7. Building Web Applications with Express

Thus far, you have been learning the fundamentals and core concepts of Node.js; armed with these ideas, you have built some simple applications, although you've had to write a lot of code to do some reasonably basic things. It's time to change gears a bit and start building more interesting web applications, using one of the biggest strengths of Node: the huge collection of libraries and modules available to you through npm. Now that you understand the core workings of Node and modules, it's time to start finding ways to make your life significantly easier and your development much faster.

In this chapter, you start working with *express*, a web application framework for Node. It takes care of many of the basic tasks you learned about in previous chapters and even exposes some powerful new functionality that you can take advantage of in your apps. I show you how to build apps with express and update your photo album app to use it. Then you look a bit more at application design, learn how to expose the API from your JSON server, and explore some new functionality you haven't seen before.

Installing Express

In this chapter, you are going to slowly change the way you install applications via npm. You might expect that, to install express, you'd just type

```
npm install express
```

and this would work just fine. However, it always installs the latest version of the express node module, which you might not want when you're deploying your application to live production servers. When you develop and test against a particular version, it's good to have that same version run on the live servers. Only after you update to a newer version and fully test against that version do

you want to use it on the live servers.

So, the basic methodology for building Node applications is now as follows:

1. Create an application folder.
2. Write a *package.json* file describing the package and stating which npm modules are needed.
3. Run npm install to make sure these modules are installed properly.

So, create a new folder called *trivial_express/* for your express test application and then put the following into *package.json* in that folder:

Click here to view code image

```
{
 "name": "Express-Demo",
 "description": "Demonstrates Using Express and Connect",
 "version": "0.0.1",
 "private": true,
 "dependencies": {
  "express": "4.x"
 }
}
```

The name and description fields are reasonably self-explanatory, and the app is just a toy right now, so it has an appropriate version number. As you saw in Chapter 5, "Modules," you can set private to true, which tells npm never to try installing it in the live npm registry because it is just a test project for your own use.

After creating the file, you can run the following to read in the dependencies section and install all the appropriate modules:

```
npm install
```

Because express itself depends on many different modules, you see a decent

amount of output before it exits, but then your *node_modules/* folder should be ready to go. If you ever make a change to your *package.json*, you can just run npm update to make sure all the modules are correctly installed.

Hello World in Express

Recall from Chapter 1, "Getting Started," that your first web server in Node.js was roughly like the following:

Click here to view code image

```
var http = require("http");

function process_request(req, res) {
  res.end("Hello World");
}

var s = http.createServer(process_request);
s.listen(8080);
```

Well, in express, it looks quite similar:

Click here to view code image

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

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

app.listen(8080);
```

Save that, run it, and then either run curl localhost:8080 or enter *localhost:8080* in the web browser, and you should get the hello world output you expect. One of the great things about express is that it *is built on top of everything you have seen so far*, so the HTTP request and response objects have the same methods

and properties they had before. To be certain, they have many new things added that make them more interesting and powerful, but you are not operating in a completely new world here—just an extended version of the one with which you are already familiar and comfortable!

Routing and Layers in Express

Express implements a system known as *middleware* that provides a collection of common functions that network applications might need. While previous versions of Express used a Node module called *connect* to implement this, versions starting with 4.0 no longer do so—it is now built in to Express.

At the heart of this middleware model is the concept of a flow of functions, or what some people like to describe as a *layered* set of functions.

Middleware: What's in a Name?

When you run into the term *middleware* for the first time, you can be forgiven for being worried that you've missed out on some key new concept or technology shift. Fortunately, it's nothing so complicated.

Effectively, middleware originally was used to describe software components (typically on the server side of things) that connected two things together, such as a business logic layer and a database server, or storage services with application modules, and so on.

Over time, however, the term seems to have become more generic and broad and now seems to cover any piece of software that connects two things together in some way, shape, or form.

For the purposes of this book, the middleware exposed by *connect* that you see in *express* is basically just a collection of modules that allow your web apps to seamlessly integrate func-

tionality from either the server or browser.

You set up your application by creating a sequence of functions (layers) through which express navigates. When one decides that it wants to take over, it can complete the processing and stop the sequence. Express then walks back up through the sequence to the end (some layers do their process "on the way in"; others "on the way out").

So, in the preceding express application, there is only one layer in the sequence:

Click here to view code image

```
app.get('/', function(req, res){ });
```

This function, which is a helper provided by express, processes the incoming request (by calling the provided function) if the following are both true:

- The HTTP request method is GET.
- The requested URL is /.

If the incoming request does not match these criteria, the function is not called. If no function matches the requested URL—for example, browse or curl to /some/other/url—express just gives a default response:

```
Cannot GET /some/other/url
```

Routing Basics

The general format of the URL routing functions is as follows:

Click here to view code image

```
app.method(url_pattern, optional_functions, request_handler_function);
```

The preceding route handler is for handling GET requests for the URL /. To support handling a POST request on a particular URL, you can use the post method

on the express application object:

Click here to view code image

```
app.post("/forms/update_user_info.json", function (req, res) { ... });
```

Similarly, express supports DELETE and PUT methods, both of which you use in the photo-sharing application, via the delete and put methods, respectively. Alternately, you can use the all method to indicate that the routing function should accept the given URL with any given HTTP method.

Click here to view code image

```
app.all("/users/marcwan.json", function (req, res) { ... });
```

The first parameter to the routing function is a regular expression to match an incoming request URL. It can be simple as follows:

```
/path/to/resources
```

Or it can contain regular expression features, such as

```
/user(s)?/list
```

which matches both /users/list and /user/list, while

```
/users/*
```

matches anything starting with /users/.

One of the most powerful features of routing is the ability to use *placeholders* to extract named values from the requested route, marked by the colon (:) character. When the route is parsed, express puts the matched placeholders on the req.params object for you. For example:

Click here to view code image

```
app.get("/albums/:album_name.json", function (req, res) {
```

```
  res.end("Requested " + req.params.album_name);
});
```

If you call the preceding app requesting */albums/italy2012.json*, you get the following output:

```
Requested italy2012.
```

You can put multiple parameters in the same URL as well:

Click here to view code image

```
app.get("/albums/:album_name/photos/:photo_id.json", function (req, res) { ... );
```

If your function is called, req.params has both album_name and photo_id set with the appropriate incoming values. Placeholders match any sequence of characters except for forward slashes.

The function you provide to the routing method is given the request and response object, as you have seen. However, it is also given a third parameter, which you are free to use or ignore. It is usually labeled next (the flow of the layered functions is always continued through functions called next), and gives you the opportunity to do some additional examinations on the incoming URL and still choose to ignore it, as follows:

Click here to view code image

```
app.get("/users/:userid.json", function (req, res, next) {
  var uid = parseInt(req.params.userid);
  if (uid < 2000000000) {
    next();          // don't want it. Let somebody else process it.
  } else {
    res.end(get_user_info(uid));
  }
});
```

Updating Your Photo Album App for Routing

The photo album app is quite easy to adapt to express. You have to do only a few things to get it working:

- Create a *package.json* and install *express*.
- Replace the http module's server with that of *express*.
- Replace the handle_incoming_request function with routing handlers.
- Update the way you get query parameters in *express* (they are placed on req.query for your convenience).

Now copy the last version of the photo app you created at the end of Chapter 6, "Expanding Your Web Server," to a new folder (you can call it basic_routing/) and place the following *package.json* inside:

Click here to view code image

```
{
 "name": "Photo-Sharing",
 "description": "Our Photo Sharing Application",
 "version": "0.0.2",
 "private": true,
 "dependencies": {
  "async": "2.x",
  "express": "4.x"
 }
}
```

You can run npm install to make sure express and async are installed to *node_modules/*. You can then change the top of the *server.js* file to use express instead of the http module:

Click here to view code image

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

```
var app = express();

var path = require("path"),
  async = require('async');
  fs = require('fs');
```

And you change

```
http.listen(8080);
```

to

```
app.listen(8080);
```

Next, you have to replace the handle_incoming_request function, which looks as follows:

Click here to view code image

```
function handle_incoming_request(req, res) {
  req.parsed_url = url.parse(req.url, true);
  var core_url = req.parsed_url.pathname;

    // test this fixed url to see what they're asking for
  if (core_url.substring(0, 7) == '/pages/') {
    serve_page(req, res);
  } else if (core_url.substring(0, 11) == '/templates/') {
    serve_static_file("templates/" + core_url.substring(11), res);
  } else if (core_url.substring(0, 9) == '/content/') {
    serve_static_file("content/" + core_url.substring(9), res);
  } else if (core_url == '/albums.json') {
    handle_list_albums(req, res);
  } else if (core_url.substr(0, 7) == '/albums'
        && core_url.substr(core_url.length - 5) == '.json') {
    handle_get_album(req, res);
  } else {
    send_failure(res, 404, invalid_resource());
```

```
    }
}
```

You replace it with these routing functions:

Click here to view code image

```
app.get('/albums.json', handle_list_albums);
app.get('/albums/:album_name.json', handle_get_album);
app.get('/content/:filename', function (req, res) {
  serve_static_file('content/' + req.params.filename, res);
});
app.get('/templates/:template_name', function (req, res) {
  serve_static_file("templates/" + req.params.template_name, res);
});
app.get('/pages/:page_name', serve_page);
app.get('*', four_oh_four);

function four_oh_four(req, res) {
  send_failure(res, 404, invalid_resource());
}
```

Not only do the routing functions seem much cleaner and simpler than the previous function you had, but it is much easier to see the URLs that will be matched and how the app will function. You added a new route matching "*" so that all other requests will be given a 404 response.

There is one problem with the routing functions right now, however. What happens if the user asks *for /pages/album/italy2012*? Currently, the app will fail: the regular expression matching for parameters never includes forward slash (/) characters, so the route /pages/:page_name does not match. To solve this problem, you must add the extra route:

Click here to view code image

```
app.get('/pages/:page_name/:sub_page', serve_page);
```

Now all the pages will route to the correct function, and you have also set yourself up to quickly identify the subpage being requested.

One of the nice things about the app now is that you no longer need to parse the incoming requests or parameters; express does all this for you. So, whereas before you had to include the *url* module to parse the incoming URL, it's now ready for you right when you get to the routing functions. Similarly, you never need to parse GET query parameters because they are also prepared for you in advance. So you can update the small helper functions as follows:

Click here to view code image

```
function get_album_name(req) {
  return req.params.album_name;
}
function get_template_name(req) {
  return req.params.template_name;
}
function get_query_params(req) {
  return req.query;
}
function get_page_name(req) {
  return req.params.page_name;
}
```

With those changes, the photo-sharing app should be updated and ready to go with express. It should behave exactly like the old one.

REST API Design and Modules

When you are designing JSON servers such as the one you are writing, it is important to spend some time thinking about the API and how clients will use it. Spending some time up front to design a good API helps you to think about how people will use the application and also helps you organize and refactor the code

to reflect this conceptual understanding.

API Design

For the photo-sharing JSON server, you are developing what's called a RESTful JSON server. The word REST comes from *Representational State Transfer*, and basically implies that you can request an accurate representation of an object from the server. REST APIs focus on four core operations (which, coincidentally, map to four HTTP request methods):

- Creation (PUT)
- Retrieval (GET)
- Updating (POST)
- Destruction (DELETE)

Some people refer to these operations as *CRUD*, and pretty much everything you do in your API centers around doing one of these things to objects, whether they are albums, photos, comments, or users. Although it is not possible to design the perfect API, I will do my absolute best to help you design one that is at least extremely intuitive and doesn't leave the client authors scratching their heads as to what you intended. Here are some principles to follow when designing RESTful interfaces:

- There are two basic kinds of URLs. Most URLs you design will be a variation on these two:
  1. Collections—for example, /albums.
  2. Specific items within these collections—for example, /albums/italy2012.
- Collections should be nouns, specifically plural nouns, such as *albums*, *users*, or *photos*.
  - PUT /albums.json—The HTTP request body contains the JSON with the data for the new album.

- You fetch or update an object by specifying the specific instance on the collection with GET or POST, respectively:
  - GET /albums/italy2012.json—The request returns this album.
  - POST /albums/italy2012.json—The HTTP request body contains the JSON with the new data for this album.
- You destroy objects with DELETE.
  - DELETE /albums/italy2012.json—Destroys this album.
- If you have collections off collections, for example, photos associated with albums, you just continue the pattern:
  - GET /albums/italy2012/photos.json—Returns all the photos in this album.
  - PUT /albums/italy2012/photos.json—Adds a new photo to this collection.
  - GET /albums/italy2012/photos/23482938424.json—Fetches the photo with the given ID.
- Slight changes to the way in which you get data from these collections, such as pagination or filtering, should all be done via GET query parameters:
  - GET /albums.json?located_in=Europe—Fetches only those albums in Europe.
  - GET /albums/japan2010/photos.json?page=1&page_size=25—Fetches the given page of 25 photos for the japan2010 album.
- You assign a version number to the API so that if you want to make major changes to it in future versions that would not be compatible, you can simply update the version number. You prefix the API URLs with */v1/* for this first version.
- Finally, you suffix all your URLs that return JSON with *.json* so that

clients know what data will be in the responses. In the future, you could also add support for *.xml* or whatever other formats you want.

With this in mind, the new API for the photo-sharing application centers around albums. It looks as follows:

```
/v1/albums.json
/v1/albums/:album_name.json

/pages/:page_name
/templates/:template_name
/content/:filename
```

Why don't you put version numbers on the static content URLs? Because they're not part of the JSON server interface, but more helpers to provide the contents for the web browser client. Users always get the latest version of this and thus get code that knows how to use the correct version of the API.

It should be trivial for you to update the server.js file with this API design change. Don't forget to also update the bootstrapping JavaScript files in /content/ with the new URLs as well! Give that a try before continuing onward.

Modules

With the APIs updated around the new REST interface, you can take advantage of some of the clarity you have given to the application. You now have clear functionality boundaries around albums and pages, so you can put those areas of functionality into their own modules.

Create a subfolder of the application folder called *handlers/* and put the new modules in there. For albums, create a file in that folder called albums.js and move the four album manipulation functions into that file. You can view the source for this project on GitHub under the folder named *handlers_as_modules/*. Effectively, all you are doing is taking the code for the functions handle_get_album and handle_list_albums and putting them, along with their accompanying