CHAPTER 22

HackHall

The HackHall app is a true MVC application. It has the REST API server with a front-end client that is written in Backbone.js and Underscore. For the purpose of this chapter, we'll illustrate how to use Express.js with MongoDB via Mongoose ORM/ODM (object-relational mapping/object-document mapping) for the back-end REST API server. In addition, the project utilizes OAuth directly and via Passport, sessions, and Mocha for TDD. It's hosted on Heroku and is in active development (see the nearby Note).

Note The HackHall source code used in this chapter is available in the public GitHub repository (`https://github.com/azat-co/hackhall`) under the v3.1.0 release (`https://github.com/azat-co/hackhall/releases/tag/v3.1.0`, `https://github.com/azat-co/hackhall/tree/v3.1.0`, and `https://github.com/azat-co/hackhall/archive/v3.1.0.zip`). Future versions might differ from this chapter's example and could have more features.

This chapter is structured as follows:

- What is HackHall?
- Running HackHall
- Structure
- Package.json
- Express.js app
- Routes
- Mongoose models
- Mocha tests

What Is HackHall?

HackHall is an open source project for online communities. Its implementation at `http://hackhall.com` is a curated social network/membership community and collaboration tool for hackers, hipsters, designers, entrepreneurs, and pirates (just kidding). The HackHall community is akin to a combination of Reddit, Hacker News, and Facebook Groups with curation. You can apply to become a member at `http://hackhall.com`.

The HackHall project is in its early stages, roughly beta. We plan to extend the code base in the future and bring in more people to share skills, wisdom, and passion for programming. You can watch a quick demo video of HackHall.com at `http://youtu.be/N1UILNqeW4k`.

In this chapter, we'll cover the v3.1.0 release, which has the following features:

- OAuth 1.0 with the `oauth` module (`https://www.npmjs.org/package/oauth`) and AngelList API (`https://angel.co/api`)
- E-mail and password authentication
- Password hashing
- Mongoose models and schemas
- Express.js structure with routes in modules
- JSON REST API
- Express.js error handling
- Front-end client Backbone.js app (for more info on Backbone.js, download, or read online, my *Rapid Prototyping with JS* tutorials, at `http://rapidprototypingwithjs.com/`)
- Environment variables with Foreman's `.env`
- TDD tests with Mocha
- Basic Makefile setup
- SendGrid e-mail notifications
- GitHub login

Running HackHall

To get the source code for HackHall, you can navigate to the `hackhall` folder or clone it from GitHub:

```
$ git clone https://github.com/azat-co/hackhall.git
$ git checkout v3.1.0
$ npm install
```

If you plan to test an AngelList, or GitHub integrations (optional), then you should sign up for their API keys as a developer. After you do so, you need to pass the values via environment variables to the app. HackHall is using a Heroku and Foreman (`http://ddollar.github.io/foreman`) setup approach (the `.env` file) for these sensitive API keys. The Foreman gem is a command-line tool that manages Procfile-based applications. The Heroku toolbelt includes it. To store keys in environment variables, simply add an `.env` file like this (replace the values following = with your own values):

```
ANGELLIST_CLIENT_ID=254C0335-5F9A-4607-87C0
ANGELLIST_CLIENT_SECRET=99F5C1AC-C5F7-44E6-81A1-8DF4FC42B8D9
GITHUB_CLIENT_ID=9F5C1AC-C5F7-44E6
GITHUB_CLIENT_SECRET=9F5C1AC-C5F7-44E69F5C1AC-C5F7-44E6
GITHUB_CLIENT_ID_LOCAL=9F5C1AC-C5F7-44E1
GITHUB_CLIENT_SECRET_LOCAL=9F5C1AC-C5F7-44E69F5C1AC-C5F7-44E6
...
```

Note that there are no spaces before or after the equal sign (=).
After you have the .env file and values, use foreman with nodemon:

```
$ foreman run nodemon server
```

If you are confused about foreman or prefer not to install it, then you can create a shell file with your environment variables and use it to launch the server.
After you create an AngelList app and register it, you can obtain the AngelList API keys at https://angel.co/api. Similarly, for GitHub, you'll need to register as a developer to be able to create an app and get API keys. SendGrid works via the Heroku add-on, so you obtain the username and password from the Heroku web interface.
The following is how my .env looks for v3.1.0 (with the keys replaced with placeholders), in which I have two sets of GitHub keys, one for the local app and one for the production (hackhall.com) app, because the callback URL for each of them is different. The callback URL is set on GitHub when you register the apps.

```
ANGELLIST_CLIENT_ID=AAAAAAAAAAAAAA
ANGELLIST_CLIENT_SECRET=AAAAAAAAAAAAAA
GITHUB_CLIENT_ID=AAAAAAAAAAAAAA
GITHUB_CLIENT_SECRET=AAAAAAAAAAAAAA
GITHUB_CLIENT_ID_LOCAL=AAAAAAAAAAAAAA
GITHUB_CLIENT_SECRET_LOCAL=AAAAAAAAAAAAAA
SENDGRID_USERNAME=AAAAAAAAAAAAAA@heroku.com
SENDGRID_PASSWORD=AAAAAAAAAAAAAA
COOKIE_SECRET=AAAAAAAAAAAAAA
SESSION_SECRET=AAAAAAAAAAAAAA
ANGELLIST_CLIENT_ID_LOCAL=AAAAAAAAAAAAAA
ANGELLIST_CLIENT_SECRET_LOCAL= AAAAAAAAAAAAAA
EMAIL=AAAAAAAAAAAAAA
```

Cookie and session secrets are used to encrypt the cookie (browser) and session (store) data.
Putting sensitive information into the environment variables allowed me to make the *entire* HackHall source code available to the public. I also have one more variable that I set in the Heroku web interface for this app (you can sync .env to/from the cloud with the Heroku config [https://devcenter.heroku.com/articles/config-vars] or use the web interface). This variable is NODE_ENV=production. I use it when I need to determine the GitHub app to use (local vs. the main live one).
Download and install MongoDB, if you don't have it already. The databases and third-party libraries are outside the scope of this book. However, you can find enough materials online (see, e.g., http://webapplog.com) and in the previously referenced *Rapid Prototyping with JS*. Before you launch the application, I recommend running the seed-script.js file or seed.js file to populate your database with information, as described next.
To seed the database hackhall with a default admin user by running the seed-script.js MongoDB script, enter

```
$ mongo localhost:27017/hackhall seed-script.js
```

Feel free to modify seed-script.js to your liking (be aware that doing so erases all previous data!). For example, use your bcryptjs hashed password (skip to the seed.js instructions for automated hashing of seed data). You'll see an example of hashing later.
First, we clean the database:

```
db.dropDatabase();
```

Then, we define an object with user information:

```
var seedUser ={
  firstName: 'Azat',
  lastName: 'Mardan',
  displayName: 'Azat Mardan',
  password: 'hashed password',
  email: '1@1.com',
  role: 'admin',
  approved: true,
  admin: true
};
```

Finally, save the object to the database using the MongoDB shell method:

```
db.users.save(seedUser);
```

Whereas `seed-script.js` is a MongoDB shell script, `seed.js` is a mini Node.js application to seed the database. You can run the Node.js database-seeding program with:

```
$ node seed.js
```

The `seed.js` program is more comprehensive (it has password hashing!) than the MongoDB shell script `seed-script.js`. We begin by importing the modules:

```
var bcrypt = require('bcryptjs');
var async = require('async');
var mongo =  require ('mongodb');
var objectId = mongo.ObjectID;
```

Similar to in `seed-script.js`, we define the user objects, only this time the passwords are plain/unhashed:

```
seedUsers = [{...},{...}];
```

The `seedUsers` array object might look like this (add your own user objects!):

```
{
  firstName:   "test",
  lastName:    "Account",
  displayName: "test Account",
  password:    "hashend password",
  email:       "1@1.com",
  role:        "user",
  admin: false,
  _id: objectId("503cf4730e9f580200000003"),
  photoUrl: "https://s3.amazonaws.com/photos.angel.co/users/68026-medium_jpg?1344297998",
  headline: "Test user 1",
  approved: true
}
```

This is the asynchronous function that will hash our plain passwords:

```
var hashPassword = function (user, callback) {
```

The `bcryptjs` module stores salts inside of the hashed passwords, so there's no need to store salts separately; `10` is the hashing complexity (the higher the better):

```
  bcrypt.hash(user.password, 10, function(error, hash) {
    if (error) throw error;
    user.password = hash;
    callback(null, user);
  });
};
```

Here, we define variables that we'll use later:

```
var db;
var invites;
var users;
var posts;
```

We connect to MongoDB with the native driver:

```
var dbUrl = process.env.MONGOHQ_URL || 'mongodb://@127.0.0.1:27017/hackhall';
mongo.Db.connect(dbUrl, function(error, client){
  if (error) throw error;
    else {
        db=client;
```

Next, we assign collections to objects and clean all the users, just in case:

```
        invites = new mongo.Collection(db, "invites");
        users = new mongo.Collection(db, "users");
        posts = new mongo.Collection(db, "posts");
        invites.remove(function(){});
        users.remove(function(){});
```

You can uncomment this line if you want this script to remove the posts as well:

```
        // posts.remove();
        invites.insert({code:'smrules'}, function(){});
```

Insert a dummy post (feel free to be creative here):

```
        posts.insert({
          title:'test',
          text:'testbody',
          author: {
           name:seedUsers[0].displayName,
           id:seedUsers[0]._id
           }
        }, function(){});
```

We use an asynchronous function, because hashing might be slow (that's a good thing, because slower hashes are harder to hack with brute force):

```
    async.map(seedUsers, hashPassword, function(error, result){
      console.log(result);
      seedUsers = result;
      users.insert(seedUsers, function(){});
      db.close();
    });
  }
});
```

To start the MongoDB server, open a new terminal window and run:

```
$ mongod
```

After MongoDB is running on localhost with default port 27017, go back to the project folder and run `foreman` (this command reads from Procfile):

```
$ foreman start
```

Or, you can use `nodemon` (`http://nodemon.io`; GitHub: `https://github.com/remy/nodemon`) with a more explicit `foreman` command:

```
$ foreman run nodemon server
```

If you open your browser to `http://localhost:3000`, you should see a login screen similar to that shown in Figure 22-1 (for v3.1.0).

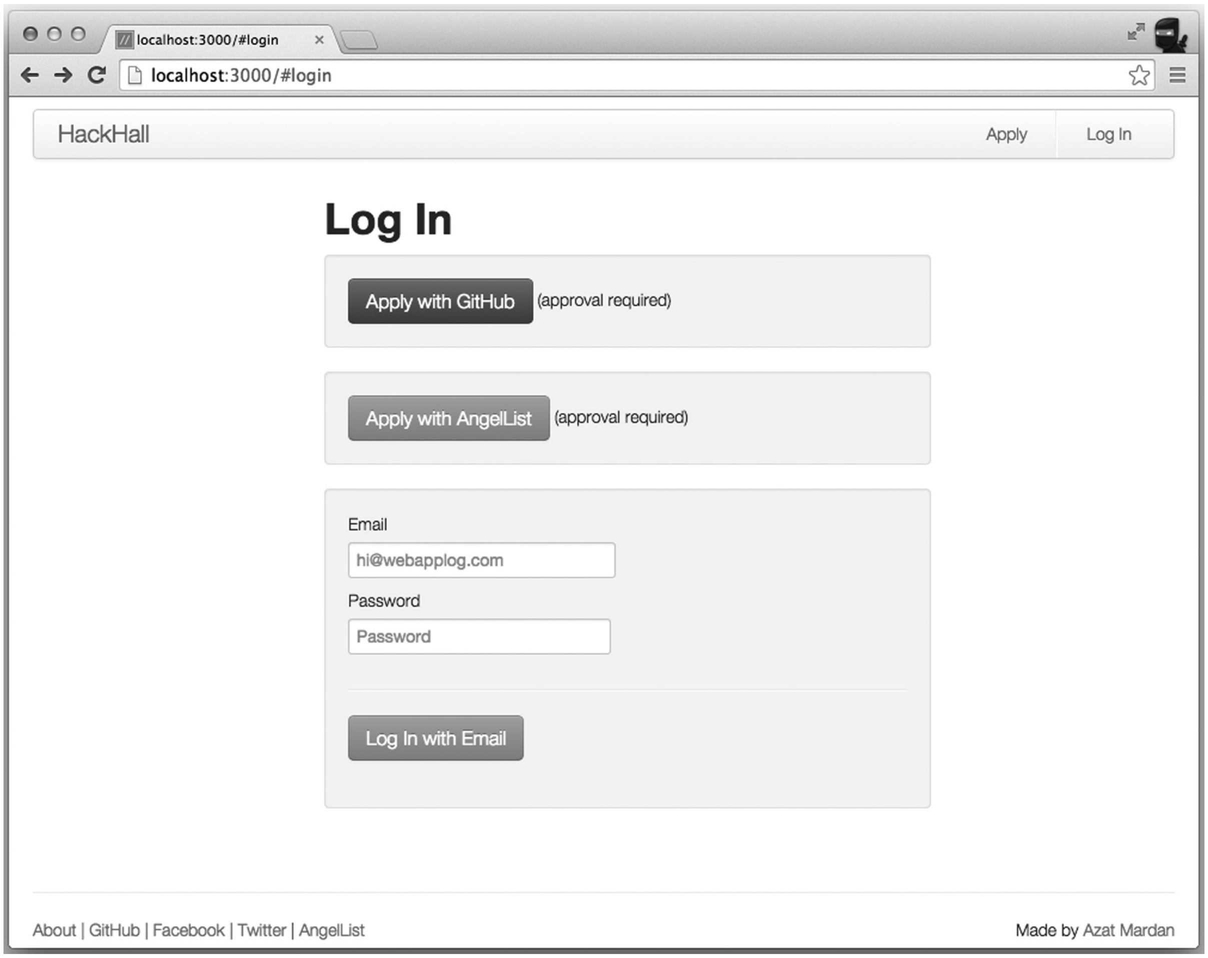


Figure 22-1. *HackHall v3.1.0 login page running locally*

Enter your username and password (the ones from your `seed.js` or `seed-script.js` file) to gain access. Use the unhashed (i.e., plain) version of the password.

After successful authentication, users are redirected to the Posts page, as shown Figure 22-2 (your data, such as post names, will be different; the “test” post is a byproduct of running Mocha tests).

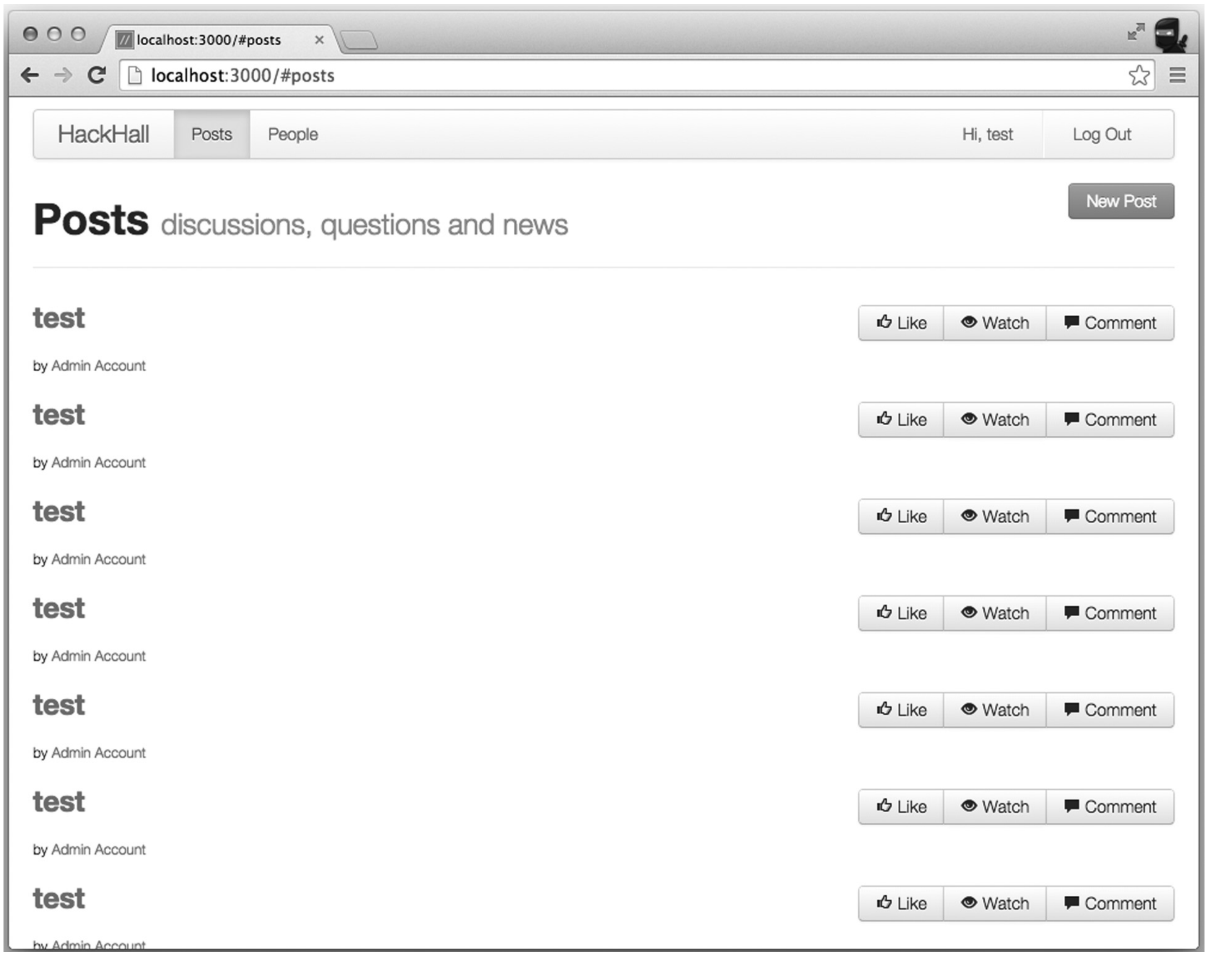
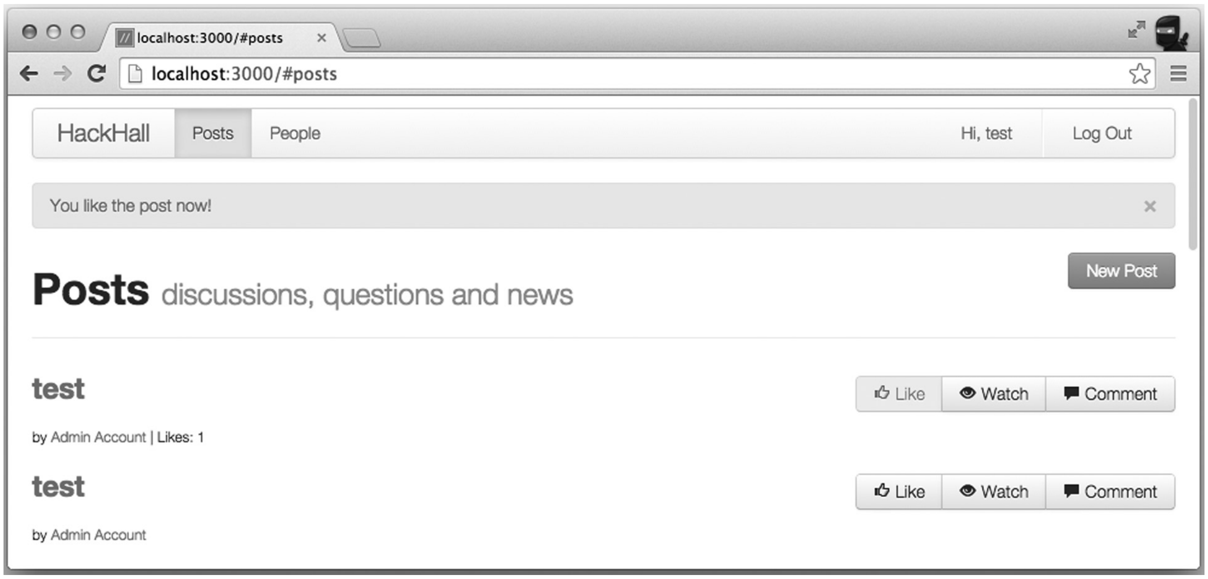


Figure 22-2. *HackHall Posts page*

If you click a “Like” button for a post, the “You like the post now!” message should be displayed and the Like counter should increase on that post, as shown in Figure 22-3. The same thing happens for the Watch buttons. Authors can edit and remove their own posts. Admins can edit and remove any posts. There are People and Profile pages, which you will see later in the chapter.



You like the post now!

Posts discussions, questions and news

test

by Admin Account | Likes: 1

test

by Admin Account

Figure 22-3. *HackHall Posts page with a liked post*

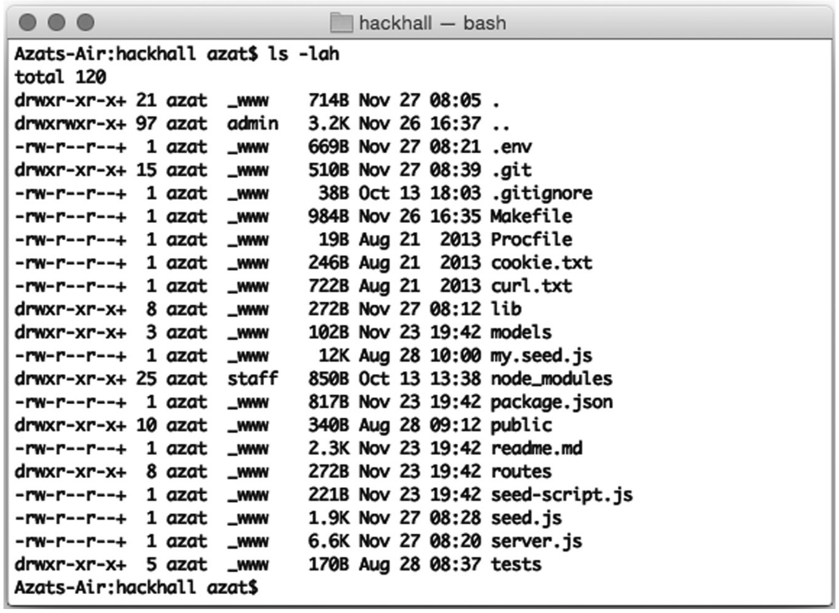
So, now you've seen how HackHall v3.1.0 might look right out of the box on your local machine. The following sections will walk you through some of the concepts and patterns that were used when implementing this application. This chapter is less detailed than previous chapters, because I'm assuming that you're already familiar with the topics covered in those chapters; repeating all the details would take too much space and would likely bore you.

Structure

Here is the structure of HackHall and a brief description of what each of the folders and files contains:

- `/api`: App-shared routes
- `/models`: Mongoose models
- `/public`: Backbone app, static files, such as front-end JavaScript, CSS, and HTML
- `/routes`: REST API routes
- `/tests`: Mocha tests
- `/lib`: Internal (in-house) libraries
- `.gitignore`: List of files that `git` should ignore
- `Makefile`: Makefile to run tests
- `Procfile`: Cedar stack file needed for Heroku deployment
- `package.json`: NPM dependencies and HackHall metadata
- `readme.md`: Description of project
- `server.js`: Main HackHall server file
- `.env`: Secret values that you don't want to share or leak to others

My project folder content is shown in Figure 22-4. The front-end application, which is written in Backbone.js with the Underscore template engine (HTML is rendered on the client side), is extensive and its coverage is beyond the scope of this book, because there are many alternatives to Backbone.js (with Angular.js being one of the most popular choices). You can always look up the source code of the browser app from the `public` folder: `https://github.com/azat-co/hackhall/tree/v3.1.0/public`.



```
Azats-Air:hackhall azat$ ls -lah
total 120
drwxr-xr-x+ 21 azat  _www    714B Nov 27 08:05 .
drwxrwxr-x+ 97 azat  admin   3.2K Nov 26 16:37 ..
-rw-r--r--+  1 azat  _www    669B Nov 27 08:21 .env
drwxr-xr-x+ 15 azat  _www    510B Nov 27 08:39 .git
-rw-r--r--+  1 azat  _www     38B Oct 13 18:03 .gitignore
-rw-r--r--+  1 azat  _www    984B Nov 26 16:35 Makefile
-rw-r--r--+  1 azat  _www     19B Aug 21  2013 Procfile
-rw-r--r--+  1 azat  _www    246B Aug 21  2013 cookie.txt
-rw-r--r--+  1 azat  _www    722B Aug 21  2013 curl.txt
drwxr-xr-x+  8 azat  _www    272B Nov 27 08:12 lib
drwxr-xr-x+  3 azat  _www    102B Nov 23 19:42 models
-rw-r--r--+  1 azat  _www     12K Aug 28 10:00 my.seed.js
drwxr-xr-x+ 25 azat  staff   850B Oct 13 13:38 node_modules
-rw-r--r--+  1 azat  _www    817B Nov 23 19:42 package.json
drwxr-xr-x+ 10 azat  _www    340B Aug 28 09:12 public
-rw-r--r--+  1 azat  _www    2.3K Nov 23 19:42 readme.md
drwxr-xr-x+  8 azat  _www    272B Nov 23 19:42 routes
-rw-r--r--+  1 azat  _www    221B Nov 23 19:42 seed-script.js
-rw-r--r--+  1 azat  _www    1.9K Nov 27 08:28 seed.js
-rw-r--r--+  1 azat  _www    6.6K Nov 27 08:20 server.js
drwxr-xr-x+  5 azat  _www    170B Aug 28 08:37 tests
Azats-Air:hackhall azat$
```

Figure 22-4. *Content of the HackHall base folder*

Package.json

As always, let's start with the `package.json` file and dependencies. The "new" libraries that we haven't used previously in this book are `passport` (OAuth integration), `sendgrid` (e-mail notifications), `mongoose` (MondoDB ORM/ODM), and `bcryptjs` (password hashing). Everything else should be familiar to you. We'll be using Express.js middleware modules and utilities (`async`, `mocha`).

This is what `package.json` looks like (use newer versions at your own discretion):

```
{
  "name": "hackhall",
  "version": "3.1.0",
  "private": true,
  "main": "server",
  "scripts": {
    "start": "node server",
    "test": "make test"
  },
  "dependencies": {
    "async": "0.9.0",
    "bcryptjs": "2.0.2",
    "body-parser": "1.6.6",
    "cookie-parser": "1.3.2",
    "csurf": "1.5.0",
    "errorhandler": "1.1.1",
    "express": "4.8.1",
    "express-session": "1.7.6",
    "method-override": "2.1.3",
    "mongodb": "1.4.9",
    "mongoose": "3.8.15",
    "mongoose-findorcreate": "0.1.2",
    "mongoskin": "1.4.4",
    "morgan": "1.2.3",
    "oauth": "0.9.12",
    "passport": "0.2.0",
    "passport-github": "0.1.5",
    "sendgrid": "1.2.0",
    "serve-favicon": "2.1.1"
  },
```

The `devDependencies` category is for modules that you don't need in production:

```
  "devDependencies": {
    "mocha": "1.21.4",
    "superagent": "0.18.2"
  },
  "engines": {
    "node": "0.10.x"
  }
}
```

Express.js App

Let's jump straight to the `server.js` file and take a look at how it's implemented. First, we declare dependencies:

```
var express = require('express'),
  routes = require('./routes'),
  http = require('http'),
  util = require('util'),
  path = require('path'),
  oauth = require('oauth'),
  querystring = require('querystring');
```

Next, we do the same for the Express.js middleware modules (no need for a separate `var`, except to show the difference in purpose of the modules):

```
var favicon = require('serve-favicon'),
  logger = require('morgan'),
  bodyParser = require('body-parser'),
  methodOverride = require('method-override'),
  cookieParser = require('cookie-parser'),
  session = require('express-session'),
  csrf = require('csurf');
```

Next, we have an internal e-mail library that uses SendGrid via a Heroku add-on:

```
var hs = require(path.join(__dirname, 'lib', 'hackhall-sendgrid'));
```

The log messages with different font colors are nice to have, but are, of course, optional. We accomplish this coloring with escape sequences in `lib/colors.js`:

```
var c = require(path.join(__dirname, 'lib', 'colors'));
require(path.join(__dirname, 'lib', 'env-vars'));
```

Passport (`http://passportjs.org`, `https://www.npmjs.org/package/passport`, `https://github.com/jaredhanson/passport`) is for GitHub OAuth. Using `passport` is a higher-level approach of implementing OAuth than using `oauth`:

```
var GitHubStrategy = require('passport-github').Strategy,
  passport = require('passport');
```

Then, we initialize the app and configure middleware. The `process.env.PORT` environment variable is populated by Heroku, and in the case of a local setup, falls back on `3000`. The rest of the configuration should be familiar to you from Chapter 4.

```
app.set('port', process.env.PORT || 3000  );
app.use(favicon(path.join(__dirname,'public','favicon.ico')));
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: true}));
app.use(methodOverride());
```

The values passed to `cookieParser` and session middleware are needed for authentication. Obviously, these secrets are supposed to be private:

```
app.use(cookieParser(process.env.COOKIE_SECRET));
app.use(session({
  secret: process.env.SESSION_SECRET,
  key: 'sid',
  cookie: {
    secret: true,
    expires: false
  },
  resave: true,
  saveUninitialized: true
}));
```

This is how we serve our front-end client Backbone.js app and other static files, such as CSS:

```
app.use(express.static(__dirname + '/public'));
```

Error handling is broken down into three functions, with `clientErrorHandler()` dedicated to AJAX/XHR requests from the Backbone.js app (responds with JSON). Right now, we only declare the functions. We'll apply them with `app.use()` later.

The first method, logErrors(), checks if err is a string and, if it is, creates an Error object. Then, the execution goes to the next error handler.

```
function logErrors(err, req, res, next) {
  if (typeof err === 'string')
    err = new Error (err);
  console.error('logErrors', err.toString());
  next(err);
}
```

As previously mentioned, clientErrorHandler is dedicated to AJAX/XHR requests from the Backbone.js app (responds with JSON) by checking req.xhr, and it will send a JSON message back or go to the next handler:

```
function clientErrorHandler(err, req, res, next) {
  if (req.xhr) {
    console.error('clientErrors response');
    res.status(500).json({ error: err.toString()});
  } else {
    next(err);
  }
}
```

The last error handler, errorHandler(), will assume that the request is not AJAX/XHR (otherwise clientErrorHandler() would have caught it, but this order will be defined later with app.use()), and will send a string back:

```
function errorHandler(err, req, res, next) {
  console.error('lastErrors response');
  res.status(500).send(err.toString());
}
```

Recall that we determine process.env.PORT and fall back on local setup value 3000 using ||. We do a similar thing with a MongoDB connection string. We pull the Heroku add-on URI string from the environment variable or fall back to local settings:

```
var dbUrl = process.env.MONGOHQ_URL
  || 'mongodb://@127.0.0.1:27017/hackhall';
var mongoose = require('mongoose');
```

Now, we create a connection:

```
var connection = mongoose.createConnection(dbUrl);
connection.on('error', console.error.bind(console,
  'connection error:'));
```

Sometimes it's a good idea to log the connection open event:

```
connection.once('open', function () {
  console.info('Connected to database')
});
```

The Mongoose models live in the `models` folder:

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

This middleware will provide access to two collections within our route methods:

```
function db (req, res, next) {
  req.db = {
    User: connection.model('User', models.User, 'users'),
    Post: connection.model('Post', models.Post, 'posts')
  };
  return next();
}
```

The lines below are just new names for the imported `routes/main.js` file authorization functions:

```
var checkUser = routes.main.checkUser;
var checkAdmin = routes.main.checkAdmin;
var checkApplicant = routes.main.checkApplicant;
```

We then move onto the AngelList OAuth routes for AngelList login. This is a standard, three-legged OAuth 1.0 strategy where we initiate the auth (`/auth/angellist`), redirect users to the service provider (AngelList), and then wait for the user to come back from the service provider (`/auth/angellist`):

```
app.get('/auth/angellist', routes.auth.angelList);
app.get('/auth/angellist/callback',
  routes.auth.angelListCallback,
  routes.auth.angelListLogin,
  db,
  routes.users.findOrAddUser);
```

■ **Tip** For more information on OAuth and Node.js OAuth examples, take a look at my book *Introduction to OAuth with Node.js* (2014), available at `https://gumroad.com/l/oauthnode`.

The next few lines of code deal with Passport and GitHub login logic. Implementing OAuth using Passport requires less manual effort than using the OAuth module.

Let's skip to the main application routes, starting with `app.get('/api/profile')`. The `api/profile` is used by the Backbone.js app, and it returns a user session, if the user is logged in. The request travels via `checkUser` and `db`, with the former authorizing and the latter populating the database info.

```
// MAIN
app.get('/api/profile', checkUser, db, routes.main.profile);
app.delete('/api/profile', checkUser, db, routes.main.delProfile);
app.post('/api/login', db, routes.main.login);
app.post('/api/logout', routes.main.logout);
```

The Posts and Users collections routes are for manipulating with posts and users:

```
// POSTS
app.get('/api/posts', checkUser, db, routes.posts.getPosts);
app.post('/api/posts', checkUser, db, routes.posts.add);
app.get('/api/posts/:id', checkUser, db, routes.posts.getPost);
app.put('/api/posts/:id', checkUser, db, routes.posts.updatePost);
app.delete('/api/posts/:id', checkUser, db, routes.posts.del);

// USERS
app.get('/api/users', checkUser, db, routes.users.getUsers);
app.get('/api/users/:id', checkUser, db,routes.users.getUser);
app.post('/api/users', checkAdmin, db, routes.users.add);
app.put('/api/users/:id', checkAdmin, db, routes.users.update);
app.delete('/api/users/:id', checkAdmin, db, routes.users.del);
```

These routes are for new members that haven't been approved yet (i.e., they've submitted an application):

```
//APPLICATION
app.post('/api/application', checkAdmin, db, routes.application.add);
app.put('/api/application', checkApplicant, db, routes.application.update);
app.get('/api/application', checkApplicant, db, routes.application.get);
```

The following is the catch-all-else route:

```
app.get('*', function(req, res){
  res.status(404).send();
});
```

We apply the error handlers in the order in which we want them to be called:

```
app.use(logErrors);
app.use(clientErrorHandler);
app.use(errorHandler);
```

The `require.main === module` is a clever trick to determine if this file is being executed as a stand-alone or as an imported module:

```
http.createServer(app);
if (require.main === module) {
  app.listen(app.get('port'), function(){
```

We show the blue log message:

```
    console.info(c.blue + 'Express server listening on port '
      + app.get('port') + c.reset);
  });
}
else {
  console.info(c.blue + 'Running app as a module' + c.reset)
  exports.app = app;
}
```

To save space, I won't list the full source code for `hackhall/server.js`, but you can view it at `https://github.com/azat-co/hackhall/blob/v3.1.0/server.js`.

Routes

The HackHall routes reside in the hackhall/routes folder and are grouped into several modules:

- hackhall/routes/index.js: Bridge between server.js and other routes in the folder
- hackhall/routes/auth.js: Routes that handle the OAuth "dance" with the AngelList API
- hackhall/routes/main.js: Login, logout, and other routes
- hackhall/routes/users.js: Routes related to users' REST API
- hackhall/routes/application.js: Routes that handle submission of an application to become a user
- hackhall/routes/posts.js: Routes related to posts' REST API

index.js

Let's peek into hackhall/routes/index.js, where we've included other modules:

```
exports.posts = require('./posts');
exports.main = require('./main');
exports.users = require('./users');
exports.application = require('./application');
exports.auth = require('./auth');
```

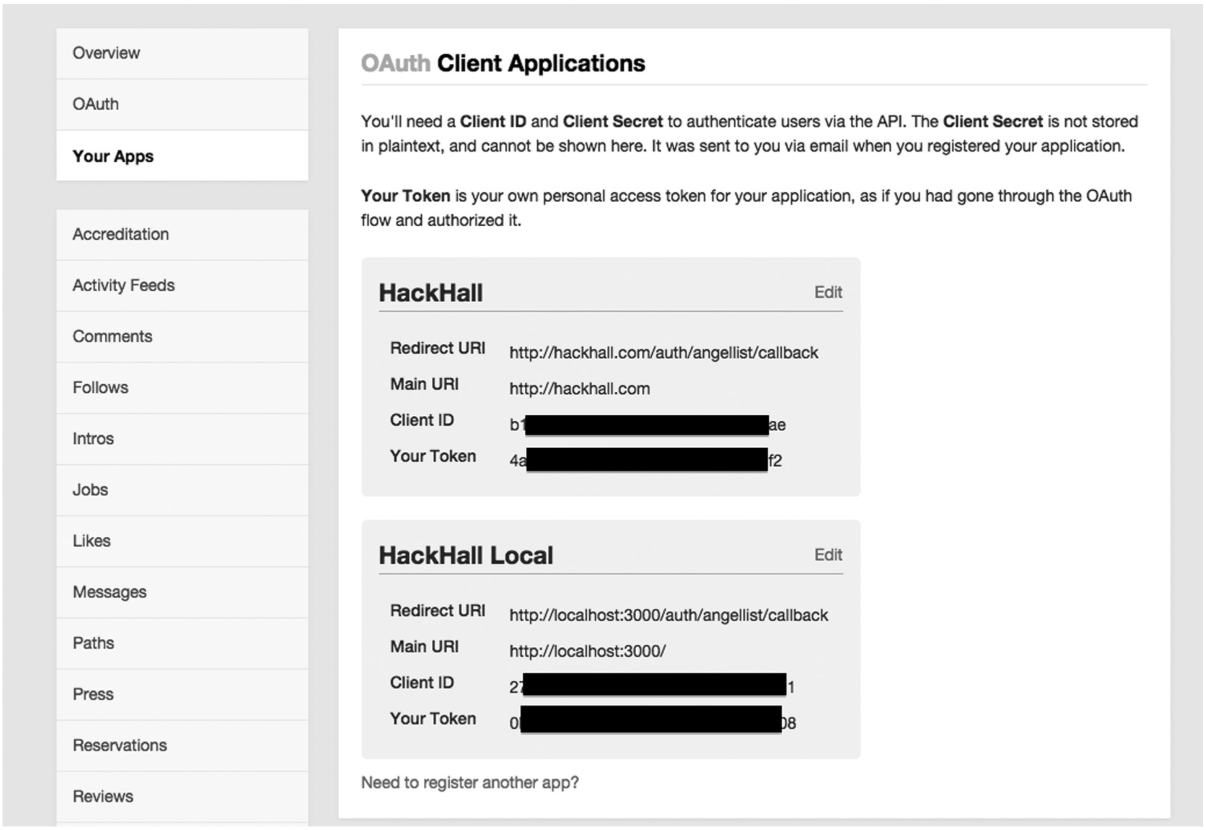
auth.js

In this module, we handle the OAuth dance with the AngelList API. To do so, we rely on the https library:

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

The AngelList API client ID and client secret are obtained at https://angel.co/api and stored in environment variables. I added two applications: one for local development and the other for production, as shown in Figure 22-5. The app picks one of them, based on the environment:

```
if (process.env.NODE_ENV === 'production') {
  var angelListClientId = process.env.ANGELLIST_CLIENT_ID;
  var angelListClientSecret = process.env.ANGELLIST_CLIENT_SECRET;
} else {
  var angelListClientId = process.env.ANGELLIST_CLIENT_ID_LOCAL;
  var angelListClientSecret = process.env.ANGELLIST_CLIENT_SECRET_LOCAL;
}
```



OAuth **Client Applications**

You'll need a **Client ID** and **Client Secret** to authenticate users via the API. The **Client Secret** is not stored in plaintext, and cannot be shown here. It was sent to you via email when you registered your application.

Your Token is your own personal access token for your application, as if you had gone through the OAuth flow and authorized it.

HackHall

Redirect URI	http://hackhall.com/auth/angellist/callback
Main URI	http://hackhall.com
Client ID	b1[redacted]ae
Your Token	4a[redacted]f2

HackHall Local

Redirect URI	http://localhost:3000/auth/angellist/callback
Main URI	http://localhost:3000/
Client ID	27[redacted]1
Your Token	0[redacted]08

Figure 22-5. *My AngelList apps*

The `exports.angelList()` method will redirect users to the `https://angel.co/api` web site for authentication. This method is invoked when we navigate to `/auth/angellist`. The request's structure is described in the documentation at `https://angel.co/api/oauth/faq`.

```
exports.angelList = function(req, res) {
  res.redirect('https://angel.co/api/oauth/authorize?client_id=' + angelListClientId +
'&scope=email&response_type=code');
}
```

After users allow our app to access their information, AngelList sends them back to this route to allow us to make a new (HTTPS) request to retrieve the token:

```
exports.angelListCallback = function(req, res, next) {
  var token;
  var buf = '';
  var data;
  var angelReq = https.request({
```

The values of host and path are specific to your service provider, so you need to consult the provider's documentation when you implement OAuth. These are the values for AngelList API:

```
    host: 'angel.co',
    path: '/api/oauth/token?client_id=' + angelListClientId +
      '&client_secret=' + angelListClientSecret + '&code=' + req.query.code +
      '&grant_type=authorization_code',
    port: 443,
    method: 'POST',
    headers: {
      'content-length': 0
    }
```

At this point, the callback should have the response with the token (or an error), so we parse the response and check for the access_token. If it's there, we save the token in the session and proceed to the next middleware in the /auth/angellist/callback, which is angelListLogin. First, let's attach an event listener that saves the response in buf:

```
  }, function(angelRes) {
    angelRes.on('data', function(buffer) {
      buf += buffer;
    });
```

Then, we attach another event listener for the end event:

```
    angelRes.on('end', function() {
```

The buf object at this moment should have the full response body in a Buffer type, so we need to convert it to a string type and parse. The data should have only two properties, access_token and token_type ('bearer'):

```
      try {
        data = JSON.parse(buf.toString('utf-8'));
      } catch (e) {
        if (e) return next(e);
      }
```

Let's check for the access_token to be 100% sure:

```
      if (!data || !data.access_token) return  next(new Error('No data from AngelList'));
      token = data.access_token;
```

Now, we can save token in a session and call the next middleware:

```
      req.session.angelListAccessToken = token;
      if (token) {
        next();
      }
      else {
        next(new Error('No token from AngelList'));
      }
    });
  });
```

The rest of the request code finishes the request and deals with an `error` event:

```
  angelReq.end();
  angelReq.on('error', function(e) {
    console.error(e);
    next(e);
  });
}
```

So, the user has been granted access to our AngelList app and we have the token (`angelListCallback`). Now, we can get the user profile information by directly calling the AngelList API with the token from the previous middleware (`angelListLogin`). The order of the middleware functions is dictated by the route `/auth/angellist/callback`, so we begin `angelListLogin` with the HTTPS request:

```
exports.angelListLogin = function(req, res, next) {
  var token = req.session.angelListAccessToken;
  httpsRequest = https.request({
    host: 'api.angel.co',
```

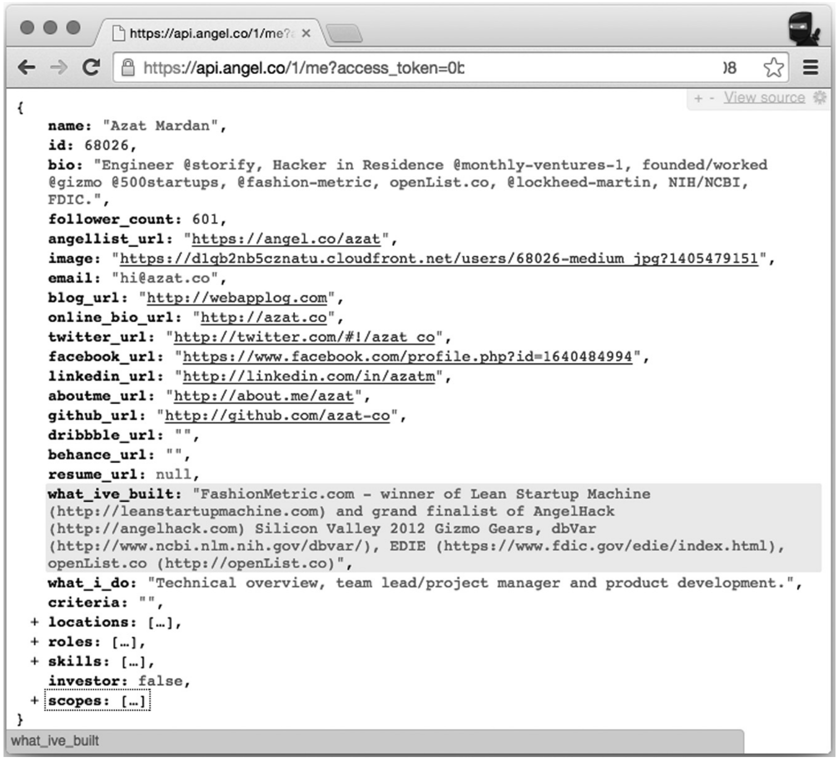
Again, the exact URL is different for each service:

```
    path: '/1/me?access_token=' + token,
    port: 443,
    method: 'GET'
  },
  function(httpsResponse) {
    var userBuffer = '';
    httpsResponse.on('data', function(buffer) {
      userBuffer += buffer;
    });
```

The next event listener will parse the buffer-type object into the normal JavaScript/Node.js object:

```
    httpsResponse.on('end', function(){
      try {
        data = JSON.parse(userBuffer.toString('utf-8'));
      } catch (e) {
        if (e) return next(e);
      }
```

At this point in the execution, the system should have data fields filled with the user information (`/1/me?access_token=...` endpoint). You can see an example of such response data in Figure 22-6.


https://api.angel.co/1/me?
https://api.angel.co/1/me?access_token=0b
+ - View source
what_ive_built

```
{
   name: "Azat Mardan",
   id: 68026,
   bio: "Engineer @storify, Hacker in Residence @monthly-ventures-1, founded/worked
   @gizmo @500startups, @fashion-metric, openList.co, @lockheed-martin, NIH/NCBI,
   FDIC.",
   follower_count: 601,
   angellist_url: "https://angel.co/azat",
   image: "https://d1qb2nb5cznatu.cloudfront.net/users/68026-medium_jpg?1405479151",
   email: "hi@azat.co",
   blog_url: "http://webapplog.com",
   online_bio_url: "http://azat.co",
   twitter_url: "http://twitter.com/#!/azat_co",
   facebook_url: "https://www.facebook.com/profile.php?id=1640484994",
   linkedin_url: "http://linkedin.com/in/azatm",
   aboutme_url: "http://about.me/azat",
   github_url: "http://github.com/azat-co",
   dribbble_url: "",
   behance_url: "",
   resume_url: null,
   what_ive_built: "FashionMetric.com - winner of Lean Startup Machine
   (http://leanstartupmachine.com) and grand finalist of AngelHack
   (http://angelhack.com) Silicon Valley 2012 Gizmo Gears, dbVar
   (http://www.ncbi.nlm.nih.gov/dbvar/), EDIE (https://www.fdic.gov/edie/index.html),
   openList.co (http://openList.co)",
   what_i_do: "Technical overview, team lead/project manager and product development.",
   criteria: "",
 + locations: […],
 + roles: […],
 + skills: […],
   investor: false,
 + scopes: […]
}
```

Figure 22-6. *Example of the AngelList user info response*

We still have to check whether the object is empty or not and, if it's not empty, we save the user data on the request object:

```
      if (data) {
        req.angelProfile = data;
        next();
      } else
        next(new Error('No data from AngelList'));
      });
    }
  );
  httpsRequest.end();
  httpsRequest.on('error', function(e) {
    console.error(e);
  });
};
```

The full source code for the `hackhall/routes/auth.js` file at the time of this writing is at `https://github.com/azat-co/hackhall/blob/v3.1.0/routes/auth.js` (which is subject to change as the HackHall version evolves).

main.js

The `hackhall/routes/main.js` file is also interesting, because it has these methods:

- `checkAdmin()`
- `checkUser()`
- `checkApplicant()`
- `login()`
- `logout()`
- `profile()`
- `delProfile()`

The `checkAdmin()` function performs authentication for admin privileges. If the session object doesn't carry the proper flag, we call the Express.js `next()` function with an error object:

```
exports.checkAdmin = function(request, response, next) {
  if (request.session
    && request.session.auth
    && request.session.userId
    && request.session.admin) {
    console.info('Access ADMIN: ' + request.session.userId);
    return next();
  } else {
    next('User is not an administrator.');
  }
};
```

Similarly, we can check for only the approved user without checking for admin rights:

```
exports.checkUser = function(req, res, next) {
  if (req.session && req.session.auth && req.session.userId
    && (req.session.user.approved || req.session.admin)) {
    console.info('Access USER: ' + req.session.userId);
    return next();
  } else {
    next('User is not logged in.');
  }
};
```

If an application is just an unapproved user object, we can also check for that:

```
exports.checkApplicant = function(req, res, next) {
  if (req.session && req.session.auth && req.session.userId
    && (!req.session.user.approved || req.session.admin)) {
    console.info('Access USER: ' + req.session.userId);
    return next();
  } else {
    next('User is not logged in.');
  }
};
```

In the login function, we search for e-mail. Because we don't store the plain password in the database—we store only its encrypted hash—we need to compare the password hashes using `bcryptjs`. Upon a successful match, we store the user object in the session, set the `auth` flag to `true` (`req.session.auth = true`), and proceed. Otherwise, the request fails:

```
var bcrypt = require('bcryptjs');
exports.login = function(req, res, next) {
  console.log('Logging in USER with email:', req.body.email)
  req.db.User.findOne({
      email: req.body.email
    },null, {
      safe: true
    }, function(err, user) {
      if (err) return next(err);
      if (user) {
```

We use the asynchronous `bcryptjs` method `compare()`, which returns `true`, if the plain password matches the saved hashed password:

```
        bcrypt.compare(req.body.password, user.password, function(err, match) {
          if (match) {
```

So, all is good: the system assigns session flags and saves user info in the session. These values will be used on all auth-required (protected) routes to identify the user:

```
            req.session.auth = true;
            req.session.userId = user._id.toHexString();
            req.session.user = user;
```

There's a separate boolean for admins:

```
            if (user.admin) {
              req.session.admin = true;
            }
            console.info('Login USER: ' + req.session.userId);
```

The JSON `{msg: 'Authorized'}` object is an arbitrary convention that you can customize, but you must keep it the same on the server and the client (to check the server response):

```
            res.status(200).json({
              msg: 'Authorized'
            });
          } else {
            next(new Error('Wrong password'));
          }
        });
      } else {
        next(new Error('User is not found.'));
      }
    });
};
```

The logging-out process removes any session information:

```
exports.logout = function(req, res) {
  console.info('Logout USER: ' + req.session.userId);
  req.session.destroy(function(error) {
    if (!error) {
      res.send({
        msg: 'Logged out'
      });
    }
  });
};
```

This route is used for the Profile page, as well as by Backbone.js for user authentication:

```
exports.profile = function(req, res, next) {
```

We don't want to expose all the user fields, so we whitelist only the fields that we want to get:

```
  var fields = 'firstName lastName displayName' +
    ' headline photoUrl admin approved banned' +
    ' role angelUrl twitterUrl facebookUrl linkedinUrl githubUrl';
```

This is a custom method created via Mongoose functionality for the reason that it has quite extensive logic and is called more than once:

```
  req.db.User.findProfileById(req.session.userId, fields, function(err, obj) {
    if (err) next(err);
    res.status(200).json(obj);
  });
};
```

It's important to allow users to delete their profiles. We utilize the `findByIdAndRemove()` method and remove the session with `destroy()`:

```
exports.delProfile = function(req, res, next) {
  console.log('del profile');
  console.log(req.session.userId);
  req.db.User.findByIdAndRemove(req.session.user._id, {},
    function(err, obj) {
      if (err) next(err);
      req.session.destroy(function(error) {
        if (err) {
          next(err)
        }
      });
      res.status(200).json(obj);
    }
  );
};
```

The full source code of the `hackhall/routes/main.js` file is available at `https://github.com/azat-co/hackhall/blob/v3.1.0/routes/main.js`.

users.js

The `routes/users.js` file is responsible for the RESTful activities related to User collections. We have these methods:

- `getUsers()`
- `getUser()`
- `add()`
- `update()`
- `del()`
- `findOrAddUser()`

First, we define some variables:

```
var path = require('path'),
  hs = require(path.join(__dirname, '..', 'lib', 'hackhall-sendgrid'));

var objectId = require('mongodb').ObjectID;

var safeFields = 'firstName lastName displayName headline photoUrl admin approved banned role
angelUrl twitterUrl facebookUrl linkedinUrl githubUrl';
```

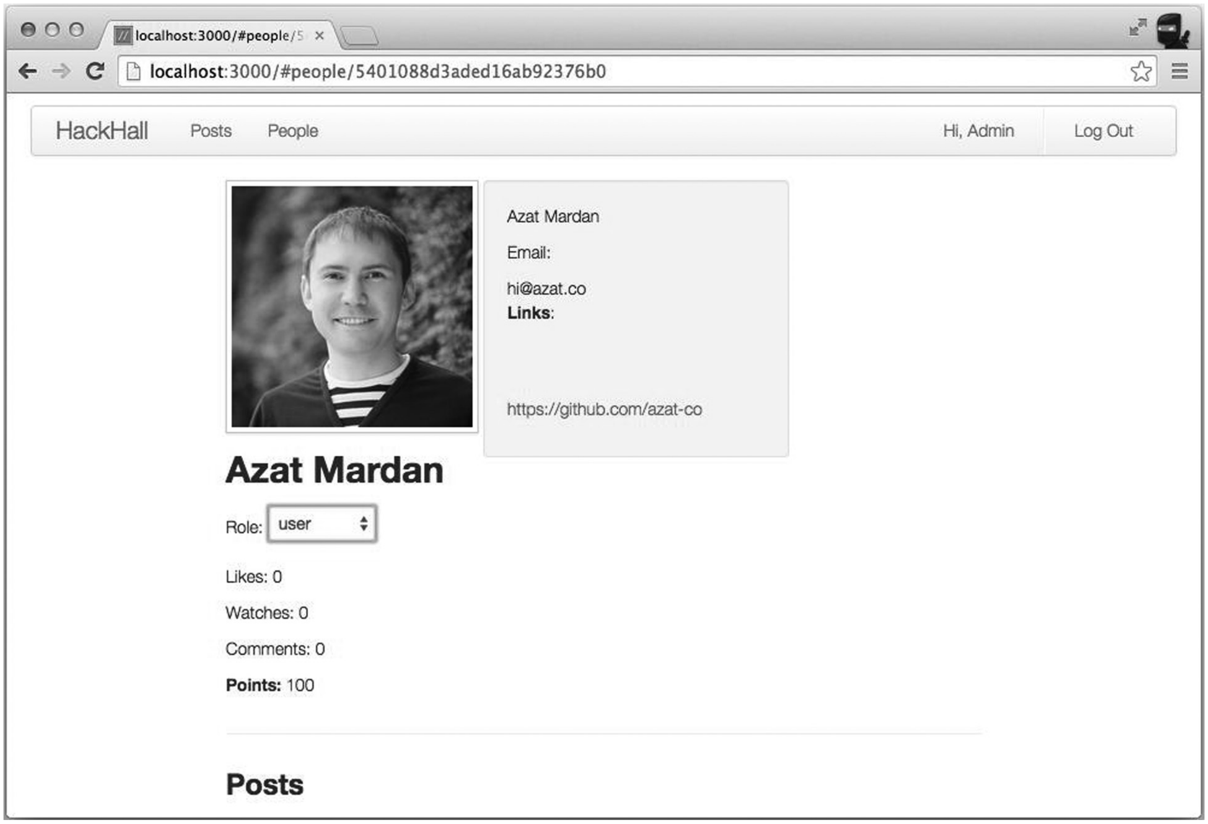
Then, we define the method `getUsers()`, which retrieves a list of users where each item has only the properties from the `safeFields` string:

```
exports.getUsers = function(req, res, next) {
  if (req.session.auth && req.session.userId) {
    req.db.User.find({}, safeFields, function(err, list) {
      if (err) return next(err);
      res.status(200).json(list);
    });
  } else {
    return next('User is not recognized.')
  }
}
```

The `getUser()` method is used on the user profile page. For administrators (the current user, not the user we fetch), we add an extra field, `email`, and invoke the custom static method `findProfileById()`:

```
exports.getUser = function(req, res, next) {
  var fields = safeFields;
  if (req.session.admin) {
    fields = fields + ' email';
  }
  req.db.User.findProfileById(req.params.id, fields, function(err, data){
    if (err) return next(err);
    res.status(200).json(data);
  })
}
```

To see the `getUser()` method in action, you can navigate to a user profile page, as shown in Figure 22-7. Admins can manage users' accounts on the Profile page. So, if you're an admin, you'll see an extra Role drop-down that sets the role for this user.



Azat Mardan

Email:

hi@azat.co

Links:

https://github.com/azat-co

Azat Mardan

Role: user

Likes: 0

Watches: 0

Comments: 0

Points: 100

Posts

Figure 22-7. *HackHall Profile page when logged in as admin*

The `add()` method is straightforward:

```
exports.add = function(req, res, next) {
  var user = new req.db.User(req.body);
  user.save(function(err) {
    if (err) next(err);
    res.json(user);
  });
};
```

The update() method is also used for the approval of new users (approvedNow == true). In case of success, we send an e-mail using the internal method notifyApproved() from the lib/hackhall-sendgrid.js file:

```
exports.update = function(req, res, next) {
var obj = req.body;
  obj.updated = new Date();
  delete obj._id;
  var approvedNow = obj.approved && obj.approvedNow;
```

The approvedNow field is not a field in the Mongoose schema, and we don't want to store it. The only purpose of the field is to let the system know whether it's a regular update call or an approval:

```
  delete obj.approvedNow;
  req.db.User.findByIdAndUpdate(req.params.id, {
    $set: obj
  }, {
```

This option will give us the new object instead of the original (the default is true):

```
    new: true
  }, function(err, user) {
    if (err) return next(err);
    if (approvedNow && user.approved) {
      console.log('Approved... sending notification!');
```

So, the approval went successfully and we can send an e-mail:

```
      hs.notifyApproved(user, function(error, user){
        if (error) return next(error);
        console.log('Notification was sent.');
        res.status(200).json(user);
      })
    } else {
```

If it was a regular update, and not an approval, then we just send back the user object:

```
      res.status(200).json(user);
    }
  });
};
```

Figure 22-8 shows how approval looks in the user interface, when you're logged in with admin rights. There is a drop-down menu that admins can use to approve, delete, or ban applicants.

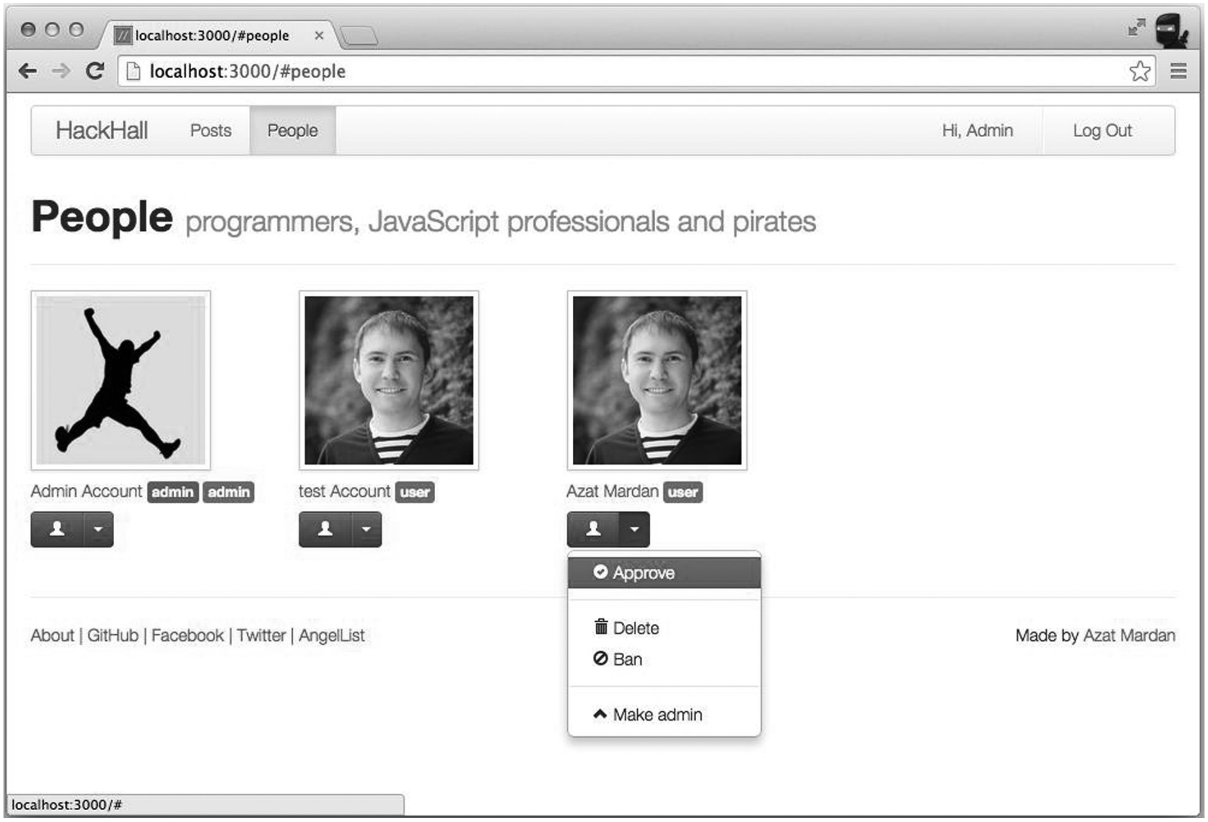


Figure 22-8. *HackHall People page with admin rights (logged in as admin)*

To delete a user, we call `findByIdAndRemove()`:

```
exports.del = function(req, res, next) {
  req.db.User.findByIdAndRemove(req.params.id, function(err, obj) {
    if (err) next(err);
    res.status(200).json(obj);
  });
};
```

Lastly, the `findOrAddUser()` method is employed when a user logs in with AngelList. You can use `findOrCreate` supplied by the plug-in (and that's what is used in the GitHub OAuth flow), but for the sake of learning, it's good to know how to implement the same functionality on your own. It will also reinforce your asynchronous way of thinking and your understanding of how to refactor code when you compare `findOrCreate` with this function:

```
exports.findOrAddUser = function(req, res, next) {
  var data = req.angelProfile;
  req.db.User.findOne({
    angelListId: data.id
  }, function(err, obj) {
    console.log('angelList Login findOrAddUser');
    if (err) return next(err);
```

Okay, so we queried the database for the user, but let's check if the user is there and, if not, create the user:

```
    if (!obj) {
      console.warn('Creating a user', obj, data);
      req.db.User.create({
```

We map/normalize all the fields that we need from the AngelList response to the user object.

```
    angelListId: data.id,
```

This token is what we can use to make subsequent API requests on behalf of the user without asking for authorization and permissions each time:

```
    angelToken: req.session.angelListAccessToken,
```

And, just in case, we store the whole AngelList object as well in `angelListProfile`:

```
    angelListProfile: data,
    email: data.email,
```

The `data.name` is the full name, so we split it by space into an array and get the first and second elements separately:

```
    firstName: data.name.split(' ')[0],
    lastName: data.name.split(' ')[1],
    displayName: data.name,
    headline: data.bio,
```

The image is just a URL to the file, not a binary field:

```
    photoUrl: data.image,
    angelUrl: data.angellist_url,
    twitterUrl: data.twitter_url,
    facebookUrl: data.facebook_url,
    linkedinUrl: data.linkedin_url,
    githubUrl: data.github_url
   }, function(err, obj) {
    if (err) return next(err);
console.log('User was created', obj);
```

Okay, so the user document has been successfully created. But the system must log in the user right away, so we set the `session` flag to `true`:

```
    req.session.auth = true;
```

We need to save the newly created user ID in the session so that we can use it on other requests from this client:

```
    req.session.userId = obj._id;
    req.session.user = obj;
```

The admin needs to be promoted by another admin. The default database value for new users is taken care of by the Mongoose schema (with the default of `false`). However, the session value needs to be set here, so we authenticate as a regular user by default:

```
      req.session.admin = false;
      res.redirect('/#application');
      }
    );
  } else {
```

When the user document is in the database, we just log in the user and redirect either to posts or their membership application:

```
        req.session.auth = true;
        req.session.userId = obj._id;
        req.session.user = obj;
        req.session.admin = obj.admin;
        if (obj.approved) {
          res.redirect('/#posts');
        } else {
          res.redirect('/#application');
        }
      }
    })
  }
```

The full source code for the `hackhall/routes/users.js` file is available at `https://github.com/azat-co/hackhall/blob/v3.1.0/routes/users.js`.

`users.js` provides functionality to the REST API routes of the People page that allows a user to visit other users' profiles, as shown in Figure 22-9. In that screenshot, the first Azat's profile is from seeding the database. The second Azat's profile is from my logging in with GitHub.

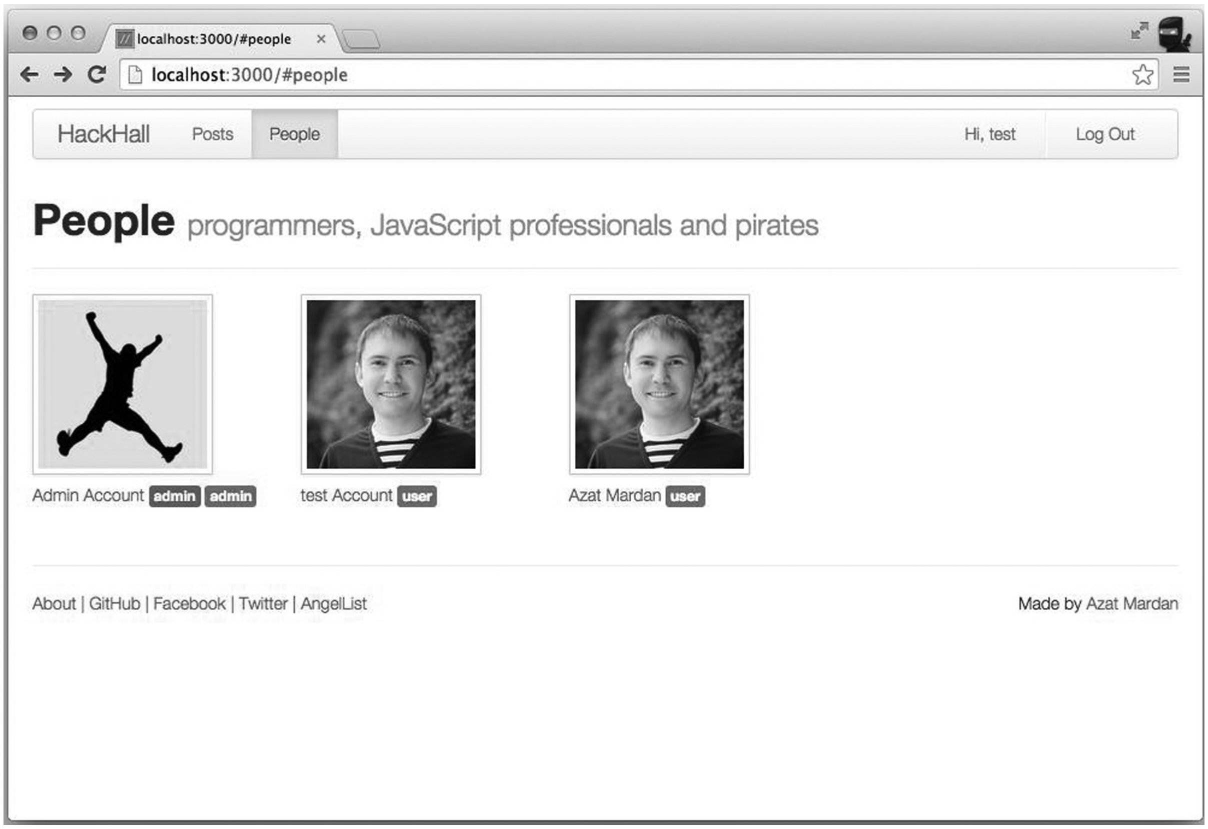


Figure 22-9. *HackHall People page*

application.js

The hackhall/routes/application.js file ("application" in the sense of applying, not as in app!) deals with new users applying to join the HackHall community. They need to be approved to ensure that only real and serious members join HackHall.com. In your local version, you might want to suppress the e-mail notifications regarding submitting and approving of an application.

Merely to add a user object (with approved=false by default) to the database (e-mail membership application), we use this method:

```
exports.add = function(req, res, next) {
  req.db.User.create({
    firstName: req.body.firstName,
    lastName: req.body.lastName,
    displayName: req.body.displayName,
    headline: req.body.headline,
    photoUrl: req.body.photoUrl,
    password: req.body.password,
    email: req.body.email,
    angelList: {
      blah: 'blah'
    },
    angelUrl: req.body.angelUrl,
    twitterUrl: req.body.twitterUrl,
    facebookUrl: req.body.facebookUrl,
    linkedinUrl: req.body.linkedinUrl,
    githubUrl: req.body.githubUrl
  }, function(err, obj) {
    if (err) return next(err);
    if (!obj) return next('Cannot create.')
    res.status(200).json(obj);
  })
};
```

We let the users update information in their applications with this method:

```
exports.update = function(req, res, next) {
  var data = req.body;
```

The _id needs to be removed first, because we don't want to change it:

```
  delete data._id;
```

In the `findByIdAndUpdate()` method, we employ the user ID from the session, not the one from the body, because it cannot be trusted:

```
  req.db.User.findByIdAndUpdate(req.session.user._id, {
    $set: data
  }, function(err, obj) {
    if (err) return next(err);
    if (!obj) return next('Cannot save.')
```

It's probably okay to send the whole thing back (`obj`), because it's this user's info anyway:

```
    res.status(200).json(obj);
  });
};
```

Select a particular object with the `get()` function:

```
exports.get = function(req, res, next) {
  req.db.User.findById(req.session.user._id,
```

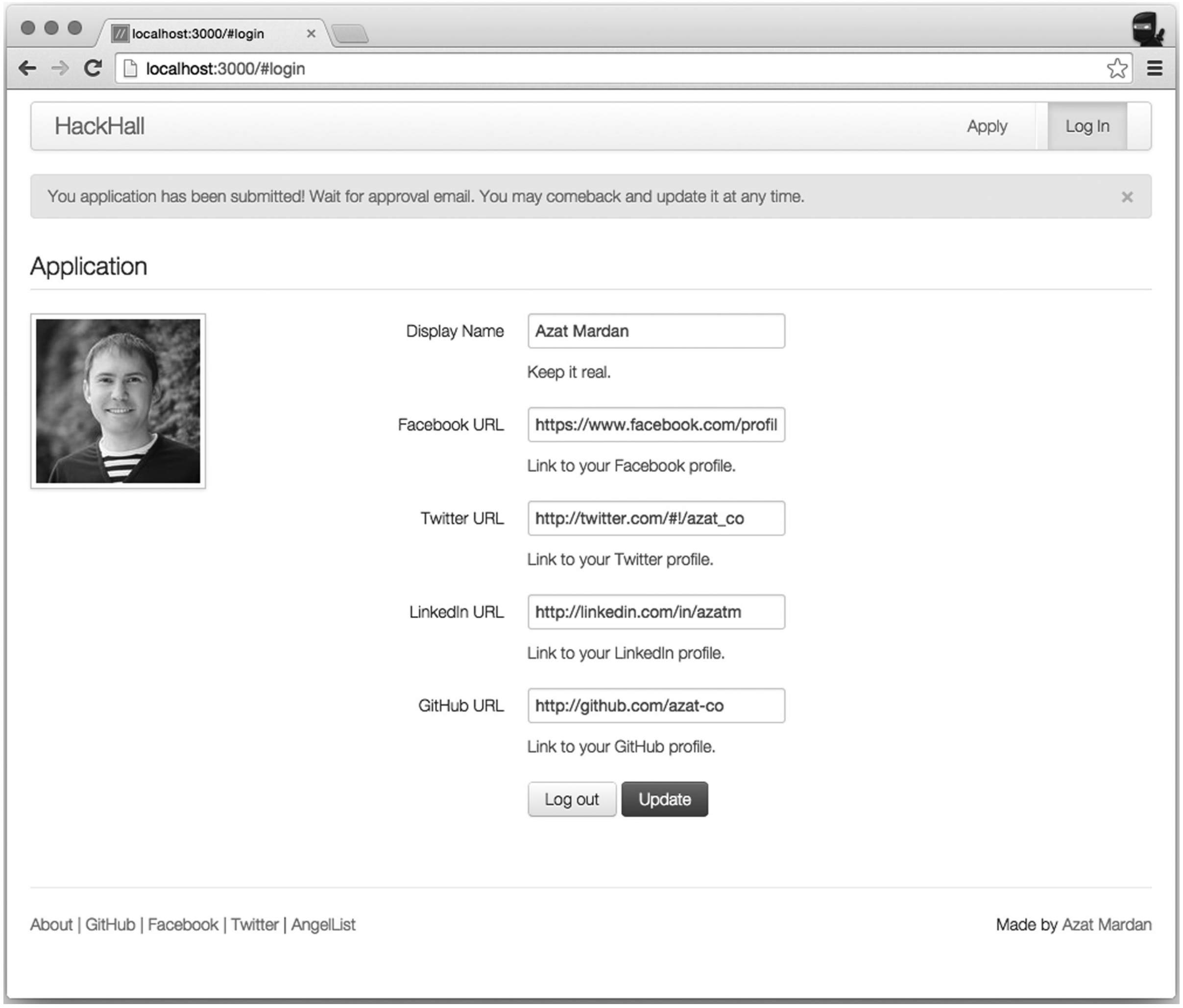
Limit the field that we get back:

```
    'firstName lastName photoUrl headline displayName'
      + 'angelUrl facebookUrl twitterUrl linkedinUrl'
      + 'githubUrl', {}, function(err, obj) {
      if (err) return next(err);
      if (!obj) return next('cannot find');
      res.status(200).json(obj);
    })
};
```

The following is the full source code of the hackhall/routes/applications.js file:

```
exports.add = function(req, res, next) {
  req.db.User.create({
    firstName: req.body.firstName,
    lastName: req.body.lastName,
    displayName: req.body.displayName,
    headline: req.body.headline,
    photoUrl: req.body.photoUrl,
    password: req.body.password,
    email: req.body.email,
    angelList: {
      blah: 'blah'
    },
    angelUrl: req.body.angelUrl,
    twitterUrl: req.body.twitterUrl,
    facebookUrl: req.body.facebookUrl,
    linkedinUrl: req.body.linkedinUrl,
    githubUrl: req.body.githubUrl
  }, function(err, obj) {
    if (err) return next(err);
    if (!obj) return next('Cannot create.')
    res.status(200).json(obj);
  })
};

exports.update = function(req, res, next) {
  var data = req.body;
  delete data._id;
  req.db.User.findByIdAndUpdate(req.session.user._id, {
    $set: data
  }, function(err, obj) {
    if (err) return next(err);
    if (!obj) return next('Cannot save.')
    res.status(200).json(obj);
  });
};

exports.get = function(req, res, next) {
  req.db.User.findById(req.session.user._id,
    'firstName lastName photoUrl headline displayName angelUrl facebookUrl twitterUrl linkedinUrl
     githubUrl', {}, function(err, obj) {
      if (err) return next(err);
      if (!obj) return next('cannot find');
      res.status(200).json(obj);
    })
};
```

Figure 22-10 shows how the membership application page looks at this point.


localhost:3000/#login

HackHall

Apply | Log In

You application has been submitted! Wait for approval email. You may comeback and update it at any time.

Application

Display Name: Azat Mardan
Keep it real.

Facebook URL: https://www.facebook.com/profil
Link to your Facebook profile.

Twitter URL: http://twitter.com/#!/azat_co
Link to your Twitter profile.

LinkedIn URL: http://linkedin.com/in/azatm
Link to your LinkedIn profile.

GitHub URL: http://github.com/azat-co
Link to your GitHub profile.

Log out | Update

About | GitHub | Facebook | Twitter | AngelList

Made by Azat Mardan

Figure 22-10. *HackHall membership application page*

posts.js

The last routes module that we need to dissect is `hackhall/routes/posts.js`. It takes care of adding, editing, and removing posts, as well as commenting, watching, and liking.

We use the object ID for conversion from HEX strings to proper objects:

```
objectId = require('mongodb').ObjectID;
```

The default values for the pagination of posts are as follows:

```
var LIMIT = 10;
var SKIP = 0;
```

The add() function handles the creation of new posts:

```
exports.add = function(req, res, next) {
  if (req.body) {
```

The req.db.Post is available here because of the custom db middleware that is used on most of the routes:

```
    req.db.Post.create({
      title: req.body.title,
      text: req.body.text || null,
      url: req.body.url || null,
```

We set the author of the post from the user's session info:

```
      author: {
        id: req.session.user._id,
        name: req.session.user.displayName
      }
    }, function(err, docs) {
      if (err) {
        console.error(err);
        next(err);
      } else {
        res.status(200).json(docs);
      }

    });
  } else {
    next(new Error('No data'));
  }
};
```

To retrieve the list of posts using either the values of limit and skip from the request query or the default values, we use the following:

```
exports.getPosts = function(req, res, next) {
  var limit = req.query.limit || LIMIT;
  var skip = req.query.skip || SKIP;
  req.db.Post.find({}, null, {
    limit: limit,
    skip: skip,
```

We sort the results by their ID, which typically lists the results in chronological order (for more precise results, we could have used the created field here):

```
    sort: {
      '_id': -1
    }
  }, function(err, obj) {
```

At this point, we check to see if there are any posts in the `obj`, and then, we perform a loop to add some helper flags such as `admin`, `own`, `like`, and `watch`:

```
    if (!obj) return next('There are not posts.');
    var posts = [];
    docs.forEach(function(doc, i, list) {
```

The `doc` object is a Mongoose document object that has a lot of magic, so it's best to convert the data to a normal object:

```
    var item = doc.toObject();
```

Now, we can check whether or not the user has admin rights, and if the user has them, then we set `item.admin` to `true`, but on the property of the new object `item`. This is redundant, as the client app has the `admin` flag elsewhere, but it's convenient to have this info on each post for representation purposes, because admins can edit and remove any post:

```
    if (req.session.user.admin) {
      item.admin = true;
    } else {
      item.admin = false;
    }
```

The next line checks whether or not the user is the author of this particular post:

```
    if (doc.author.id == req.session.userId) {
      item.own = true;
    } else {
      item.own = false;
    }
```

This line checks whether or not this user liked this post:

```
    if (doc.likes && doc.likes.indexOf(req.session.user._id) > -1) {
      item.like = true;
     } else {
      item.like = false;
     }
```

This line checks whether or not this user watches this post:

```
      if (doc.watches && doc.watches.indexOf(req.session.user._id) > -1) {
        item.watch = true;
        } else {
          item.watch = false;
        }
        posts.push(item);
      });
```

Here's where we form the response body:

```
    var body = {};
    body.limit = limit;
    body.skip = skip;
    body.posts = posts;
```

To include the total number of documents (posts) for pagination, we need this quick query:

```
    req.db.Post.count({}, function(err, total) {
      if (err) return next(err);
      body.total = total;
      res.status(200).json(body);
    });
  });
};
```

For the individual post page, we need the `getPost()` method. We can pass the properties that we want, not as a string, as in `users.js`, but as an object as well:

```
exports.getPost = function(req, res, next) {
  if (req.params.id) {
    req.db.Post.findById(req.params.id, {
```

This is another way to limit fields that we want to be returned from the database:

```
      title: true,
      text: true,
      url: true,
      author: true,
      comments: true,
      watches: true,
      likes: true
    }, function(err, obj) {
      if (err) return next(err);
      if (!obj) {
        next('Nothing is found.');
      } else {
        res.status(200).json(obj);
      }
    });
  } else {
    next('No post id');
  }
};
```

The del() function removes specific posts from the database. The findById() and remove() methods from Mongoose are used in this snippet. However, the same thing can be accomplished with just remove().

```
exports.del = function(req, res, next) {
  req.db.Post.findById(req.params.id, function(err, obj) {
    if (err) return next(err);
```

The following is just a sanity check to confirm that the client is either an admin or the author of the post that we are about to delete:

```
    if (req.session.admin || req.session.userId === obj.author.id) {
      obj.remove();
      res.status(200).json(obj);
    } else {
      next('User is not authorized to delete post.');
    }
  })
};
```

To like the post, we update the post item by prepending the post.likes array with the ID of the user:

```
function likePost(req, res, next) {
  req.db.Post.findByIdAndUpdate(req.body._id, {
```

This is a neat MongoDB operand to add values to arrays:

```
    $push: {
      likes: req.session.userId
    }
  }, {}, function(err, obj) {
    if (err) {
      next(err);
    } else {
      res.status(200).json(obj);
    }
  });
};
```

Likewise, when a user performs the watch action, the system adds a new ID to the post.watches array:

```
function watchPost(req, res, next) {
  req.db.Post.findByIdAndUpdate(req.body._id, {
    $push: {
      watches: req.session.userId
    }
  }, {}, function(err, obj) {
    if (err) next(err);
    else {
      res.status(200).json(obj);
    }
  });
};
```

The updatePost() method is what calls like or watch functions, based on the action flag sent with the request (req.body.action):

```
exports.updatePost = function(req, res, next) {
  var anyAction = false;
  if (req.body._id && req.params.id) {
```

This logic is for adding a like:

```
    if (req.body && req.body.action == 'like') {
      anyAction = true;
      likePost(req, res);
```

The next condition is for adding a watch:

```
    } else if (req.body && req.body.action == 'watch') {
      anyAction = true;
      watchPost(req, res);
```

This one is for adding a comment to the post:

```
    } else if (req.body && req.body.action == 'comment'
      && req.body.comment && req.params.id) {
      anyAction = true;
      req.db.Post.findByIdAndUpdate(req.params.id, {
        $push: {
          comments: {
            author: {
              id: req.session.userId,
              name: req.session.user.displayName
            },
            text: req.body.comment
          }
        }
      }, {
        safe: true,
        new: true
      }, function(err, obj) {
        if (err) throw err;
        res.status(200).json(obj);
      });
```

Lastly, when none of the previous conditions of action are met, `updatePost()` processes the changes to the post itself (title, text, etc.) made by the author or admin (`req.body.author.id == req.session.user._id || req.session.user.admin`):

```
    } else if (req.session.auth && req.session.userId && req.body
      && req.body.action != 'comment' &&
      req.body.action != 'watch' && req.body != 'like' &&
      req.params.id && (req.body.author.id == req.session.user._id
      || req.session.user.admin)) {
      req.db.Post.findById(req.params.id, function(err, doc) {
```

In this context, the doc object is a Mongoose document object, so we assign the new values to its properties and invoke `save()`, which triggers the pre-save hook defined in the model (covered in the next section, “Mongoose Models”):

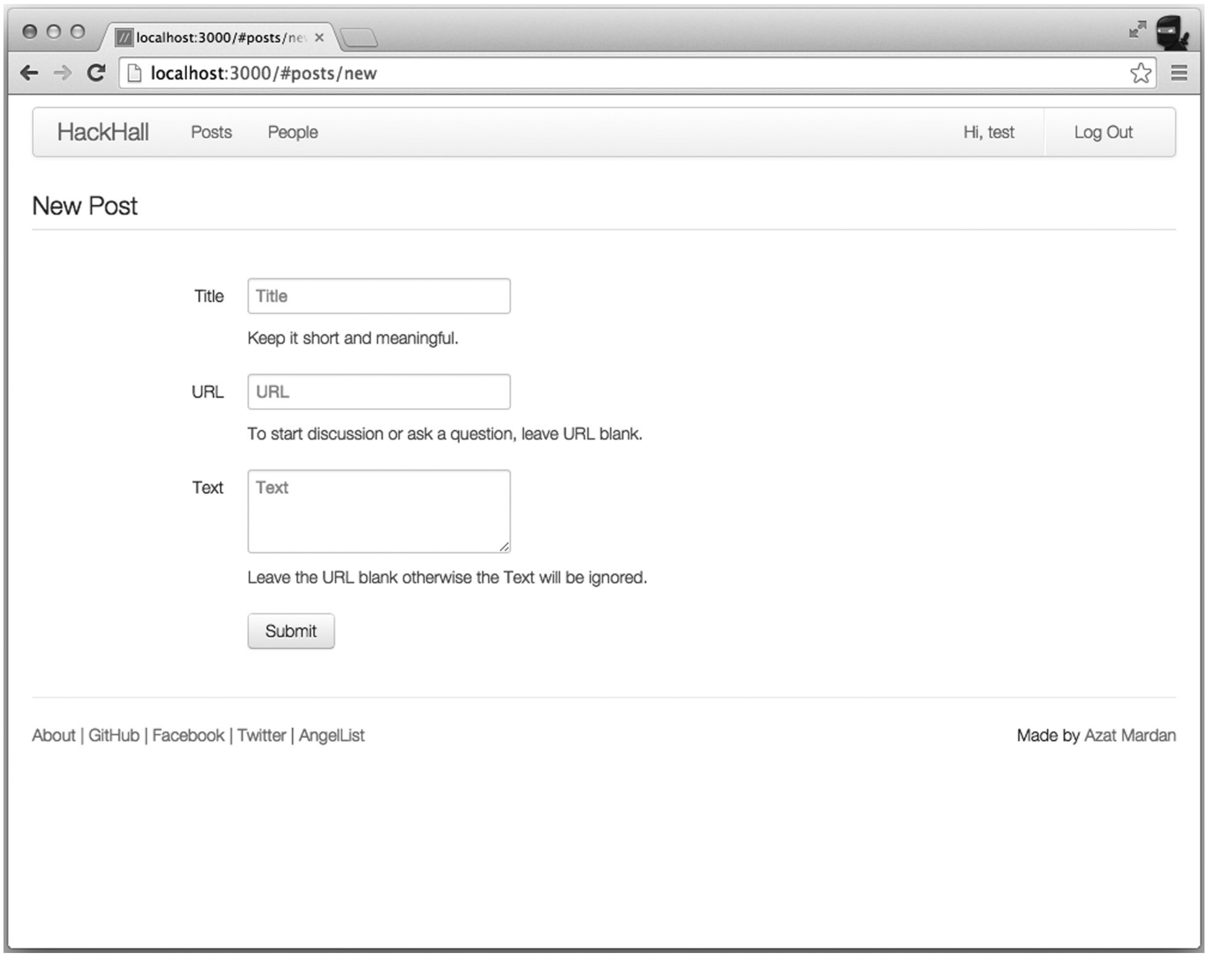
```
        if (err) next(err);
        doc.title = req.body.title;
        doc.text = req.body.text || null;
        doc.url = req.body.url || null;
        doc.save(function(e, d) {
          if (e) return next(e);
```

It’s a rule to send back the updated object:

```
          res.status(200).json(d);
        });
      })
    } else {
      if (!anyAction) next('Something went wrong.');
    }

  } else {
    next('No post ID.');
  }
};
```

The full source code for the `hackhall/routes/posts.js` file is available at `https://github.com/azat-co/hackhall/blob/v3.1.0/routes/posts.js`.

This concludes coding the routes for the New Post page (see Figure 22-11), where users can create a post (e.g., a question).


localhost:3000/#posts/new
HackHall
Posts
People
Hi, test
Log Out
Submit
About | GitHub | Facebook | Twitter | AngelList

New Post

Title [Title]
Keep it short and meaningful.

URL [URL]
To start discussion or ask a question, leave URL blank.

Text [Text]
Leave the URL blank otherwise the Text will be ignored.

Made by Azat Mardan

Figure 22-11. *HackHall New Post page*

We're done with the routes files! Do you remember that HackHall is a true MVC application? Next, we'll cover the models.

Mongoose Models

Ideally, in a big application, we would break down each model into a separate file. Right now, in the HackHall app, we have them all in `hackhall/models/index.js`.

As always, our dependencies look better at the top:

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

This reference will be used for Mongoose data types:

```
var Schema = mongoose.Schema;
```

This array will be used as an enumerated type:

```
var roles = 'user staff mentor investor founder'.split(' ');
```

The Post model represents a post with its likes, comments, and watches. Each property in the schema sets a certain behavior for the property. For example, `required` means that this property is required, and `type` is the Mongoose/BSON data type.

■ **Tip** For more information on Mongoose, please consult its official documentation (`https://gumroad.com/l/mongoose`), *Practical Node.js* (Apress, 2014), and a new online course.

We define `Schema` with the `new` operand:

```
var Post = new Schema ({
```

Then, we have a `title` field that is required (`type` of `String`) and that is automatically trimmed of whitespace at the beginning and end:

```
  title: {
    required: true,
    type: String,
    trim: true,
```

The RegExp says "a word, a space, or any of the characters `,.!?`" and be between 1 and 100 characters in length":

```
    match: /^([\w ,.!?]{1,100})$/
  },
```

Then, we define `url` with the maximum of 1000 characters (that should be enough for long URLs, right?) and turn trimming on:

```
  url: {
    type: String,
    trim: true,
    max: 1000
  },
```

We define similar field properties for `text`:

```
  text: {
    type: String,
    trim: true,
    max: 2000
  },
```

comments is an array of comments for this post. Each comment object has a text and author. The author id is a reference to the User model:

```
comments: [{
  text: {
    type: String,
    trim: true,
    max:2000
  },
  author: {
    id: {
      type: Schema.Types.ObjectId,
      ref: 'User'
    },
    name: String
  }
}],
```

The post can be watched or liked by a user. These features are implemented by having arrays watches and likes with user IDs:

```
watches: [{
  type: Schema.Types.ObjectId,
  ref: 'User'
}],
likes: [{
  type: Schema.Types.ObjectId,
  ref: 'User'
}],
```

Next, we enter the author information and make each field in the nested object a required field:

```
author: {
  id: {
    type: Schema.Types.ObjectId,
    ref: 'User',
    required: true
  },
  name: {
    type: String,
    required: true
  }
},
```

Lastly, we add the time and date fields. It's good to have timestamps of events, such as when this post was created and when it was updated the last time. For this, we use `Date.now` as the default field. The `updated` property will be set by the pre save hook, or it can be set manually on each `save()`. (The pre save hook code is provided after this schema code.):

```
  created: {
    type: Date,
    default: Date.now,
    required: true
  },
  updated:  {
    type: Date,
    default: Date.now,
    required: true
  }
});
```

Going back to the `updated` field, to ensure that we don't have to set the timestamp manually each time we update (`save()`) the post, we utilize a pre save hook that checks if the field has been modified (has a new value). If it has not been modified, then we set it with a new date and time. This hook works only when you call `save()`; not when you use `update()` or a similar method. The callback has an asynch `next()` function, as you might see in Express.js middleware:

```
Post.pre('save', function (next) {
  if (!this.isModified('updated')) this.updated = new Date;
  next();
})
```

The `User` model can also serve as an application object (when `approved=false`). Let's define the schema as this:

```
var User = new Schema({
  angelListId: String,
```

The `Mixed` type allows us to store anything:

```
  angelListProfile: Schema.Types.Mixed,
  angelToken: String,
    firstName: {
    type: String,
    required: true,
    trim: true
  },
  lastName: {
    type: String,
    required: true,
    trim: true
  },
  displayName: {
    type: String,
    required: true,
    trim: true
  },
  password: String,
  email: {
    type: String,
    required: true,
    trim: true
  },
```

Roles are enum, because the value can be only one of the values from the roles array ([user, staff, mentor, investor, founder]):

```
  role: {
    type: String,
    enum: roles,
    required: true,
    default: roles[0]
  },
```

Here are some of the required boolean flags:

```
  approved: {
    type: Boolean,
    default: false
  },
  banned: {
    type: Boolean,
    default: false
  },
  admin: {
    type: Boolean,
    default: false
  },
```

Now a short bio statement:

```
  headline: String,
```

We won't store the photo binary, only the URL to it:

```
  photoUrl: String,
```

The angelList is a loose type that will have the AngelList profile:

```
  angelList: Schema.Types.Mixed,
```

It's good to keep logs to track when this document was created and when it was updated the last time (we set the time manually in the update() method of users.js):

```
  created: {
    type: Date,
    default: Date.now
  },
  updated:  {
    type: Date,
    default: Date.now
  },
```

We need some social media URLs:

```
  angelUrl: String,
  twitterUrl: String,
  facebookUrl: String,
  linkedinUrl: String,
  githubUrl: String,
```

We reference the IDs of posts that this user authored, liked, watched, and commented on as arrays of objects (they will be `ObjectIDs`):

```
  posts: {
    own: [Schema.Types.Mixed],
    likes: [Schema.Types.Mixed],
    watches: [Schema.Types.Mixed],
    comments: [Schema.Types.Mixed]
  }
});
```

For convenience, we apply the `findOrCreate` plug-in (`https://www.npmjs.org/package/mongoose-findorcreate`):

```
User.plugin(findOrCreate);
```

Mongoose plug-ins act like mini modules. This allows you to add extra functionality to your models. Another way to add extra functionality is to write your own custom method. Such a method can be static (attached to the entire category of entities) or an instance (attached to the specific model).

In routes, you've seen `findProfileById()` twice: once in `main.js` and once in `users.js`. To avoid duplication, the code was abstracted as a Mongoose static method of the `User` schema. It retrieves information, such as comments, likes, etc., which is why we have multiple nested Mongoose calls.

The `findProfileById()` method might look a bit complicated initially, but there is nothing difficult here—just a few nested database calls so that we can have full user information. This info includes, not only the username, e-mail address, and so forth, but all the posts, likes, watches, and comments made by the user. This information is used for gamification purposes on the Profile page to convert the number of comments, likes, and watches into points. But let's start with the first basic query and limit the fields that we request (to avoid leaking passwords and e-mail addresses):

```
User.statics.findProfileById = function(id, fields, callback) {
  var User = this;
  var Post = User.model('Post');

  return User.findById(id, fields, function(err, obj) {
    if (err) return callback(err);
    if (!obj) return callback(new Error('User is not found'));
```

After a user is found, we find the user's posts by using the `_id` and `displayName`. The fields option is set to `null` so we can pass other arguments, and the results are sorted by the creation date. In the callback we check for errors and exit (`callback(err)`) if there is an error.

```
Post.find({
  author: {
    id: obj._id,
    name: obj.displayName
  }
}, null, {
  sort: {
    'created': -1
  }
}, function(err, list) {
```

It's vital to handle errors on each nested callback:

```
  if (err) return callback(err);
  obj.posts.own = list || [];
```

Now that we have saved the list of this user's posts into `obj.posts.own`, the next query finds all the posts this user liked:

```
  Post.find({
    likes: obj._id
  }, null, {
```

The chronological order is ensured by `created`:

```
    sort: {
      'created': -1
    }
  }, function(err, list) {
    if (err) return callback(err);
```

In case, this user didn't like any of the posts, yet we account for that with an empty array:

```
    obj.posts.likes = list || [];
```

This query gets the posts that this user watches:

```
    Post.find({
      watches: obj._id
    }, null, {
      sort: {
        'created': -1
      }
    }, function(err, list) {
```

The err and list objects from the previous context are masked by this closure's err and list, but we don't really care. This style allows for variable name reuse:

```
        if (err) return callback(err);
        obj.posts.watches = list || [];
```

The last query finds posts where this user left comments:

```
        Post.find({
          'comments.author.id': obj._id
        }, null, {
          sort: {
            'created': -1
          }
        }, function(err, list) {
          if (err) return callback(err);
          obj.posts.comments = [];
```

After we get the list of posts in which this user left comments, there might be some posts where this particular user left more than one comment. For this reason, we need to go through that list of posts, and through each comment, and compare the author's ID with the user ID. If they match, then we include that comment into the list:

```
          list.forEach(function(post, key, arr) {
            post.comments.forEach(function(comment, key, arr) {
              if (comment.author.id.toString() == obj._id.toString())
                obj.posts.comments.push(comment);
            });
          });
```

Finally, we invoke the callback with proper data and null errors:

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

Lastly, we export the schema objects so they can be compiled into models in another file:

```
exports.Post = Post;
exports.User = User;
```

The full source code for hackhall/models/index.js is available at https://github.com/azat-co/hackhall/blob/v3.1.0/models/index.js.

Mocha Tests

One of the benefits of using REST API server architecture is that each route, and the application as a whole, become very testable. The assurance of the passed tests is a wonderful supplement during development—the so-called test-driven development approach introduced in Chapter 21.

The HackHall tests live in the `tests` folder and consist of:

- `hackhall/tests/application.js`: Functional tests for unapproved users' information
- `hackhall/tests/posts.js`: Functional tests for posts
- `hackhall/tests/users.js`: Functional tests for users

To run tests, we utilize a Makefile. I like to have various targets in a Makefile, because it gives me more flexibility. Here are the tasks in this example:

- `test`: Run all tests from the `tests` folder
- `test-w`: Rerun tests each time there is a file change
- `users`: Run the `tests/users.js` tests for user-related routes
- `posts`: Run the `tests/posts.js` tests for posts-related routes
- `application`: Run the `tests/application.js` tests for application-related routes

This is how the Makefile might look, starting with options for Mocha:

```
REPORTER = list
MOCHA_OPTS = --ui tdd
```

Then we define a task `test`:

```
test:
	clear
	echo Seeding **********************
	node seed.js
	echo Starting test **********************
	foreman run ./node_modules/mocha/bin/mocha \
	--reporter $(REPORTER) \
	$(MOCHA_OPTS) \
	tests/*.js
	echo Ending test
```

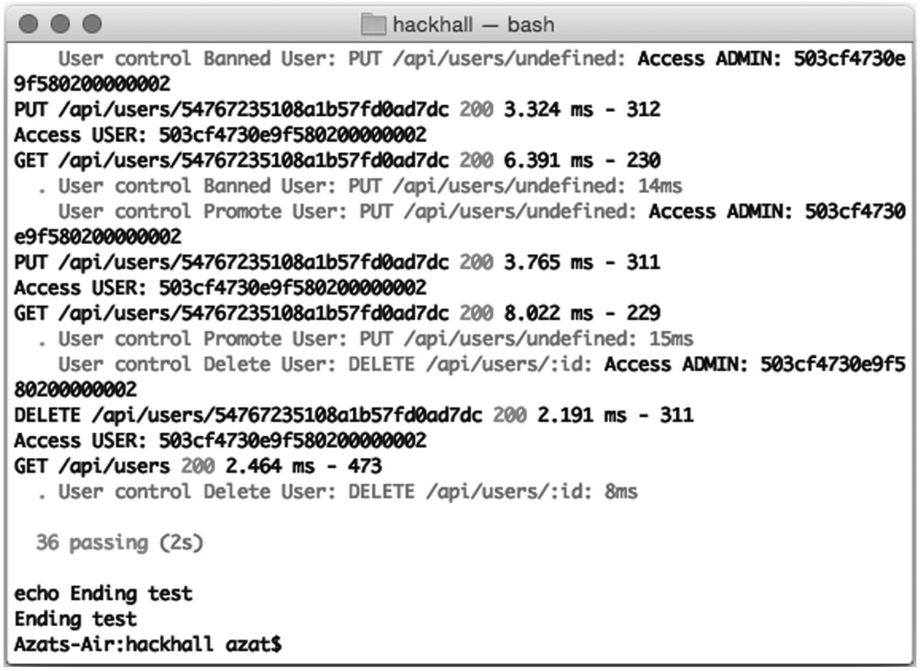
Similarly, we define other targets:

```
test-w:
	./node_modules/mocha/bin/mocha \
	--reporter $(REPORTER) \
	--growl \
	--watch \
	$(MOCHA_OPTS) \
	tests/*.js
```

```
users:
	clear
	echo Starting test *********************
	foreman run ./node_modules/mocha/bin/mocha \
	--reporter $(REPORTER) \
	$(MOCHA_OPTS) \
	tests/users.js
	echo Ending test

posts:
	clear
	echo Starting test *********************
	foreman run ./node_modules/mocha/bin/mocha \
	--reporter $(REPORTER) \
	$(MOCHA_OPTS) \
	tests/posts.js
	echo Ending test

application:
	clear
	echo Starting test *********************
	foreman run ./node_modules/mocha/bin/mocha \
	--reporter $(REPORTER) \
	$(MOCHA_OPTS) \
	tests/application.js
	echo Ending test

.PHONY: test test-w users posts application
```

Therefore, we can start tests with the `$ make` or `$ make test` command (to run the Makefile in the example, you have to have the `foreman` tool and `.env` variables).

All 36 tests should pass (as of this writing in HackHall v3.1.0), as shown in Figure 22-12.



```
    User control Banned User: PUT /api/users/undefined: Access ADMIN: 503cf4730e
9f580200000002
PUT /api/users/54767235108a1b57fd0ad7dc 200 3.324 ms - 312
Access USER: 503cf4730e9f580200000002
GET /api/users/54767235108a1b57fd0ad7dc 200 6.391 ms - 230
  . User control Banned User: PUT /api/users/undefined: 14ms
    User control Promote User: PUT /api/users/undefined: Access ADMIN: 503cf4730
e9f580200000002
PUT /api/users/54767235108a1b57fd0ad7dc 200 3.765 ms - 311
Access USER: 503cf4730e9f580200000002
GET /api/users/54767235108a1b57fd0ad7dc 200 8.022 ms - 229
  . User control Promote User: PUT /api/users/undefined: 15ms
    User control Delete User: DELETE /api/users/:id: Access ADMIN: 503cf4730e9f5
80200000002
DELETE /api/users/54767235108a1b57fd0ad7dc 200 2.191 ms - 311
Access USER: 503cf4730e9f580200000002
GET /api/users 200 2.464 ms - 473
  . User control Delete User: DELETE /api/users/:id: 8ms

  36 passing (2s)

echo Ending test
Ending test
Azats-Air:hackhall azat$
```

Figure 22-12. *Results of running all Mocha tests*

Tests use a library called superagent (https://npmjs.org/package/superagent; GitHub: https://github.com/visionmedia/superagent). The tests are similar in concept to those for the REST API in Chapter 21. We log in, then make a few requests, while checking for their correct response.

For example, this is the beginning of hackhall/tests/application.js in which we have a dummy user object with a hashed password (bcrypt.hashSync()):

```
var bcrypt = require('bcryptjs');

var user3 = {
  firstName: 'Dummy',
  lastName: 'Application',
  displayName: 'Dummy Application',
  password: bcrypt.hashSync('3', 10),
  email: '3@3.com',
  headline: 'Dummy Application',
  photoUrl: '/img/user.png',
  angelList: {blah:'blah'},
  angelUrl: 'http://angel.co.com/someuser',
  twitterUrl: 'http://twitter.com/someuser',
  facebookUrl: 'http://facebook.com/someuser',
  linkedinUrl: 'http://linkedin.com/someuser',
  githubUrl: 'http://github.com/someuser'
}
```

```
var app = require ('../server').app,
  assert = require('assert'),
  request = require('superagent');
```

We start the server:

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

The next line will store the client object so that we can log in and make authorized requests as that user:

```
var user1 = request.agent();
var port = 'http://localhost:'+app.get('port');
var userId;
```

We use the admin user that was created by `seed.js`:

```
var adminUser = {
  email: 'admin-test@test.com',
  password: 'admin-test'
};
```

Next, we create a test suite:

```
suite('APPLICATION API', function (){
```

This is a test suite preparation (empty at the moment):

```
  suiteSetup(function(done){
    done();
  });
```

The following is the first test case definition with the POST call to `/api/login`:

```
  test('log in as admin', function(done){
    user1.post(port+'/api/login').send(adminUser).end(function(res){
        assert.equal(res.status,200);
      done();
    });
  });
```

Let's check that we can get protected resource `/api/profile`:

```
  test('get profile for admin',function(done){
    user1.get(port+'/api/profile').end(function(res){
        assert.equal(res.status,200);
      done();
    });
  });
  test('submit application for user 3@3.com', function(done){
```

Here, we create a new application for membership using `user3` data and a hashed password:

```
    user1.post(port+'/api/application').send(user3).end(function(res){
      assert.equal(res.status,200);
      userId = res.body._id;
      done();
    });
  });
```

Then, we log out `user1` and check that we've logged out:

```
  test('logout admin',function(done){
    user1.post(port+'/api/logout').end(function(res){
        assert.equal(res.status,200);
      done();
    });
  });
  test('get profile again after logging out',function(done){
    user1.get(port+'/api/profile').end(function(res){
        assert.equal(res.status,500);
      done();
    });
  });
```

Now, we try to log in as `user3` by using a plain password, as though it was entered on the web page (system will hash it to compare with the hashed password):

```
  test('log in as user3 - unapproved', function(done){
    user1.post(port+'/api/login').send({email:'3@3.com', password:'3'}).end(function(res){
        assert.equal(res.status, 200);
      done();
    });
  });

...
```

Assuming you got the general idea from this test case, there's no need to list all the mundane test cases. Of course, you can get the full content for `hackhall/tests/application.js`, `hackhall/tests/posts.js`, and `hackhall/tests/users.js` at `https://github.com/azat-co/hackhall/tree/v3.1.0/tests`.

■ **Caution** Do not store plain passwords/keys in the database. Any serious production app should, at least, salt the passwords[1] before storing them. Use `bcryptjs` instead!

---

[1]`https://crackstation.net/hashing-security.htm`

By now, you should be able to run the application and tests locally (either by copying from the book or downloading the code). If you get the API keys, you should be able to log in with AngelList and GitHub, as well as send/receive e-mails with SendGrid. At the bare minimum, you should be able to log in locally using the e-mail and password that you specified in your database seeding script.

Summary

Now you know all the tips and tricks used in building HackHall, which includes important, real production application components, such as REST API architecture, OAuth, Mongoose and its models, MVC structure of Express.js apps, access to environment variables, and so on.

As mentioned in this chapter, HackHall is still in active development, so the code will continue to evolve. Make sure you follow the repository on GitHub. You can visit the live HackHall.com app and join the community by applying for membership! And, of course, you can make a contribution by submitting a pull request.

This chapter concludes our journey into Express.js and related web development topics. The task of covering an evolving framework is a difficult one, akin to shooting at a moving target, so my goal in this chapter was to get you the most up-to-date information and, most importantly, to show you some of the more fundamental aspects, such as code organization. I also put a lot of energy into explaining and repeating examples of the middleware pattern. If you are under the pressure of a deadline, or simply prefer just-in-time learning (learn when you need it vs. learn for the future), then you will find plenty of code to copy and paste into your own project. I know that building your own project is more fun than borrowing another abstract application from a tutorial.

I hope you have enjoyed the examples and the book! I want to hear from you via Twitter (`@azat_co`) and e-mail (`hi@azat.co`). The appendices that follow will serve as a reference. And don't forget to claim your two-page print-ready Express.js 4 cheat sheet (the link to download it is in Appendix C).