. This event will filter down to all of the account's contacts and cause the status to instantly display onscreen for anyone who happens to be looking at the account's profile view.

Example 10-29. routes/accounts.js

```
module.exports = function(app, models) {
  app.get('/accounts/:id/contacts', function(req, res) {
    var accountId = req.params.id == 'me'
                      ? req.session.accountId
                      : req.params.id;
    models.Account.findById(accountId, function(account) {
      res.send(account.contacts);
    });
  });

  app.get('/accounts/:id/activity', function(req, res) {
    var accountId = req.params.id == 'me'
                      ? req.session.accountId
                      : req.params.id;
    models.Account.findById(accountId, function(account) {
      res.send(account.activity);
    });
  });

  app.get('/accounts/:id/status', function(req, res) {
    var accountId = req.params.id == 'me'
                      ? req.session.accountId
                      : req.params.id;
    models.Account.findById(accountId, function(account) {
      res.send(account.status);
    });
  });

  app.post('/accounts/:id/status', function(req, res) {
    var accountId = req.params.id == 'me'
                      ? req.session.accountId
                      : req.params.id;
    models.Account.findById(accountId, function(account) {
      status = {
        name: account.name,
        status: req.param('status', '')
      };
      account.status.push(status);

      // Push the status to all friends
      account.activity.push(status);
      account.save(function (err) {
        if (err) {
          console.log('Error saving account: ' + err);
        } else {
```

```
        app.triggerEvent('event:' + accountId, {
          from: accountId,
          data: status,
          action: 'status'
        });
      }
    });
  });
  res.send(200);
});

app.delete('/accounts/:id/contact', function(req,res) {
  var accountId = req.params.id == 'me'
                    ? req.session.accountId
                    : req.params.id;
  var contactId = req.param('contactId', null);

  // Missing contactId, don't bother going any further
  if ( null == contactId ) {
    res.send(400);
    return;
  }

  models.Account.findById(accountId, function(account) {
    if ( !account ) return;
    models.Account.findById(contactId, function(contact,err) {
      if ( !contact ) return;

      models.Account.removeContact(account, contactId);
      // Kill the reverse link
      models.Account.removeContact(contact, accountId);
    });
  });

  // Note: Not in callback - this endpoint returns immediately and
  // processes in the background
  res.send(200);
});

app.post('/accounts/:id/contact', function(req,res) {
  var accountId = req.params.id == 'me'
                    ? req.session.accountId
                    : req.params.id;
  var contactId = req.param('contactId', null);

  // Missing contactId, don't bother going any further
  if ( null == contactId ) {
    res.send(400);
    return;
  }

  models.Account.findById(accountId, function(account) {
```

```
      if ( account ) {
        models.Account.findById(contactId, function(contact) {
          models.Account.addContact(account, contact);

          // Make the reverse link
          models.Account.addContact(contact, account);
          account.save();
        });
      }
    });

    // Note: Not in callback - this endpoint returns immediately and
    // processes in the background
    res.send(200);
  });

  app.get('/accounts/:id', function(req, res) {
    var accountId = req.params.id == 'me'
                      ? req.session.accountId
                      : req.params.id;
    models.Account.findById(accountId, function(account) {
      if ( accountId == 'me'
           || models.Account.hasContact(account, req.session.accountId) ) {
        account.isFriend = true;
      }
      res.send(account);
    });
  });
}
```

The authentication routes in Example 10-30 add the account ID data to the `login` and `authenticated` responses so the Backbone application can compare incoming events to figure out if they are applicable to the currently logged-in user.

Example 10-30. routes/authentication.js

```
module.exports = function(app, models) {
  app.post('/login', function(req, res) {
    var email = req.param('email', null);
    var password = req.param('password', null);

    if ( null == email || email.length < 1
        || null == password || password.length < 1 ) {
      res.send(400);
      return;
    }

    models.Account.login(email, password, function(account) {
      if ( !account ) {
        res.send(401);
        return;
      }
```

```
    req.session.loggedIn = true;
    req.session.accountId = account._id;
    res.send(account._id);
  });
});

app.post('/register', function(req, res) {
  var firstName = req.param('firstName', '');
  var lastName = req.param('lastName', '');
  var email = req.param('email', null);
  var password = req.param('password', null);

  if ( null == email || email.length < 1
       || null == password || password.length < 1 ) {
    res.send(400);
    return;
  }

  models.Account.register(email, password, firstName, lastName);
  res.send(200);
});

app.get('/account/authenticated', function(req, res) {
  if ( req.session && req.session.loggedIn ) {
    res.send(req.session.accountId);
  } else {
    res.send(401);
  }
});

app.post('/forgotpassword', function(req, res) {
  var hostname = req.headers.host;
  var resetPasswordUrl = 'http://' + hostname + '/resetPassword';
  var email = req.param('email', null);
  if ( null == email || email.length < 1 ) {
    res.send(400);
    return;
  }

  models.Account.forgotPassword(email, resetPasswordUrl, function(success){
    if (success) {
      res.send(200);
    } else {
      // Username or password not found
      res.send(404);
    }
  });
});

app.get('/resetPassword', function(req, res) {
  var accountId = req.param('account', null);
  res.render('resetPassword.jade', {locals:{accountId:accountId}});
```

```
  });

  app.post('/resetPassword', function(req, res) {
    var accountId = req.param('accountId', null);
    var password = req.param('password', null);
    if ( null != accountId && null != password ) {
      models.Account.changePassword(accountId, password);
    }
    res.render('resetPasswordSuccess.jade');
  });
}
```

Example 10-31 shows the finalized *chat.js* route. Now when your user logs in, he will loop through each of his contacts and listen to events originated from them. When your user logs out or is disconnected, he will loop through each of his contacts and remove those listeners to prevent hanging processes. This is similar to the `changeView` function in the Backbone.js router: the goal is to prevent "zombie" listeners from consuming resources by reacting to events for users and views that no longer exist. With the listener callback removed from the event list, Node.js is free to garbage-collect the socket because there will be no forgotten references to it hanging around.

Example 10-31. routes/chat.js

```
module.exports = function(app, models) {
  var io = require('socket.io');
  var utils = require('connect').utils;
  var cookie = require('cookie');
  var Session = require('connect').middleware.session.Session;

  var sio = io.listen(app.server)

  sio.configure(function() {
    app.isAccountOnline = function(accountId) {
      var clients = sio.sockets.clients(accountId);
      return (clients.length > 0);
    };

    sio.set('authorization', function( data, accept) {
      var signedCookies = cookie.parse(data.headers.cookie);
      var cookies = utils.parseSignedCookies(signedCookies,app.sessionSecret);
      data.sessionID = cookies['express.sid'];
      data.sessionStore = app.sessionStore;
      data.sessionStore.get(data.sessionID, function(err, session) {
        if ( err || !session ) {
          return accept('Invalid session', false);
        } else {
          data.session = new Session(data, session);
          accept(null, true);
        }
      });
    });
```

```
sio.sockets.on('connection', function(socket) {
  var session = socket.handshake.session;
  var accountId = session.accountId;
  var sAccount = null;
  socket.join(accountId);

  app.triggerEvent('event:' + accountId, {
    from: accountId,
    action: 'login'
  });

  var handleContactEvent = function(eventMessage) {
    socket.emit('contactEvent', eventMessage);
  };

  var subscribeToAccount = function(accountId) {
    var eventName = 'event:' + accountId;
    app.addEventListener(eventName, handleContactEvent);
    console.log('Subscribing to ' + eventName);
  };

  models.Account.findById(accountId, function subscribeToFriendFeeds(account) {
    var subscribedAccounts = {};
    sAccount = account;
    account.contacts.forEach(function(contact) {
      if ( !subscribedAccounts[contact.accountId]) {
        subscribeToAccount(contact.accountId);
        subscribedAccounts[contact.accountId] = true;
      }
    });

    if (!subscribedAccounts[accountId]) {
      // Subscribe to my own updates
      subscribeToAccount(accountId);
    }
  });

  socket.on('disconnect', function() {
    sAccount.contacts.forEach(function(contact) {
      var eventName = 'event:' + contact.accountId;
      app.removeEventListener(eventName, handleContactEvent);
      console.log('Unsubscribing from ' + eventName);
    });
    app.triggerEvent('event:' + accountId, {
      from: accountId,
      action: 'logout'
    });
  });

  socket.on('chatclient', function(data) {
    sio.sockets.in(data.to).emit('chatserver', {
```

```
          from: accountId,
          text: data.text
        });
      });
    });
  });
}
```

Static Files

Static files are supporting files that do not contain executable code but are needed by the user interface.

Example 10-32 contains the new stylesheet for this project. The `online_indicator` class will contain the traffic light status indicator for the updated chat list. This will be a 15×16 pixel container with a background image, which will be indistinguishable from a real image element in the web browser. The importance of this is that the `online_indicator` container element is smaller than the size of the traffic light background image. When the `online` CSS class is added to the indicator container, the background position shifts 15 px, giving the visual effect of the traffic light changing color.

Example 10-32. public/styles/styles.css

```
form {
  width: 400px;
}

#chat form {
  width: auto;
}

#chat {
  position: absolute;
  right: 0;
  bottom: 0;
}

.chat_list {
  float: right;
  border: 1px solid black;
  list-style-type: none;
  overflow: auto;
  width: 120px;
  height: 300px;
  margin: 0;
  padding: 0;
}

.chat_list li {
  width: 100%;
  padding: 10px 0;
```

```
  background-color: #0099ff;
}

.chat_list li:nth-child(odd) {
  background-color: #80ccff;
}

.chat_list span {
  margin: 10px;
}

.chat_session {
  float: left;
  width: 250px;
  height: 300px;
}

.online_indicator {
  float: left;
  width: 15px;
  height: 16px;
  background-image: url('/images/trafficlight.png');
}

.online_indicator.online {
  background-position: -15px 0;
}
```

Glossary

bootstrap
: In software development a bootstrap is a simple computer program whose purpose is to launch a more complicated program. When applied to Backbone.js, a bootstrap is a small class that takes a minimum amount of parameters and is capable of initializing the entire application.

denial of service (DOS)
: A denial of service attack is an attempt to render a web server inoperable for its intended users, often by saturating its designed capacity with unnecessary requests.

event
: An action that happens outside of an application's regular flow, and handled by dedicated code inside the application. Events can be triggered by user input (keyboard events, mouse events) or from computer input (disk events, operating system events, application events).

Internet Information Services (IIS)
: A web server created by Microsoft used to serve, among many types of content, websites and applications.

middleware
: Makes input and output easier by interfacing between high level applications and low level system. Connect is an example of middleware that abstracts HTTP server functionality and eases the work of dealing with sessions, cookies, and data transport.

namespace
: Contains a set of variables and functions grouped by similar functionality. In Node.js, each source file contains a set of code that is not directly available to code from other source files unless they are explicitly exported.

payload
: A set of information delivered to an end user. This could refer to raw bits, a JSON response, or HTML data.

prototype
: Unlike class-based programming languages, JavaScript only has a single instance type: `object`. Prototypes are JavaScript's way of sharing functions and variables across objects of the same type, similarly to classes.

prototype chain
: Prototype chaining provides class inheritance by linking the constructor for one prototype to another to achieve a class who contains all of the properties of both the parent and child classes.

render

Rendering is the process of generating formatted content from source data. In Node.js, a template engine such as Jade is responsible for converting source models into HTML for consumption by web browsers. In Backbone.js, Underscore's template engine provides the same functionality.

About the Author

Mike Wilson has had the privilege of working with some of the largest and most influential brands in the world, including Disney, Microsoft, and McDonalds. He has years of web development experience, designing and building everything from small business sites to large MMO server clusters hosting millions of players. In his free time, Mike maintains his personal blog and contributes to forums and experiments with emerging frameworks and software. Mike lives in Vancouver with his wife and their three children.

Colophon

The animal on the cover of *Building Node Applications with MongoDB and Backbone* is the small Indian civet (*Viverricula indica*), which is found across south and southeast Asia as well as in the Indonesian archipelago. The animal is named for the thick yellowish musky-odored substance it produces in self defense.

The small Indian civet is slender, agile in climbing trees, has no erectile mane, and lives in holes in rocky and brushy locations. It is nocturnal, solitary, and usually arboreal; that is to say, it climbs trees and stays there. However, when hunting, it opts for ground level. The civet is basically omnivorous—its diet consists of lizards, rodents, birds, insects, and even eggs. In captivity, it is easily tamed and feeds on small animals, which it catches with cat-like dexterity.

The small Indian civet is gray or tawny in color with rows of black spots on the body and stripes on the tail. Its legs, ears, and muzzle are also black.

The civet produces a musk (also called civet), which is highly valued as a fragrance and stabilizing agent for perfume. Both male and female civets produce the strong-smelling secretion, which is produced by the civet's perineal glands. Humans have been known to hunt the civet for its meat, and purify its skin into medicine.

The cover image is from *Shaw's Zoology*. The cover font is Adobe ITC Garamond. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.