



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Express.js Blueprints

Learn to use Express.js pragmatically by creating five fun and robust real-world APIs, with a bonus chapter on Koa.js

Ben Augarten, Marc Kuo, Eric Lin, Aidha Shaikh,
Fabiano Pereira Soriani, Geoffrey Tisserand,
Chiqing Zhang, Kan Zhang

www.allitebooks.com

[PACKT] open source*
PUBLISHING community experience distilled

Express.js Blueprints

Learn to use Express.js pragmatically by creating five fun and robust real-world APIs, with a bonus chapter on Koa.js

**Ben Augarten, Marc Kuo, Eric Lin, Aidha Shaikh,
Fabiano Pereira Soriani, Geoffrey Tisserand,
Chiqing Zhang, Kan Zhang**



BIRMINGHAM - MUMBAI

Express.js Blueprints

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: May 2015

Production reference: 1080515

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78398-302-5

www.packtpub.com

Credits

Authors

Ben Augarten
Marc Kuo
Eric Lin
Aidha Shaikh
Fabiano Pereira Soriani
Geoffrey Tisserand
Chiqing Zhang
Kan Zhang

Reviewers

Aristides Villarreal Bravo
Aritrik Das
John Fawcett
Ajmal Sali
Dan Williams

Commissioning Editor

Ashwin Nair

Acquisition Editor

James Jones

Content Development Editors

Priyanka Shah
Ritika Singh

Technical Editor

Deepti Tuscano

Copy Editors

Vikrant Phadke
Adithi Shetty

Project Coordinator

Izzat Contractor

Proofreaders

Simran Bhogal
Martin Diver
Safis Editing

Indexer

Priya Sane

Production Coordinator

Nitesh Thakur

Cover Work

Nitesh Thakur

About the Authors

Ben Augarten is a senior software engineer at Twitter, in the beautiful city of Boulder, Colorado, USA. He is an electrical engineering and computer science graduate from the University of California, Berkeley. He is the author of node-RESTful, a library used to generate RESTful endpoints for domain resources. Ben writes programs that are widely used, functional, and scalable. He primarily works on distributed systems, data processing, and web technologies.

My thanks to Packt Publishing for their help and support while writing this book. Also, thanks to the Axiom Zen team for their help in crafting the content. Finally, thanks to my friends and family for their support, and my co-workers for refraining from making too many Node.js jokes.

Marc Kuo has a neat freak personality, which is reflected in his clean and proficient code. As a full stack developer at Axiom Zen, he architects shipshape infrastructure and efficient model databases. He loves to hack in Common Lisp, CoffeeScript, Angular, and Node.js.

The neat freak in Marc comes from two important principles: efficiency and optimization. He is the founder of Routific, a routing solution that reduces waste in the transportation sector. In the open source field, he is the author of Alike and Look-Alike(recommendation engines for Node.js), T3 (*Ultimate Tic-Tac-Toe*), and Zenbase-angular (gulp-angular-coffee-stylus-jade boilerplate).

Infinite gratitude goes out to my wife, Suzanne Ma, the cofounder of Routific as well as my life. Thanks for always joining me on my crazy adventures. I'd also like to express (no pun intended) exponential gratitude towards Axiom Zen for giving us this unique opportunity and for always fostering creativity.

Eric Lin is a software engineer at Axiom Zen in Vancouver, British Columbia, Canada. He completed his master's degree in statistics and has been developing software professionally for 2 years. He is always looking forward to picking up new languages and technologies while having a personal interest in data analytics and predictive modeling.

My thanks to Packt Publishing for their support throughout the writing process and their flexibility on deadlines; to Axiom Zen, for the numerous opportunities they gave me to expand my knowledge in new technologies; finally, to my wife, Min-Chee Lo, for always being patient and understanding, especially on the days where I ended up working late.

Aidha Shaikh has a doctoral degree in chemistry from the University of British Columbia, where she researched enzymes that cleave blood antigens to make universal O-type blood, and published several first-authored papers. After completing an NSERC Post Doctoral Industrial R&D Fellowship, she embraced her love for coding. She recently stepped out of her lab coat and donned a coder hoodie with pride.

Aidha's research-rich past stays with her as she constantly looks for new ways to solve problems, and she loves learning new things everyday. She started off with frontend web development. She really loves to hack into Node.js and Express.js.

My deepest thanks go to Axiom Zen for the amazing opportunities, learning experiences, and creative avenues I've been given. I would also like to thank the Packt Publishing team for all their support and work on this book with us.

Fabiano Pereira Soriani does what he loves as a software developer at Axiom Zen in Santiago, Chile. He holds a bachelor's degree in computer science from the Universidade Estadual de Londrina, Brazil, and a certification in project management from Canada. He has developed software professionally for over 5 years, always focusing on new and productive web technologies, with an intense focus on the impact they have on users and other stakeholders alike. He aims for excellence in product and agile product lifecycles.

Fabiano has published open source Node.js packages and a number of how-to blog posts, ranging from backend concepts and Ruby on Rails all the way through to the cutting-edge frontend.

Thanks to the talented and patient staff at Packt Publishing for helping us instill the best book we could. It has been quite a long journey. Also, thanks to Axiom Zen, for the vision and encouragement, and allowing time for this project—this is a part of what makes this company so unique.

Finally, thanks to my companion, Asuka Kiriyama, who was kind and tolerant of the long work hours required to write the content of this book.

Geoffrey Tisserand is a full stack software engineer, who focuses on building reliable and scalable applications and services for Axiom Zen's products. He completed his master's degree in computer science at the Université de technologie in Belfort-Montbéliard, France. He is a nitpicky and detail-oriented JavaScript and Ruby ninja, who really enjoys discovering new technologies, APIs, and frameworks to play with.

A start-up enthusiast, Geoffrey is thrilled to be in an environment where he is constantly learning and improving his skills. He also loves to build side-projects and create experiments, and is always thinking about his next idea for a start-up.

Chiqing Zhang is an exceptional software architect, whose clean and simple code has leveraged scalable and maintainable systems for some of the world's top technology companies, such as Microsoft, Baidu, and AppAnnie. As a full stack developer at Axiom Zen, he is passionate about building highly reliable systems and delivering products with the best user experience. Chiqing was granted a patent for multilayer structured data operations and he has published a book on Microsoft Silverlight technologies.

Many thanks to the Packt Publishing team and the technical reviewers, whose insightful comments and kind suggestions were essential for improving the content. Thanks to Axiom Zen for this opportunity. Finally, thanks to my wife, Hanna Yang, for her constant support and patience.

Kan Zhang is an experienced software engineer with both a bachelor's degree in civil engineering and a bachelor's degree in computer science. He has gained substantial project management experience from leading personal team projects as well as previous civil engineering internships.

Kan has also worked on many Android apps, mobile games, and backend systems for various applications and services. He is currently working as a software engineer at Axiom Zen, discovering his love for new technologies, innovative products, and exciting start-ups.

My thanks for the amazing support from all the people at both Axiom Zen and Packt Publishing. And thanks to Irene Fung for her continued patience and support in everything I do.

About the Reviewers

Aristides Villarreal Bravo is a Java developer, a member of the NetBeans Dream Team, and a Java User Groups leader. He lives in Panama. He has organized and participated in various conferences and seminars related to Java, JavaEE, NetBeans, the NetBeans platform, free software, and mobile devices, both nationally and internationally. He is the author of tutorials and blogs about Java, NetBeans, and web development.

Aristides has participated in several interviews on sites about topics such as NetBeans, NetBeans DZone, and JavaHispano. He is a developer of plugins for NetBeans and the technical reviewer of a book about PrimeFaces. He is the CEO of Javscnaz Software Developers.

Aritrik Das is a web developer with expertise in various cutting-edge technologies used in the Web arena. He has strong analytical skills and a broad knowledge of open source technologies. He has gained skills in many web frameworks, both frontend and backend. Aritrik is an excellent problem solver, able to quickly grasp complex systems, and identify opportunities for improvement and the resolution of critical issues. He also has quite a good hold on various deployment techniques and infrastructure designs. As such, he can develop a scalable web application from the groundwork to deployment in live environments.

John Fawcett is a JavaScript application developer with over 10 years of professional development experience. In his home at Austin, Texas, USA, he organizes a small tech meetup and is an active member of the web development community. He contributes to, and is the author of, many open source projects, and is the CTO of the local start-up, www.goodybag.com. In his free time, he writes for his blog, performs visual experiments with processing, composes music, and drinks craft beer with his lovely girlfriend and partner, Courtney.

Ajmal M Sali is a technophile from Haripad, Kerala—a tourist destination. He started programming at the age of 12. He likes to help others in solving technical problems, and these problems range from setting up clustered messaging servers to fixing Tetra Pak machines. He has worked with few startups such as Sourcebits Inc. and Rightaway Inc. and has worked as a remote consultant. He has strong expertise with many frameworks and languages and focuses more on Angular.js, RubyOnRails, PHP and Android. He believes in continuous integration and deployment. He blogs at <https://ajm.al>.

Dan Williams has been programming since he was in high school. Having worked from the microcontroller level up to large-scale enterprise applications, he has now found a new home as a senior developer at Keaton Row. Developing with Node.js in the backend and React on the browser, he enjoys being fully immersed in JavaScript. Dan can often be found around Ontario giving talks and facilitating workshops on emerging technologies.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit
www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	v
Chapter 1: Building a Basic Express Site	1
Setting up Express for a static site	1
Saying Hello, World in Express	2
Jade templating	3
Local user authentication	4
User object modeling	5
Introducing Express middleware	7
Setting up passport	8
Registering users	10
Authenticating users	11
OAuth with passport	12
Adding OAuth to user model	12
Getting API tokens	13
Third-party registration and login	14
Profile pages	15
URL params	15
Profile templates	16
Testing	16
Introducing Mocha	17
Testing API endpoints	18
Automate builds and deploys	19
Introducing the Gruntfile	20
Continuous integration with Travis	21
Deploying Node.js applications	22
Summary	24

Table of Contents

Chapter 2: A Robust Movie API	25
Folder structure and organization	25
Responding to CRUD operations	27
Retrieving an actor with GET	28
Creating a new actor with POST	29
Updating an actor with PUT	30
Removing an actor with DELETE	30
Object modeling with Mongoose	32
Generating unique IDs	34
Validating your database	36
Extracting functions to reusable middleware	36
Testing	40
Summary	44
Chapter 3: Multiplayer Game API – Connect 4	45
Modeling game state with Mongoose	46
Creating a new game	49
Input validation	54
Getting the game state	56
Joining a game	58
Playing the game	61
Testing for a tie	71
Summary	73
Chapter 4: MMO Word Game	75
Gameplay	75
Real-time application overview	76
Keeping track of active users	77
Schema design	77
User schema	77
User join	78
Promises	79
The then and catch method	83
Chain multiple Promises	83
Prevent duplicates	84
User leaves the game	86
Show all active users	87
The words – Subdocuments	88
Validate input	89
Dealing with race conditions	91
Test case to test race conditions	93

Table of Contents

Socket.IO	94
Socket handshaking, user join	94
Adding and pushing updates to clients	95
Launch Socket.IO applications	97
Test Socket.IO applications with the Socket.IO client	98
Debug Socket.IO with Chrome Developer Tools	103
Summary	106
Chapter 5: Coffee with Strangers	107
Code structure	108
Defining routes	109
Persisting data	110
Exception handling	113
Naive pairing	113
Notes about tests	118
Considering user history	118
Optimizing for distance	124
E-mail follow up	126
Periodical tasks with node-cron	135
Summary	137
Chapter 6: Hacker News API on Koa.js	139
Generator syntax	140
Middleware philosophy	143
Context versus req,res	145
The link model	145
The link routes	146
Tying it together	147
Validation and error handling	148
Update route	150
Let's perform some tests	152
Parallel requests	154
Rendering HTML pages	155
Serving static assets	160
Summary	161
Appendix: Connect 4 – Game Logic	163
Index	171

Preface

APIs are at the core of every serious web application. Node.js is an especially exciting tool that is easy to use, allows you to build APIs, and develop your backend code in JavaScript. It powers the server side of web apps, including PayPal, Netflix, and Zenhub.

Express.js is the most popular framework that can be used to build on top of Node.js—it provides an essential level of abstraction to develop robust web applications. With the emergence of this minimal and flexible Node.js web application framework, creating Node.js applications has become much simpler, faster, and also requires minimal effort.

This book takes a pragmatic approach to leveraging what Express.js has to offer, introduces key libraries, and fully equips you with the skills and tools necessary to build scalable APIs from start to finish while offering subtle details and nuggets of wisdom that come from years of experience.

What this book covers

Chapter 1, Building a Basic Express Site, will provide a basic application (scaffolding), which we will use for the upcoming examples. You will get an insight into what Express applications look like.

Chapter 2, A Robust Movie API, will walk you through building a movie API that allows you to add actor and movie information to a database and connect actors to movies and vice versa.

Chapter 3, Multiplayer Game API – Connect 4, will revolve around building a multiplayer game API. We will also build the app using test-driven development with maximum code coverage.

Chapter 4, MMO Word Game, will teach you how to build a real-time application with Express and SocketIO, perform authentication for socket handshaking, and deal with race conditions using MongoDB's atomic update.

Chapter 5, Coffee with Strangers, will enable you to write an API that allows users to go for a coffee! It will comprise a simple, yet extendable user-matching system.

Chapter 6, Hacker News API on Koa.js, will take you through building a CRUD backend to post links and upvote on Koa.js. We will also look at centralized error handling and avoid callback hell with thunks.

Appendix, Connect 4 – Game Logic, shows the accompanying game logic that we omitted in Chapter 3, Multiplayer Game API – Connect 4.

What you need for this book

You'll need the following to get started with the examples in this book:

- Nvm: <https://github.com/creationix/nvm>
- MongoDB: <https://www.mongodb.org/downloads>
- RoboMongo: <http://robomongo.org/>
- Mocha: Use the `npm i -g mocha` command to download it

Mac OS is preferred but not a necessity.

Who this book is for

This book is for beginners to Node.js and also for those who are technically advanced. By the end of this book, every developer will have the expertise to build web applications with Express.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "If it is, then we render the `users/profile.jade` template with `req.user` as the data."

A block of code is set as follows:

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

app.get('/', function(req, res, next) {
  res.send('Hello, World!');
});

app.listen(3000);
console.log('Express started on port 3000');
```

Any command-line input or output is written as follows:

```
$ npm install --save express
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "You can also right click on the page, and select **Inspect Element**."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

Building a Basic Express Site

Express is a web development framework for Node.js. Node.js is an open source, cross-platform runtime environment for server-side and networking applications. It uses Google Chrome's JavaScript engine, V8, to execute code. Node.js is single-threaded and event-driven. It uses non-blocking I/O to squeeze every ounce of processing power out of the CPU. Express builds on top of Node.js, providing all of the tools necessary to develop robust web applications with node.

In addition, by utilizing Express, one gains access to a host of open source software to help solve common pain points in development. The framework is unopinionated, meaning it does not guide you one way or the other in terms of implementation or interface. Because it is unopinionated, the developer has more control and can use the framework to accomplish nearly any task; however, the power Express offers is easily abused. In this book, you will learn how to use the framework in the right way by exploring the following different styles of an application:

- Setting up Express for a static site
- Local user authentication
- OAuth with passport
- Profile pages
- Testing

Setting up Express for a static site

To get our feet wet, we'll first go over how to respond to basic HTTP requests. In this example, we will handle several `GET` requests, responding first with plaintext and then with static HTML. However, before we get started, you must install two essential tools: `node` and `npm`, which is the `node` package manager.



Navigate to <https://nodejs.org/download/> to install node and npm.



Saying Hello, World in Express

For those unfamiliar with Express, we will start with a basic example – Hello World! We'll start with an empty directory. As with any Node.js project, we will run the following code to generate our package.json file, which keeps track of metadata about the project, such as dependencies, scripts, licenses, and even where the code is hosted:

```
$ npm init
```

The package.json file keeps track of all of our dependencies so that we don't have versioning issues, don't have to include dependencies with our code, and can deploy fearlessly. You will be prompted with a few questions. Choose the defaults for all except the entry point, which you should set to `server.js`.

There are many generators out there that can help you generate new Express applications, but we'll create the skeleton this time around. Let's install Express. To install a module, we use `npm` to install the package. We use the `--save` flag to tell `npm` to add the dependency to our `package.json` file; that way, we don't need to commit our dependencies to the source control. We can just install them based on the contents of the `package.json` file (`npm` makes this easy):

```
$ npm install --save express
```

We'll be using Express v4.4.0 throughout this book.



Warning: Express v4.x is not backwards compatible with the versions before it.



You can create a new file `server.js` as follows:

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

app.get('/', function(req, res, next) {
  res.send('Hello, World!');
}
```

```
) ;  
  
app.listen(3000) ;  
console.log('Express started on port 3000') ;
```

This file is the entry point for our application. It is here that we generate an application, register routes, and finally listen for incoming requests on port 3000. The `require('express')` method returns a generator of applications.

We can continually create as many applications as we want; in this case, we only created one, which we assigned to the variable `app`. Next, we register a GET route that listens for GET requests on the server root, and when requested, sends the string 'Hello, World' to the client. Express has methods for all of the HTTP verbs, so we could have also done `app.post`, `app.put`, `app.delete`, or even `app.all`, which responds to all HTTP verbs. Finally, we start the app listening on port 3000, then log to standard out.

It's finally time to start our server and make sure everything works as expected.

```
$ node server.js
```

We can validate that everything is working by navigating to `http://localhost:3000` in our browser or `curl -v localhost:3000` in your terminal.

Jade templating

We are now going to extract the HTML we send to the client into a separate template. After all, it would be quite difficult to render full HTML pages simply by using `res.send`. To accomplish this, we will use a templating language frequently in conjunction with Express -- jade. There are many templating languages that you can use with Express. We chose Jade because it greatly simplifies writing HTML and was created by the same developer of the Express framework.

```
$ npm install --save jade
```

After installing Jade, we're going to have to add the following code to `server.js`:

```
app.set('view engine', 'jade') ;  
app.set('views', __dirname + '/views') ;  
  
app.get('/', function(req, res, next) {  
  res.render('index') ;  
}) ;
```

The preceding code sets the default view engine for Express – sort of like telling Express that in the future it should assume that, unless otherwise specified, templates are in the Jade templating language. Calling `app.set` sets a key value pair for Express internals. You can think of this sort of application like wide configuration. We could call `app.get` (view engine) to retrieve our set value at any time.

We also specify the folder that Express should look into to find view files. That means we should create a `views` directory in our application and add a file, `index.jade` to it. Alternatively, if you want to include many different template types, you could execute the following:

```
app.engine('jade', require('jade').__express);
app.engine('html', require('ejs').__express);
app.get('/html', function(req, res, next) {
  res.render('index.html');
});

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

Here, we set custom template rendering based on the extension of the template we want to render. We use the Jade renderer for `.jade` extensions and the `ejs` renderer for `.html` extensions and expose both of our index files by different routes. This is useful if you choose one templating option and later want to switch to a new one in an incremental way. You can refer to the source for the most basic of templates.

Local user authentication

The majority of applications require user accounts. Some applications only allow authentication through third parties, but not all users are interested in authenticating through third parties for privacy reasons, so it is important to include a local option. Here, we will go over best practices when implementing local user authentication in an Express app. We'll be using MongoDB to store our users and Mongoose as an ODM (Object Document Mapper). Then, we'll leverage passport to simplify the session handling and provide a unified view of authentication.



Downloading the example code

You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

User object modeling

We will leverage passportjs to handle user authentication. Passport centralizes all of the authentication logic and provides convenient ways to authenticate locally in addition to third parties, such as Twitter, Google, Github, and so on. First, install passport and the local authentication strategy as follows:

```
$ npm install --save passport-local
```

In our first pass, we will implement a local authentication strategy, which means that users will be able to register locally for an account. We start by defining a user model using Mongoose. Mongoose provides a way to define schemas for objects that we want to store in MongoDB and then provide a convenient way to map between stored records in the database and an in-memory representation.

Mongoose also provides convenient syntax to make many MongoDB queries and perform CRUD operations on models. Our user model will only have an e-mail, password, and timestamp for now. Before getting started, we need to install Mongoose:

```
$ npm install --save mongoose bcrypt validator
```

Now we define the schema for our user in `models/user.js` as follows:

```
var mongoose = require('mongoose');

var userSchema = new mongoose.Schema({
  email: {
    type: String,
    required: true,
    unique: true
  },
  password: {
    type: String,
    required: true
  },
  created_at: {
    type: Date,
    default: Date.now
  }
});
```

```
userSchema.pre('save', function(next) {
  if (!this.isModified('password')) {
    return next();
  }
  this.password = User.encryptPassword(this.password);
  next();
});
```

Here, we create a schema that describes our users. Mongoose has convenient ways to describe the required and unique fields as well as the type of data that each property should hold. Mongoose does all the validations required under the hood. We don't require many user fields for our first boilerplate application—e-mail, password, and timestamp to get us started.

We also use Mongoose middleware to rehash a user's password if and when they decide to change it. Mongoose exposes several hooks to run user-defined callbacks. In our example, we define a callback to be invoked before Mongoose saves a model. That way, every time a user is saved, we'll check to see whether their password was changed.

Without this middleware, it would be possible to store a user's password in plaintext, which is not only a security vulnerability but would break authentication. Mongoose supports two kinds of middleware – serial and parallel. Parallel middleware can run asynchronous functions and gets an additional callback to invoke; you'll learn more about Mongoose middleware later in this book.

Now, we want to add validations to make sure that our data is correct. We'll use the `validator` library to accomplish this, as follows:

```
var validator = require('validator');

User.schema.path('email').validate(function(email) {
  return validator.isEmail(email);
});

User.schema.path('password').validate(function(password) {
  return validator.minLength(password, 6);
});

var User = mongoose.model('User', userSchema);
module.exports = User;
```

We added validations for e-mail and password length using a library called `validator`, which provides a lot of convenient validators for different types of fields. `Validator` has validations based on length, URL, int, upper case; essentially, anything you would want to validate (and don't forget to validate all user input!).

We also added a host of helper functions regarding registration, authentication, as well as encrypting passwords that you can find in `models/user.js`. We added these to the user model to help encapsulate the variety of interactions we want using the abstraction of a user.



For more information on Mongoose, see <http://mongoosejs.com/>. You can find more on passportjs at <http://passportjs.org/>.



This lays out the beginning of a design pattern called MVC – model, view, controller. The basic idea is that you encapsulate separate concerns in different objects: the model code knows about the database, storage, and querying; the controller code knows about routing and requests/responses; and the view code knows what to render for users.

Introducing Express middleware

Passport is authentication middleware that can be used with Express applications. Before diving into passport, we should go over Express middleware. Express is a connect framework, which means it uses the connect middleware. Connecting internally has a stack of functions that handle requests.

When a request comes in, the first function in the stack is given the request and response objects along with the `next()` function. The `next()` function when called, delegates to the next function in the middleware stack. Additionally, you can specify a path for your middleware, so it is only called for certain paths.

Express lets you add middleware to an application using the `app.use()` function. In fact, the HTTP handlers we already wrote are a special kind of middleware. Internally, Express has one level of middleware for the router, which delegates to the appropriate handler.

Middleware is extraordinarily useful for logging, serving static files, error handling, and more. In fact, passport utilizes middleware for authentication. Before anything else happens, passport looks for a cookie in the request, finds metadata, and then loads the user from the database, adds it to `req.user`, and then continues down the middleware stack.

Setting up passport

Before we can make full use of passport, we need to tell it how to do a few important things. First, we need to instruct passport how to serialize a user to a session. Then, we need to deserialize the user from the session information. Finally, we need to tell passport how to tell if a given e-mail/password combination represents a valid user as given in the following:

```
// passport.js
var passport = require('passport');
var LocalStrategy = require('passport-local').Strategy;
var User = require('mongoose').model('User');

passport.serializeUser(function(user, done) {
  done(null, user.id);
});

passport.deserializeUser(function(id, done) {
  User.findById(id, done);
});
```

Here, we tell passport that when we serialize a user, we only need that user's id. Then, when we want to deserialize a user from session data, we just look up the user by their ID! This is used in passport's middleware, after the request is finished, we take `req.user` and serialize their ID to our persistent session. When we first get a request, we take the ID stored in our session, retrieve the record from the database, and populate the request object with a `user` property. All of this functionality is provided transparently by passport, as long as we provide definitions for these two functions as given in the following:

```
function authFail(done) {
  done(null, false, { message: 'incorrect email/password
combination' });
}

passport.use(new LocalStrategy(function(email, password, done) {
  User.findOne({
    email: email
  }, function(err, user) {
    if (err) return done(err);
    if (!user) {
      return authFail(done);
    }
    if (!user.validPassword(password)) {
      return authFail(done);
    }
  });
});
```

```
        }
        return done(null, user);
    });
});
```

We tell passport how to authenticate a user locally. We create a new `LocalStrategy()` function, which, when given an e-mail and password, will try to lookup a user by e-mail. We can do this because we required the e-mail field to be unique, so there should only be one user. If there is no user, we return an error. If there is a user, but they provided an invalid password, we still return an error. If there is a user and they provided the correct password, then we tell passport that the authentication request was a success by calling the `done` callback with the valid user.

In order to utilize passport, we need to add the middleware we talked about. We actually need to add a few different kinds of middleware. The great part about Express middleware is that it encourages developers to write small, focused modules so that you can bring in functionality that you want and exclude functionality that you don't need.

```
// server.js
var mongoose = require('mongoose');
var User = require('./models/user');
var passport = require('./passport');

mongoose.connect('mongodb://localhost/chapter01', function(err) {
    if (err) throw err;
});

...
app.use(require('cookie-parser')('my secret string'));
app.use(require('express-session')({ secret: "my other secret
string" }));
app.use(require('body-parser')());
app.use(passport.initialize());
app.use(passport.session());
```

In order to use passport, we have to enable a few things for our server. First we need to enable cookies and session support. To enable session support, we add a cookie parser. This middleware parses a cookie object into `req.cookies`. The session middleware lets us modify `req.session` and have that data persist across requests. By default, it uses cookies, but it has a variety of session stores that you can configure. Then, we have to add body-parsing middleware, which parses the body of HTTP requests into a JavaScript object `req.body`.

In our use case, we need this middleware to extract the e-mail and password fields from POST requests. Finally, we add the passport middleware and session support.

Registering users

Now, we add routes for registration, both a view with a basic form and backend logic to create a user. First, we will create a user controller. Up until now, we have thrown our routes in our `server.js` file, but this is generally bad practice. What we want to do is have separate controllers for each kind of route that we want. We have seen the model portion of MVC. Now it's time to take a look at controllers. Our user controller will have all the routes that manipulate the user model. Let's create a new file in a new directory, `controllers/user.js`:

```
// controllers/user.js
var User = require('mongoose').model('User');

module.exports.showRegistrationForm = function(req, res, next) {
    res.render('register');
};

module.exports.createUser = function(req, res, next) {
    User.register(req.body.email, req.body.password, function(err,
user) {
        if (err) return next(err);
        req.login(user, function(err) {
            if (err) return next(err);
            res.redirect('/');
        });
    });
};
```

 Note that the `User` model takes care of the validations and registration logic; we just provide callback. Doing this helps consolidate the error handling and generally makes the registration logic easier to understand. If the registration was successful, we call `req.login`, a function added by passport, which creates a new session for that user and that user will be available as `req.user` on subsequent requests.

Finally, we register the routes. At this point, we also extract the routes we previously added to `server.js` to their own file. Let's create a new file called `routes.js` as follows:

```
// routes.js
app.get('/users/register', userRoutes.showRegistrationForm);
app.post('/users/register', userRoutes.createUser);
```

Now we have a file dedicated to associating controller handlers with actual paths that users can access. This is generally good practice because now we have a place to come visit and see all of our defined routes. It also helps unclutter our `server.js` file, which should be exclusively devoted to server configuration.



For details, as well as the registration templates used, see the preceding code.



Authenticating users

We have already done most of the work required to authenticate users (or rather, passport has). Really, all we need to do is set up routes for authentication and a form to allow users to enter their credentials. First, we'll add handlers to our user controller:

```
// controllers/user.js
module.exports.showLoginForm = function(req, res, next) {
  res.render('login');
}

module.exports.createSession = passport.authenticate('local', {
  successRedirect: '/',
  failureRedirect: '/login'
});
```

Let's deconstruct what's happening in our login post. We create a handler that is the result of calling `passport.authenticate('local', ...)`. This tells passport that the handler uses the local authentication strategy. So, when someone hits that route, passport will delegate to our LocalStrategy. If they provided a valid e-mail/password combination, our LocalStrategy will give passport the now authenticated user, and passport will redirect the user to the server root. If the e-mail/password combination was unsuccessful, passport will redirect the user to /login so they can try again.

Then, we will bind these callbacks to routes in `routes.js`:

```
app.get('/users/login', userRoutes.showLoginForm);
app.post('/users/login', userRoutes.createSession);
```

At this point, we should be able to register an account and login with those same credentials. (see tag 0.2 for where we are right now).

OAuth with passport

Now we will add support for logging into our application using Twitter, Google, and GitHub. This functionality is useful if users don't want to register a separate account for your application. For these users, allowing OAuth through these providers will increase conversions and generally make for an easier registration process for users.

Adding OAuth to user model

Before adding OAuth, we need to keep track of several additional properties on our user model. We keep track of these properties to make sure we can look up user accounts provided there is information to ensure we don't allow duplicate accounts and allow users to link multiple third-party accounts by using the following code:

```
var userSchema = new mongoose.Schema({
  email: {
    type: String,
    required: true,
    unique: true
  },
  password: {
    type: String,
  },
  created_at: {
    type: Date,
    default: Date.now
  },
  twitter: String,
  google: String,
  github: String,
  profile: {
    name: { type: String, default: '' },
    gender: { type: String, default: '' },
    location: { type: String, default: '' },
    website: { type: String, default: '' },
    picture: { type: String, default: '' }
  },
});
```

First, we add a property for each provider, in which we will store a unique identifier that the provider gives us when they authorize with that provider. Next, we will store an array of tokens, so we can conveniently access a list of providers that are linked to this account; this is useful if you ever want to let a user register through one and then link to others for viral marketing or extra user information. Finally, we keep track of some demographic information about the user that the providers give to us so we can provide a better experience for our users.

Getting API tokens

Now, we need to go to the appropriate third parties and register our application to receive application keys and secret tokens. We will add these to our configuration. We will use separate tokens for development and production purposes (for obvious reasons!). For security reasons, we will only have our production tokens as environment variables on our final deploy server, not committed to version control.

I'll wait while you navigate to the third-party websites and add their tokens to your configuration as follows:

```
// config.js
twitter: {
  consumerKey: process.env.TWITTER_KEY ||
'VRE4lt1y0W3yWTpChzJHcAaVf',
  consumerSecret: process.env.TWITTER_SECRET ||
'TOA4rNzv9Cn8IwrOi6M0myV894hyaJks6393V6cyLdtmFfkWqe',
  callbackURL: '/auth/twitter/callback'
},
google: {
  clientID: process.env.GOOGLE_ID || '627474771522-
uskkhdsevat3rn15kgrqt62bdft15cpu.apps.googleusercontent.com',
  clientSecret: process.env.GOOGLE_SECRET ||
'FwVkn76DKx_0BBaIAmRb6mjB',
  callbackURL: '/auth/google/callback'
},
github: {
  clientID: process.env.GITHUB_ID || '81b233b3394179bfe2bc',
  clientSecret: process.env.GITHUB_SECRET ||
'de0322c0aa32eafaa84440ca6877ac5be9db9ca6',
  callbackURL: '/auth/github/callback'
}
```



Of course, you should never commit your development keys publicly either. Be sure to either not commit this file or to use private source control. The best idea is to only have secrets live on machines ephemerally (usually as environment variables). You especially should not use the keys that I provided here!

Third-party registration and login

Now we need to install and implement the various third-party registration strategies. To install third-party registration strategies run the following command:

```
npm install --save passport-twitter passport-google-oAuth passport-github
```

Most of these are extraordinarily similar, so I will only show the TwitterStrategy, as follows:

```
passport.use(new TwitterStrategy(config.twitter, function(req,
  accessToken, tokenSecret, profile, done) {
  User.findOne({ twitter: profile.id }, function(err,
  existingUser) {
    if (existingUser) return done(null, existingUser);
    var user = new User();
    // Twitter will not provide an email address. Period.
    // But a person's twitter username is guaranteed to be
    unique
    // so we can "fake" a twitter email address as follows:
    // username@twitter.mydomain.com
    user.email = profile.username + "@twitter." + config.domain +
    ".com";
    user.twitter = profile.id;
    user.tokens.push({ kind: 'twitter', accessToken:
    accessToken, tokenSecret: tokenSecret });
    user.profile.name = profile.displayName;
    user.profile.location = profile._json.location;
    user.profile.picture = profile._json.profile_image_url;
    user.save(function(err) {
      done(err, user);
    });
  });
}));
```

Here, I included one example of how we would do this. First, we pass a new TwitterStrategy to passport. The TwitterStrategy takes our Twitter keys and callback information and a callback is used to make sure we can register the user with that information. If the user is already registered, then it's a no-op; otherwise we save their information and pass along the error and/or successfully saved user to the callback. For the others, refer to the source.

Profile pages

It is finally time to add profile pages for each of our users. To do so, we're going to discuss more about Express routing and how to pass request-specific data to Jade templates. Often times when writing a server, you want to capture some portion of the URL to use in the controller; this could be a user id, username, or anything! We'll use Express's ability to capture URL parts to get the id of the user whose profile page was requested.

URL params

Express, like any good web framework, supports extracting data from URL parts. For example, you can do the following:

```
app.get('/users/:id', function(req, res, next) {
  console.log(req.params.id);
})
```

In the preceding example, we will print whatever comes after `/users/` in the request URL. This allows an easy way to specify per user routes, or routes that only make sense in the context of a specific user, that is, a profile page only makes sense when you specify a specific user. We will use this kind of routing to implement our profile page. For now, we want to make sure that only the logged-in user can see their own profile page (we can change this functionality later):

```
app.get('/users/:id', function(req, res, next) {
  if (!req.user || (req.user.id != req.params.id)) {
    return next('Not found');
  }
  res.render('users/profile', { user: req.user.toJSON() });
});
```

Here, we check first that the user is signed in and that the requested user's id is the same as the logged-in user's id. If it isn't, then we return an error. If it is, then we render the `users/profile.jade` template with `req.user` as the data.

Profile templates

We already looked at models and controllers at length, but our templates have been underwhelming. Finally, we'll show how to write some basic Jade templates. This section will serve as a brief introduction to the Jade templating language, but does not try to be comprehensive. The code for Profile templates is as follows:

```
html
  body
    h1
      =user.email
    h2
      =user.created_at
    - for (var prop in user.profile)
      if user.profile[prop]
        h4
          =prop + " = " + user.profile[prop]
```

Notably, because in the controller we passed in the user to the view, we can access the variable user and it refers to the logged-in user! We can execute arbitrary JavaScript to render into the template by prefixing it with `= --`. In these blocks, we can do anything we would normally do, including string concatenation, method invocation, and so on.

Similarly, we can include JavaScript code that is not intended to be written as HTML by prefixing it with `-` like we did with the `for` loop. This basic template prints out the user's e-mail, the `created_at` timestamp, as well as all of the properties in their profile, if any.



For a more in-depth look at Jade, please see <http://jade-lang.com/reference/>.



Testing

Testing is essential for any application. I will not dwell on the whys, but instead assume that you are angry with me for skipping this topic in the previous sections. Testing Express applications tend to be relatively straightforward and painless. The general format is that we make fake requests and then make certain assertions about the responses.

We could also implement finer-grained unit tests for more complex logic, but up until now almost everything we did is straightforward enough to be tested on a per route basis. Additionally, testing at the API level provides a more realistic view of how real customers will be interacting with your website and makes tests less brittle in the face of refactoring code.

Introducing Mocha

Mocha is a simple, flexible, test framework runner. First, I would suggest installing Mocha globally so you can easily run tests from the command line as follows:

```
$ npm install --save-dev -g mocha
```

The `--save-dev` option saves `mocha` as a development dependency, meaning we don't have to install Mocha on our production servers. Mocha is just a test runner. We also need an assertion library. There are a variety of solutions, but `should.js` syntax, written by the same person as Express and Mocha, gives a clean syntax to make assertions:

```
$ npm install --save-dev should
```

The `should.js` syntax provides BDD assertions, such as `'hello'.should.equal('hello')` and `[1, 2].should.have.length(2)`. We can start with a Hello World test example by creating a test directory with a single file, `hello-world.js`, as shown in the following code:

```
var should = require('should');

describe('The World', function() {
  it('should say hello', function() {
    'Hello, World'.should.equal('Hello, World');
  });
  it('should say hello asynchronously!', function(done) {
    setTimeout(function() {
      'Hello, World'.should.equal('Hello, World');
      done();
    }, 300);
  });
});
```

We have two different tests both in the same namespace, `The World`. The first test is an example of a synchronous test. Mocha executes the function we give to it, sees that no exception gets thrown and the test passes. If, instead, we accept a `done` argument in our callback, as we do in the second example, Mocha will intelligently wait until we invoke the callback before checking the validity of our test. For the most part, we will use the second version, in which we must explicitly invoke the `done` argument to finish our test because it makes more sense to test Express applications.

Now, if we go back to the command line, we should be able to run Mocha (or `node_modules/.bin/mocha` if you didn't install it globally) and see that both of the tests we wrote pass!

Testing API endpoints

Now that we have a basic understanding of how to run tests using Mocha and make assertions with `should` syntax, we can apply it to test local user registration. First, we need to introduce another `npm` module that will help us test our server programmatically and make assertions about what kind of responses we expect. The library is called `supertest`:

```
$ npm install --save-dev supertest
```

The library makes testing Express applications a breeze and provides chainable assertions. Let's take a look at an example usage to test our create user route, as shown in the following code:

```
var should = require('should'),
    request = require('supertest'),
    app = require('../server').app,
    User = require('mongoose').model('User');

describe('Users', function() {
  before(function(done) {
    User.remove({}, done);
  });
  describe('registration', function() {
    it('should register valid user', function(done) {
      request(app)
        .post('/users/register')
        .send({
```

```
        email: "test@example.com",
        password: "hello world"
    })
    .expect(302)
    .end(function(err, res) {
        res.text.should.containEql("Redirecting to /");
        done(err);
    });
});
});
});
```

First, notice that we used two namespaces: `Users` and `registration`. Now, before we run any tests, we remove all users from the database. This is useful to ensure we know where we're starting the tests. This will delete all of your saved users though, so it's useful to use a different database in the test environment. Node detects the environment by looking at the `NODE_ENV` environment variable. Typically it is `test`, `development`, `staging`, or `production`. We can do so by changing the database URL in our configuration file to use a different local database when in a test environment and then run Mocha tests with `NODE_ENV=test mocha`.

Now, on to the interesting bits! Supertest exposes a chainable API to make requests and assertions about responses. To make a request, we use `request(app)`. From there, we specify the HTTP method and path. Then, we can specify a JSON body to send to the server; in this case, an example user registration form. On registration, we expect a redirect, which is a `302` response. If that assertion fails, then the `err` argument in our callback will be populated, and the test will fail when we use `done(err)`. Additionally, we validate that we were redirected to the route we expect, the server root `/`.

Automate builds and deploys

All of this development is relatively worthless without a smooth process to build and deploy your application. Fortunately, the node community has written a variety of task runners. Among these are Grunt and Gulp, two of the most popular task runners. Both work seamlessly with Express and provide a set of utilities for us to use, including concatenating and uglifying JavaScript, compiling sass/less, and reloading the server on local file changes. We'll focus on Grunt, for simplicity.

Introducing the Gruntfile

Grunt itself is a simple task runner, but its extensibility and plugin architecture lets you install third-party scripts to run in predefined tasks. To give us an idea of how we might use Grunt, we're going to write our `css` in `sass` and then use Grunt to compile `sass` to `css`. Through this example, we'll explore the different ideas that Grunt introduces. First, you need to install `cli` globally to install the plugin that compiles `sass` to `css`:

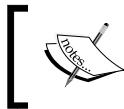
```
$ npm install -g grunt-cli  
$ npm install --save grunt grunt-contrib-sass
```

Now we need to create `Gruntfile.js`, which contains instructions for all of the tasks and build targets that we need. To do this perform the following:

```
// Gruntfile.js  
module.exports = function(grunt) {  
    grunt.loadNpmTasks('grunt-contrib-sass');  
    grunt.initConfig({  
        sass: {  
            dist: {  
                files: [{  
                    expand: true,  
                    cwd: "public/styles",  
                    src: ["**.scss"],  
                    dest: "dist/styles",  
                    ext: ".css"  
                }]  
            }  
        }  
    });  
}
```

Let's go over the major parts. Right at the top, we require the plugin we will use, `grunt-contrib-sass`. This tells `grunt` that we are going to configure a task called `sass`. In our definition of the task `sass`, we specify a target, `dist`, which is commonly used for tasks that produce production files (minified, concatenated, and so on).

In that task, we build our file list dynamically, telling Grunt to look in `/public/styles/` recursively for all `.scss` files, then compile them all to the same paths in `/dist/styles`. It is useful to have two parallel static directories, one for development and one for production, so we don't have to look at minified code in development. We can invoke this target by executing `grunt sass` or `grunt sass:dist`.



It is worth noting that we don't explicitly concatenate the files in this task, but if we use `@imports` in our main `sass` file, the compiler will concatenate everything for us.

We can also configure Grunt to run our test suite. To do this, let's add another plugin `-- npm install --save-dev grunt-mocha-test`. Now we have to add the following code to our `Gruntfile.js` file:

```
grunt.loadNpmTasks('grunt-mocha-test');
grunt.registerTask('test', 'mochaTest');
...

mochaTest: {
  test: {
    src: ["test/**.js"]
  }
}
```

Here, the task is called `mochaTest` and we register a new task called `test` that simply delegates to the `mochaTest` task. This way, it is easier to remember how to run tests. Similarly, we could have specified a list of tasks to run if we passed an array of strings as the second argument to `registerTask`. This is a sampling of what can be accomplished with Grunt. For an example of a more robust `Gruntfile`, check out the source.

Continuous integration with Travis

Travis CI provides free continuous integration for open source projects as well as paid options for closed source applications. It uses a git hook to automatically test your application after every push. This is useful to ensure no regression was introduced. Also, there could be dependency problems only revealed in CI that local development masks; Travis is the first line of defense for these bugs. It takes your source, runs `npm install` to install the dependencies specified in `package.json`, and then runs the `npm test` to run your test suite.

Travis accepts a configuration file called `travis.yml`. These typically look like this:

```
language: node_js
node_js:
  - "0.11"
  - "0.10"
  - "0.8"
services:
  - mongodb
```

We can specify the versions of node that we want to test against as well as the services that we rely on (specifically MongoDB). Now we have to update our test command in `package.json` to run `grunt test`. Finally, we have to set up a webhook for the repository in question. We can do this on Travis by enabling the repository. Now we just have to push our changes and Travis will make sure all the tests pass! Travis is extremely flexible and you can use it to accomplish most tasks related to continuous integration, including automatically deploying successful builds.

Deploying Node.js applications

One of the easiest ways to deploy Node.js applications is to utilize Heroku, a platform as a service provider. Heroku has its own toolbelt to create and deploy Heroku apps from your machine. Before getting started with Heroku, you will need to install its toolbelt.



Please go to <https://toolbelt.heroku.com/> to download the Heroku toolbelt.



Once installed, you can log in to Heroku or register via the web UI and then run Heroku login. Heroku uses a special file, called the Procfile, which specifies exactly how to run your application.

1. Our Procfile looks like this:

```
web: node server.js
```

Extraordinarily simple: in order to run the web server, just run `node server.js`.

2. In order to verify that our Procfile is correct, we can run the following locally:

```
$ foreman start
```

3. Foreman looks at the Procfile and uses that to try to start our server. Once that runs successfully, we need to create a new application and then deploy our application to Heroku. Be sure to commit the Procfile to version control:

```
$ heroku create  
$ git push heroku master
```

Heroku will create a new application and URL in Heroku, as well as a git remote repository named heroku. Pushing that remote actually triggers a deploy of your code.

If you do all of this, unfortunately your application will not work. We don't have a Mongo instance for our application to talk to!

4. First we have to request MongoDB from Heroku:

```
$ heroku addons:add mongolab // don't worry, it's free
```

This spins up a shared MongoDB instance and gives our application an environment variable named MONOLAB_URI, which we should use as our MongoDB connect URI. We need to change our configuration file to reflect these changes.

In our configuration file, in production, for our database URL, we should look at the environment variable MONGOLAB_URI. Also, be sure that Express is listening on process.env.PORT || 3000, or else you will receive strange errors and/or timeouts.

5. With all of that set up, we can commit our changes and push the changes once again to Heroku. Hopefully, this time it works! To view the application logs for debugging purposes, just use the Heroku toolbelt:

```
$ heroku logs
```

6. One last thing about deploying Express applications: sometimes applications crash, software isn't perfect. We should anticipate crashes and have our application respond accordingly (by restarting itself). There are many server monitoring tools, including pm2 and forever. We use forever because of its simplicity.

```
$ npm install --save forever
```

7. Then, we update our Procfile to reflect our use of forever:

```
// Procfile
web: node_modules/.bin/forever server.js
```

Now, forever will automatically restart our application, if it crashes for any strange reason. You can also set up Travis to automatically push successful builds to your server, but that goes beyond the deployment we will do in this book.

Summary

In this chapter, we got our feet wet in the world of node and using the Express framework. We went over everything from Hello World and MVC to testing and deployments. You should feel comfortable using basic Express APIs, but also feel empowered to own a Node.js application across the entire stack.

In the following chapters, we will build on the core ideas introduced in this chapter in order to create rich user experiences and compelling applications.

2

A Robust Movie API

We will build a movie API that allows you to add actor and movie information to a database and connect actors with movies, and vice versa. This will make use of the information introduced in *Chapter 1, Building a Basic Express Site*, and give you a hands-on feel for what Express.js offers. We will cover the following topics in this chapter:

- Folder structure and organization
- Responding to CRUD operations
- Object modeling with Mongoose
- Generating unique IDs
- Testing

Folder structure and organization

Folder structure is a very controversial topic. Though there are many clean ways to structure your project, we will use the following code for the remainder of our chapters:

```
chapter2
├── app.js
├── package.json └── node_modules
|   └── npm package folders └── src
|       ├── lib
|       ├── models
|       ├── routes
|       └── test
```

Let's take a look this at in detail:

- `app.js`: It is conventional to have the main `app.js` file in the root directory. The `app.js` is the entry point of our application and will be used to launch the server.
- `package.json`: As with any Node.js app, we have `package.json` in the root folder specifying our application name and version as well as all of our npm dependencies.
- `node_modules`: The `node_modules` folder and its content are generated via npm installation and should usually be ignored in your version control of choice because it depends on the platform the app runs on. Having said that, according to the npm FAQ, it is probably better to commit the `node_modules` folder as well.

 Check `node_modules` into git for things you deploy, such as websites and apps. Do not check `node_modules` into git for libraries and modules intended to be reused.
Refer to the following article to read more about the rationale behind this:
<http://www.futurealoof.com/posts/nodemodules-in-git.html>

- `src`: The `src` folder contains all the logic of the application.
- `lib`: Within the `src` folder, we have the `lib` folder, which contains the core of the application. This includes the middleware, routes, and creating the database connection.
- `models`: The `models` folder contains our `mongoose` models, which defines the structure and logic of the models we want to manipulate and save.
- `routes`: The `routes` folder contains the code for all the endpoints the API is able to serve.
- `test`: The `test` folder will contain our functional tests using Mocha as well as two other node modules, `should` and `supertest`, to make it easier to aim for 100 percent coverage.

Responding to CRUD operations

The term CRUD refers to the four basic operations one can perform on data: create, read, update, and delete. Express gives us an easy way to handle those operations by supporting the basic methods `GET`, `POST`, `PUT`, and `DELETE`:

- `GET`: This method is used to retrieve the existing data from the database. This can be used to read single or multiple rows (for SQL) or documents (for MongoDB) from the database.
- `POST`: This method is used to write new data into the database, and it is common to include a JSON payload that fits the data model.
- `PUT`: This method is used to update existing data in the database, and a JSON payload that fits the data model is often included for this method as well.
- `DELETE`: This method is used to remove an existing row or document from the database.

Express 4 has dramatically changed from version 3. A lot of the core modules have been removed in order to make it even more lightweight and less dependent. Therefore, we have to explicitly `require` modules when needed.

One helpful module is `body-parser`. It allows us to get a nicely formatted body when a `POST` or `PUT` HTTP request is received. We have to add this middleware before our business logic in order to use its result later. We write the following in `src/lib/parser.js`:

```
var bodyParser = require('body-parser');
module.exports = function(app) {
  app.use(bodyParser.json());
  app.use(bodyParser.urlencoded({ extended: false }));
};
```

The preceding code is then used in `src/lib/app.js` as follows:

```
var express = require('express'); var app = express();
require('./parser')(app);
module.exports = app;
```

The following example allows you to respond to a GET request on `http://host/path`. Once a request hits our API, Express will run it through the necessary middleware as well as the following function:

```
app.get('/path/:id', function(req, res, next) {  
  res.status(200).json({ hello: 'world' });  
});
```

The first parameter is the path we want to handle a GET function. The path can contain parameters prefixed with `:`. Those path parameters will then be parsed in the request object.

The second parameter is the callback that will be executed when the server receives the request. This function gets populated with three parameters: `req`, `res`, and `next`.

The `req` parameter represents the HTTP request object that has been customized by Express and the middlewares we added in our applications. Using the path `http://host/path/:id`, suppose a GET request is sent to `http://host/path/1?a=1&b=2`. The `req` object would be the following:

```
{  
  params: { id: 1 }, query: { a: 1, b: 2 }  
}
```

The `params` object is a representation of the path parameters. The `query` is the query string, which are the values stated after `?` in the URL. In a POST request, there will often be a body in our request object as well, which includes the data we wish to place in our database.

The `res` parameter represents the response object for that request. Some methods, such as `status()` or `json()`, are provided in order to tell Express how to respond to the client.

Finally, the `next()` function will execute the next middleware defined in our application.

Retrieving an actor with GET

Retrieving a movie or actor from the database consists of submitting a GET request to the route: `/movies/:id` or `/actors/:id`. We will need a unique ID that refers to a unique movie or actor:

```
app.get('/actors/:id', function(req, res, next) {  
  //Find the actor object with this :id  
  //Respond to the client  
});
```

Here, the URL parameter `:id` will be placed in our request object. Since we call the first variable in our callback function `req` as before, we can access the URL parameter by calling `req.params.id`.

Since an actor may be in many movies and a movie may have many actors, we need a nested endpoint to reflect this as well:

```
app.get('/actors/:id/movies', function(req, res, next) {
  //Find all movies the actor with this :id is in
  //Respond to the client
});
```

If a bad GET request is submitted or no actor with the specified ID is found, then the appropriate status code bad request `400` or not found `404` will be returned. If the actor is found, then success request `200` will be sent back along with the actor information. On a success, the response JSON will look like this:

```
{
  "_id": "551322589911fefaf1f656cc5", "id": 1,
  "name": "AxiomZen", "birth_year": 2012, "__v": 0, "movies": []
}
```

Creating a new actor with POST

In our API, creating a new movie in the database involves submitting a `POST` request to `/movies` or `/actors` for a new actor:

```
app.post('/actors', function(req, res, next) {
  //Save new actor
  //Respond to the client
});
```

In this example, the user accessing our API sends a `POST` request with data that would be placed into `request.body`. Here, we call the first variable in our callback function `req`. Thus, to access the body of the request, we call `req.body`.

The request body is sent as a JSON string; if an error occurs, a `400` (bad request) status would be sent back. Otherwise, a `201` (created) status is sent to the response object. On a success request, the response will look like the following:

```
{
  "__v": 0, "id": 1,
  "name": "AxiomZen", "birth_year": 2012,
  "_id": "551322589911fefaf1f656cc5", "movies": []
}
```

Updating an actor with PUT

To update a movie or actor entry, we first create a new route and submit a `PUT` request to `/movies/:id` or `/actors/:id`, where the `:id` parameter is unique to an existing movie/actor. There are two steps to an update. We first find the movie or actor by using the unique id and then we update that entry with the body of the request object, as shown in the following code:

```
app.put('/actors/:id', function(req, res) {
  //Find and update the actor with this :id
  //Respond to the client
});
```

In the request, we would need `request.body` to be a JSON object that reflects the actor fields to be updated. The `request.params.id` would still be a unique identifier that refers to an existing actor in the database as before. On a successful update, the response JSON looks like this:

```
{
  "_id": "551322589911fefaf656cc5",
  "id": 1,
  "name": "Axiomzen", "birth_year": 99, "__v": 0, "movies": []
}
```

Here, the response will reflect the changes we made to the data.

Removing an actor with DELETE

Deleting a movie is as simple as submitting a `DELETE` request to the same routes that were used earlier (specifying the ID). The actor with the appropriate id is found and then deleted:

```
app.delete('/actors/:id', function(req, res) {
  //Remove the actor with this :id
  //Respond to the client
});
```

If the actor with the unique id is found, it is then deleted and a response code of `204` is returned. If the actor cannot be found, a response code of `400` is returned. There is no response body for a `DELETE()` method; it will simply return the status code of `204` on a successful deletion.

Our final endpoints for this simple app will be as follows:

```
//Actor endpoints
app.get('/actors', actors.getAll);
app.post('/actors', actors.createOne);
app.get('/actors/:id', actors.getOne);
app.put('/actors/:id', actors.updateOne);
app.delete('/actors/:id', actors.deleteOne)
app.post('/actors/:id/movies', actors.addMovie);
app.delete('/actors/:id/movies/:mid', actors.deleteMovie);
//Movie endpoints
app.get('/movies', movies.getAll);
app.post('/movies', movies.createOne);
app.get('/movies/:id', movies.getOne);
app.put('/movies/:id', movies.updateOne);
app.delete('/movies/:id', movies.deleteOne);
app.post('/movies/:id/actors', movies.addActor);
app.delete('/movies/:id/actors/:aid', movies.deleteActor);
```

In Express 4, there is an alternative way to describe your routes. Routes that share a common URL, but use a different HTTP verb, can be grouped together as follows:

```
app.route('/actors')
  .get(actors.getAll)
  .post(actors.createOne);
app.route('/actors/:id')
  .get(actors.getOne)
  .put(actors.updateOne)
  .delete(actors.deleteOne);
app.post('/actors/:id/movies', actors.addMovie);
app.delete('/actors/:id/movies/:mid', actors.deleteMovie);
app.route('/movies')
  .get(movies.getAll)
  .post(movies.createOne);
app.route('/movies/:id')
  .get(movies.getOne)
  .put(movies.updateOne)
  .delete(movies.deleteOne);
app.post('/movies/:id/actors', movies.addActor);
app.delete('/movies/:id/actors/:aid', movies.deleteActor);
```

Whether you prefer it this way or not is up to you. At least now you have a choice!

We have not discussed the logic of the function being run for each endpoint. We will get to that shortly.

Express allows us to easily CRUD our database objects, but how do we model our objects?

Object modeling with Mongoose

Mongoose is an object data modeling library (ODM) that allows you to define schemas for your data collections. You can find out more about Mongoose on the project website: <http://mongoosejs.com/>.

To connect to a MongoDB instance using the `mongoose` variable, we first need to install npm and save Mongoose. The `save` flag automatically adds the module to your `package.json` with the latest version, thus, it is always recommended to install your modules with the `save` flag. For modules that you only need locally (for example, Mocha), you can use the `savedev` flag.

For this project, we create a new file `db.js` under `/src/lib/db.js`, which requires Mongoose. The local connection to the `mongodb` database is made in `mongoose.connect` as follows:

```
var mongoose = require('mongoose');
module.exports = function(app)
{
  mongoose.connect('mongodb://localhost/movies', {
    mongoose: { safe: true
  }, function(err) { if (err)
  {
    return console.log('Mongoose - connection error:', err);
  }
});
return mongoose;
};
```

In our movies database, we need separate schemas for actors and movies. As an example, we will go through object modeling in our actor database `/src/models/actor.js` by creating an actor schema as follows:

```
// /src/models/actor.js
var mongoose = require('mongoose');
var generateId = require('./plugins/generateId');
```

```
var actorSchema = new mongoose.Schema({
  id: {
    type: Number,
    required: true,
    index: {
      unique: true
    }
  },
  name: {
    type: String,
    required: true
  },
  birth_year: {
    type: Number,
    required: true
  },
  movies: [
    {
      type: mongoose.Schema.ObjectId,
      ref: 'Movie'
    }
  ]
});
actorSchema.plugin(generateId());
module.exports = mongoose.model('Actor', actorSchema);
```

Each actor has a unique id, a name, and a birth year. The entries also contain validators such as the type and boolean value that are required. The model is exported upon definition (`module.exports`), so that we can reuse it directly in the app.

Alternatively, you could fetch each model through Mongoose using `mongoose.model('Actor', actorSchema)`, but this would feel less explicitly coupled compared to our approach of directly requiring it.

Similarly, we need a movie schema as well. We define the movie schema as follows:

```
// /src/models/movies.js
var movieSchema = new mongoose.Schema({
  id: {
    type: Number,
    required: true,
    index: {
      unique: true
    }
},
```

```
title: {
  type: String,
  required: true
},
year: {
  type: Number,
  required: true
},
actors: [
  {
    type : mongoose.Schema.ObjectId,
    ref : 'Actor'
  }
);
movieSchema.plugin(generateId());
module.exports = mongoose.model('Movie', movieSchema);
```

Generating unique IDs

In both our movie and actor schemas, we used a plugin called `generateId()`.

While MongoDB automatically generates `ObjectID` for each document using the `_id` field, we want to generate our own IDs that are more human readable and hence friendlier. We also would like to give the user the opportunity to select their own id of choice.

However, being able to choose an id can cause conflicts. If you were to choose an id that already exists, your `POST` request would be rejected. We should autogenerate an ID if the user does not pass one explicitly.

Without this plugin, if either an actor or a movie is created without an explicit ID passed along by the user, the server would complain since the ID is required.

We can create middleware for Mongoose that assigns an `id` before we persist the object as follows:

```
// /src/models/plugins/generateId.js
module.exports = function() {

  return function generateId(schema) {
    schema.pre('validate',function(next, done) {
      var instance = this;
      var model = instance.model(instance.constructor.modelName);
```

```
if( instance.id == null ) {
  model.findOne().sort("-id").exec(function(err,maxInstance) {
    if (err) {
      return done(err);
    } else {
      var maxId = maxInstance.id || 0;
      instance.id = maxId+1;
      done();
    }
  })
} else {
  done();
}
}
};
```

There are a few important notes about this code.

See what we did to get the `var model`? This makes the plugin generic so that it can be applied to multiple Mongoose schemas.

Notice that there are two callbacks available: `next` and `done`. The `next` variable passes the code to the next pre-validation middleware. That's something you would usually put at the bottom of the function right after you make your asynchronous call. This is generally a good thing since one of the advantages of asynchronous calls is that you can have many things running at the same time.

However, in this case, we cannot call the `next` variable because it would conflict with our model definition of `id` required. Thus, we just stick to using the `done` variable when the logic is complete.

Another concern arises due to the fact that MongoDB doesn't support transactions, which means you may have to account for this function failing in some edge cases. For example, if two calls to `POST /actor` happen at the same time, they will both have their IDs auto incremented to the same value.

Now that we have the code for our `generateId()` plugin, we require it in our actor and movie schema as follows:

```
var generateId = require('./plugins/generateId');
actorSchema.plugin(generateId);
```

Validating your database

Each key in the Mongoose schema defines a property that is associated with a SchemaType. For example, in our `actors.js` schema, the actor's name key is associated with a string SchemaType. String, number, date, buffer, boolean, mixed, objectId, and array are all valid schema types.

In addition to schema types, numbers have min and max validators and strings have enum and match validators. Validation occurs when a document is being saved (`.save()`) and will return an error object, containing type, path, and value properties, if the validation has failed.

Extracting functions to reusable middleware

We can use our anonymous or named functions as middleware. To do so, we would export our functions by calling `module.exports` in `routes/actors.js` and `routes/movies.js`:

Let's take a look at our `routes/actors.js` file. At the top of this file, we require the Mongoose schemas we defined before:

```
var Actor = require('../models/actor');
```

This allows our variable `Actor` to access our MongoDB using mongo functions such as `find()`, `create()`, and `update()`. It will follow the schema defined in the file `/models/actor`.

Since actors are in movies, we will also need to require the `Movie` schema to show this relationship by the following.

```
var Movie = require('../models/movie');
```

Now that we have our schema, we can begin defining the logic for the functions we described in endpoints. For example, the endpoint `GET /actors/:id` will retrieve the actor with the corresponding ID from our database. Let's call this function `getOne()`. It is defined as follows:

```
getOne: function(req, res, next) { Actor.findOne({ id: req.params.id }) .populate('movies') .exec(function(err, actor) {
```

```
    if (err) return res.status(400).json(err); if (!actor) return
    res.status(404).json(); res.status(200).json(actor);
  });
},
```

Here, we use the mongo `findOne()` method to retrieve the actor with `id: req.params.id`. There are no joins in MongoDB so we use the `.populate()` method to retrieve the movies the actor is in.

The `.populate()` method will retrieve documents from a separate collection based on its `ObjectId`.

This function will return a status 400 if something went wrong with our Mongoose driver, a status 404 if the actor with `:id` is not found, and finally, it will return a status 200 along with the JSON of the actor object if an actor is found.

We define all the functions required for the actor endpoints in this file. The result is as follows:

```
// /src/routes/actors.js
var Actor = require('../models/actor');
var Movie = require('../models/movie');

module.exports = {

  getAll: function(req, res, next) {
    Actor.find(function(err, actors) {
      if (err) return res.status(400).json(err);

      res.status(200).json(actors);
    });
  },

  createOne: function(req, res, next) {
    Actor.create(req.body, function(err, actor) {
      if (err) return res.status(400).json(err);

      res.status(201).json(actor);
    });
  },
}
```

```
getOne: function(req, res, next) {
  Actor.findOne({ id: req.params.id })
    .populate('movies')
  .exec(function(err, actor) {
    if (err) return res.status(400).json(err);
    if (!actor) return res.status(404).json();

    res.status(200).json(actor);
  });
},

updateOne: function(req, res, next) {
  Actor.findOneAndUpdate({ id: req.params.id }, req.
body, function(err, actor) {
    if (err) return res.status(400).json(err);
    if (!actor) return res.status(404).json();

    res.status(200).json(actor);
  });
},

deleteOne: function(req, res, next) {
  Actor.findOneAndRemove({ id: req.params.id }, function(err) {
    if (err) return res.status(400).json(err);

    res.status(204).json();
  });
},

addMovie: function(req, res, next) {
  Actor.findOne({ id: req.params.id }, function(err, actor) {
    if (err) return res.status(400).json(err);
    if (!actor) return res.status(404).json();

    Movie.findOne({ id: req.body.id }, function(err, movie) {
      if (err) return res.status(400).json(err);
      if (!movie) return res.status(404).json();
    });
  });
},
```

```
    actor.movies.push(movie);
    actor.save(function(err) {
        if (err) return res.status(500).json(err);

        res.status(201).json(actor);
    });
})
})
});

},
}

deleteMovie: function(req, res, next) {
    Actor.findOne({ id: req.params.id }, function(err, actor) {
        if (err) return res.status(400).json(err);
        if (!actor) return res.status(404).json();

        actor.movies = [];
        actor.save(function(err) {
            if (err) return res.status(400).json(err);

            res.status(204).json(actor);
        })
    });
}
};
```

For all of our movie endpoints, we need the same functions but applied to the movie collection.

After exporting these two files, we require them in `app.js` (`/src/lib/app.js`) by simply adding:

```
require('../routes/movies'); require('../routes/actors');
```

By exporting our functions as reusable middleware, we keep our code clean and can refer to functions in our CRUD calls in the `/routes` folder.

Testing

Mocha is used as the test framework along with `should.js` and `supertest`. The principles behind why we use testing in our apps along with some basics on Mocha are covered in *Chapter 1, Building a Basic Express Site*. Testing `supertest` lets you test your HTTP assertions and testing API endpoints.

The tests are placed in the root folder `/test`. Tests are completely separate from any of the source code and are written to be readable in plain English, that is, you should be able to follow along with what is being tested just by reading through them. Well-written tests with good coverage can serve as a `readme` for its API, since it clearly describes the behavior of the entire app.

The initial setup to test our movies API is the same for both `/test/actors.js` and `/test/movies.js` and will look familiar if you have read *Chapter 1, Building a Basic Express Site*:

```
var should = require('should'); var assert = require('assert');
var request = require('supertest');
var app = require('../src/lib/app');
```

In `src/test/actors.js`, we test the basic CRUD operations: creating a new actor object, retrieving, editing, and deleting the actor object. An example test for the creation of a new actor is shown as follows:

```
describe('Actors', function() {
  describe('POST actor', function() {
    it('should create an actor', function(done) {
      var actor = {
        'id': '1',
        'name': 'AxiomZen', 'birth_year': '2012',
      };

      request(app)
        .post('/actors')
        .send(actor)
        .expect(201, done)
    });
  });
});
```

We can see that the tests are readable in plain English. We create a new POST request for a new actor to the database with the id of 1, name of AxiomZen, and birth_year of 2012. Then, we send the request with the `.send()` function. Similar tests are present for GET and DELETE requests as given in the following code:

```
describe('GET actor', function() {
  it('should retrieve actor from db', function(done) {
    request(app)
      .get('/actors/1')
      .expect(200, done);
  });
  describe('DELETE actor', function() {
    it('should remove a actor', function(done) {
      request(app)
        .delete('/actors/1')
        .expect(204, done);
    });
  });
});
```

To test our PUT request, we will edit the name and birth_year of our first actor as follows:

```
describe('PUT actor', function() {
  it('should edit an actor', function(done) {
    var actor = {
      'name': 'ZenAxiom',
      'birth_year': '2011'
    };

    request(app)
      .put('/actors/1')
      .send(actor)
      .expect(200, done);
  });

  it('should have been edited', function(done) {
    request(app)
      .get('/actors/1')
      .expect(200)
```

```
        .end(function(err, res) {
          res.body.name.should.eql('ZenAxiom');
          res.body.birth_year.should.eql(2011);
          done();
        });
      });
    });
  });
});
```

The first part of the test modifies the actor name and birth_year keys, sends a PUT request for /actors/1 (1 is the actors id), and then saves the new information to the database. The second part of the test checks whether the database entry for the actor with id 1 has been changed. The name and birth_year values are checked against their expected values using .should.eql().

In addition to performing CRUD actions on the actor object, we can also perform these actions to the movies we add to each actor (associated by the actor's ID).

The following snippet shows a test to add a new movie to our first actor (with the id of 1):

```
describe('POST /actors/:id/movies', function() {
  it('should successfully add a movie to the actor', function(done) {
    var movie = {
      'id': '1',
      'title': 'Hello World',
      'year': '2013'
    }
    request(app)
      .post('/actors/1/movies')
      .send(movie)
      .expect(201, done)
    });
  });

  it('actor should have array of movies now', function(done) {
    request(app)
      .get('/actors/1')
      .expect(200)
      .end(function(err, res) {
        res.body.movies.should.eql(['1']);
        done();
      });
    });
  });
});
```

The first part of the test creates a new movie object with `id`, `title`, and `year` keys, and sends a `POST` request to add the movies as an array to the actor with `id` of 1. The second part of the test sends a `GET` request to retrieve the actor with `id` of 1, which should now include an array with the new movie input.

We can similarly delete the movie entries as illustrated in the `actors.js` test file:

```
describe('DELETE /actors/:id/movies/:movie_id', function() {
  it('should successfully remove a movie from actor', function(done) {
    request(app)
      .delete('/actors/1/movies/1')
      .expect(200, done);
  });

  it('actor should no longer have that movie id', function(done) {
    request(app)
      .get('/actors/1')
      .expect(201)
      .end(function(err, res) {
        res.body.movies.should.eql([]);
        done();
      });
  });
});
```

Again, this code snippet should look familiar to you. The first part tests that sending a `DELETE` request specifying the actor ID and movie ID will delete that movie entry. In the second part, we make sure that the entry no longer exists by submitting a `GET` request to view the actor's details where no movies should be listed.

In addition to ensuring that the basic CRUD operations work, we also test our schema validations. The following code tests to make sure two actors with the same ID do not exist (IDs are specified as unique):

```
it('should not allow you to create duplicate actors', function(done) {
  var actor = {
    'id': '1',
    'name': 'AxiomZen',
    'birth_year': '2012',
  };
});
```

```
request(app)
  .post('/actors')
  .send(actor)
  .expect(400, done);
}) ;
```

We should expect code 400 (bad request) if we try to create an actor who already exists in the database.

A similar set of tests is present for `tests/movies.js`. The function and outcome of each test should be evident now.

Summary

In this chapter, we created a basic API that connects to MongoDB and supports CRUD methods. You should now be able to set up an API complete with tests, for any data, not just movies and actors!

The astute reader will have noticed that we have not addressed some issues in the current chapter such as dealing with race conditions in MongoDB. These will be clarified in detail in the following chapters.

We hope you found that this chapter has laid a good foundation for the Express and API setup.

3

Multiplayer Game API – Connect 4

Connect 4 is a turn-based two-player game, where each player would drop a chip down a column, with the objective to get four chip of the same color in a row. It can be vertical, horizontal, or diagonal.

In this chapter, we will build Connect4-as-a-Service. An API that allows you to build a game of Connect 4 on any client, be it a website, mobile app, or just play it from the command line; why not?

In *Chapter 1, Building a Basic Express Site*, and *Chapter 2, MMO Word Game*, we covered the most generic use cases for an Express backed API, which is to serve and persist data to and from a database. In this chapter, we'll cover something more fun. We'll build a multiplayer game API!

Some topics that will be covered include authentication, game state modeling, and validation middleware. Also, we will build an app using test-driven development with maximum code coverage.

For your reference, this is the folder structure of our app, which we will build throughout the chapter:

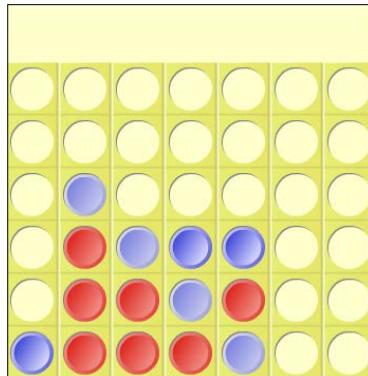
```
.
+-- README.md
+-- app.js
+-- config.js
+-- package.json
+-- src
|   +-- lib
|       +-- app.js
|       +-- connect4.js
|       +-- db.js
|       +-- parser.js
|       +-- utils.js
|       +-- validators.js
|   +-- models
|       +-- game.js
|   +-- routes
|       +-- games.js
+-- test
    +-- createGame.js
    +-- gameLogic.js
    +-- joinGame.js
    +-- makeMove.js
    +-- tieGame.js
```

How do you create a game? How do you join a game? How do you make a move? And how do you persist the game state in a DB?

It is always a good idea to start with the data structure. So let's get to it!

Modeling game state with Mongoose

We will represent the board as a 2-dimensional array, with the values being '`'x'`', '`'o'`', or '`' '`', representing the three possible states for each location on the grid. Here's an example, where player 2 wins the game:



This game state would be represented in an array as follows:

```
[ [ ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
  [ ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
  [ ' ', 'o', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
  [ ' ', 'x', 'o', 'o', 'o', ' ', ' ', ' ', ' ', ' ', ' ' ],
  [ ' ', 'x', 'x', 'o', 'x', ' ', ' ', ' ', ' ', ' ', ' ' ],
  [ 'o', 'x', 'x', 'x', 'o', ' ', ' ', ' ', ' ', ' ' ] ]
```

This would suffice if the game were to be played locally with the state being stored in memory. In our case, we want to play on the internet, so we will need a way to identify which game we are playing, as well as which player you are, and whose turn it is. A game document would look as follows:

```
{
  boardId: '<id>',
  p1Key: '<p1key>',
  p1Name: 'express',
  p2Key: '<p2key>',
  p2Name: 'koalala',
  columns: 7,
  rows: 6,
  status: 'Game in progress',
  winner: undefined,
  turn: 1,
  board: [...]
}
```

Here are the parameters:

Parameter	Description
boardId	This is a unique ID that you'll need if you want to take a look at the current game state.
p1Key	This is a secret token to identify player 1; we want to avoid the possibility of cheating of course
p1Name	This is player 1's name
p2Key	This is a secret token to identify player 2
p2Name	This is a player 2's name
turn	This is the total number of turns played on this board
rows	This is the number of rows of the game board
columns	This is the number of columns of the game board
board	This is the game state stored in a 2D array
status	This is either Game in progress or Game Over.
winner	This is the name of the winner once the game is over

Let's use the same app folder structure as was introduced in *Chapter 2, Building a Basic Express Site*, and let's define the preceding as a Mongoose model in `src/models/game.js`:

```
var mongoose = require('mongoose');

var gameSchema = new mongoose.Schema({
    type: String,
    required: true
},
{
    p2Key: {
        type: String,
        required: true
    },
    p1Name: {
        type: String,
        required: true
    },
    p2Name: {
        type: String
    },
    turn: {
        type: Number,
        required: true
    },
    boardId: {
        type: String,
        required: true,
        index: {
            unique: true
        }
    },
    board: {
        type: Array,
        required: true
    },
    rows: {
        type: Number,
        required: true
    },
    columns: {
        type: Number,
        required: true
    },
}
```

```
status: {
  type: String
},
winner: {
  type: String
}
);

module.exports = mongoose.model('Game', gameSchema);
```

Creating a new game

Now that we have defined the data structure of our game, let's start with implementing the logic to create and persist a new game document in the database, all the while following Test-Driven Development practices.

In order to create a new game, we need to accept a POST to `/create` with your name in the POST body:

```
{ name: 'player1' }
```

There are a few things we should think about:

- We need to return the board information to the user, and whether or not game creation was successful
- We need to ensure the player can access the game they just created, so we must send them the `boardId`
- In order for the player to identify themselves, we also need to ensure that we send them the `p1Key`, which will be needed for all future moves that Player One wishes to play to this board

Since we're building the game, we have the power to bend the rules of the game. So let's allow player 1 to optionally configure the size of the playing board! We should have a minimum size of 6×7 , though.

So let's start with the tests for creating a game and fetch the game information:

```
var expect = require('chai').expect,
request = require('supertest');

var app = require('../src/lib/app');
describe('Create new game | ', function() {
  var boardId;
```

```
it('should return a game object with key for player 1',
  function(done) {
    request(app).post('/create')
      .send({name: 'express'})
      .expect(200)
      .end(function(err, res) {
        var b = res.body;
        expect(b.boardId).to.be.a('string');
        expect(b.p1Key).to.be.a('string');
        expect(b.p1Name).to.be.a('string').and.equal('express');
        expect(b.turn).to.be.a('number').and.equal(1);
        expect(b.rows).to.be.a('number');
        expect(b.columns).to.be.a('number');

        // Make sure the board is a 2D array
        expect(b.board).to.be.an('array');
        for(var i = 0; i < b.board.length; i++){
          expect(b.board[i]).to.be.an('array');
        }

        // Store the boardId for reference
        boardId = b.boardId;
        done();
      });
  });
})
```



Throughout this chapter, we will use the `expect` assertion library. The only difference with `should` is the syntax, and the way it handles `undefined` more gracefully. The `should` library patches the object prototype, which means that if the object is `undefined`, it will throw a `TypeError: Cannot read property 'should' of undefined`.

The test will use `supertest` to simulate POSTing data to the `/create` endpoint, and we describe everything that we expect from the response.

1. Now let's create a `POST` route in `src/routes/games.js` to create a game in the database, and make the first test pass:

```
var Utils = require('../lib/utils');
var connect4 = require('../lib/connect4');
```

```

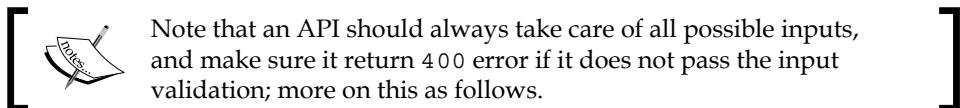
var Game = require('../models/game');

app.post('/create', function(req, res) {
    if(!req.body.name) {
        res.status(400).json({
            "Error": "Must provide name field!"
        });
    }

    var newGame = {
        p1Key: Utils.randomUUID(25),
        p2Key: Utils.randomUUID(25),
        boardId: Utils.randomUUID(6),
        p1Name: req.body.name,
        board: connect4.initializeBoard(req.body.rows,
        req.body.columns),
        rows: req.body.rows || app.get('config').MIN_ROWS,
        columns: req.body.columns || app.get('config').MIN_COLUMNS,
        turn: 1,
        status: 'Game in progress'
    };

    Game.create(newGame, function(err, game) {
        if (err) {
            return res.status(400).json(err);
        }
        game.p2Key = undefined;
        res.status(201).json(game);
    });
});

```



2. The `Utils.randomUUID()` method will return a random string, which we use to generate a token as well as `boardId`. Instead of defining it in the preceding file, let's package it up nicely in `src/lib/utils.js`:

```

var crypto = require('crypto');

module.exports = {
    randomUUID: function(len) {

```

```
        return crypto.randomBytes(Math.ceil(len/2))
            .toString('hex')
            .slice(0,len);
    }
}
```

All the game logic of Connect4 is in `src/lib/connect4.js`, which you can find in the Appendix. We'll use that library to initialize the board.

3. Also notice that rows and columns are optional arguments. We don't want to be hardcoding the default values in the code, so we have the following `config.js` file in the root folder:

```
module.exports = {
  MIN_ROWS: 6,
  MIN_COLUMNS: 7
};
```

4. As we initiate the app in `src/lib/app.js`, we can attach this config object onto the app object, so we have app-wide access to the config:

```
var express = require('express'),
  app = express(),
  config = require('../config'),
  db = require('./db');

app.set('config', config);
db.connectMongoDB();
require('./parser')(app);
require('../routes/games')(app);

module.exports = app;
```

By now, your first pass should pass—congratulations! We can now be rest assured that the `POST` endpoint is working, and will keep working as expected. It's a great feeling because if we ever break something in the future, the test will fail. Now you don't have to worry about it anymore and focus on your next task.

5. You do have to be diligent about getting as much code coverage as possible. For instance, we allow the client to customize the size of the board, but we have not written tests to test this feature yet, so let's get right to it:

```
it('should allow you to customize the size of the board',
function(done) {
  request(app).post('/create')
    .send({
```

```
        name: 'express',
        columns: 8,
        rows: 16
    })
    .expect(200)
    .end(function(err, res) {
        var b = res.body;
        expect(b.columns).to.equal(8);
        expect(b.rows).to.equal(16);
        expect(b.board).to.have.length(16);
        expect(b.board[0]).to.have.length(8);
        done();
    });
});
```

6. We should also enforce a minimum size of the board; otherwise, the game can't be played. Remember how we defined MIN_ROWS and MIN_COLUMNS in the config.js file? We can reuse that in our tests as well, without having to resort to hardcoded the tests. Now if we want to be changing the minimum size of the game, we can do it one place! As given in the following:

```
it('should not accept sizes < ' + MIN_COLUMNS + ' for
columns', function(done) {
    request(app).post('/create')
    .send({
        name: 'express',
        columns: 5,
        rows: 16
    })
    .expect(400)
    .end(function(err, res) {
        expect(res.body.error).to.equal('Number of columns
has to be >= ' + MIN_COLUMNS);
        done();
    });
});
```



```
it('should not accept sizes < ' + MIN_ROWS + ' rows',
function(done) {
    request(app).post('/create')
    .send({
        name: 'express',
        columns: 8,
        rows: -6
    })
    .expect(400)
    .end(function(err, res) {
        expect(res.body.error).to.equal('Number of rows
has to be >= ' + MIN_ROWS);
        done();
    });
});
```

```
        })
        .expect(400)
        .end(function(err, res) {
            expect(res.body.error).to.equal('Number of rows has
            to be >= ' + MIN_ROWS);
            done();
        });
    });
});
```

As described in the preceding test cases, we should make sure that if the player is customizing the size of the board, that the size is not less than the minimum size. There are many more validation checks that we'll be doing, so let's start to get a bit more organized.

Input validation

We should always check that the inputs we receive from a `POST` request are indeed what we expect, and return a `400` input error otherwise. This requires thinking about as many edge-case scenarios as possible. When an API is used by thousands of users, it is guaranteed that some users will abuse or misuse it, be it either intentional or unintentional. However, it is your responsibility to make the API as user-friendly as possible.

The only input validation that we covered in the preceding `/create` route is to make sure that there is a name in the `POST` body. Now we can just add two more `if` blocks to cover the board-size cases to make the tests pass.

In true TDD philosophy, you should write the least amount of code to make the tests pass first. They call it red-green-refactor. First, write tests that fail (red), make them pass as quickly as possible (green), and refactor after.

We urge you to try the preceding first. The following is the result after refactoring.

1. A lot of the input validation checks would be useful across multiple routes, so let's package it nicely as a collection of middleware in `src/lib/validators.js`:

```
// A collection of validation middleware

module.exports = function(app) {
    var MIN_COLUMNS = app.get('config').MIN_COLUMNS,
        MIN_ROWS = app.get('config').MIN_ROWS;
```

```

// Helper to return 400 error with a custom message
var _return400Error = function(res, message) {
    return res.status(400).json({
        error: message
    });
};

return {
    name: function(req, res, next) {
        if(!req.body.name) {
            return _return400Error(res, 'Must provide name
field!');
        }
        next();
    },
    columns: function(req, res, next) {
        if(req.body.columns && req.body.columns <
MIN_COLUMNS) {
            return _return400Error(res, 'Number of columns has
to be >= ' + MIN_COLUMNS);
        }
        next();
    },
    rows: function(req, res, next) {
        if(req.body.rows && req.body.rows < MIN_ROWS) {
            return _return400Error(res, 'Number of rows has to
be >= ' + MIN_ROWS);
        }
        next();
    }
}

```

The preceding packages three validation checkers in a reusable fashion. It returns an object with three middleware. Note how we DRYed up the code using a private `_return400Error` helper, to make it even cleaner.

2. Now we can refactor the `/create` route as follows:

```

module.exports = function(app) {
    // Initialize Validation middleware with app to use
config.js
    var Validate = require('../lib/validators')(app);

```

```
app.post('/create', [Validate.name, Validate.columns,
Validate.rows], function(req, res) {

  var newGame = {
    p1Key: Utils.randomUUID(25),
    p2Key: Utils.randomUUID(25),
    boardId: Utils.randomUUID(6),
    p1Name: req.body.name,
    board: connect4.initializeBoard(req.body.rows,
req.body.columns),
    rows: req.body.rows || app.get('config').MIN_ROWS,
    columns: req.body.columns || app.get('config').MIN_COLUMNS,
    turn: 1,
    status: 'Game in progress'
  };
  Game.create(newGame, function(err, game) {
    if (err) return res.status(400).json(err);

    game.p2Key = undefined;
    return res.status(201).json(game);
  });
});
}
```

This will create a nice separation of concerns, where each of the routes that we will define will accept an array of (reusable!) validation middleware that it has to go through, before it reaches the controller logic of the route.



Make sure your tests still pass before you proceed with the next endpoint.



Getting the game state

Both players need a way to check on the state of a game that they are interested in. To do this, we can send a GET request to /board/{boardId}. This will return the current state of the game, allowing players to see the state of the board, as well as whose turn is next.

We will create another endpoint to fetch a board, so let's first write the test for that:

```
it('should be able to fetch the board', function(done) {
  request(app).get("/board/" + boardId)
    .expect(200)
    .end(function(err, res) {
      var b = res.body;
      expect(b.boardId).to.be.a('string').and.equal(boardId);
      expect(b.turn).to.be.a('number').and.equal(1);
      expect(b.rows).to.be.a('number');
      expect(b.columns).to.be.a('number');
      expect(b.board).to.be.an('array');
      done();
    });
});
```

Note that we want to make sure that we don't accidentally leak the player tokens. The response should be basically identical to the one received by the player that most recently made a move as given in the following:

```
app.get('/board/:id', function(req, res) {
  Game.findOne({boardId: req.params.id}, function(err, game) {
    if (err) return res.status(400).json(err);

    res.status(200).json(_sanitizeReturn(game));
  });
});
```

Here, `_sanitizeReturn(game)` is a simple helper that just copies the game object, except for the player tokens.

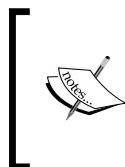
```
// Given a game object, return the game object without tokens
function _sanitizeReturn(game) {
  return {
    boardId: game.boardId,
    board: game.board,
    rows: game.rows,
    columns: game.columns,
    turn: game.turn,
    status: game.status,
    winner: game.winner,
    p1Name: game.p1Name,
    p2Name: game.p2Name
  };
}
```

Joining a game

This game would be no fun if played alone, so we need to allow a second player to join the game.

1. In order to join a game, we need to accept POST to /join with the name of player2 in the POST body:

```
{ name: 'player2' }
```



For this to work, we need to implement a rudimentary match-making system. An easy one is to simply have a queue of games in a joinable state, and popping one off when the /join API is hit. We chose to use Redis as our Queue implementation to keep track of the joinable games.



Once a game is joined, we will send boardId and p2Key back to the player, so that they can play on this board with player 1. This will intrinsically avoid a game to be joined multiple times.

2. All we need to do is add this line to push boardId onto the queue, once the game is created and stored in the DB:

```
client.lpush('games', game.boardId);
```

3. We glanced over database connections when we showed `app.js`. The way to set up a MongoDB connection was covered in *Chapter 2, A Robust Movie API*. The following is how we'll connect to a `redis` database in `src/lib/db.js`:

```
var redis = require('redis');
var url = require('url');

exports.connectRedis = function() {
  var urlRedisToGo = process.env.REDISTOGO_URL;
  var client = {};

  if (urlRedisToGo) {
    console.log('using redistogo');
    rtg = url.parse(urlRedisToGo);
    client = redis.createClient(rtg.port, rtg.hostname);
    client.auth(rtg.auth.split(':')[1]);
  } else {
    console.log('using local redis');
    // This would use the default redis config: { port
    6347, host: 'localhost' }
```

```

        client = redis.createClient();
    }

    return client;
};

```



Note that in production, we'll be connecting to Redis To Go (you can start with a 2MB instance for free). For local development, all you need to do is `redis.createClient()`.



4. Now we can write the tests to join a game, TDD style:

```

var expect = require('chai').expect,
    request = require('supertest'),
    redis = require('redis'),
    client = redis.createClient();

var app = require('../src/lib/app');

describe('Create and join new game | ', function() {
  before(function(done) {
    client.flushall(function(err, res) {
      if (err) return done(err);
      done();
    });
  });
});

```

5. Note that we flush the `redis` queue each time we run this test suite, just to make sure that the stack is empty. In general, it is a good idea to write atomic tests that can run on their own, without reliance on outside state.

```

it('should not be able to join a game without a name',
function(done) {
  request(app).post('/join')
    .expect(400)
    .end(function(err, res) {
      expect(res.body.error).to.equal("Must provide name
field!");
      done();
    });
});

it('should not be able to join a game if none exists',
function(done) {
  request(app).post('/join')
    .send({name: 'koalabear'})
    .expect(400)
    .end(function(err, res) {
      expect(res.body.error).to.equal("Game does not exist!");
      done();
    });
});

```

```
        .expect(418)
        .end(function(err, res) {
            expect(res.body.error).to.equal("No games to
join!");
            done();
        });
    });
}
```

6. Always remember to cover input the edge-cases! In the preceding test, we make sure that we cover the case that we have no games left to join. If not, we might crash the server or return the 500 error (which we should attempt to eradicate because that means it's your fault, not the user!).

Now let's write the following code:

```
it('should create a game and add it to the queue',
function(done) {
    request(app).post('/create')
        .send({name: 'express'})
        .expect(200)
        .end(function(err, res) {
            done();
        });
});

it('should join the game on the queue', function(done) {
    request(app).post('/join')
        .send({name: 'koaloid'})
        .expect(200)
        .end(function(err, res) {
            var b = res.body;
            expect(b.boardId).to.be.a('string');
            expect(b.p1Key).to.be.undefined;
            expect(b.p1Name).to.be.a('string').and.equal('express');
            expect(b.p2Key).to.be.a('string');
            expect(b.p2Name).to.be.a('string').and.equal('koaloid');
            expect(b.turn).to.be.a('number').and.equal(1);
            expect(b.rows).to.be.a('number');
            expect(b.columns).to.be.a('number');
            done();
        });
});
```

7. These tests describe the core logic of creating a game and joining it. Enough tests to describe this endpoint. Let's now write the accompanying code:

```
app.post('/join', Validate.name, function(req, res) {
  client.rpop('games', function(err, boardId) {
    if (err) return res.status(418).json(err);

    if (!boardId) {
      return res.status(418).json({
        error: 'No games to join!'
      });
    }

    Game.findOne({ boardId: boardId }, function (err,
game) {
      if (err) return res.status(400).json(err);

      game.p2Name = req.body.name;
      game.save(function(err, game) {
        if (err) return res.status(500).json(err);
        game.p1Key = undefined;
        res.status(200).json(game);
      });
    });
  });
});
```

We reuse the `Validate.name` middleware here to make sure that we have a name for player 2. If so, we will look for the next joinable game in the queue. When there are no joinable games, we will return an appropriate 418 error.

If we successfully retrieve the next joinable `boardId`, we will fetch the board from the database, and store the name of player 2 in it. We also have to make sure that we do not return player 1's token along with the game object.

Now that both players have fetched their respective tokens, let the games begin!

Playing the game

The game state is stored in the database and can be retrieved with a GET request to `/board/{boardId}`. The essence of making a move is a change to the game state. In familiar CRUD terms, we would be updating the document. Following REST conventions whenever possible, a PUT request to `/board/{boardId}` would be the logical choice to make a move.

To make a valid move, a player needs to include an X-Player-Token in their request header matching that of the corresponding player, as well as a request body identifying which column to make a move in:

```
{ column: 2 }
```

However, not all moves are legal, for example, we need to ensure that players only play moves when it is their turn. There are a few more things that need to be checked for every move:

- Is the move valid? Does the column parameter specify an actual column?
- Does the column still have space?
- Is the X-Player-Token a valid token for the current game?
- Is it your turn?
- Did the move create a victory condition? Did this player win with this move?
- Did the move fill up the board and cause a draw game?

Now we will model all these scenarios.

1. Let's play a full game with the following tests:

```
var expect = require('chai').expect,
    request = require('supertest'),
    redis = require('redis'),
    client = redis.createClient();

var app = require('../src/lib/app'),
    p1Key, p2Key, boardId;

describe('Make moves | ', function() {
  before(function(done) {
    client.flushall(function(err, res) {
      if (err) return done(err);
      done();
    });
  });
  it('create a game', function(done) {
    request(app).post('/create')
      .send({name: 'express'})
      .expect(200)
      .end(function(err, res) {
        p1Key = res.body.p1Key;
```

```
        boardId = res.body.boardId;
        done();
    });
});

it('join a game', function(done) {
    request(app).post('/join')
        .send({name: 'koal'})
        .expect(200)
        .end(function(err, res) {
            p2Key = res.body.p2Key;
            done();
        });
});
});
```

The first test creates the game and the second test joins it. The next six tests are validation tests to make sure that the requests are valid.

2. Make sure that the X-Player-Token is present:

```
it('Cannot move without X-Player-Token', function(done) {
    request(app).put('/board/' + boardId)
        .send({column: 1})
        .expect(400)
        .end(function(err, res) {
            expect(res.body.error).to.equal('Missing X-Player-
Token!');
            done();
        });
});
```

3. Make sure that the X-Player-Token is the correct one:

```
it('Cannot move with wrong X-Player-Token',
function(done) {
    request(app).put('/board/' + boardId)
        .set('X-Player-Token', 'wrong token!')
        .send({column: 1})
        .expect(400)
        .end(function(err, res) {
            expect(res.body.error).to.equal('Wrong X-Player-
Token!');
            done();
        });
});
```

4. Make sure that the board you move on exists:

```
it('Cannot move on unknown board', function(done) {  
    request(app).put('/board/3213')  
        .set('X-Player-Token', p1Key)  
        .send({column: 1})  
        .expect(404)  
        .end(function(err, res) {  
            expect(res.body.error).to.equal('Cannot find  
board!');  
            done();  
        });  
});
```

5. Make sure that a column parameter is sent when making a move:

```
it('Cannot move without a column', function(done) {  
    request(app).put('/board/' + boardId)  
        .set('X-Player-Token', p2Key)  
        .expect(400)  
        .end(function(err, res) {  
            expect(res.body.error).to.equal('Move where?  
Missing column!');  
            done();  
        });  
});
```

6. Make sure that the column is not off the board:

```
it('Cannot move outside of the board', function(done) {  
    request(app).put('/board/' + boardId)  
        .set('X-Player-Token', p1Key)  
        .send({column: 18})  
        .expect(200)  
        .end(function(err, res) {  
            expect(res.body.error).to.equal('Bad move.');//  
            done();  
        });  
});
```

7. Make sure that the wrong player cannot move:

```
it('Player 2 should not be able to move!', function(done)  
{  
    request(app).put('/board/' + boardId)  
        .set('X-Player-Token', p2Key)  
        .send({column: 1})
```

```
.expect(400)
.end(function(err, res) {
  console.log(res.body);
  expect(res.body.error).to.equal('It is not your
turn!');
  done();
});
});
```

8. Now that we have covered all the validation cases, let's test the entire game play:

```
it('Player 1 can move', function(done) {
  request(app).put('/board/' + boardId)
    .set('X-Player-Token', p1Key)
    .send({column: 1})
    .expect(200)
    .end(function(err, res) {
      var b = res.body;
      expect(b.p1Key).to.be.undefined;
      expect(b.p2Key).to.be.undefined;
      expect(b.turn).to.equal(2);
      expect(b.board[b.rows-1][0]).to.equal('x');
      done();
    });
});
```

9. Just a quick check that player 1 cannot move again, before player 2 makes a move:

```
it('Player 1 should not be able to move!', function(done) {
  request(app).put('/board/' + boardId)
    .set('X-Player-Token', p1Key)
    .send({column: 1})
    .expect(400)
    .end(function(err, res) {
      expect(res.body.error).to.equal('It is not your
turn!');
      done();
    });
});
```



```
it('Player 2 can move', function(done) {
  request(app).put('/board/' + boardId)
```

```
.set('X-Player-Token', p2Key)
.send({column: 1})
.expect(200)
.end(function(err, res) {
  var b = res.body;
  expect(b.p1Key).to.be.undefined;
  expect(b.p2Key).to.be.undefined;
  expect(b.turn).to.equal(3);
  expect(b.board[b.rows-2][0]).to.equal('o');
  done();
});
});
```

10. The remainder of this test suite plays out a full game. We won't show it all here, but you may refer to the source code. The last three tests are still interesting though because we cover the final game state and prevent any more moves.

```
it('Player 1 can double-check victory', function(done) {
  request(app).get('/board/' + boardId)
    .set('X-Player-Token', p1Key)
    .expect(200)
    .end(function(err, res) {
      var b = res.body;
      expect(b.winner).to.equal('express');
      expect(b.status).to.equal('Game Over.');
      done();
    });
});

it('Player 2 is a loser, to be sure', function(done) {
  request(app).get('/board/' + boardId)
    .set('X-Player-Token', p2Key)
    .expect(200)
    .end(function(err, res) {
      var b = res.body;
      expect(b.winner).to.equal('express');
      expect(b.status).to.equal('Game Over.');
      done();
    });
});
```

```

it('Player 1 cannot move anymore', function(done) {
  request(app).put('/board/' + boardId)
    .set('X-Player-Token', p1Key)
    .send({column: 3})
    .expect(400)
    .end(function(err, res) {
      expect(res.body.error).to.equal('Game Over. Cannot
move anymore!');
      done();
    });
});
});

```

Now that we have described our expected behavior, let's begin with implementing the move endpoint.

1. First, let's cover the validation pieces, making the first 8 tests pass.

```

app.put('/board/:id', [Validate.move, Validate.token],
function(req, res) {

  Game.findOne({boardId: req.params.id }, function(err,
game) {

```

2. We fetch the board that the move is sent to. If we cannot find the board, we should return a 400 error. This should make the test 'Cannot move on unknown board' pass.

```

    if (!game) {
      return res.status(400).json({
        error: 'Cannot find board!'
      });
    }
  }

```

3. If the game is over, you cannot make any moves.

```

    if(game.status !== 'Game in progress') {
      return res.status(400).json({
        error: 'Game Over. Cannot move anymore!'
      });
    }
  }

```

4. The following code will make sure that the token is either p1Key or p2Key. If not, return the 400 error with the according message:

```

    if(req.headers['x-player-token'] !== game.p1Key &&
req.headers['x-player-token'] !== game.p2Key) {
      return res.status(400).json({

```

```
        error: 'Wrong X-Player-Token!'
    });
}
```

Now that we have verified that the token is indeed a valid one, we still need to check if it is your turn.

The `game.turn()` method will increment with each turn, so we have to take the modulo to check who's turn it is. Incrementing the turn, instead of toggling, will have the benefit of keeping a count on the number of turns that have been played, which will also be handy later, when we want to check whether the board is full, and end in a tie.

Now we know which key to compare the token with.

1. If your token does not match, then it is not your turn:

```
var currentPlayer = (game.turn % 2) === 0 ? 2 : 1;
var currentPlayerKey = (currentPlayer === 1) ? game.p1Key :
game.p2Key;
if (currentPlayerKey !== req.headers['x-player-token']){
    return res.status(400).json({
        error: 'It is not your turn!'
    });
}
```

2. We added two more validation middleware for this route, move and token, which we can add to the validators library in `src/lib/validators.js`:

```
move: function(req, res, next) {
    if (!req.body.column) {
        return _return400Error(res, 'Move where? Missing
column!');
    }
    next();
},
token: function(req, res, next) {
    if (!req.headers['x-player-token']) {
        return _return400Error(res, 'Missing X-Player-
Token!');
    }
    next();
}
```

3. Since we are sending the 400 error four times in the preceding code, let's dry it up and reuse the same helper we had in `validators.js`, by extracting that helper into `src/lib/utils.js`:

```
var crypto = require('crypto');

module.exports = {
    randomValueHex: function(len) {
        return crypto.randomBytes(Math.ceil(len/2))
            .toString('hex')
            .slice(0,len);
    },
    // Helper to return 400 error with a custom message
    return400Error: function(res, message) {
        return res.status(400).json({
            error: message
        });
    }
}
```

4. Don't forget to update `src/lib/validators.js` to use this `utils`, by replacing the line with the following:

```
var _return400Error = require('../utils').return400Error;
```

5. Now we can refactor the move route to make a move as follows:

```
app.put('/board/:id', [Validate.move, Validate.token],
function(req, res) {

    Game.findOne({boardId: req.params.id }, function(err,
game) {
        if (!game) {
            return _return400Error(res, 'Cannot find board!');
        }

        if(game.status !== 'Game in progress') {
            return _return400Error(res, 'Game Over. Cannot move
anymore!');
        }
    })
})
```

```
        if(req.headers['x-player-token'] !== game.p1Key &&
req.headers['x-player-token'] !== game.p2Key) {
    return _return400Error(res, 'Wrong X-Player-
Token!');
}

var currentPlayer = (game.turn % 2) === 0 ? 2 : 1;
var currentPlayerKey = game['p' + currentPlayer +
'Key'];
if(currentPlayerKey !== req.headers['x-player-
token']){
    return _return400Error(res, 'It is not your
turn!');
```

Much cleaner, ain't it!

For the remainder of the controller logic, we will use the `connect4.js` library (see Appendix), which implements the `makeMove()` and `checkForVictory()` methods.

The `makeMove()` method will return a new board that results from the move, or return false if the move is invalid. Invalid here means that the column is already full, or the column is out of bounds. No turn validation is done here.

```
// Make a move, which returns a new board; returns false if
// the move is invalid
var newBoard = connect4.makeMove(currentPlayer,
req.body.column, game.board);
if(newBoard) {
    game.board = newBoard;
    game.markModified('board');
} else {
    return _return400Error(res, 'Bad move.');
}
```

One really important thing to point out is the line `game.markModified('board')`. Since we are using a 2D array for board, Mongoose is unable to auto-detect any changes. It can only do so with the basic field types. So if we do not explicitly mark the board as modified, it will not persist any changes when we call `game.save()`.

```
// Check if you just won
var win = connect4.checkForVictory(currentPlayer,
req.body.column, newBoard);
if(win) {
    game.winner = game['p'+ currentPlayer + 'Name'];
```

```
        game.status = 'Game Over.';
    } else if(game.turn >= game.columns*game.rows) {
        game.winner = 'Game ended in a tie!';
        game.status = 'Game Over.';
    }
}
```

The `checkForVictory()` method is a predicate that will check for victory based on the last move by the last player. We don't need to be checking the entire board each time. If the last move was a winning move, this method will return true; otherwise, it will return false.

```
// Increment turns
game.turn++;

game.save(function(err, game) {
    if (err) return res.status(500).json(err);
    return res.status(200).json(_sanitizeReturn(game));
});

});

});
```

It is a good idea to keep the controller logic as thin as possible and defer as much of the business logic as possible to the libraries or models. This decoupling and separation of concerns improves maintainability and testability, as well as modularity and reusability. Given the current architecture, it would be very easy to reuse the core components of our application in another Express project.

Testing for a tie

The only thing we haven't covered yet in our test suite is a tie game. We could create another test suite that would fill the entire board manually using 42 individual moves, but that would be too tedious. So let's fill the board programmatically.

That may sound easy, but it can be a bit tricky with JavaScript's asynchronous control flow. What would happen if we were to simply wrap the move request in a `for` loop?

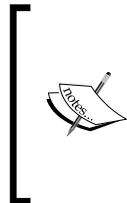
In short, it would be a mess. All requests would go out at the same time, and there will be no order. And how would you know that all moves are finished? You would need to maintain a global state counter that increments with each callback.

This is where the `async` library becomes indispensable from Github.

Async is a utility module, which provides straightforward, powerful functions to work with asynchronous JavaScript.

There is so much that you can do with async that would make your life easier; definitely a library that you should acquaint yourself with and add to your toolbox.

In our situation, we will use `async.series`, which allows us to send a flight of requests serially. Each request will wait until the previous request has returned.



Run the functions in the tasks array in series, each one running once the previous function has completed. If any functions in the series pass an error to its callback, no more functions can be run, and callback is immediately called with the value of the error; otherwise, callback receives an array of results when tasks are completed.



So to prepare our moves to be passed to `async.series`, we will use the following helper to create a thunk:

```
function makeMoveThunk(player, column) {
  return function(done) {
    var token = player === 1 ? p1Key : p2Key;
    request(app).put('/board/' + boardId)
      .set('X-Player-Token', token)
      .send({column: column})
      .end(done);
  };
}
```

A thunk is simply a subroutine; in this case calling the API to make a move, that is wrapped in a function, to be executed later. In this case, we create a thunk that accepts a callback parameter (as required by `async`), which notifies `async` that we're done.

Now we can fill the board programmatically and check for the tie state:

```
it('Fill the board! Tie the game!', function(done) {
  var moves = [],
    turn = 1,
    nextMove = 1;

  for(var r = 0; r < rows; r++) {
    for(var c = 1; c <= columns; c++) {
```

```
    moves.push(makeMoveThunk(turn, nextMove));
    turn = turn === 1 ? 2 : 1;
    nextMove = ((nextMove + 2) % columns) + 1;
  }
}

async.series(moves, function(err, res) {
  var lastResponse = res[rows*columns-1].body;
  console.log(lastResponse);
  expect(lastResponse.winner).to.equal('Game ended in a
tie!');
  expect(lastResponse.status).to.equal('Game Over.');
  done();
});
});
```

Summary

Congratulations! By now, all your tests should be passing and your game should be complete. You have mastered developing a robust and well-tested API and deal with validation using reusable middleware. Along the way you've also learned to use Redis for a simple queue.

Now you can deploy your API and you'll have your Connect4-as-a-Service available for the world to build their own connect 4 game upon, using their own favorite platform. Whether it is an HTML5 interface, a mobile app, or a command-line interface, it will all be powered by your backend!

In the next chapter, we'll be taking game development to another level – it will be a realtime massively multiplayer online game!

4

MMO Word Game

Word Chain Game is a real-time, massive multiplayer online game. Each player will be able to see the other online players when playing the game, along with a score leaderboard for score. In this chapter we will introduce the Promise pattern and explain how Promises simplify asynchronous operations. You will learn how to build a real-time application with Express and SocketIO, perform authentication over socket handshaking, and deal with race conditions using the atomic update of MongoDB. You will also learn how to build the game client to connect to the game server over socket, and how to debug WebSocket on the client side using Chrome Developer Tools.

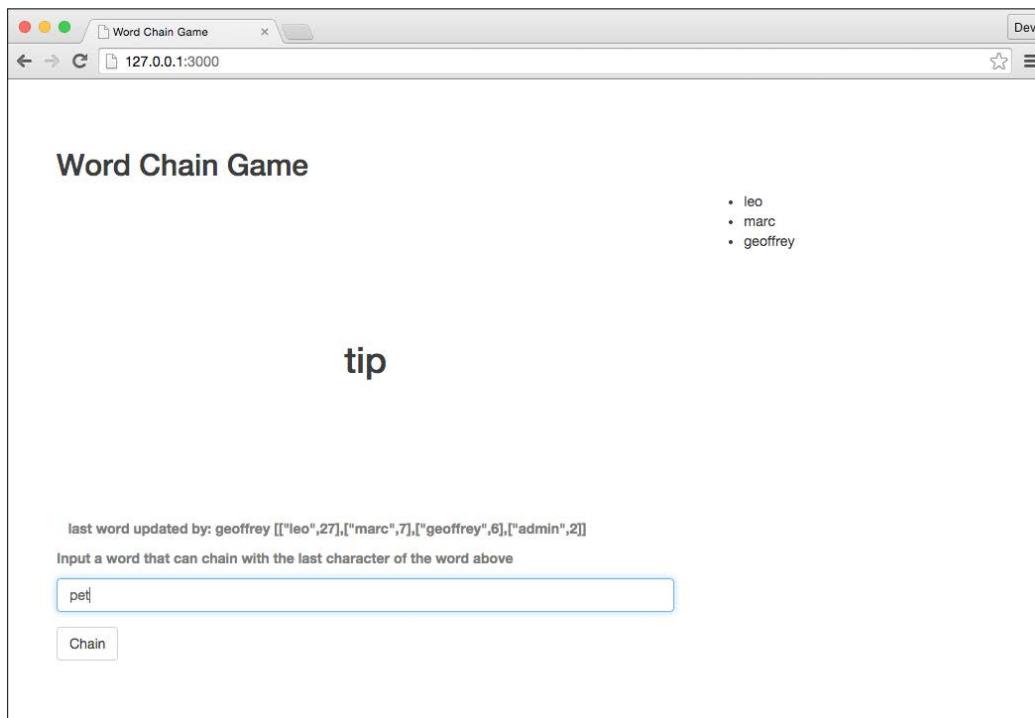
Once you have mastered this, you can build similar games such as online quiz competitions.

Gameplay

The game starts with a randomly selected English word and each player tries to submit a word where the first character of their submission matches the last character of the current word; we call this chaining with the current word. For example, if the game starts with the word Today, then players can send words such as Yes or Yellow.

The first person to submit a valid word will have their word become the starting word for the next round and gets the points for that round. Once the new word is accepted, the server will broadcast the change to all online players. The players will see the new word and submit another word to chain with it.

For example, if player 1 sends yes to chain with Today, the server will accept the word and broadcast the current word Yes to all other players. If a player submits a word that is invalid based on the dictionary we have or was submitted by another player earlier, the game server will ignore that request. If multiple players submit valid words simultaneously, the server will only accept the first submitted word.



Real-time application overview

In this game, we will introduce the Promise pattern and explain how Promise will simplify async operations.

Despite this being a real-time game, we will not rush into implementing a real-time feature at the beginning. Instead, we will first build a game model, which contains all the game logic.

In the game logic, we first introduce how to keep track of active users and then explain how we verify users' input. After verifying the input, during the updating game state phase, we deal with race conditions by utilizing the atomic operation of MongoDB. We also look into how to cover the race conditions with test cases.

After the game logic is done, we will introduce how to broadcast game state changes to all players using Socket.IO.

In the end, we will create an Express app, a Socket.IO server, and a game client that can talk to our server using the `socket.io-client` libraries.

Keeping track of active users

Since the game is a multiplayer game, players can see the number of players and their usernames. To keep track of active users, we need to track when a player joins the game and when a player leaves the game.

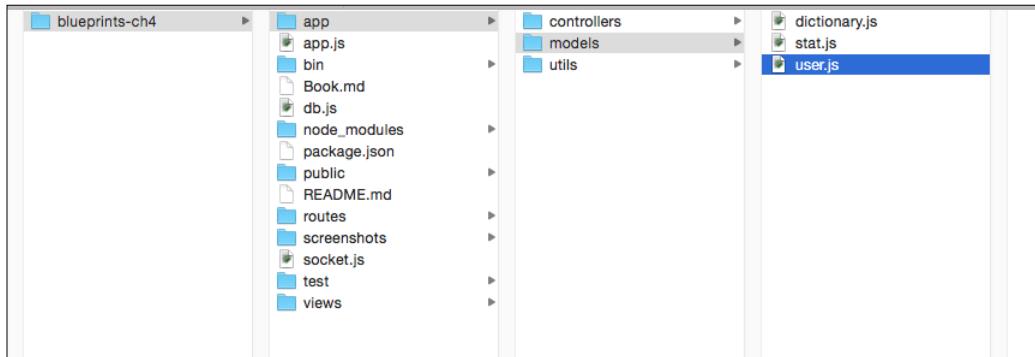
Schema design

Each player can simply be represented by a document with a single field for the name:

```
{ name: 'leo' }
```

User schema

We will use Mongoose for our data modeling. Let's start with designing our user schema. The schemas are placed in the `models` folder in the app. The following screenshot shows the folder structure. The schema will have one required field `name`, this is done by adding `required: true` to the `name` object in the schema.



To make querying a user by name fast, we can add an index to name. By default, only the `_id` field that MongoDB generates will be indexed. This means, to perform a search by name, the database will need to iterate over all the documents in the collection to find a matching name. When you add an index to `name`, you can query by name as quickly as when you query by `_id`. Now, when a user leaves, we can find the user by name directly and remove that user.

Also, we add the `unique : true` property to the index to avoid having multiple users with the same name, as shown in the following code:

```
var mongoose = require('mongoose');
var schema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    index: {
      unique: true
    }
  }
});

var User = mongoose.model('user', schema);
module.exports = User;
```

User join

When a user joins a game, we create a user with the key `name` and save this user to MongoDB, as follows:

```
schema.statics.join = function(name, callback) {
  var user = new User({
    name: name
  });

  user.save(function(err, doc) {
    if (!callback) { return; }
    if (err) { return callback(err); }

    callback(null, doc);
  });
};
```

The `save()` method in the preceding code uses callback patterns, which is also known as callback hell. If an error occurs, we make a call to the callback function passing the error as a parameter; otherwise, the operation succeeds and it returns the updated document.

The preceding callback pattern involves a lot of logic and condition checks. The nested callback pattern of JavaScript can quickly turn into a spaghetti nightmare. A good alternative is to use Promises to simplify things.

Mongoose's `model.create()` method (http://mongoosejs.com/docs/api.html#model_Model.create) can create and save a new document into the database if valid. Functions and documents such as objects and arrays are valid parameters for the `model.create()` method. The `create` method returns a Promise.

With this Promise, the caller of the `join` method can define the success and fail callbacks, simplifying the code:

```
schema.statics.join = function(name) {
  return User.create({
    name: name
  });
};
```

Promises

A Promise is the eventual result of an asynchronous operation, just like giving someone a promise. Promises help handle errors, which results in writing cleaner code without callbacks. Instead of passing in an additional function that takes an error and result as parameters to every function, you can simply call your function with its parameter and get a Promise:

```
getUserinfo('leo', function(err, user){
  if (err) {
    // handle error
    onFail(err);
    return;
  }

  onSuccess(user);
});
```

versus

```
var promiseUserInfo = getUserinfo('leo') ;

promiseUserInfo.then(function(user) {
    onSuccess(user);
}) ;

promiseUserInfo.catch(function(error) {
    // code to handle error
    onFail(user);
}) ;
```

The benefit of using Promises isn't obvious if there is only one async operation. If there are many async operations with one depending on another, the callback pattern will quickly turn into a deeply nested structure, while Promises can keep your code shallow and easier to read.

Promises can centralize your error handling and when an exception happens, you will get stack traces that reference actual function names instead of anonymous ones.

In our word game, you could use Promises to turn this:

```
var onJoinSuccess = function(user) {
    console.log('user', user.name, 'joined game!');
    return user;
};

var onJoinFail = function(err) {
    console.error('user fails to join, err', err);
};

User.join('leo', function(err, user) {
    if (err) {
        return onJoinFail(err);
    }

    onJoinSuccess(user);
});
```

into this:

```
User.join('leo')
  .then(function(user) {
    onJoinSuccess(user);
  })
  .catch(function(err) {
    onJoinFail(err);
  });
}
```

or even simpler:

```
User.join('leo')
  .then(onJoinSuccess)
  .catch(onJoinFail);
```

To understand the execution flow of the preceding, let's create a complete example that calls the user model's `join()` method, and then add some log statements to see the output:

```
var User = require('../app/models/user.js');
var db = require('../db');

var onJoinSuccess = function(user) {
  console.log('user', user.name, 'joined game!');
  return user;
};

var onJoinFail = function(err) {
  console.error('user fails to join, err', err);
};

console.log('Before leo send req to join game');

User.join('leo')
  .then(onJoinSuccess)
  .catch(onJoinFail);

console.log('After leo send req to join game');
```

If a user joins the game successfully, the Promise returned by the `User.join()` method will be resolved. A newly created user document object will be passed to the `onJoinSuccess` callback and the output result will be printed as follows:

```
# leo at LeoAZ in ~/az/blueprints-ch4/test on git:master x [22:36:31]
$ node user.js
connecting db...
Before leo send req to join game
After leo send req to join game
db connected
user leo joined game!
```

If we run this script again, we will see that the user fails to join the game and the error is printed. It fails because the user model already has an index on name property because a user with the name `leo` was created when we ran the script the first time. When we run it again, we can't create another user with the same name `leo`, so the Promise fails and the error is passed into `onJoinFail`.

```
# leo at LeoAZ in ~/az/blueprints-ch4/test on git:master x [22:37:55]
$ node user.js
connecting db...
Before leo send req to join game
After leo send req to join game
db connected
user fails to join, err { [MongoError: E11000 duplicate key error index: blueprints-ch4.users.$name_1  dup key: { : "leo" }]
  name: 'MongoError',
  err: 'E11000 duplicate key error index: blueprints-ch4.users.$name_1  dup key: { : "leo" }',
  code: 11000,
  n: 0,
  connectionId: 80,
  ok: 1 }
```

A Promise has three states: pending, fulfilled, or rejected; a Promise's initial state is pending, then it promises that it will either succeed (fulfilled) or fail (rejected). Once it is fulfilled or rejected, it cannot change again. A major benefit of this is that you can chain multiple Promises together and define one error handler to handle all the errors.

As the `join()` method returns a Promise, we can define the success and fail callbacks as follows.

The then and catch method

The `then` and `catch` methods are used to define success and fail callbacks; you might wonder when they are actually being called. When the `User.create()` method is called, it will return a Promise object and at the same time send an async query to MongoDB. The success callback, `onJoinSuccess`, is then passed into the `then` method and will be called when the async query is successfully completed, resolving the Promise.

Once the Promise is resolved, it can't be resolved again, so `onJoinSuccess` won't be called again, it will only be called once at the most.

Chain multiple Promises

You can chain Promise operations by calling them on the Promise that is returned by the previous `then()` function. We use the `.then()` method when we want to do something with the result from the Promise (once *x* resolves, then do *y*) as follows:

```
var User = require('../app/models/user.js');
var db = require('../db');

var onJoinSuccess = function(user) {
  console.log('user', user.name, 'joined game!');
  return user;
};

var onJoinFail = function(err) {
  console.error('user fails to join, err', err);
};

console.log('Before leo send req to join game');
User.join('leo')
  .then(onJoinSuccess)
  .then(function(user) {
    return User.findAllUsers();
  })
  .then(function(allUsers) {
    return JSON.stringify(allUsers);
  })
```

```
.then(function(val) {
  console.log('all users json string:', val);
})
.catch(onJoinFail);

console.log('After leo send req to join game');
```

We can centralize the error handling at the end. It's much easier to deal with errors with Promise chains. If we run the code, we will get the following result:

```
# leo at LeoAZ in ~/az/blueprints-ch4/test on git:master ✘ [22:54:01]
$ node allusers.js
connecting db...
Before leo send req to join game
After leo send req to join game
db connected
user leo joined game!
will call User.findAllUsers
will call JSON.stringify
all users json string: ["leo"]
```

Now that we've gone through all the logic and error handling of creating a new user, let's look into how we will ensure that multiple users with the same name can't join.

Prevent duplicates

Earlier, when we defined our user schema, we added `index` with a `unique` set to `true` on the `name` field:

```
var schema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    index: {
      unique: true
    }
  }
});
```

MongoDB will issue a query to see whether there is another record with the same value for the unique property and, if that query comes back empty, it allows the save or update to proceed. If another user joins with the same name, Mongo throws the error: Duplicate Key Error. This prevents the user from being saved and the player must choose another name to join with.

To make sure our code works as we want it to, we need to create tests; we will create a test case with Mocha. The test case will pass a username to the `User.join` method and expect that the username of the newly created user is valid. The `User.join` method returns a Promise. If it succeeds, the object returned from the Promise will be sent to the `then` method; otherwise it fails and the Promise will `.reject` with an error that will be caught by the `catch` method.

In the case of the success callback, we have the newly created user, and we can check whether it's correct by expecting `user.name` to return `leo`, since `leo` was entered as the username (illustrated in the following code).

In the case of fail callback, we can pass the error object to Mocha, `done(error)`, to fail the test case. Since we created a user named `leo` for the first time, we expect this test to pass. Since Mocha tests are synchronous and Promises are `async`, we need to wait for the function to be done. When the code is successful, it will call the `done()` function and report success to Mocha; if it fails, the `catch` method will catch the error and return the error to the `done` method, which will tell Mocha to fail the test case.

```
var User = require('../app/models/user');

describe('when leo joins', function() {
  it('should return leo', function(done) {
    User.join('leo')
      .then(function(user) {
        expect(user.name).to.equal('leo');
        done();
      })
      .catch(function(error) {
        done(error);
      });
  });
});
```

Version 1.18.0 or above of Mocha allows you to return a Promise in a test case. Mocha will fail the test case if the Promise fails without needing to explicitly catch the error as given in the following:

```
describe('when leo joins', function() {
  it('should return leo', function() {
    return User.join('leo')
      .then(function(user) {
        expect(user.name).to.equal('leo');
      });
  });
});
```

Now that we tested that submitting the first user with a unique name works, we want to test what happens when another user with the same name joins:

```
describe('when another leo joins', function() {
  it('should be rejected', function() {
    return User.join('leo')
      .then(function() {
        throw new Error('should return Error');
      })
      .catch(function(err) {
        expect(err.code).to.equal(11000);
        return true;
      });
  });
});
```

When we submit `leo` again as a username, the Promise of `Game.join` comes back rejected and goes to the `.catch` method. The `return true` turns a failed Promise into a success, which tells us that it succeeded in rejecting the second `leo` and that we successfully caught the error; we basically swallow the error to tell Mocha that this is the correct behavior we expect.

User leaves the game

When a user leaves the game, we need to remove their entry in the database; this would also free up their user name so that a new user can take it. Mongoose has a delete method called `findOneAndRemove`, which can find that player by name, and then remove it as shown in the following code.

For our Promises, we use Bluebird (<https://github.com/petkaantonov/bluebird>) (spec: PromiseA) because of its better performance, utility, and popularity (support).

We call the `Promise.resolve` method, which creates a Promise that is resolved with the value inside: `Promise.resolve(value)`. Therefore, we can take a method that does not normally return a Promise and wrap it with the Bluebird `Promise.resolve` method to get a Promise back, which we can then chain with `then` if it succeeds or `catch` if it fails. Receiving Promises from our methods will ensure that we deal with successes and errors efficiently and also lets the callee deal with the error when it runs `(.exec())`.

```
schema.statics.leave = function(name) {
  return Promise.resolve(this.findOneAndRemove({
    name: name
  })
  .exec());
};
```

Show all active users

So far we demonstrated how to add and remove users, we will now dive into how we will display the game data to a user that's joined. To show the total active users, we could simply return all users, as offline users have already been removed. In order to return an array of just the user names, rather than an array of the entire user object, we could use the `Promise.map()` method to convert each user object in the array into a user name.

Since `User.find` returns an array of users, we use the `Promise.map()` method to return the values from the `name` key. This effectively turns the array of user objects into an array of user names. Again, notice that we use the `promise.resolve()` method to obtain a Promise from our input. This will allow us to display a list of the currently logged in users by their user name.

```
schema.statics.findAllUsers = function() {
  return Promise.resolve(User.find({}).exec()
  .map(function(user) {
    return user.name;
  }));
};
```

The words – Subdocuments

We have gone through the game logic involving creating, displaying, and deleting users but what about the meat of the game itself—the words?

In `app/models/stat.js`, we see how we model our word data. The `word` field shows the current word, and the `used` field saves the game's history.

We embedded the `used` list as subdocuments into the Stat document, so that we can update stats atomically. We will explain this later.

```
{  
  'word': 'what',  
  'used': [  
    { 'user': 'admin', 'word': 'what' },  
    { 'user': 'player1', 'word': 'tomorrow' },  
    { 'user': 'player2', 'word': 'when' },  
    { 'user': 'player2', 'word': 'nice' },  
    { 'user': 'player1', 'word': 'egg' },  
  ]  
}
```

The preceding code gives you an overview of what we will store in the database.

We first create a model for our word inputs, new word (`word`) and used words (`used`), in a similar method to our user's model, by defining a type (string for new and array for old). The old words are stored in an array so that they can be accessed when we check whether or not a new word has been used before.

```
var schema = new mongoose.Schema({  
  word: {  
    type: String,  
    required: true  
  },  
  used: {  
    type: Array  
  },  
});
```

Further logic about validating word inputs and scoring will be described after we create a new game.

When we create a new game, we want to make sure that no old game data exists and that all values in our database are reset, so we will first remove the existing game, and then create a new one, as shown in the following code:

```
schema.statics.newGame = function() {
  return Promise.resolve(Stat.remove().exec())
    .then(function() {
      return Stat.create({
        word: 'what',
        used: [
          {
            word: 'what',
            user: 'admin'
          }
        ]
      });
    })
};
```

In the preceding example, we use `Stat.remove()` to remove all old game data and when the Promise is fulfilled, we create a new game using `Stat.create()` by passing a new word, 'what', to start off the new round and also submit both the word and the user who submitted the word into the used array. We want to submit the user in addition to the word so that other users can see who submitted the current word and also use that information to calculate scores.

Validate input

We can't just accept any word a user might input; users might enter an invalid word (as determined by our internal dictionary), a word that can't chain with the current word or a word that has already been used before in this game.

Our internal dictionary model is found in `models/dictionary.js` and consists of the `dictionary.json`. Requests with an invalid word should be ignored and should not change the game's state (see `app/controllers/game.js`); if the word is not in the dictionary, the Promise will be rejected and will not go to `Stat.chain()`.

In the following code example, we illustrate how to check whether the submitted word chains with the current word:

```
schema.statics.chain = function(word, user) {
  var first = word.substr(0, 1).toLowerCase();

  return Promise.resolve(Stat.findOne({}).exec())
    .then(function(stat) {
      var currentWord = stat.word;
```

```
if (currentWord.substr(-1) !== first) {
    throw Helper.build400Error('not match');
}

return currentWord;
})
.then(function(currentWord) {
    return Promise.resolve(Stat.findOneAndUpdate({
        word: currentWord,
    }, {
        $push: {
            used: { 'word': word, 'user': user }
        }
    }, {
        upsert: false
    }).exec();
});
});
```

The first step is to query the `Stat` collection to get the current game state. From the game state, we know the current word that needs to be matched by calling `stat.word` and assigning it to the variable `currentWord`.

We then compare the current word with the user's input. First we determine the first letter of the submitted word using calling `substr(0, 1)` and then we compare it to the last letter of the current word (`currentWord`) by calling `substr(-1)`. If the first character of the user's input doesn't match with the last character of the current word of the game, we throw a 400 error. The Promise will catch this error, and call the `catch` callback to handle the error.

Here, in the model's method, we let the model object return a `Promise` object. Later on, we will introduce how to catch this error in the controller's method and return a 400 response to the user.

```
throw Helper.build400Error('not match');
```

The `Helper.build400Error()` function is a utility function that returns a 400 Error with an error message:

```
exports.build400Error = function(message) {
    var error = new Error(message);
    error.status = 400;
    return error;
};
```

If the word can chain with the current word, it's a valid request. We will get back a successful Promise, which allows us to chain with the next then and save the word along with the player's username to the database.

To save the data into the database, we use Mongoose's `findOneAndUpdate` method, which takes three arguments. The first is a query object to find the document to be updated. We find the stat document where the word is `currentWord` we get from `Stat.findOneQuery`. The second argument is the update object. This defines what to update.

We use Mongo's modifier `$push` to push a word chain history into the `used` field, which is an array. The last argument is options.

We use the `{ upsert: false }` option, which means if we can't find the document with the query defined in the first argument, we won't update or insert a new document. This makes sure no other operation occurs in between the time it takes to find the document and update the document, that is, we don't insert a new word if the current word cannot be found. Therefore, the game status doesn't change because the current word is assigned to `word` and is still the same.

If we successfully find the word, we add a new used word object to the `used` word array consisting of the new valid word and the username that submitted it.

```
Stat.findOneAndUpdate({
  word: currentWord,
}, {
  $push: {
    used: { 'word': word, 'user': user }
  }
}, {
  upsert: false
}).exec();
```

Dealing with race conditions

You might have questions about the preceding code. Finding a document and updating a document seem like two separate operations; what if two users send the same request? It may cause a race condition.

For example, if the current word is `Today`, Player 1 submits `yes`, and Player 2 submits `yellow`; both players chain a valid word. While both these words are valid, we can't accept both of them for two reasons; only one player can win each round, and if we had two or more winning words, the words could end with different letters, which would affect the next word chain.

If yes arrives at the server first and gets accepted, then the next word should start with an s, and yellow from Player 2 should become invalid and be rejected. This is called a race condition.

How do we solve this? We need to combine the two database operations, finding a document and updating a document, into one operation. We could use Mongoose model's `findOneAndUpdate` method. This method will actually call the `findAndModify` method of MongoDB, which is an isolated update and return operation. Since it becomes one database operation, MongoDB will update the document atomically.

```
schema.statics.chain = function(word, user) {
  var first = word.substr(0, 1);

  return Promise.resolve(Stat.findOne({}).exec())
    .then(function(stat) {
      var currentWord = stat.word;

      if (currentWord.substr(-1).toLowerCase() !== first.toLowerCase())
      {
        throw Helper.build400Error('not match');
      }

      return currentWord;
    })
    .then(function(currentWord) {
      return Promise.resolve(Stat.findOneAndUpdate({
        word: currentWord,
        'used.word': { $ne: word }
      }, {
        word: word,
        $push: {
          used: { 'word': word, 'user': user }
        }
      }, {
        upsert: false,
      })
      .exec());
    })
    .then(function(result) {
      if (!result) {
        throw Helper.build404Error('not found');
      }

      return result;
    });
};
```

When a user submits a word, we first query the current game state, when the Promise is resolved and successful, and then check that the first letter of our submitted word (first) and last letter of the current word (currentWord) are the same.

If they are the same, we call `findOneAndUpdate()` to search for the submitted word and make sure that it is not present in the array of previously used words. `used.word: { $ne: word }` then returns a Promise object.

If the Promise comes back fulfilled, then we push the submitted word and user to the used words array.

If the Promise is rejected and/or the conditions are not satisfied, then no data will be pushed into the array (`upsert: false`).

The last `then` statement returns the new result; if none was returned, then the `not found` error will be thrown.

Test case to test race conditions

Now that we implemented the logic, we want to test it out. The test case is shown as follows:

```
describe('when player1 and player2 send different valid word together', function() {
  it('should accept player1\'s word, and reject player2\'s word', function(done) {
    Game.chain('geoffrey', 'hello')
      .then(function(state) {
        expect(state.used.length).to.equal(4);
        expect(state.used[3].word).to.equal('hello');
        expect(state.used[3].user).to.equal('geoffrey');

        expect(state.word).to.equal('hello');
      });

    Game.chain('marc', 'helium')
      .then(function(state) {
        done(new Error('should return Error'));
      })
      .catch(function(err) {
        expect(err.status).to.equal(400);
        done();
      });
  });
});
```

As the word by player 1 goes in first, the `hello` word by player 1 should increase the length of the used array to 4, the current word position in the array should be equal to `hello`, and the successful user who submitted it should be updated to be `geoffrey`.

When marc submits a word beginning with `h`, it should return an error because the last letter of the current word, `hello`, is `o` and `helium` does not begin with `o`.

Socket.IO

We can send information to our servers when we submit user info or words but how do we get the server to update us without requesting updates manually? We use Socket.IO to enable real-time two-way event-based communication. Documentation for Socket.IO is available at socket.io/docs. We install it by executing the following code:

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

Socket handshaking, user join

First, we require `socket.io` and our game in `socket.js`:

```
var socketIO = require('socket.io');
var Game = require('./app/controllers/game');
```

Authorization takes place during handshaking, which is when the socket connection is established. Without handshaking, we would not know which socket connection belongs to which Express session. As given in the following code:

```
module.exports = function(server) {

  var io = socketIO(server, {
    transports: ['websocket']
  });

  io.use(function(socket, next) {
    var handshakeData = socket.request;
    console.log('socket handshaking',
    handshakeData._query.username);
    socket.user = handshakeData._query.username;

    Game.join(socket.user)
    .then(function(users) {
      console.log('game joined successfully', socket.user);
      socket.broadcast.emit('join', users);
    })
  });
}
```

```
    next();
  })
  .catch(function(err) {
    console.log('game joined fail', socket.user);
    next(err);
  });
});

};


```

The `io.use()` method lets you give the Socket.IO server functions to run after a socket is created.

The request sent from the client (consisting of a URL and name) will be stored in `handshakeData`. The console will output the username and make sure that the sockets are handshaking.

Next, it will assign the username to `socket.user` so that it can be passed in to the `join()` function. The socket will call the `Game.join()` function and if the user is able to join, a console message will be displayed with the message `game joined successfully` and the name of the user.

The `Socket.broadcast.emit` method sends the message to all other clients except the newly created connection telling them that a new user has joined.

If the user was not successfully created (that is, there were two users with the same name) the error will be sent to the `catch` method and the console will log that the user was not able to join the game. Then, `next (err)` will send the error message back to the connecting client, so that on the client side we can show a pop-up message telling the user that the name is being used.

Adding and pushing updates to clients

With Socket.IO, you can send and receive any events you want as well as any data you want in the JSON format.

There are three additional socket events (after connecting) that we're going to need for our game: disconnect, chain (chain new word to last), and game status.

In `socket.js`, add these three socket events:

```
io.sockets.on('connection', function(socket) {
  console.log('client connected via socket', socket.user);
```

```
socket.on('disconnect', function() {
  console.log('socket user', socket.user, 'leave');
  Game.leave(socket.user)
  .then(function(users) {
    socket.broadcast.emit('leave', users);
  });
});

socket.on('chain', function(word, responseFunc) {
  console.log('socket chain', word);
  Game.chain(socket.user, word)
  .then(function(stat) {
    console.log('successful to chain', stat);
    if (responseFunc) {
      responseFunc({
        status: 200,
        resp: stat
      });
    }
    console.log('broadcasting from', socket.user, stat);
    socket.broadcast.emit('stat', stat);
  })
  .catch(function(err) {
    console.log('fail to chain', err);
    if (responseFunc) {
      responseFunc(err);
    }
  });
});

socket.on('game', function(query, responseFunc) {
  console.log('socket stat', socket.user);
  Game.state()
  .then(function(game) {
    console.log('socket stat end', game);
    if (responseFunc) {
      responseFunc(game);
    }
  });
});

socket.on('error', function(err) {
  console.error('error', err);
});
});
```

The first socket event `connection` we subscribe to will be triggered when a user establishes a socket connection with the server. Once a client is connected, we log that event and display their name on to the console so that we know who is connected.

The second event `disconnect` will be triggered when users are disconnected from the server. It happens when they leave the game or the network connection is broken. Once this event is triggered, we broadcast to all other sockets that the user has left (via `socket.broadcast.emit`) so that the other users can see that the disconnected user is no longer in the list of active players.

The last two socket events, `chain` and `game`, are game actions.

The `chain` takes in the user's submitted word and calls the `Game.chain()` function; if it succeeds, then it logs that the chain was successful and broadcasts the status to all other users.

The `game` responds with the latest game status.

Launch Socket.IO applications

To launch our game, let's create a launch script called `www`, and place it under the `bin` folder. This is our code for `./bin/www` as given in the following:

```
#!/usr/bin/env node
var app = require('../app');
var socket = require('../socket');

app.set('port', process.env.PORT || 3000);

var server = app.listen(app.get('port'), function() {
  console.log('Express server listening on port ' + server.address().port);
});
socket(server);
```

The first line tells shell which interpreter should be used to execute this script. Here, we tell shell that the interpreter is `node`. Then, we can launch the server locally with the following command:

```
./bin/www
```

Next, in `bin/www`, we will set up an Express application listening on a port, which is defined in the environment variable or `3000` if nothing is there.

We then bind socket to our HTTP server, which is created by our Express application. Since the Socket.IO server needs to be attached to an HTTP server, we pass the server object to the socket function, where we initialize the socket server.

So now we have the launch script in place. If we launch the server locally, we will see the following message printed to the console:

```
$ ./bin/www
connecting db...
Express server listening on port 3000
db connected
```

Test Socket.IO applications with the Socket.IO client

We will write the JavaScript for the client-side frontend application, which we will test our game with.

You can find the JavaScript file under `public/javascripts/app.js` and the view in `app/views/index.jade`. We will not be covering frontend components such as `jade` and `stylus/css` in this book.

We begin by setting up our game with all our variables, which are classes in the `index.jade` file that we will refer to. We also initialize our game with the `init()` function, which will be described in the next code block:

```
$(function() {
    var game = new Game({
        $viewLogin: $('.view-login'),
        $viewGame: $('.view-game'),
        $username: $('.username'),
        $wordInput: $('.word-input'),
        $word: $('.word'),
        $bywho: $('.bywho'),
        $users: $('.users'),
    });
});

var Game = function(views) {
    this.views = views;

    this.init();
};
```

The `Game.prototype` adds functions to our `Game` method in `app/controllers/game.js`. We will break this up into several smaller code blocks to show the client-side logic that we're working with.

The `init()` function begins by bringing the `username` input box into focus, and then when the submit button is pressed, obtain the value of the user input and assign it to the variable `username`.

We then send the user name to the `join()` function listed as follows, in the next code block.

We also set up a function that will take the input from the submit button for `chain` (which is where you input the word you would like to chain with the current word), store it in the `chain` variable, and then send it to the `chain` function (discussed later) and clear out the text input box.

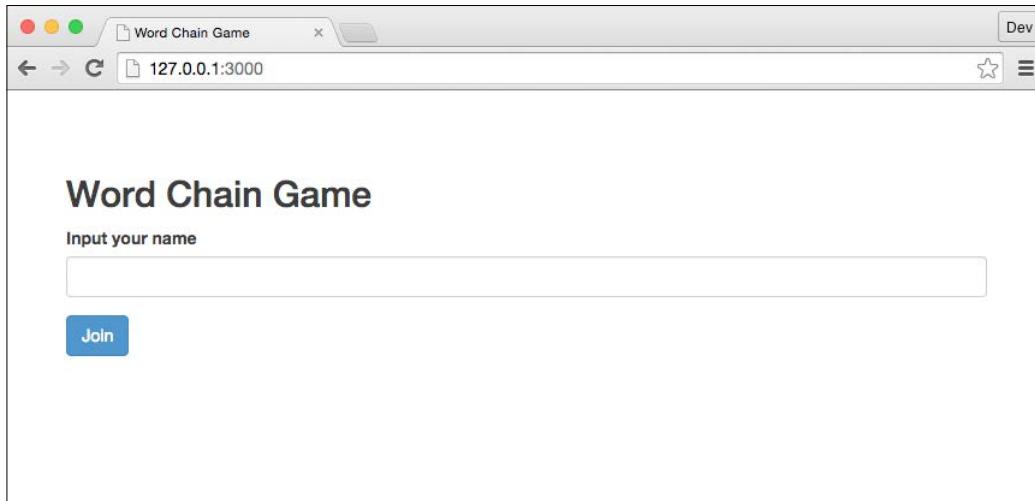
```
Game.prototype = {
  init: function() {
    var me = this;

    this.views.$username.focus();

    this.views.$viewLogin.submit(function() {
      var username = me.views.$username.val();
      me.join(username);
      return false;
    });

    this.views.$viewGame.submit(function() {
      var word = me.views.$wordInput.val();
      me.chain(word);
      me.views.$chain.val('');
      return false;
    });
  };
}
```

The login UI will look like this:



When a user submits a user name, it is passed on to the `join` function, which first establishes a socket connection and then calls `User.join()` (covered earlier) on the server (`game.js`) and initializes a socket handshake (with the configuration to only use WebSocket as the transport) with the submitted username and a URL that consists of `/?username= + username`.

When the connection is established, the socket emits the game status and users list (`updateStat()` and `updateUsers()` functions, which we will discuss later) and calls the `showGameView()` function.

The `showGameView()` function (see the following code block) hides the login form, displays the view-game form where you can input a word to chain, and focuses on the chain input box.

```
join: function(username) {
  var socket = io.connect('/?username=' + username, {
    transports: ['websocket'],
  });

  this.socket = socket;

  var me = this;
  this.socket.on('connect', function() {
    console.log('connect');
```

```
me.socket.emit('game', null, function(game) {
  console.log('stat', game);
  me.updateStat(game.stat);
  me.updateUsers(game.users);
});

  me.showGameView();
});

this.socket.on('join', function(users) {
  me.updateUsers(users);
});

this.socket.on('leave', function(users) {
  me.updateUsers(users);
});

this.socket.on('stat', function(stat) {
  me.updateStat(stat);
});
},
showGameView: function() {
  this.views.$viewLogin.hide();
  this.views.$viewGame.show();
  this.views.$wordInput.focus();
},

```

When a user joins or leaves the game, it's passed to the socket server (`game.js`) `join` or `leave` functions, and the client-side `updateUsers()` function is called.

The `updateUsers()` function takes the array of users that was returned by the server and maps it to get the usernames that are displayed as a list.

Similarly, when a stat call is made to the server, `updateStat()` method gets called, which receives the current word (`stat.word`) from the server and displays it.

Additionally, the input box will contain the last letter of that word as a placeholder and the user who submitted the current word will be displayed by accessing the user array and popping out the last user.

```
updateStat: function(stat) {
  this.views.$word.html(stat.word);
```

```
        this.views.$wordInput.attr('placeholder', stat.word.substr(-1));
        this.views.$bywho.html('current word updated by: ' + stat.used.
        pop().user);
    },

updateUsers: function(users) {
    this.views.$users.html(users.map(function(user) {
        return '<li>' + user + '</li>';
    }).join(''));
},
```

The chain function given in the following alerts a user if they try to submit without entering a word; it then sends a call to the server's chain function, the input word, and the callback function, which will output the data received from the server (which is the response word and used array).

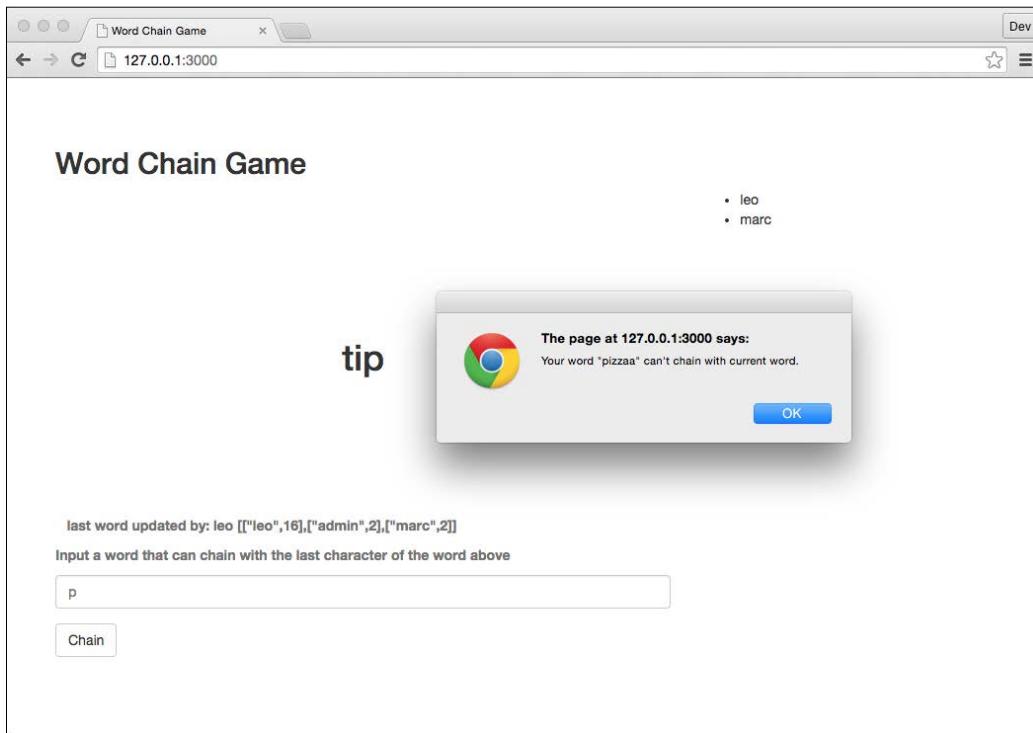
Looking in the server's socket code (socket.js line 47), if a callback is present, and the function was successful, then a status of 200 is sent back.

If the client side receives a status of 200, then it will call the updateStat() function with data.resp, which is the stat object containing the word and used words; otherwise, if no data was received from the server or the chain was unsuccessful and a status code that is not 200 is sent back, the user will see an alert telling them that their input word doesn't chain with the current word.

```
chain: function(word) {
    if (!word) {
        return alert('Please input a word');
    }

    var me = this;
    this.socket.emit('chain', word, function(data) {
        console.log('chain', data);
        if (!data || data.status !== 200) {
            return alert('Your word "' + word + ' can\'t chain with
current word.');
        }

        me.updateStat(data.resp);
    });
}
```



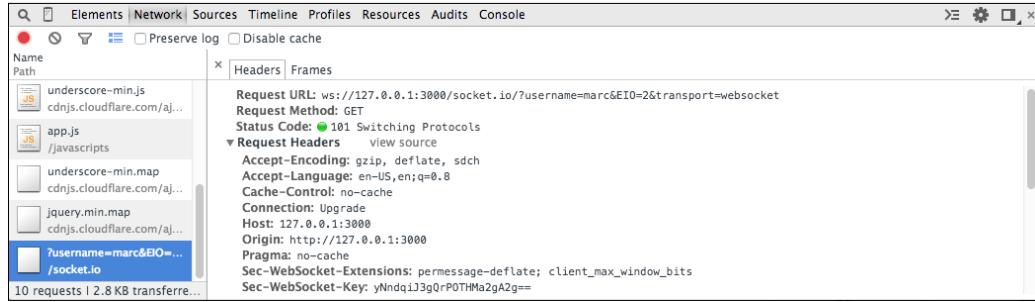
Debug Socket.IO with Chrome Developer Tools

To debug Socket.IO, we want to know what socket request we send to our server, what the request arguments are, and what the broadcast messages look like. Chrome has a built-in powerful WebSocket debugging tool; let's see how to use it.

To open Chrome Developer Tools, go to the menu, select **View | Developer | Developer Tools**. You can also right click on the page, and select **Inspect Element**.

MMO Word Game

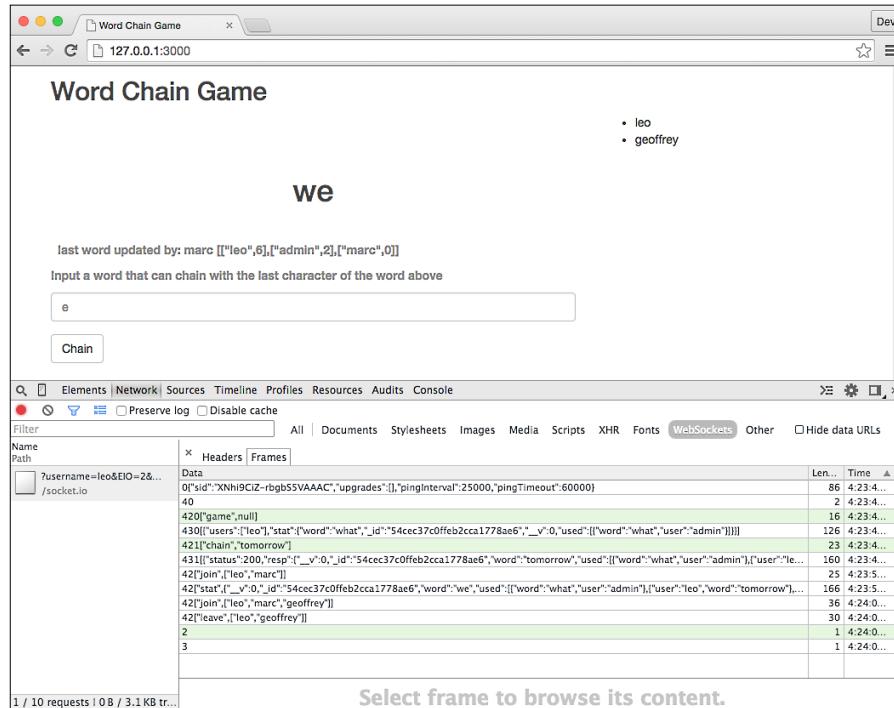
From the Developer tools, select the **Network** panel.



The screenshot shows the Network tab in the Chrome DevTools. A single request is highlighted: `ws://127.0.0.1:3000/socket.io/?username=marc&EIO=2&transport=websocket`. The status code is listed as **101 Switching Protocols**. The Headers section shows standard HTTP headers like Accept-Encoding, Accept-Language, Cache-Control, Connection, Host, Origin, Pragma, Sec-WebSocket-Extensions, and Sec-WebSocket-Key.

Now when we go back to the page and join the game, we will see a Socket.IO request in the **Network** panel of the Chrome Developer Tools. The request URL is `ws://127.0.0.1:3000/socket.io/?username=marc&EIO=2&transport=websocket` and **Status Code is 101 Switching Protocols**, meaning we passed the handshaking and established a socket connection with the server.

Now, click on the **Frames** tab on the right-hand side panel. We will see some messages there in the table. The white rows are the messages our client sent to the server and the green rows are the messages that the server sent to the client.



The screenshot shows the 'Word Chain Game' application running in a browser window. It displays a list of users: leo and geoffrey. Below the list is a word input field with the placeholder 'Input a word that can chain with the last character of the word above' and a 'Chain' button. The Network tab in the DevTools shows a table of WebSockets messages. The table has columns for Data, Len..., and Time. Most messages are green, indicating they were sent by the server. Some white rows are also present, likely representing client-to-server messages.

Data	Len...	Time
0{"id": "XNhj9CIZ-rbgb55VAAAC", "upgrades": [], "pingInterval": 25000, "pingTimeout": 60000}	86	4:23:4...
40	2	4:23:4...
420{"game": null}	16	4:23:4...
430["users": [{"name": "leo"}, {"name": "geoffrey"}], "status": 200, "resp": {"id": "54cec37c0ffeb2cca1778ae6", "word": "we", "used": [{"word": "what", "user": "admin"}, {"word": "the", "user": "leo"}, {"word": "tomorrow", "user": "geoffrey"}]}]	126	4:23:4...
421["chain": "tomorrow"]	23	4:23:4...
431["status": 200, "resp": {"id": "54cec37c0ffeb2cca1778ae6", "word": "tomorrow", "used": [{"word": "what", "user": "admin"}, {"word": "the", "user": "leo"}, {"word": "tomorrow", "user": "geoffrey"}]}]	160	4:23:4...
421["join": {"name": "marc"}]	25	4:23:5...
421["stat": {"v": 0, "id": "54cec37c0ffeb2cca1778ae6", "word": "we", "used": [{"word": "what", "user": "admin"}, {"word": "the", "user": "leo"}, {"word": "tomorrow", "user": "geoffrey"}]}]	166	4:23:5...
421["join": {"name": "marc", "name": "geoffrey"}]	36	4:24:0...
421["leave": {"name": "geoffrey"}]	30	4:24:0...
2	1	4:24:0...
3	1	4:24:0...

Select frame to browse its content.

Let's take a look at each row and understand what happened in the game.

0{"sid":"XNhi9CiZ-rbgbS5VAAAC","upgrades":[],"pingInterval":25000,"pingTimeout":60000}: After the connection is established, the server returned some configuration to the client such as the socket session id (sid), pingInterval, and pingTimeout.

420["game",null]: The client sent a socket request to get the latest game status.

430[{"users":["leo"],"stat":{"word":"what","_id":"54cec37c0ffeb2cca1778ae6",__v:0,"used":[{"word":"what","user":"admin"}]}]: The server responded with the latest game status, which shows that the current word is **what**.

421["chain","tomorrow"]: The client sent a request to chain the current word **what** with **tomorrow**.

431[{"status":200,"resp":{"__v":0,"_id":"54cec37c0ffeb2cca1778ae6","word":"tomorrow","used":[{"word":"what","user":"admin"}, {"user":"leo","word":"tomorrow"}]}]: The server accepted the request and returned the updated game status.
So now, the current word is **tomorrow**

42["join",["leo","marc"]]: marc joined the game. Now we have leo and marc in the game.

42["stat",__v:0,_id:"54cec37c0ffeb2cca1778ae6","word":"we","used":[{"word":"what","user":"admin"}, {"user":"leo","word":"tomorrow"}, {"user":"marc","word":"we"}]]: Here **marc** chained the current word **tomorrow** with **we**. So the server pushed the game status to the client.

42["join",["leo","marc","geoffrey"]]: **geoffrey** joined the game. Now we have three players in the game: **leo**, **marc**, and **geoffrey**.

42["leave",["leo","geoffrey"]]: **marc** left the game, **leo** and **geoffrey** are still in the game.

Now you've had a chance to actually test the game developed for this app and can see how the different aspects intertwine.

Summary

We created an Express app, a Socket.IO server, and a game client that can talk to our server using the `socket.io-client` library, and receive the push updates from our server. We've also gone through the user creation and word chaining logic so that we can validate new users and words to be chained. In this process, we dived into the world of Promises; hopefully, illustrating their versatility and how they can simplify your code.

In the next chapter, we will introduce how to build a user matching system, and make it a service. You will also learn how to set up periodical tasks with node-cron.

5

Coffee with Strangers

In this chapter, we will write an API that allows users to go for a coffee! This comprises of a simple yet extendable user matching system.

Initially, we'll just ask the user to enter their name and e-mail, which is stored on MongoDB. Whenever we can match these with the nearest other user, e-mails are sent to both sides and then it's coffee time. After we set up the base, it's time to make sure we keep a record of the matches and avoid duplicates from happening for a better user experience.

Soon after, let's make ourselves ready to go global and take into account their geo positioning.

Assuming everything goes well (which is a mistake), we are validated. So it's time to refactor to a more maintainable architecture, where the pairing becomes a service by itself.

Finally, let's allow our users to rate how their meeting was and tell us whether it was a successful meeting or not in real-world applications, the usage of user generated feedback is invaluable!

We expect that this sort of application structure will offer the reader inspiration to create real world matching applications.

Code structure

Before getting into actual code, we want to provide a heads up on the structure for the code in this chapter, which is a bit different than before, and we hope it adds another view to structure code for Express and Node.js in general.

Some may call it a **Factory pattern**; it consists of wrapping each of the file's code with a function that can be used to configure or test it. While this requires a bit more of scaffolding, it frees our code from depending on a static state. It will often look as follows:

```
'''javascript
module.exports = function (dependency1) {
    // these will be public
    var methods = {}

    // individual for each instance
    var state = 0

    // some core functionality of this file
    methods.addToState = function(x) {
        state += x
    };

    methods.getResult = function() {
        return dependency1.getVforX(state)
    };

    return methods
}
'''
```

A corollary of this structure is that each invocation of this file will have its own state, exactly like the instance of a class, except we don't depend on this, but the scope that never goes missing.

Going a bit further, we'll try centralizing the structure of the pieces per folder, each with a respective `index.js`, the main responsibilities of which are to initialize instances when needed, keep references to dependencies that will be passed down, and return only public methods.

Defining routes

Let's start by defining the first routes we need and how we want them to behave and simple logical steps building what's strictly essential first, in a TDD style.

1. The first thing is that we need users to be able to register; the smallest test case to register our user is as follows:

```
'''javascript
var dbCleanup = require('./utils/db')
var expect = require('chai').expect;
var request = require('supertest');
var app = require('../src/app');

describe('Registration', function() {
  it("shoots a valid request", function(done) {
    var user = {
      'email': 'supertest'+Math.random()+'@example.com',
      'name': 'Super'+Math.random(),
    };

    request(app)
      .post('/register')
      .send(user)
      .expect(200, done);
  })
})
```

2. Assuming you have Mocha installed with `npm i -g mocha`, execute `mocha`.
3. See 404? Good start! Now let's expand and create a file, `src/route/index.js`, which will declare all the routes known to the app. It uses controllers that handle each concern.
4. Start with `user.js`, which implements a create action, as shown in the following code:

```
'''javascript
// src/routes/index.js
module.exports = function() {
  var router = require('express').Router();
  var register = require('../user')();
  router.post("/user", user.create);
  return router;
};
```

```
// src/routes/user.js
module.exports = function() {
  var methods = {};

  methods.create = function(req, res, next) {
    res.send({});
  }

  return methods;
};
```

5. This amount of code should be enough to make the tests pass with Mocha.

```
~/workspace/az/blueprints-ch5 $ mocha -R spec

  Registration
    ✓ shoots a valid request

  1 passing (23ms)
```

6. For this app, we'll have all route definitions in one place, that is, `routes/index.js`.

At this stage, we know that the testing setup works. Next, let's move onto persistence and some business logic!

Persisting data

Adding some diversity to the libraries, let's experiment with Mongojs (<https://github.com/mafintosh/mongojs>), a simple library for MongoDB that aims to be as close as possible to the native client.

1. First things first, let's create a small config file, `./config.js`, to store all the common data and just return a simple object with relevant configurations for each environment. For now let's just make sure we have a URL in a format accepted by Mongojs.

2. This file should be able to hold all global configs for the app. It ensures we also have different settings depending on the environment.

```
module.exports = function(env) {
  var configs = {};
  configs.dbUrl = "localhost/coffee_"+env;
  return configs;
};
```

3. This file needs to be in `app.js`, a central place to initialize and gather dependencies, and it will be passed to our DB, which then returns public methods. Let's see that happening in the following code:

```
'''javascript
//...
var config = require('../config')(app.get('env'));
var models = require('./models')(config.dbUrl);
app.set('models', models);
//..
```

4. For our models, let's define one file to rule them all `src/models/index.js` with its main responsibilities being to instantiate the db and expose public methods to other modules so that storage details stay encapsulated, keeping the code clean and decoupled.

```
'''javascript
module.exports = function(dbUrl) {
  var mongojs = require('mongojs');
  var db = mongojs(dbUrl);
  var models = {
    User: require('./user')(db)
  };
  return models;
};
```

5. Our first model, `user`, has the ability to create one user. Notice that we are not making any validations in this model to keep things simple. Don't go to production without having the models double-checked.

```
'''javascript
module.exports = function (db) {
  var methods = {};
  var User = db.collection('users');

  methods.create = function(name, email, cb) {
    User.insert({
      name: name,
```

```
        email: email
    }, cb)
};

return methods;
}
```

6. Let's update our `user.js` route to make use of our DB:

```
'''javascript
module.exports = function(Model) {
    var methods = {};

    methods.create = function(req,res,next) {
        Model.User.create(req.param('name'), req.param('email'),
            function(err, user) {
                if(err) return next(err);
                res.send(user);
            });
    }
}
```

7. With this simple change, we should have a user created in our DB.

Let's open Robomongo (<http://robomongo.org/>) to see what user data was created; it's super handy to lookup what data we have in MongoDB no matter what library we use.

The screenshot shows the Robomongo interface. On the left is a tree view of databases and collections. Under 'localhost (20)', 'System' is expanded, showing 'coffee_development'. Under 'coffee_development', 'Collections (3)' is expanded, showing 'System', 'users', and 'Functions'. The 'Users' collection is selected. On the right, a table displays the contents of the 'users' collection. The table has columns 'Key', 'Value', and 'Type'. One row is shown, representing a single document:

Key	Value	Type
{ 1 ObjectId("549...") }	{ 3 fields }	Object
name	Super0.013509273529052734	String
email	supertest0.861390647245571...	String
_id	ObjectId("54986caa53f74ac30...")	ObjectId

Exception handling

Let's open a parenthesis here and talk about the `if(err) return next(err);` command. This is a pattern that is used to abstract error handling in a single action that is supposed to be treated in Express further down the stack, via `app.use`.

1. To keep things neat, we can abstract error handling to a file of its own where we will define specific handlers for each type of error `src/routes/errorHandler.js`.
2. Let's define a `catchAll()` method for now. Express will know the type of use for this function because its functionality is 4.

```
''''
module.exports = function() {
  var methods = {};

  methods.catchAll = function(err, req, res, next) {
    console.warn("catchAll ERR:", err);
    res.status(500).send({
      error: err.toString() ? err.toString() : err
    });
  }

  return methods;
};

''''
```

3. Lastly, it's activated in `routes/index.js`. Error handling should be the very last middleware(s):

```
//..
  router.use(errorHandler.catchAll);

  return router;
};
```

Naive pairing

The simplest pairing system we can implement is to simply lookup if there is another user available without a pair whenever someone signs up.

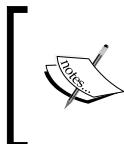
In order to do so, we'll start a new collection and model: `Meeting`, which will be the base matching structure we'll be expanding on. The fundamental idea here is that each document will represent a meeting; either it's in the request phase, already set or occurred, finally will also store the feedback.

We'll be elaborating and defining the structure for it as it goes. For an initial implementation, let's run the scheduling logic right when the user decides to be matched. The strategy will be to look for a meeting document, where only the first user is set, and update it. In case there is no document like that, let's create a new one.

There are a couple of race conditions that might kick in, which we certainly want to avoid. These are as follows:

- The user who's trying to find someone to schedule gets scheduled in the middle of the process.
- The user who's available to be scheduled is selected but then reserved by someone else.

Lucky MongoDB offers the `findAndModify()` method, which can find and update on a single document automatically, while also returning the updated document. Keep in mind that it also offers an `update()` method to update multiple methods.



Check out the docs from the following link: <http://docs.mongodb.org/manual/reference/method/db.collection.findAndModify/#comparisons-with-the-update-method>



Let's get started with a new collection, `Meeting`, where we will keep track of a user's interest in finding a pair as well as keeping track of meetings as follows:

1. This document will contain all of the user's info up to that point in time, so we can use it as a history, as well as use its contents to send e-mails and setup reviews.
2. Let's see what the code looks like in `src/models/meeting.js`:

```
'''javascript
var arrangeTime = function() {
    var time = moment().add(1, 'd');
    time.hour(12);
    time.minute(0);
    time.second(0);
    return time.toDate();
};

methods.pairNaive = function(user, done) {
    /**
     * ...
     */
}
```

```

* Try to find an unpaired User in Meeting collection,
* at the same time, update with an pair id, it either:
* 1. Add the new created user to Meeting collection, or
* 2. The newly created user was added to a Meeting document
*/
Meeting.findAndModify({
    new: true,
    query: {
        user2: { $exists: false },
    },
    update: {
        $set: {
            user2: user,
            at: arrangeTime()
        }
    }
}, function(err, newPair) {
    if (err) { return done(err) }

    if (newPair){
        return done(null, newPair);
    }

    // no user currently waiting for a match
    Meeting.insert({user1: user}, function(err,meeting) {
        done();
    })
});
};

```

```

3. In the case of a successful pairing, `user2` would be set in the `Meeting` object to meet the following day at noon, as you can see on the attribute `at`, which we set via the aux `arrangeTime()` function and the lightweight library `moment.js` (<http://momentjs.com/>). It's amazing to deal with dates in a super readable way. It is recommended that you take a look and become more familiar with it.

Also, notice `new: true` as a parameter. It ensures that MongoDB returns the updated version of the object, so we don't need to duplicate the logic in the app.

The new object `Meeting` needs to be created, as it carries the information of the users at that point in time and can be used to compose the emails/notification for both.

This is a good opportunity to define some basic structure for our tests that will follow a pattern of making several calls to the endpoints and asserting the response. There is a thorough explanation about the decisions to implement tests immediately, as shown in the following code:

```
...
describe('Naive Meeting Setup', function() {
 // will go over each collection and remove all documents
 before(dbCleanup);

 var userRes1, userRes2;

 it("register user 1", function(done) {
 var seed = Math.random()
 var user = {
 'name': 'Super'+seed,
 'email': 'supertest'+seed+'@example.com',
 }

 request(app)
 .post('/register')
 .send(user)
 .expect(200, function(err,res){
 userRes1 = res.body
 done(err)
 })
 });

 it('should be no meeting for one user', function(done) {
 models.Meeting.all(function(err,meetings) {
 expect(meetings).to.have.length(1);
 var meeting = meetings[0];
 expect(meeting.user1).to.be.an("object");
 expect(meeting.user2).to.be.an("undefined");
 done(err);
 });
 });
});
```

```
it("register user 2", function(done) {
 var seed = Math.random();
 var user = {
 'name': 'Super'+seed,
 'email': 'supertest'+seed+'@example.com',
 };

 request(app)
 .post('/register')
 .send(user)
 .expect(200, function(err,res){
 userRes2 = res.body
 done(err)
 });
});

it('should be a meeting setup with the 2 users', function(done) {
 models.Meeting.all(function(err,meetings) {
 expect(meetings).to.have.length(1)
 var meeting = meetings[0]
 expect(meeting.user1.email).to.equal(userRes1.email)
 expect(meeting.user2.email).to.equal(userRes2.email)
 done(err)
 });
});
});
}
});
```

(Source:git checkout e4fbf672d409482028de7c7427eab769ab0a20d2)

```
~/workspace/az/blueprints-ch5 $ mocha -R spec test/meeting_naive.js

 Naive Meeting Setup
 ✓ register user 1 (57ms)
 ✓ should be no meeting for one user
 ✓ register user 2
 ✓ should be a meeting setup with the 2 users

 4 passing (106ms)
```

## Notes about tests

When using Mocha, a test is much like any javascript file and is executed as expected, allowing for any sort of regular Node.js require as you would do usually.

The `describe()` method is the context on which our tests execute; in our case, it's a full run of a certain functionality. The `before()` method will run once; in this case, our logic is set to clean up all of our MongoDB collections.

It stands for a simple expectation to be fulfilled. It will run in the same order it's declared, and we will try to make the assertions small and predictable as far as possible. Each of these functions defines steps and in this case, because we are doing end-to-end tests, we make requests to the API and check the results, and sometimes save it to variables that are used later to assert.

There are advices that say that tests shouldn't be dependent on the previous state, but those don't usually test the application flow, rather, individual portions of logic. For this particular test scenario, in case of a failure, it's important to interpret the error from the first `it` that fails; fixing it will likely fix errors after it. You can configure Mocha to stop at the first error by using the `-b` flag.

While testing, the most important point to make sense is that our test cases should make sure all of the expected cases are checked, and bad behaviors don't happen. We can never expect to predict everything that may go wrong of course, but it's still our duty to test as many points as we are certain about common issues.

## Considering user history

Our users will probably want to always be paired to meet new people, so we have to avoid repetitive meetings. How should we handle this?

First, we need to allow for a method to set up new meetings. Think of it as a button in an app that would trigger a request to the route `POST/meeting/new`.

This endpoint will reply with the status `200` when the request is allowed and a pair is found, or if there is no pair but they are now attached to a `meeting` object and can now be matched with another user; `412` if the user is already scheduled in another meeting and `400` in case the expected e-mail of the user isn't sent; in this case, it can't be fulfilled because the user wasn't specified.



The usage of status codes is somewhat subjective, (see a more comprehensive list on Wikipedia at [http://en.wikipedia.org/wiki/List\\_of\\_HTTP\\_status\\_codes](http://en.wikipedia.org/wiki/List_of_HTTP_status_codes)). However, having distinct responses is important so that the client can display meaningful messages to the user.

Let's implement an Express.js middleware, that requires an e-mail for all requests that are made on behalf of the user. It should also load their document and attach it to `res.locals`, which can be used in subsequent routes.

Our `src/routes/index.js` will look like this:

```
'''javascript
//...
 app.post("/register", register.create);
 app.post("/meeting", [filter.requireUser], meeting.create);
//...
'''
```

The filter in '`src/routes/filter.js`' is:

```
'''javascript
module.exports = function(Model) {
 var methods = {};

 methods.loadUser = function(req,res,next) {
 var email = req.query.email || req.body.email
 if(!email) return res.status(400).send({error: "email missing, it
should be either in the body or querystring"});
 Model.User.loadByEmail(email, function(err,user) {
 if(err) return next(err);
 if(!user) return res.status(400).send({error: "email not
associated with an user"});
 res.locals.user = user;
 next();
 })
 }

 return methods;
};
```

The goal of this middleware is to stop and return an error message for every request that doesn't have a user email. It's a validation that would usually require a username and password or a secret token.

Let's set up a small but important test suite for this middleware:

- Clear DB
- Try to get me without email and fail
- Create a valid user; it succeeds
- Try to get me with another email; it fails
- Access me with the email we registered and it works!

Now that we have a way to load the user who's making the request, let's go back to the goal of matching people without repetition. As a pre-condition, their past meeting time has to be in the past already, otherwise it returns a 412 code.

If we want to schedule a meeting for our users but any scheduled meeting will be set for tomorrow, how can we test it? Meet timekeeper (<https://github.com/vesln/timekeeper>), library with a simple interface to alter the system dates in Node.js; this is especially useful for tests. Look closely for the snippet of this test:

```
'''javascript
describe('Meeting Setup', function() {
 before(dbCleanup);

 after(function() {
 timekeeper.reset();
 });

 // ...

 it('should try matching an already matched user', function(done) {
 request(app)
 .post('/meeting')
 .send({email:userRes1.email})
 .expect(412, done);
 });

 it('should be able match the user again, 2 days later', function() {
 var nextNextDay = moment().add(2,'d');
 timekeeper.travel(nextNextDay.toDate());
 });
});
```

```
request(app)
 .post('/meeting')
 .send({email:userRes1.email})
 .expect(200, function(err,res) {
 done(err);
 });
});
```

It's of vital importance to set an `after` hook to reset timekeeper so that the dates go back to normal after the scenario is finished in either success or failure; otherwise, there is a chance it will alter the results of other tests. It's also worth checking how date manipulation is made easy with `moment()` method and once you use `timekeeper.travel()` function, the time is warped to that date. For all Node.js knows, the new warped time is the actual time (although it does not affect any other applications). We can also switch it back and forth as required.

The `Meeting` method to perform this check on our user (defined at `models/meeting.js`) is as follows:

```
methods.isUserScheduled = function(user, cb) {
 Meeting.count({
 $or: [
 {'user1.email': user.email},
 {'user2.email': user.email}
],
 at: {$gt: new Date()}
 }, function(err, count) {
 cb(err, count > 0);
 });
};
```

The `$or` operator is necessary because we don't know whether the user we are looking for is going to be `user1` or `user2`, so we take advantage of the query capabilities of MongoDB that can look inside objects in a document and match the `email` as a `String`, and the `at` field as mentioned earlier.

Our newly created `src/routes/meeting.js`, is given as follows:

```
'''javascript
module.exports = function(Model) {
 var methods = {};

 methods.create = function(req,res,next) {
 var user = res.locals.user;
```

---

*Coffee with Strangers*

---

```
 Model.Meeting.isUserScheduled(user, function(err,isScheduled) {
 if(err) return next(err);
 if(isScheduled) return res.status(412).send({error: "user is
already scheduled"});
 Model.Meeting.pair(user, function(err,result) {
 // we don't really expect this function to fail, if that's the
 case it should be an internal error
 if(err) return next(err);
 res.send({});
 })
 })
 }

 return methods;
};
```

Moving on, we'll define a very important helper function that finds previous meetings involving the user who's making the request and returns the emails of everyone they have been matched with, so we can avoid matching those two users again.

Helper functions like this are super useful to keep the code understandable when dealing with complicated pieces of logic. As a rule of thumb, always separate into smaller functions when a chunk of code can be abstracted into a concept.

```
/**
 * the callback returns an array with emails that have previously
been
 * matched with this user
*/
methods.userMatchHistory = function(user,cb) {
 var email = user.email;
 Meeting.find({
 $or:[
 {'user1.email': email},
 {'user2.email': email}
],
 user1: {$exists: true},
 user2: {$exists: true}
 }, function(err, meetings) {
 if(err) return cb(err);
 var pastMatches = meetings.map(function(m) {
```

```

 if(m.user1.email != email) return m.user1.email;
 else return m.user2.email;
 });
 // avoid matching themselves!
 pastMatches.push(user.email);
 cb(null, pastMatches);
}
}

```

The key to `userMatchHistory` object; is through the MongoDB `$nin` operator, which performs a match when the element doesn't match what's in the array. The matching logic follows the very same logic we had in naive pairing.

In our `Meeting` model, we removed our previous `pairNaive` method with the `pair` method, which does similar, but first build a list of the previous matches to ensure we don't match those again.

```

methods.pair = function(user,done) {
 // find the people we shouldn't be matched with again
 methods.userMatchHistory(user, function(err, emailList) {
 if(err) return done(err);

 Meeting.findAndModify({
 new: true,
 query: {
 user2: { $exists: false },
 'user1.email': {$nin: emailList}
 },
 update: {
 $set: {
 user2: user,
 at: arrangeTime()
 }
 }
 }, function(err, newPair) {
 if (err) { return done(err); }

 if (newPair){
 return done(null, newPair);
 }

 Meeting.insert({user1: user}, function(err,meeting) {
 done();
 })
 })
 })
}

```

```
 })
 return;
 });
}
}
```

## Optimizing for distance

Let's use a down-to-earth geolocation approach (ah! I'm so funny) to the match. We have to be realistic. Our service was born in Smallville but it's going global and we can't match people who are too far apart.

Because our meetings are arranged free of racing conditions on the `Meeting` collection and we would like to keep it that way, let's adapt our existing `pair` method to incorporate the user's location. We can assume that at registration, they will supply their location (or we could also easily update the meeting document once they provide the location). In our existing strategy, we have one user who creates a meeting document; in this case, let's also set their location, so the next user looking for a match will have to be in a similar location as an additional constraint, as shown in the following code:

```
'''javascript
Meeting.ensureIndex({location1: "2dsphere"});

//..

methods.pair = function(user,done) {
 methods.userMatchHistory(user, function(err, emailList) {
 if(err) return done(err);

 Meeting.findAndModify({
 new: true,
 query: {
 user2: {$exists: false},
 'user1.email': {$nin: emailList},
 'location1': {$nearSphere:{
 $geometry :
 { type : 'Point',
 coordinates : user.location } ,
 $maxDistance : 7*1000
 }}}
 },
 },
}
```

```
 update: {
 $set: {
 user2: user,
 at: arrangeTime()
 }
 },
 function(err, newPair) {
 if (err) { return done(err); }

 if (newPair){
 return done(null, newPair);
 }

 Meeting.insert({
 user1: user,
 location1: user.location
 }, function(err,meeting) {
 done();
 })
 return;
 });
 });
 }
 ...
}
```

Our Meeting collection now has `location1` indexed as `2dsphere`. The geoquery for this field can easily be integrated with our previous query, using the operator `$nearSphere` to match geo positions in a sphere object. `$maxDistance` is the maximum radius for the match. It's expressed in meters and in this case, we intersect the coordinates with `Point`, which is a previously registered user. `7km` was chosen arbitrarily because it seems like a reasonable enough radius to meet someone.

If we changed `$maxDistance` to something considerably smaller, some of our tests would fail because matches wouldn't happen; see `test/meeting_near.js`.

- Clear DB
- Create user 1 in Santiago
- Create user 4 in Valparaiso
- Check whether there is a match
- Create user 2 in Santiago

- Check whether there was a match between 1 and 2
- Create user 3 in Vancouver
- Check whether 3 has a match
- Create user 5 in Valparaiso
- Check whether there is a match between 4 & 5

(Source:git checkout 52e8f80b7fe3b9482ff27ea1bcc410270752a796)

## E-mail follow up

Users can now be matched. The meetings are unique and made between people that are nearby, which is awesome! There is no end to possible improvements on a matching system; so instead, let's now collect some data about how their meeting went!

To do so, we'll send an email to each of the attendees, which will consist of a few simple options to promote engagement. Some of them are listed as follows:

- It was awesome
- It was awful
- Meh...
- My pair didn't show up!

Those values are added to `src/models/meeting.js` as key-value pairs, which we can store for ratings and use them to communicate back to users.

```
methods.outcomes = function() {
 return {
 awesome : "It was awesome",
 awful : "It was awful",
 meh : "Meh",
 noshow : "My pair didn't show up!"
 }
}
```

We could store these responses in the respective `meeting` object, associating it with the user who responded.

For this purpose, we'll rely primarily on the package `Nodemailer` (<https://github.com/andris9/Nodemailer>). It is broadly used and offers support for a number of integrations, including transport providers and templates so we can make our e-mails dynamic.

Coming to the setup decision, as you probably realized Node.js & Express are free of conventions about how to set up your code because these apps may do very different things and there is no one-size-fits-all. Let's make mailing a concern of its own, as much as persistence and routes are separated concerns integrated into `src/app.js`.

The `src/mailer/index.js` will be our entry point and its main responsibility is to instantiate the `nodemailer` variable and provide public methods other files can refer to.

```
...
var nodemailer = require('nodemailer')

module.exports = function (mailConfig) {
 var methods = {};
 var transporter;

 // Setup transport
 if(process.env.NODE_ENV == 'test'){
 var stubTransport = require('nodemailer-stub-transport');
 transporter = nodemailer.createTransport(stubTransport());
 } else if(mailConfig.service === 'Mailgun'){
 transporter = nodemailer.createTransport({
 service: 'Mailgun',
 auth: {
 user: mailConfig.user,
 pass: mailConfig.password
 }
 });
 } else {
 throw new Error("email service missing");
 }

 // define a simple function to deliver mails
 methods.send = function(recipients, subject, body, cb) {
 // small trick to ensure dev & tests emails go to myself
 if(process.env.NODE_ENV !== 'production') {
 recipients = ["my.own.email@provider.com"];
 }
 transporter.sendMail({
 to: recipients,
 from: mailConfig.from,
 subject: subject,
 generateTextFromHTML: true,
 html: body
 }
 }
}
```

```
 }, function(err, info) {
 // console.info("nodemailer::send",err,info)
 if(typeof cb === 'function'){
 cb(err,info);
 }
 })
}

return methods;
}
```

When it comes to the test environment, we definitely don't want to be sending real e-mails, that's why we register the stub transport. For other environments, we decided to go with Mailgun but we could also go with any service that integrates via SMTP (remember to use Gmail since there is a risk of failing to send e-mails, as they have a bunch of heuristics to prevent spam).

When it comes to testing, this section is one of the harder ones to test, we will implement something very basic in `test/send_mail.js`

```
var dbCleanup = require('../utils/db');
var app = require('../src/app');
var mailer = app.get('mailer');

describe('Meeting Setup', function() {

 it('just send one.', function(done) {
 this.timeout(10*1000);
 mailer.send(
 "my.own.email@provider.com",
 "Test "+(new Date()).toLocaleString(),
 "Body "+Math.random()+"
"+Math.random()
 , done);
 })
})
```

Add to `config.js`, and have the correspondent environment variables defined because it's not a good idea to keep our secrets in the code.

```
var ENV = process.env;
configs.email = {
 service: "Mailgun",
 from: ENV.MAIL_FROM,
 user: ENV.MAIL_USER,
 password: ENV.MAIL_PASSWORD
};
```

When I disable the `test` environment, I can actually see the email in my inbox. Win! To make the service look better, let's experiment with some templates, which is what `email-templates` (<https://github.com/niftylettuce/node-email-templates>) is all about.

It makes it easy to implement dynamic e-mails including packing the CSS inline; these are required to be inline by many e-mail clients.

On `src/mailers/followUp.js`

```
'''javascript
module.exports = function(sendMail, models) {
 //..

 function sendForUser (user1, user2, id, date, cb) {
 emailTemplates(templatesDir, function(err,template) {
 if(err) return cb(err);

 template('followup', {
 meetingId: id.toString(),
 user1: user1,
 user2: user2,
 date: date,
 outcomes: Meeting.outcomes()
 }, function(err,html) {
 if(err) return cb(err);
 sendMail(
 user1.email,
 "How was your meeting with "+user2.name+"?",
 html,
 cb
)
 });
 });
 });

 // call done() when both emails are sent
 return function followUp(meeting, done) {
 async.parallel([
 function(cb) {
 sendForUser(meeting.user1, meeting.user2, meeting._id,
meeting.at, cb);
 },

```

```
 function(cb) {
 sendForUser(meeting.user2, meeting.user1, meeting._id,
meeting.at, cb);
 },
], done)
}
}
```

Essentially, we send two identical emails so we get feedback from both users. There is a bit of complexity there that we will manage by using `async.parallel()` method. It allows us to start two asynchronous operations and callbacks (`done`) when both are completed. See <https://github.com/caolan/async#parallel>.

The actual print of the email is created by two files, `src/mailer/templates/followup/followUp.html.swig` and `style.css`, which are combined and set via our transport solution, respectively:

```
'''html
<h4 class="title">
 Hey {{user1.name}},
</h4>
<div class="text">
 We hope you just had an awesome meeting with {{user2.name}}!
 You guys were supposed to meetup at {{date|date('JS \o\f F H:i')}},
 how did it go?
</div>

 {% for id, text in outcomes %}
 <a href="http://127.0.0.1:8000/followup/{{meetingId}}/{{user2._
id.toString()}}/{{id}}">{{text}}
 {% endfor %}

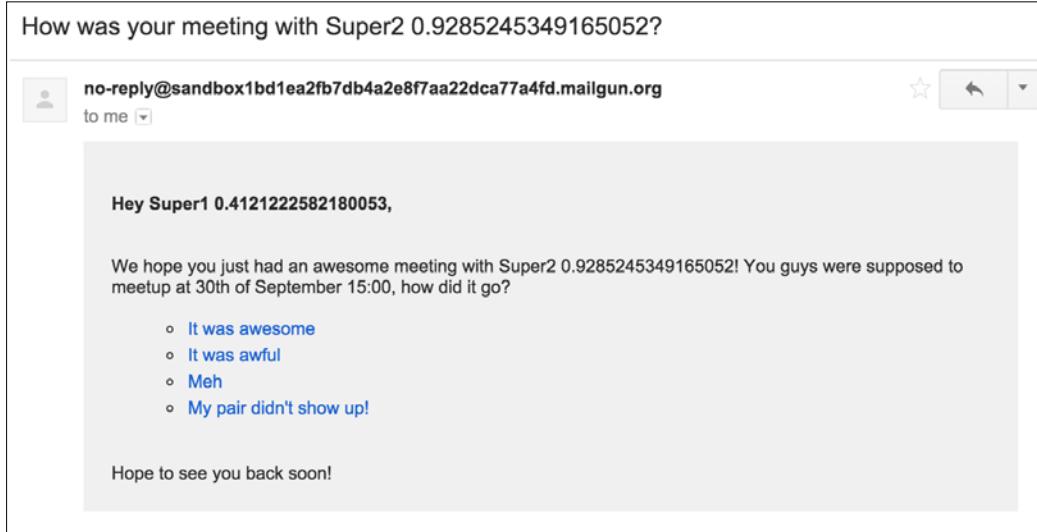
<div class="text">
 Hope to see you back soon!
</div>
'''

'''css
body{
 background: #EEE;
 padding: 20px;
}
.text{
 margin-top: 30px;
}
```

```
ul{
 list-style-type: circle;
}
ul li{
 line-height: 150%;
}
a{
 text-decoration: none;
}
```

We can choose from many template solutions. swig (<http://paularmstrong.github.io/swig/docs/>) comes with convenient helpers, makes it easy to work with lists, and has the familiar HTML visual. A bit of insight is given as follows:

- `{ {string} }` is the general interpolating method
- `|` is for helpers (aka filters); you can use built-ins or define your own
- `for k,v in obj` is a tag and works looping over key-value pairs



When it came to the logic for the follow-up links, we made it really easy for the user to provide feedback; usually, the less friction, the better for outstanding UX. All they have to do is click on the link and their review is instantly recorded! In terms of Express.js, this means we have to set up a route that links all the piece of data together; in this case, in `src/routes/index.js`:

```
app.get("/followup/:meetingId/:reviewedUserId/:feedback",
 meeting.followUp);
```

To have an endpoint that actually changes the data defined as a GET is an exception to HTTP & REST conventions, but the reason is that email clients will send the request as a GET; not a lot we can do about it.

The method is defined at `src/routes/meeting.js` as follows:

```
methods.followUp = function(req, res, next) {
 var meetingId = req.param("meetingId");
 var reviewedUserId = req.param("reviewedUserId");
 var feedback = req.param("feedback");
 // validate feedback
 if(!(feedback in Model.Meeting.outcomes())) return
 res.status(400).send("Feedback not recognized");
 Model.Meeting.didMeetingHappened(meetingId, function(err,
 itDid) {
 if(err){
 if(err.message == "no meeting found by this id"){
 return res.status(404).send(err.message);
 } else {
 return next(err);
 }
 }
 if(!itDid){
 return res.status(412).send("The meeting didn't happen
 yet, come back later!");
 }
 Model.Meeting.rate(meetingId, reviewedUserId, feedback,
 function(err, userName, text) {
 if(err) return next(err);
 res.send("You just rated your meeting with "+userName+
 " as
 "+text+". Thanks!");
 });
 });
}
```

This method does quite a bit of checking and that's because there is a considerable amount of input that needs validation along with providing the appropriate response. First, we check whether the feedback provided is valid, since we are only taking quantitative data. `didMeetingHappened` returns two important pieces of info about the meeting; the ID may be completely wrong, or it might not have happened yet. Both scenarios should deliver different results. Finally, if everything looks good, we attempt to rate the meeting, which should work just fine and return some data to respond with and finish the request with an implied 200 status.

The implementation of the preceding methods are available at `src/models/meeting.js`

```

 ...
 // cb(err, itDid)
 methods.didMeetingHappened = function(meetingId, cb) {
 if(!db.ObjectId.isValid(meetingId)) return cb(new Error("bad
ObjectID"));
 Meeting.findOne({
 user1: {$exists: true},
 user2: {$exists: true},
 _id: new db.ObjectId(meetingId)
 }, function(err, meeting) {
 if(err) return cb(err);
 if(!meeting) return cb(new Error('no meeting found by this
id'));
 if(meeting.at > new Date()) return cb(null, false);
 cb(null, true);
 })
 }

 // cb(err, userName, text)
 methods.rate = function(meetingId, reviewedUserId, feedback, cb) {
 Meeting.findOne({
 _id: new db.ObjectId(meetingId),
 }, function(err, meeting) {
 if(err) return cb(err);
 var update = {};
 // check the ids present at the meeting object, if user 1 is
 being reviewed then the review belongs to user 2
 var targetUser = (meeting.user1._id.toString() ==
reviewedUserId) ? '1' : '2';
 update["user"+targetUser+"review"] = feedback;
 Meeting.findAndModify({
 new: true,
 query: {
 _id: new db.ObjectId(meetingId),
 },
 update: {
 $set: update
 }
 }, function(err, meeting) {
 if(err) return cb(err);
 var userName = (meeting["user"+targetUser].name);
 var text = methods.outcomes()[feedback];
 cb(null, userName, text);
 })
 })
 }
}
...

```

The implementation method should be pretty readable. The `didMeetingHappened()` method looks for a maximum of one document with `_id`, where `user1` and `user2` are filled. When this document is found, we look at the `at` field and compare with the current time to check whether it already happened.

The rate is a bit longer but just as simple. We find the `meeting` object and figure out which user is being rated. Such feedback belonging to the opposite user is stored in an atomic operation, setting either field `user1reviewed` or `user2reviewed` with the key for the feedback.

We have a thorough test suite implemented for this case, where we mind both success & failure cases. It can be used to check the emails by simply calling the test with `NODE_ENV=development mocha test/meeting_followup.js`, which then overrides the test environment with development and delivers emails to our provider, so we can see how it looks and fine-tune it.

Our test for this whole scenario is a bit long but we need to test several things!

- Clean up DB
- Setting up the meeting
- Register user 1
- Register user 2 at the same position
- STest that non-existent meetings can't be reviewed
- Status 412 on meeting reviews that still didn't happen
- Travel time two days ahead
- Send an email
- Taking up a review that makes sense
- User 1 should be able to review the meeting
- User 2 should be able to review the meeting as well

Seems like we can now send emails and receive reviews, which is great, but how do we send the emails in a time-sensitive manner? A couple of minutes after the meeting has started, the emails should be sent to both parties.

(Source:git checkout 7f5303ef10bfa3d3bfb33469dea957f63e0ab1dc)

## Periodical tasks with node-cron

Maybe you are familiar with cron (<http://en.wikipedia.org/wiki/Cron>). It's a Unix-based task scheduling system that makes running tasks easy. One problem with it is that it's linked to your platform, and it's not trivial to turn it on and off from code.

Meet node-cron (<https://github.com/ncb000gt/node-cron>). It's basically the same task scheduler but it runs directly from your Node application, so as long as it is up, your jobs should run.

Our strategy is simple: Periodically select all meetings that need mailing, call our mailer with each of these meetings, and then mark it as emailed.

Following this app's convention, let's separate concerns into their own folders, starting with `src/tasks/index.js`, as shown in the following code:

```
var CronJob = require('cron').CronJob;

module.exports = function(models, mailer) {
 var tasks = {};

 tasks.followupMail = require('./followupMail')(models, mailer);

 tasks.init = function() {
 (new CronJob('00 */15 * * *', tasks.followupMail)).start();
 };

 return tasks;
}
```

It needs to take `models` and `mailer` as parameters, which can be used inside tasks. `followupMail` is the single user defined for now because it's all we need. The exported method `init` is what will kick start the cron job, the timer presenting respectively: `00` defined as the seconds, meaning it will run at second `00`, for every `*/15` minutes, any hour, any day of the month, any month, any day of the week. For the actual task, see `src/mailers/followUp.js`

```
...
module.exports = function(Model, mailer) {
 return function() {
 Model.Meeting.needMailing(function(err, meetings) {
 if(err) return console.warn("needMailing", err);
 if(!meetings || meetings.length < 1) return;
 meetings.forEach(function(meeting) {
```

```
 mailer.followUp(meeting, function(err) {
 if(err) return console.warn("needMailing followup failed
"+meeting._id.toString(), err);
 Model.Meeting.markAsMailed(meeting._id);
 });
 });
 Model.Meeting.markAsMailed(meetings);
});
};

```
It returns a function, which when executed, looks up all meeting documents that still need to be mailed and for each one, use mailer.followUp as we defined before and upon completion, mark each email as sent. Notice that fails here have nowhere to communicate and that's because it's an automated task. It's important for web servers to have meaningful log reporting, so in this case, the warning messages should be reported.

Of course, this would require us to add two methods to src/models/meeting.js, which you should be able to easily make sense of by now:



```
// all meetings that are due and not mailed yet
methods.needMailing = function(cb) {
 Meeting.find({
 at: {$lt: new Date},
 mailed: {$exists: false}
 },cb);
};

// mark a meeting as mailed
methods.markAsMailed = function(id,cb) {
 Meeting.findAndModify({
 query: {
 _id: id
 },
 update:{
 $set: {mailed: new Date()}
 }
 },cb);
};
```



---



[ 136 ]


```

For our final test, we'll be creating four users implying 2 meetings, travel 2 days in the future and try sending the emails through the task, it should work and mark both emails as sent.

- Clear DB
- Register users 1, 2, 3, and 4 at the same location
- Travel time after the meeting is done
- Task should send an email
- Verify that the emails were sent

Summary

In this chapter, we created an API that can set up meetings between users taking into account their matching history and the pair of longitude and latitude, while providing them the chance to give feedback on how it went-essential information which can be used in many ways to further improve the algorithm!

We hope you learned about many interesting and practical concepts, such as making geo queries, testing time-sensitive code, sending e-mails with style, and tasks that run periodically.

Besides the technical bits, hope you had fun and perhaps were able to spark some insight on the framework behind match-making apps!

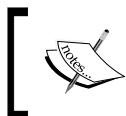
Next on, we'll see how `Koa.js` works by leveraging the power of generators, bringing the readability of synchronous code on top of the `async` features of `Node.js`.

6

Hacker News API on Koa.js

In this chapter, we will build an API to power our own Hacker News! While technically this wouldn't be very different from the previous chapters, we will use a different framework altogether, Koa.js (<http://koajs.com/>).

Koa.js is a new web framework designed by the team behind Express. Why did they create a new framework? Because it is designed from the bottom up, with a minimalistic core for more modularity, and to make use of the new generator syntax, proposed in ECMAScript 6, but already implemented in node 0.11.



The odd version releases of node are considered unstable. At the time of writing, the latest stable release was version 0.10. However, when this book went to print, node 0.12 was finally released and is the latest stable version.

An alternative to node 0.11 would be io.js, which at the time of writing reached version 1.0, and also implements ES6 goodies (forked from Node.js and maintained by a handful of node core contributors). In this chapter, we will stick to node 0.11. (When this book went to print, node 0.12 was finally released and is the latest stable version of node.)

One of the main benefits of the generator syntax is that you can very elegantly avoid callback hell, without the use of complicated promise patterns. You can write your APIs even more cleanly than ever before. We'll go over the subtleties as well as some caveats that come with the bleeding edge.

Some things we will cover in this chapter are as follows:

- Generator syntax
- Middleware philosophy
- Context, versus req,res

- Centralized error handling
- Mongoose models in Koa.js
- Thunkify to use Express modules
- Testing generator functions with Mocha
- Parallel HTTP requests using co-mocha
- Rendering views with koa-render
- Serve static assets with koa-mount and koa-static

Generator syntax

Generator functions are at the core of Koa.js, so let's dive right into dissecting this beast. Generators allow adept JavaScript users to implement functions in completely new ways. Koa.js makes use of the new syntax to write code in a synchronous-looking fashion while maintaining the performance benefits of an asynchronous flow.

The following defines a simple generator function in `src/helloGenerator.js` (note the asterisk syntax):

```
module.exports = function *() {
  return 'hello generator';
};
```

To use Mocha with Koa.js:

1. You will need to include `co-mocha` to add generator support, requiring once at the first line of each test file is the safe way to do it. Now you can pass generator functions to Mocha's `it` function as follows:

```
require('co-mocha');
var expect = require('chai').expect;
var helloGenerator = require('../src/helloGenerator');

describe('Hello Generator', function() {
  it('should yield to the function and return hello',
    function *() {
      var ans = yield helloGenerator();
      expect(ans).to.equal('hello generator');
    });
});
```

2. In order to run this code, you will need to have node 0.11 installed, and use the `--harmony-generators` flag as you run Mocha:

```
./node_modules/mocha/bin/mocha --harmony-generators
```

3. If all is well, congratulations, you have just written your first generator function! Now let's explore the execution flow of generator functions a little more.

 Note the magic use of the `yield` keyword. The `yield` keyword can only be used within a `Generator` function, and works somewhat similar to `return`, expecting a single value to be passed, that can also be a generator function (also accepts other yieldables – more on that later), and yields the process to that function.

When a `function*` is passed, the execution flow will wait until that function returns before it continues further down. In essence, it would be equivalent to the following callback pattern:

```
helloGenerator(function(ans) {
  expect(ans).to.equal('hello generator');
});
```

Much cleaner, right? Compare the following code:

```
var A = yield foo();
var B = yield bar(A);
var C = yield baz(A, B);
```

With the nasty callback hell if we didn't have generator functions:

```
var A, B, C;
```

```
foo(function(A) {
  bar(A, function(B) {
    baz(A, B, function(C) {
      return C;
    });
  });
});
```

Another neat advantage is super clean error handling, which we will get into later.

The preceding example is not too interesting because the `helloGenerator()` function is a synchronous function anyway, so it would've worked the same, even if we didn't use generator functions!

4. So let's make it more interesting and change `helloGenerator.js` to the following:

```
module.exports = function *() {
  setTimeout(function(){
    return 'hello generator';
  }, 1000);
}
```

Wait! Your test is failing?! What is going on here? Well, `yield` should have given the flow to the `helloGenerator()` function, let it run asynchronously, and wait until it is done before continuing. Yet, `ans` is undefined. And nobody is lying.

The reason why it is undefined is because the `generator()` function returns immediately after calling the `setTimeout` function, which is set to `ans`.

The message that should have returned from within the `setTimeout` function is broadcast into the infinite void, nowhere to be seen, ever again.

[ One thing to keep in mind with generator functions is that once you use a generator function, you better commit, and not resort to callbacks down the stack! Recall that we mentioned that `yield` expects a generator function. The `setTimeout` function is not a generator function, so what do we do? The `yield` method can also accept a Promise or a Thunk (more on this later).]

5. The `setTimeout()` function isn't a Promise, so we have two options left; we can thunkify the function, which basically takes a normal node function with a callback pattern and returns a Thunk, so we can yield to it; alternatively, we use co-sleep, which is basically a minuscule node package that has done it for you as follows:

```
module.exports = sleep;
function sleep(ms) {
```

```
        return function (cb) {
            setTimeout(cb, ms);
        };
    }
}
```

6. We will talk about how to thunkify later, so let's use `co-sleep`. Generally a good idea to reuse what's available is to just do a quick search in the npm registry. There are numerous `co` packages out there!

```
var sleep = require('co-sleep');
```

```
module.exports = function *() {
    yield sleep(1000);
    return 'hello generator';
}
```

7. Now all should be good; your tests should pass after sleeping for 1 second.
8. Note that the `co` library is what's under the hood of Koa.js, giving it the generator-based control flow goodies. If you want to use this sort of flow outside Koa.js, you can use something like this:

```
var co = require('co');
var sleep = require('co-sleep');

co(function*(){
    console.log('1');
    yield sleep(10);
    console.log('3');
})();

console.log('2');
```

Middleware philosophy

You should be familiar by now with the middlewares in Express. We used them a lot to dry out code, especially for validation and authentication. In Express, middleware is placed between the server that receives the request and the handler that responds to a request. The request flows one way, until it terminates at `res.send` or something equivalent.

In Koa.js, everything is a middleware, including the handler itself. As a matter of fact, a Koa.js application is just an object, which contains an array of middleware generator functions! The request flows all the way down the stack of middlewares, and back up again. This is best explained with a simple example:

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

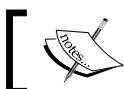
app.use(function *(next) {
  var start = new Date();
  yield next;
  var ms = new Date() - start;
  this.set('X-Response-Time', ms + 'ms');
});

app.use(function *() {
  this.body = 'Hello World';
});

app.listen(3000);
```

Here we have a Koa.js application with two middlewares. The first middleware adds an `X-Response-Time` header to the response, whereas the second middleware simply sets the response body to `Hello World` for each request. The flow is as follows:

- The request comes in on port 3000.
- The first middleware receives the execution flow.
- A new `Date` object is created and assigned to `start`.
- The flow yields to the next middleware on the stack.
- The second middleware sets `body` on the Context to `Hello World`.
- Since there is no more middleware down the stack to be yielding to, the flow returns back upstream.
- The first middleware receives the execution flow again and continues down.
- The response time is calculated and the response header is set.
- The request has reached the top and the Context is returned.



The Koa.js does not use `req`, `res` anymore; they are encapsulated into a single Context.



To run this app, we can use the following command:

```
node --harmony app.js
```

Context versus req,res

A Koa.js Context is created for each incoming request. Within each middleware, you can access the Context using the `this` object. It includes the Request and Response object in `this.request` and `this.response`, respectively, although most methods and accessors are directly available from the Context.

The most important property is `this.body`, which sets the response body. The response status is automatically set to 200 when the response body is set. You may override this by setting `this.status` manually.

Another very useful syntactic sugar is `this.throw`, which allows you to return an error response by simply calling `this.throw(400)`, or if you want to override the standard HTTP error message, you may pass a second argument with the error message. We will get to Koa.js slick error handling later in this chapter.

Now that we've got the basics down, let's start building a Hacker News API!

The link model

The following code describes the straightforward link document model in `src/models/links.js`:

```
var mongoose = require('mongoose');

var schema = new mongoose.Schema({
  title: { type: String, require: true },
  URL: { type: String, require: true },
  upvotes: { type: Number, require: true, 'default': 0 },
  timestamp: { type: Date, require: true, 'default': Date.now }
});

schema.statics.upvote = function *(linkId) {
  return yield this.findByIdAndUpdate(linkId, {
    $inc: {
      upvotes: 1
    }
  })
}
```

```
        }) .exec();
    };

var Links = mongoose.model('links', schema);
module.exports = Links;
```

Note that this is pretty much identical to how you would define a model in Express, with one exception: the `upvotes` static method. Since `findByIdAndUpdate` is an asynchronous I/O operation, we need to make sure that we `yield` to it, so as to make sure we wait for this operation to complete, before we continue the execution.

Earlier we noted that not only generator functions can be yielded to; it also accepts Promises, which is awesome, because they are quite ubiquitous. Using Mongoose, for example, we can turn Mongoose query instances into Promises by calling the `exec()` method.

The link routes

With the link model in place, let's set up some routes in `src/routes/links.js`:

```
var model = require('../models/links');

module.exports = function(app) {
    app.get('/links', function *(next) {
        var links = yield model.find({}).sort({upvotes: 'desc'}).exec();
        this.body = links;
    });

    app.post('/links', function *(next) {
        var link = yield model.create({
            title: this.request.body.title,
            URL: this.request.body.URL
        });
        this.body = link;
    });

    app.delete('/links/:id', function *(next) {
        var link = yield model.remove({ _id: this.params.id }).exec();
        this.body = link;
    });
}
```

```
app.put('/links/:id/upvote', function *(next) {
  var link = yield model.upvote(this.params.id);
  this.body = link;
});
```

This should start to look familiar. Instead of function handlers with the signature (`req, res`) that we are used to in Express, we simply use middleware generator functions and set the response body in `this.body`.

Tying it together

Now that we have our model and routes defined perform the following steps:

1. Let's tie it together in a Koa.js application in `src/app.js`:

```
var koa = require('koa'),
    app = koa(),
    bodyParser = require('koa-body-parser'),
    router = require('koa-router');

// Connect to DB
require('./db');

app.use(bodyParser());
app.use(router(app));
require('./routes/links')(app);

module.exports = app;
```

 Note that we use `koa-body-parser` to parse the request body in `this.request.body` and `koa-router`, which allows you to define Express style routes, the kind you saw earlier.

2. Next, we connect to the database, which isn't different from the previous chapters, so we will omit the code here.

3. Finally, we define the Koa app, mount the middleware, and load the routes. Then, in the root folder, we have `/app.js` as given in the following:

```
var app = require('./src/app.js');
app.listen(3000);
console.log('Koa app listening on port 3000');
```

This just loads the app and starts an HTTP server, which listens on port 3000. Now to start the server, make sure you use the `--harmony-generators` flag. You should now have a working Koa API to power a Hacker News-like website!

Validation and error handling

Error-handling is one of the fortés of Koa.js. Using generator functions we don't need to deal with error handling in every level of the callbacks, avoiding the use of `(err, res)` signature callbacks popularized by Node.js. We don't even need to use the `.error` or `.catch` methods known to Promises. We can use plain old `try/catch` that ships with JavaScript out of the box.

The implication of this is that we can now have the following centralized error handling middleware:

```
var logger = console;

module.exports = function *(next) {
  try {
    yield next;
  } catch (err) {
    this.status = err.status || 500;
    this.body = err.message;
    this.app.emit('error', err, this);
  }
};
```

When we include this as one of the first middlewares on the Koa stack, it will basically wrap the entire stack, which is yielded to downstream, in a giant `try/catch` clause. Now we don't need to worry about exceptions being thrown into the ether. In fact, you are now encouraged to throw common JavaScript errors, knowing that this middleware will gracefully unpack it for you, and present it to the client.

Now this may not always be exactly what you want though. For instance, if you try to upvote an ID that is not a valid BSON format, Mongoose will throw `CastError` with the message `Cast to ObjectId failed for value xxx at path _id`. While informative for you, it is pretty dirty for the client. So here's how you can override the error by returning a 400 error with a nice, clean message:

```
app.put('/links/:id/upvote', function *(next) {
  var link;
  try {
    link = yield model.upvote(this.params.id);
  } catch (err) {
    if (err.name === 'CastError') {
      this.throw(404, 'link can not be found');
    }
  }

  // Check that a link document is returned
  this.assert(link, 404, 'link not found');

  this.body = link;
});
```

We basically catch the error where it happens, as opposed to let it bubble up all the way to the error handler. While we could throw a JavaScript error object with the `status` and `message` fields set to pass it along to the `errorHandler` middleware, we can also handle it here directly with the `this.throw` helper of the Context object.

Now if you pass a valid BSON ID, but the link does not exist, Mongoose will not throw an error. Therefore, you still have to check whether the value of `link` is not `undefined`. Here is yet another gorgeous helper of the Context object: `this.assert`. It basically asserts whether a condition is met, and if not, it will return a 400 error with the message `link not found`, as passed in the second and third argument.

Here are a few more validations to the submission of links:

```
app.post('/links', function *(next) {
  this.assert(typeof this.request.body.title === 'string', 400,
  'title is required');
  this.assert(this.request.body.title.length > 0, 400, 'title is
  required');
```

```
    this.assert(utils.isValidURL(this.request.body.URL), 400, 'URL  
is invalid');

    // If the above assertion fails, the following code won't be  
executed.
    var link = yield model.create({
      title: this.request.body.title,
      URL: this.request.body.URL
    });
    this.body = link;
  });
}
```

We ensure that a title is being passed, as well as a valid URL, for which we use the following RegEx util:

```
module.exports = {
  isValidURL: function(url) {
    return /(ftp|http|https):\/\/((\w+:{0,1}\w*@\)?(\s+)(:[0-  
9]+)?(\//|\//([\w#!:.?+=&%@!-\-/]))?/;
  }
};
```

Now there are still ways to refactor the validation checks into modular middleware; similar to what we did in *Chapter 3, Multiplayer Game API – Connect* this is left as an exercise to the reader.

Update route

A CRUD API is not complete with the update route! If you are a Hacker News frequenter, you'll know that titles of the submissions can change (but not the URL). This route should be straightforward, but there is one caveat! Yes, you could use `findByIdAndUpdate`, which is used by `upvote`, but what if you wanted to use Mongoose's instance method `.save()`?

Well, it does not return a Promise, so therefore we cannot `yield` to it. In fact, at the time of writing, there is still an open issue about this. Using `save()`, we can only use the traditional callback pattern. However, remember the rule – do not mix generator functions with callbacks!

So now what? Well, it will be quite common for certain node modules to be only available in the callback format. While most common modules are ported to a Koa version, you can still use Express packages; you just have to `thunkify` them. In fact, you could turn any callback style function into a thunk.

```
npm install --save thunkify
```

Now here's how to turn a function that accepts a callback into a yieldable thunk:

```
var thunk = require('thunkify');

...

// Thunkify save method
Links.prototype.saveThunk = thunk(Links.prototype.save);
```

Adding the preceding code to `model/links.js`, we can now do the following in the update route:

```
app.put('/links/:id', function *(next) {
  this.assert((this.request.body.title || '').length > 0, 400,
  'title is required');

  var link;
  try {
    link = yield model.findById(this.params.id).exec();
  } catch (err) {
    if (err.name === 'CastError') {
      this.throw(400, 'invalid link id');
    }
  }

  // Check that a link document is returned
  this.assert(link, 400, 'link not found');

  link.title = this.request.body.title;
  link = yield link.saveThunk() [0];
  this.body = link;
}) ;
```

Notice the use of `saveThunk()` near the bottom. It is basically a `thunkified` version of the original `save()` method. This means that an error that would originally be passed as the first argument in the callback is now thrown as an Error. We can afford not to wrap it in a `try/catch` block because the `errorHandler` middleware will catch it and throw a `500` error, which would be appropriate in this case.

Also, note how the thunk returns an array. This is because the original callback has an arity of `3`. The first argument is the error, the second argument is the new document, while the third argument is the number of affected documents. The array returned by the thunk contains the latter two values. If the arity of the callback was `2`, it would've just returned the value; something to keep in mind.

Let's perform some tests

In this chapter we omitted the disciplined TDD approach, since it has been covered multiple times in previous chapters. However, testing is slightly different in Koa.js, so let's highlight some of those differences.

We can still use `supertest` in the neat way that we did before, with one slight adjustment as follows:

```
var app = require('../src/app').callback();
```

We need to call the `.callback()` method to return an object that we can pass to `supertest`. In fact, the returned object can even be mounted on top of an Express app.

Testing the routes to submit links is pretty straightforward:

```
var app = require('../src/app').callback(),
    Links = require('../src/models/links');

describe('Submit a link', function() {

  before(function(done) {
    Links.remove({}, function(err) {
      done();
    });
  });

  it('should add a link', function() {
    var link = {
      title: 'The Zen of Python',
      href: 'http://www.python.org/peps/pep-0020.html'
    };

    return request(app)
      .post('/links')
      .send(link)
      .expect(201)
      .end(function(res) {
        var body = JSON.parse(res.text);
        expect(body).to.contain(link);
      });
  });
});
```

```
it('should successfully submit a link', function (done) {
  request(app).post('/links')
    .send({title: 'google', URL: 'http://google.com'})
    .expect(200, done);
});
```

At the start of this test suite, we clear the collection in the DB and submit a link using a post request. Nothing special here; note that we use Mocha's default callbacks for the asynchronous requests, and not co-mocha.

Let's submit a few more links, and check that they are indeed stored in the DB:

```
it('should successfully submit another link', function (done) {
  request(app).post('/links')
    .send({title: 'Axiom Zen', URL: 'http://axiomzen.co'})
    .expect(200, done);
});

it('should successfully submit a third link', function (done) {
  request(app).post('/links')
    .send({title: 'Hacker News', URL:
'http://news.ycombinator.com'})
    .expect(200, done);
});

// To be used in next test
var linkIDs = [];
it('should list all links', function (done) {
  request(app).get('/links')
    .expect(200)
    .end(function(err, res) {
      var body = res.body;
      expect(body).to.have.length(3);

      // Store Link ids for next test
      for(var i = 0; i < body.length; i++) {
        linkIDs.push(body[i]._id);
      }
      done();
    });
});
```

Notice that we store link IDs in an array for the next test case to demonstrate the final, most awesome bonus feature of Koa.js, parallel asynchronous requests, out of the box!

Parallel requests

The backend of Hacker News should be able to deal with the race condition, that is, it should handle hundreds of concurrent upvote requests without losing data (recall *Chapter 4, MMO Word Game* on race conditions). So let's write a test case that simulates parallel requests.

Traditionally, you would immediately think of using the extremely powerful and popular `async` library, which has a lot of very useful tools to deal with complex asynchronous execution flows. One of the most useful tools that `async` offers is `async.parallel`, with which you can make asynchronous requests in parallel. It is used to be the go-to solution for parallel requests, but now Koa offers something out of the box and with a much cleaner syntax!

Recall that `co` is actually what gives Koa the power of generator functions, so refer to the readme page of the `co` project to read more about all the patterns that it has to offer.

So far we yielded to generator functions, Promises, and thunks. However, that is not all. You can also yield to an array of the preceding which would execute them in parallel! Here's how:

```
// Add to top of file
require('co-mocha');
var corequest = require('co-supertest');

...
it('should upvote all links in parallel', function *() {
  var res = yield linkIDs.map(function(id) {
    return corequest(app)
      .put('/links/' + id + '/upvote')
      .end()
  });
});
```

```
// Assert that all Links have been upvoted
for(var i = 0; i < res.length; i++) {
    expect(res[i].body.upvotes).to.equal(1);
}

});
```

Firstly, notice how we use a generator function, so be sure that you have `require('co-mocha')` on top of your test file.

Secondly, `supertest` does not return a thunk or a promise, which we can yield to, so we require `co-supertest` for this test case:

```
npm install co-supertest --save-dev
```

Thirdly, we build an array of requests to be executed later. We are basically pushing thunks into an array; they could be promises too. Now when we yield the array, it will execute all requests in parallel, and return an array of all the response objects!

Quite mind blowing if you're used to `async.parallel` for these things, right?

Rendering HTML pages

At this point, we have a simple Koa API that has all the basic functionalities quite well tested. Let's add a simple view layer on top to show how you can serve static files from a Koa app as well. So if the app receives a request from a browser for HTML content, we'll serve a functional web page, where we can see the links submitted, submit a link, as well as upvote a link.

Let's pause here for a quick real-developer-life anecdote to implement the preceding. The tendency for modularity is an empowering force of the open source community. A modern day developer has access to a plethora of well-tested modules. Oftentimes, the majority of the developer's work is simply to compose an app of several such modules. We learn of these modules from prior experience, books, news websites, social media, and so on. So how do we go about choosing the right tools instead of reinventing the wheel?

It is always recommended to do a simple search to see whether a module is already available. In this case, we are interested in rendering views with Koa.js, so let's try the search term `koa-render` on www.npmjs.com. Two popular packages come up that seem to quite fit our needs, as shown in the following screenshot:

The screenshot shows the npmjs.com search results for the package `koa-views`. The page header includes the npm logo, a search bar, and a sign-up/login link. The main content area displays the package details for `koa-views`, which is described as "Template rendering middleware for koa, supporting many template engines". It shows the npm version (v2.1.2), dependencies (out-of-date), and license (MIT). Below this, there is a code example for installation:

```
$ npm install koa-views
```

The "Example" section contains the following code snippet:

```
// Must be used before any router is used
app.use('views', {
  map: {
    html: underscore
  }
});

app.use(function* (next) {
  this.locals = {
    session: this.session,
    ...
  };
});
```

On the right side of the page, there are sections for "Stats" (74 downloads in the last day, 228 downloads in the last week, 1,173 downloads in the last month), "One open issue on GitHub", "2 open pull requests on GitHub", and "Keywords" (koa, render, views, app-wide, templating, templates).

The `koa-views` is a template rendering middleware for Koa, supporting many template engines. Sounds promising! `koa-render` adds a `render()` method to Koa that allows you to render almost any templating engine. Not bad either. As shown in the following screenshot:

The screenshot shows the npmjs.com search results for the package `koa-render`. The page header includes the npm logo, a search bar, and a sign-up/login link. The main content area displays the package details for `koa-render`, which is described as "Add a `render()` method to koa that allows you to render almost any templating engine". It shows the npm version (v0.2.1), dependencies (out-of-date), and license (MIT). Below this, there is a code example for installation:

```
$ npm i --save koa-render
```

The "Usage" section contains the following code snippet:

```
views(path, [ext, map])
```

- `ext`: define the default extension name. Defaults to `html`.
- `map`: custom mapping.

On the right side of the page, there are sections for "Stats" (149 downloads in the last day, 1,034 downloads in the last week, 1,454 downloads in the last month), "Have an issue? File it.", and "Keywords" (koa, render, views, app-wide).

One of the things we can look at to guide our choice is the number of downloads; both packages have a decent amount of downloads, which shows some credibility. The `koa-views` has about 5 times more downloads than `koa-render` per month. While these badges are a minor touch, it does show that the author cared enough and is likely to support it. The number of recent commits is also a good indicator that can be found on the GitHub page for the project, the number issues that have been resolved, and so on.

At the time of writing, both projects' GitHub links redirect to `koa-views`, which is unexpected, but good for us! Looking at the GitHub account of the author of `koa-render`, we cannot find the project anymore, so it's safe to assume it was discontinued; avoid it! When you can, try to avoid using non-maintainable packages as it might pose a threat given the fact that Node.js (and io.js) are rapidly evolving ecosystems.

Back to rendering HTML pages, Koa, unlike Express, has no pre-baked opinion about the rendering of views. However, it does provide us with some mechanisms for content negotiation, some of which we can use to enhance and reuse the routes we already have for our API. Let's see what our `/links` handler will look like:

```
app.get('/links', function *(next) {
  var links = yield model.find({}).sort({upvotes: 'desc'}).exec();
  if( this.accepts('text/html') ){
    yield this.render('index', {links: links});
  } else {
    this.body = links;
  }
}
```

Our use case is rather simple; we either serve JSON or HTML. When the request header `accepts` is set to `text/html`, something browsers set automatically, we'll render the HTML. For the rendering of dynamic jade views to work as expected, we must not forget to include the `koa-views` middleware in `app.js` somewhere before the router middleware:

```
var views = require('koa-views');

...
app.use(views('../views', {default: 'jade'}));
```

The middleware points to a folder with a relative path that will contain the templates. Right now, we just need a single template `views/index.jade`:

```
doctype html
html(lang="en")
head
  title Koa News
body
  h1 Koa News
  div
    each link in links
      .link-row
        a(href='#', onclick="upvote('#{link._id}', $(this))") ^
          span &nbsp;
        a(href=link.URL)= link.title
        .count= link.upvotes
        | votes
  h2 Submit your link:
  form(action='/links', method='post')
    label Title:
    input(name='title', placeholder="Title")
    br
    label URL:
    input(name='URL', placeholder="https://")
    br
    br
    button.submit-btn Submit
script(src="https://code.jquery.com/jquery-2.1.3.min.js")
script.
  var upvote = function(id, elem) {
    $.ajax({url:'links/'+id+'/upvote', type:'put' })
    .done(function(data) {
      elem.siblings('.count').text(data.upvotes + ' votes');
    })
  }
}
```

It's a jade file similar to the ones presented before in this book. It loops over every link loaded at the controller, which has a single action to upvote. Links are displayed in the descending order of votes, which only happens when the page is reloaded. There is also a simple form that allows the user to submit new links.

We chose to load jQuery from a CDN simply in order to make the `PUT` request for upvotes. Notice that our use of inline JavaScript as well as adding a click event using the `onclick` element is highly discouraged, other than to make this example simple to digest.

Now if you have your app running and you go to `localhost:3000/links`, here's the result:



So that's a start from a functional standpoint! Clearly not good enough if we want to add more frontend JavaScript and CSS styling to it; we still need to be able to serve static files.

Serving static assets

Although usually you'd be incentivized to create a separate server for your assets, let's keep things simple and dive straight to the goal. We want to serve any files from a certain folder to a certain base path. For that purpose, we'll need two small middlewares, respectively, `koa-static` and `koa-mount`. In `src/app.js`, we add the following:

```
var serve = require('koa-static');
var mount = require('koa-mount');

// ...

app.use(mount('/public', serve('./public')));
```

The function `mount()` will namespace the request for each middleware that follows, in this particular case being combined with `serve`, which will serve any file inside the `public/` directory. If we decide not to mount to any particular URL, serving files would still work; it just won't have a nice namespace.

Now all you need to do is create a `public/` directory in the root folder with `filepublic/main.css` and it will able to serve a stylesheet.

This method allows to serve all static files you'd expect; CSS, JavaScript, images, and even videos.

To take it even further, there are many build tools and best practices for front-end assets, including ways to set up asset pipelines with Grunt, Gulp, Browserify, SASS, CoffeeScript, and many others tools. Not to mention front-end frameworks such as Angular, Ember, React, and so on. This is only the beginning.

Hope you enjoyed the introduction to Koa.js!

Summary

We built an API with which you can now host your own Hacker News of X! Obviously, we're still missing the sort and decay algorithm, as well as comments, but since you reached this far, it should be an easy exercise for you.

The purpose of this chapter was really to give you a taste of the neat features of Koa.js, and demonstrate the use of the generator function pattern, which will be available in ECMAScript 6. If you like being on the bleeding edge, and enjoy the generator syntax, it is definitely a good alternative to Express.js.

Connect 4 – Game Logic

In *Chapter 3, Multiplayer Game API – Connect* we built a multiplayer game API for the game Connect 4 in which we focused on the general mechanics of creating a game, joining a game, and playing it. This Appendix shows the accompanying game logic that we omitted in *Chapter 3, Multiplayer Game API* main text.

```
src/lib/connect4.js

/*
  Connect 4 Game logic

  Written for Blueprints: Express.js, Chapter 3

*/
var MIN_ROWS = 6,
    MIN_COLUMNS = 7,
    players = ['x', 'o'];

// Initializes and returns the board as a 2D array.
// Arguments accepted are int rows, int columns,
// Default values: rows = 6, columns = 7
exports.initializeBoard = function initializeBoard(rows, columns) {
  var board = [];
  rows = rows || MIN_ROWS;
  columns = columns || MIN_COLUMNS;

  // Default values is minimum size of the game
  if (rows < MIN_ROWS) {
    rows = MIN_ROWS;
  }
}
```

Connect 4 - Game Logic

```
if (columns < MIN_COLUMNS) {
    columns = MIN_COLUMNS;
}

// Generate board
for (var i = 0; i < rows; i++) {
    var row = [];
    for (var j = 0; j < columns; j++) {
        row.push(' ');
    }
    board.push(row);
}
return board;
};

// Used to draw the board to console, mainly for debugging
exports.drawBoard = function drawBoard(board) {
    var numCols = board[0].length,
        numRows = board.length;
    consolePrint(' ');
    for (var i = 1; i <= numCols; i++) {
        consolePrint(i+'');
        consolePrint(' ');
    }
    consolePrint('\n');
    for (var j = 0; j < numCols*2+1; j++) {
        consolePrint('-');
    }
    consolePrint('\n');
    for (i = 0; i < numRows; i++) {
        consolePrint('|');
        for (j = 0; j < numCols; j++) {
            consolePrint(board[i][j] + '');
            consolePrint('|');
        }
        consolePrint('\n');
        for (j = 0; j < numCols*2+1; j++) {
            consolePrint('-');
        }
        consolePrint('\n');
    }
};
```

```
// Make a move for the specified player, at the indicated column
for this board
// Player should be the player number, 1 or 2
exports.makeMove = function makeMove(player, column, board) {
    if (player !== 1 && player !== 2) {
        return false;
    }
    var p = players[player-1];
    for (var i = board.length-1; i >= 0; i--) {
        if (board[i][column-1] === ' ') {
            board[i][column-1] = p;
            return board;
        }
    }
    return false;
}

// Check for victory on behalf of the player on this board,
// starting at location (row, column)
// Player should be the player number, 1 or 2
exports.checkForVictory = function checkForVictory(player,
lastMoveColumn, board) {
    if (player !== 1 && player !== 2) {
        return false;
    }
    var p = players[player-1],
        directions = [[1,0],[1,1],[0,1],[1,-1]],
        rows = board.length,
        columns = board[0].length,
        lastMoveRow;
    lastMoveColumn--;
    // Get the lastMoveRow based on the lastMoveColumn
    for (var r = 0; r < rows; r++) {
        if(board[r][lastMoveColumn] !== ' ') {
            lastMoveRow = r;
            break;
        }
    }

    for (var i = 0; i<directions.length; i++) {
        var matches = 0;
        // Check in the 'positive' direction
```

Connect 4 - Game Logic

```
for (var j = 1; j < Math.max(rows,columns); j++) {
    if (board[lastMoveRow + j*directions[i][1]] && p ===
board[lastMoveRow + j*directions[i][1]][lastMoveColumn +
j*directions[i][0]]) {
        matches++;
    } else {
        break;
    }
}
// Check in the 'negative' direction
for (j = 1; j < Math.max(rows,columns); j++) {
    if (board[lastMoveRow - j*directions[i][1]] && p ===
board[lastMoveRow - j*directions[i][1]][lastMoveColumn -
j*directions[i][0]]) {
        matches++;
    } else {
        break;
    }
}
// If there are greater than three matches, then that means
there are at least 4 in a row
if (matches >= 3) {
    return true;
}
return false;
};

function consolePrint(msg) {
    process.stdout.write(msg);
}
And the accompanying unit tests:
var expect = require('chai').expect;

var connect4 = require('../src/lib/connect4');

describe('Connect 4 Game Logic | ', function() {
    describe('#Create a board ', function() {
        var board = connect4.initializeBoard();

        it('should return game boards of the defaults length when too
small', function(done) {
            var board2 = connect4.initializeBoard(3,3),
                board3 = connect4.initializeBoard(5),
```

```
board4 = connect4.initializeBoard(3,10),
board5 = connect4.initializeBoard(10,3);

// Make sure the board is a 2D array
expect(board2).to.be.an('array');
expect(board2.length).to.equal(board.length);
expect(board2[0].length).to.equal(board[0].length);
for(var i = 0; i < board2.length; i++){
    expect(board2[i]).to.be.an('array');
}

// Make sure the board is a 2D array
expect(board3).to.be.an('array');
expect(board3.length).to.equal(board.length);
expect(board3[0].length).to.equal(board[0].length);
for(var i = 0; i < board3.length; i++){
    expect(board3[i]).to.be.an('array');
}
// Board initialized with 3 rows, but should default to 6
expect(board4).to.be.an('array');
expect(board4.length).to.equal(board.length);
for(var i = 0; i < board4.length; i++){
    expect(board4[i]).to.be.an('array');
}
// Board initialized with 3 columns, but should default to 7
expect(board5).to.be.an('array');
expect(board5[0].length).to.equal(board[0].length);
for(var i = 0; i < board5.length; i++){
    expect(board5[i]).to.be.an('array');
}

done();
});

it('should only allow pieces to be placed #row amount of times', function(done) {
    board = connect4.initializeBoard();
    for (var i = 0; i < board.length; i++) {
        board = connect4.makeMove(1, 1, board);
    }
    // Column should be full
```

Connect 4 - Game Logic

```
    expect(connect4.makeMove(1, 1,
board)).to.be.an('boolean').and.equal(false);
    // Out of bounds
    expect(connect4.makeMove(1, 0,
board)).to.be.an('boolean').and.equal(false);
    expect(connect4.makeMove(1, board[0].length+1,
board)).to.be.an('boolean').and.equal(false);

    done();
}

it('should return victory if there are 4 in a row',
function(done) {
    // Vertical Win
    board = connect4.initializeBoard();
    for (var i = 0; i < 3; i++) {
        board = connect4.makeMove(1, 1, board);
        expect(connect4.checkForVictory(1, 1,
board)).to.equal(false);
    }
    board = connect4.makeMove(1, 1, board);
    expect(connect4.checkForVictory(1, 1,
board)).to.equal(true);

    // Horizontal Win
    board = connect4.initializeBoard();
    for (var i = 1; i < 4; i++) {
        board = connect4.makeMove(1, i, board);
        expect(connect4.checkForVictory(1, 1,
board)).to.equal(false);
    }
    board = connect4.makeMove(1, 4, board);
    expect(connect4.checkForVictory(1, 4,
board)).to.equal(true);

    // Diagonal Win
    board = connect4.initializeBoard();
    for (var i = 1; i < 4; i++) {
        for (var j = 1; j <= i; j++){
            if (j==i){
                board = connect4.makeMove(1, i, board);
            } else {
```

```
        board = connect4.makeMove(2, i, board);
    }
    expect(connect4.checkForVictory(1, 1,
board)).to.equal(false);
}
for (var i = 0; i < 3; i++) {
    board = connect4.makeMove(2, 4, board);
    expect(connect4.checkForVictory(2, 4,
board)).to.equal(false);
}
board = connect4.makeMove(1, 4, board);
expect(connect4.checkForVictory(1, 4,
board)).to.equal(true);

done();
});
});
});
});
```


Index

A

actor
 creating, with POST 29
 removing, with DELETE 30-32
 retrieving, with GET 28
 updating, with PUT 30
API endpoints
 testing 18

B

Bluebird
 URL 87
builds
 automating 19

C

catch method 83
Chrome Developer Tools
 used, for debugging Socket.IO 103-105
code structure 108
Connect 4
 about 45
 new game, creating 49
CRUD operations
 about 27, 28
 actor, creating with POST 29
 actor, removing with DELETE 30-32
 actor, retrieving with GET 28
 actor, updating with PUT 30
 DELETE 27
 GET 27
 POST 27
 PUT 27

D

data
 exception handling 113
 persisting 110-112
database
 validating 36
DELETE
 actor, removing with 30-32
deploys
 automating 19
distance optimization 124, 125

E

e-mail follow up 126-134
error handling 148-150
exception handling 113
Express
 about 1
 Hello World! 2, 3
 Jade templating 3, 4
 setting, for static site 1
Express middleware 7

F

Factory pattern 108
folder structure 25, 26
functions
 extracting, to reusable middleware 36-39

G

game
 creating 50-54
 game state, obtaining 56

input validation 54, 55
joining 58-61
playing 61-71
testing, for tie 71-73
game logic, Connect 4 163
game state
modeling, with Mongoose 46-48
generator syntax
about 140-143
benefits 139
Context, versus req 145
Context, versus res 145
middleware 143, 144
GET
actor, retrieving with 28

H

Heroku
about 22
URL 22
HTML pages
rendering 155-159

J

Jade
URL 16

K

Koa.js
about 139, 140, 147, 148
URL 139

L

link model 145, 146
link routes 146, 147
local user authentication
about 4
Express middleware 7
passport, setting up 8, 9
user object modeling 5, 6
users, authenticating 11
users, registering 10

M

Mocha 17
moment.js
URL 115
Mongoose
object modeling, using with 32
URL 7, 32
multiplayer game API
building 45, 46

N

naive pairing
about 113-115
tests 118
node-cron
periodical tasks, used with 135-137
URL 135
Node.js
about 1
deploying 22, 23
URL 2

O

OAuth, with passport
about 12
adding, to user model 12
API tokens, obtaining 13
third party registration strategies,
installing 14, 15
object modeling
with Mongoose 32

P

parallel requests 154, 155
params object 28
periodical tasks
with node-cron 135-137
POST
actor, creating with 29
profile pages
about 15
profile templates 16
URL params 15

Promise

- about 79-82
- active users, displaying 87
- benefits 80
- catch method 83
- duplicates, preventing 84-86
- input, validating 89, 90
- multiple Promises, chaining 83, 84
- race conditions, dealing with 91-93
- race conditions, testing 93, 94
- subdocuments 88
- then method 83
- user exit 86

PUT

- actor, updating with 30

R

res parameter 28

- reusable middleware**
 - functions, extracting to 36-39

route

- defining 109, 110
- updating 150-152

S

schema

- user, joining 78
- user schema, designing 77

Socket.IO

- applications, launching 97
- debugging, with Chrome Developer Tools 103-105
- documentation 94
- updates, adding to clients 95-97
- updates, pushing to clients 95-97

Socket.IO applications

- launching 97
- testing, with Socket.IO client 98-102

Socket.IO client

- used, for testing Socket.IO applications 98-102

static assets

- serving 160

swig

- URL 131

T

testing

- about 16, 40-43
- API endpoints 18, 19

tests

- about 118
- HTML pages, rendering 155-159
- parallel requests 154, 155
- performing 152-154
- static assets, serving 160

then method 83

timekeeper

- URL 120

Travis CI 21

U

unique IDs

- generating 34, 35

user history 118-122

user object modeling 5, 6

V

validation 148-150

W

Word Chain Game

- about 75
- active users, tracking 77
- real-time application overview 76



Thank you for buying Express.js Blueprints

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

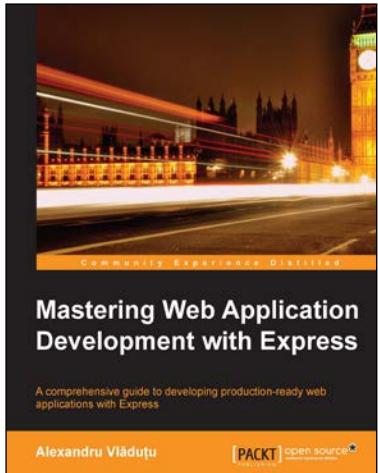


Express Web Application Development

ISBN: 978-1-84969-654-8 Paperback: 236 pages

Learn how to develop web applications with the Express framework from scratch

1. Exploring all aspects of web development using the Express framework.
2. Starts with the essentials.
3. Expert tips and advice covering all Express topics.



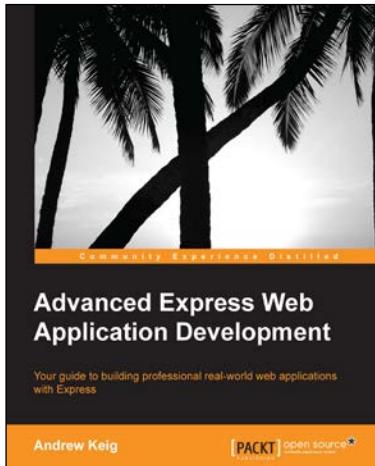
Mastering Web Application Development with Express

ISBN: 978-1-78398-108-3 Paperback: 358 pages

A comprehensive guide to developing production-ready web applications with Express

1. Create fast, secure and reliable production-ready web applications using Express.
2. Packed with the latest techniques for tackling real world issues.
3. Improve code quality and speed up development by using a variety of patterns and tools.

Please check www.PacktPub.com for information on our titles



Advanced Express Web Application Development

ISBN: 978-1-78328-249-4 Paperback: 148 pages

Your guide to building professional real-world web applications with Express

1. Learn how to build scalable, robust, and reliable web applications with Express using a test-first, feature-driven approach.
2. Full of practical tips and real world examples, and delivered in an easy-to-read format.
3. Explore and tackle the issues you encounter in commercially developing and deploying an Express application.



Learning Express Web Application Development [Video]

ISBN: 978-1-78398-988-1 Duration: 2:27 hours

Build powerful and modern web apps that run smoothly on the webserver with Express.js

1. Use Express.js and get the best out of JavaScript to build robust server based web apps.
2. Incorporate MongoDB, the blazingly fast document-based database into your applications.
3. Impress your colleagues with production ready code through test-driven development.

Please check www.PacktPub.com for information on our titles