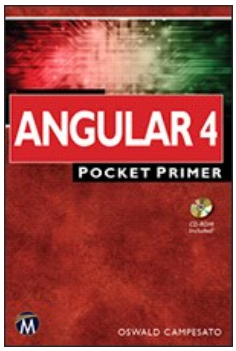

Angular 4 Pocket Primer

by Oswald Campesato

Chapter 6: Angular and Express

Overview

This chapter contains Angular applications that work in conjunction with server-side technologies that are often referred to as the "Node stack." You will learn how to create Angular applications that use `Express` on the server and the NoSQL database Mongo. You will also get an introduction to microservices in this chapter. However, this chapter does not cover the introductory-level material for the Node itself, which is a prerequisite for this chapter (many online tutorials are available).

The first section shows you how to set up a simple JavaScript Object Notation (JSON)-based server (called `json-server`) that you can use with many client-side applications (not just Angular). The second section introduces you to basic `Express`-based applications that can process client-side data requests.

The third section takes a slight detour by introducing you to microservices, along with an Angular application that contains the `forkJoin()` method to simulate a very simple example of microservices. The server-side code consists of three `Express` applications that listen for client-side requests on three different ports.

The fourth section combines an `Express`-based application and an Angular application that issues a data request to the `Express` application. The data request retrieves a set of users from a `Mongo` instance and then returns that list of users to the Angular application. In addition, this code sample uses a technique that you saw in Chapter 4 for returning `JSON`-based data from the server that is converted to a `Promise`, which in turn is converted into an `Observable` that is processed inside the Angular application.

A Minimalistic Node Application

In Chapter 4 you learned how to create Angular applications that can issue `HTTP GET` and `HTTP POST` requests to a file server using `json-server`. The advantage of using `json-server` is "zero configuration," which is convenient for creating prototypes and demos. In this section you will see how to create a minimalistic code sample using `Express`, which requires only the file `server.js`.

DVD Copy the directory `SimpleNode` from the companion disc to a convenient location. This directory contains all the files for this section. However, if you want to perform the steps yourself, the next section explains how to do so. If you want to use the existing code, skip the following setup section.

Set Up a Node Environment (Optional)

After installing NodeJS on your machine, navigate to a convenient directory and enter the following command:

```
npm init –y
```

The second step is to install `Express` (which we need in the next section) with the following command:

```
npm install express –save
```

After the preceding command has completed its execution, the current directory will contain the file `package.json` whose contents are similar to the following:

```
{
  "name": "application-name",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.13.4"
  }
}
```

Create an Express Application

DVD In case you haven't already done so, copy the directory `SimpleNode` from the companion disc to a convenient location. Listing 6.1 displays the contents of `server.js`, which is a simple `Express`-based application.

Listing 6.1: server.js

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

// send a message for the root path ("/"):
app.get('/', function(req, res){
```

```
    res.send('hello world');
});


// send a message for the path /pasta:
app.get('/pasta', function(req, res){
    res.send('hello pasta');
});

console.log("Listening on port 3000");
app.listen(3000);
```

Listing 6.1 contains a `require` statement and then a code snippet to initialize the `app` variable. There are two defined routes: The first route is the default route "`/`" and the second route is "`/pasta`," which displays a corresponding message when users navigate to the `URL` `localhost:3000/pasta`.

Launch the Express Application

Before launching the application, you must install the Node-related dependencies with this command:

```
npm install
```

When the preceding step has completed, invoke the following command:

```
npm start
```

This command will display the following output:

```
Listening on port 3000
```

Now, launch a browser session and navigate to `localhost:3000/pasta`, and if everything was set up correctly, you will see the string `hello pasta` in your browser.

An Application with Angular and Express

In Chapter 4 you saw how to use the `forkJoin()` method in the `Observable` class to issue multiple `HTTP` requests concurrently and then aggregate the results of those requests. This section shows you how replace the `json-server` file server in Chapter 4 with an `Express`-based application, and how to aggregate multiple responses from that Express application. (In a later section you will see how to serve data from multiple Express-based applications.)

The code sample has two parts: The first part shows you how to create a `Node`-based application that returns JSON-based data about individual students (based on the student `id` that is sent to the server). The second part contains an Angular application that uses the `forkJoin()` method to send multiple concurrent requests to the server.

Hence, you need two command shells to do the following:

```
invoke node server.js (from NodeForkJoin/server)
invoke ng serve       (from NodeForkJoin/src)
```

Starting the Server and the Angular Application

DVD Copy the directory called `NodeForkJoin` from the companion disc into a convenient location. Let's start by launching the Express-based application, and then we'll launch the Angular application.

Navigate to the `src/server` directory that contains the file `package.json` (discussed in the next section) and `server.js`. Perform the following step to install the necessary code:

```
npm install
```

Now launch the Express-based application with this command:

```
node server.js
```

This command displays the following output:

```
Listening on port 3000
```

Next, launch the Angular application by opening another command shell, navigating to the `NodeForkJoin/src` directory, and typing the following command:

```
ng serve
```

Navigate to the URL `localhost:4200;` if everything is working correctly, you will see the following text:

Angular HTTP and Observables

A List of Students

- John Smith
- Dave Stone
- Miko Mason

Now that the Angular application is running correctly, the next two sections examine the files `package.json` and `server.js` for the Express-based application, followed by a section that discusses the Angular code.

The Server Code: package.json

Listing 6.2 displays the contents of `package.json` that contains a single dependency that is required for the `Node` application for this section.

Listing 6.2: package.json

```
{
  "name": "application-name",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.13.4"
  }
}
```

The Server Code: server.js

Listing 6.3 displays the contents of `server.js`, which is the `Node` application for this section.

Listing 6.3: server.js

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

var students = {
                  "1100":    {"fname": "John",      "lname":"Smith"},
                  "1200":    {"fname": "Jane",      "lname":"Jones"},
                  "1300":    {"fname": "Dave",      "lname":"Stone"},
                  "1400":    {"fname": "Miko",      "lname":"Mason"},
                  "1500":    {"fname": "Yuki",      "lname":"Smith"}
               }

// add CORS support to allow cross-domain requests
app.use(function(req, res, next) {
  res.header("Access-Control-Allow-Origin",  "*");
  res.header("Access-Control-Allow-Headers", "Origin,
                    X-Requested-With, Content-Type, Accept");
  next();
});

// send a message for the root path ("/"):
app.get('/', function(req, res){
    res.send('hello world');
});

// get individual student:
app.get('/json/:id', function(req, res){
    studentid = req.params.id;
    res.send(students[studentid]);
});
```

```
console.log("Listening on port 3000");
app.listen(3000);
```

Listing 6.3 contains the `students` array with `JSON`-based data for convenience. In a realistic application, you would retrieve such data from a `Mongo` database (or some other data store).

Listing 6.3 contains three routes, the first of which is the "`/`" default route, which returns a `hello world` message. The second route handles requests that contain an `id` value, which is an exercise for you.

The third route also supports an `id` parameter that is used as an index for the `students` array. The information about the user that is associated with the specified `id` is returned to the client with this code snippet:

```
res.send(students[studentid]);
```

As you can see, Listing 6.3 serves an individual request. The next section shows you how to issue multiple concurrent requests to retrieve student-related information.

The Angular Code

Listing 6.4 displays the contents of `app.component.ts` that queries the `Express`-based server for multiple students and then displays the first name and last name of those students.

Listing 6.4: app.component.ts

```
import { Component }      from '@angular/core';
import { Http }           from '@angular/http';
import { Observable }     from 'rxjs/Observable';
import 'rxjs/Rx';
import 'rxjs/add/operator/map'

@Component({
  selector: 'app-root',
  template:'
  <h2>Angular HTTP and Observables</h2>
  <h3>A List of Students</h3>
  <ul>
    <li *ngFor="let student of students">
      {{student.fname}} {{student.lname}}
    </li>
  </ul>
  '
})
export class AppComponent {
  students = [];
  studentids:number[] = [1100,1300,1400];

  constructor(private http:Http) { }

  ngOnInit() {
    this.getStudents();
  }
  getStudents() {
    // map them into a array of observables and forkJoin
    Observable.forkJoin(
      this.studentids.map(
        id => this.http.get('http://localhost:3000/json/'+id)
                  .map(res => res.json())
      ))
      .subscribe(data => {
         this.students = data;
         console.log("subscribe students = "+data);
      })
  }
}
```

Listing 6.4 contains a `template` property that displays an unordered list of student-related information by iterating through the `students` array. The `AppComponent` class in Listing 6.4 initializes `students` as an empty array and `studentids` as an array with four integers. The `AppComponent` class also contains a constructor that initializes an instance of the `Http` class.

Notice the `ngOnInit()` lifecycle method. This method invokes the `getStudents()` method, which is also defined in the `AppComponent` class.

Next, the `getStudents()` method invokes the `forkJoin` method of the `Observable` class to aggregate the result of issuing multiple concurrent `HTTP GET` requests. These multiple requests are made by using the `map()` method to iterate through the values in the `studentids` array. The last line of code in the `getStudents()` method invokes the `subscribe()` method to initialize the contents of the `students` array with the data retrieved from the server. In this example, the `studentids` array contains three values, and therefore the `students` array will contain `JSON`-based data for three students.

The next section contains a generalized version of the code sample in this section: You will learn how to create an Angular application that makes HTTP requests to multiple endpoints to retrieve and aggregate data from those endpoints.

Concurrent Requests and Angular (Version 2)

The code sample in the previous section creates a Node application that serves data requests from an Angular application via the `forkJoin()` method. This method makes multiple concurrent requests and then aggregates the responses. This example also launches `Express` applications that listen on three different port numbers: `4500`, `5500`, and `6500`.

DVD Copy the directory `NodeForkJoin2` from the companion disc into a convenient location so that you can view the code that is discussed in the following subsections.

How to Start the Express Server Code

The `NodeForkJoin2` directory contains the subdirectory `server`, which includes a configuration file `package.json`, four simple `Express` applications, and a simple shell script (for your convenience) for launching the `Express` applications that will serve data to the Angular application. The list of files in the `server` subdirectory is as follows:

```
package.json
server.js
server4500.js
server5500.js
server6500.js
start.sh
```

The file `server.js` is a generic Express application that responds to requests from a given port number. In addition, `server.js` was copied to the three JS files `server4500.js`, `server5500.js`, and`server6500.js`. Note that each of these JS files serves data from their respective port numbers (and nothing more). Thus, the files `server4500.js`, `server5500.js`, and `server6500.js` will "listen" on ports `4500`, `5500`, and `6500`, respectively.

Navigate to the `server` subdirectory (as you did in the previous section) and launch the following command to install the Express-related code:

```
npm install
```

The shell script `start.sh` contains `node` commands that launch the three `Express` applications, and its contents are shown here:

```
node server4500.js &
node server5500.js &
node server6500.js &
```

You can invoke the three preceding commands manually, or you can invoke the following command to launch the three `Express` applications in the background:

```
./start.sh
```

Launch a browser and navigate to each of the three preceding port numbers (on `localhost`) to confirm that they are responding correctly.

At this point, you are ready to look at the Angular code for this application, which is discussed in the next section.

The Angular Code

Listing 6.5 displays the contents of the file `app.component.ts` that requests data from each of the three `Express` applications that you launched in the previous section.

Listing 6.5: app.component.ts

```
import {Component}      from '@angular/core';
import {Http}           from '@angular/http';
import {Observable}     from 'rxjs/Observable';
import 'rxjs/Rx';
import 'rxjs/add/operator/map'
```

```
@Component({
  selector:  'app-root',
  template:'
  <h2>Angular HTTP and Observables</h2>
  <h3>A List of Students</h3>
  <ul>
    <li *ngFor="let student of students">
      {{student.fname}} {{student.lname}}
    </li>
  </ul>
  '
})
export class AppComponent {
  students = [];

  studentids:any[] = [{url:"http://localhost:4500", sid:1100}
                      {url:"http://localhost:5500", sid:1300}
                      {url:"http://localhost:6500",
sid:1400}];

  constructor(private http:Http) { }

  ngOnInit() {
    this.getStudents();
  }

  getStudents() {
    // map them into a array of observables and forkJoin
    Observable.forkJoin(
      this.studentids.map(
        loc => this.http.get(loc.url+'/json/'+loc.sid)
                  .map(res => res.json())
      ))
      .subscribe(data => {
         this.students = data;
         console.log("subscribe students = "+data);
      })
  }
}
```

Listing 6.5 is similar to Listing 6.4, and the modified code is shown in bold (the original code is kept intact for comparison purposes). The key idea is to specify a `URL` with a port number and a student id for each request. The modified `forkJoin()` command in the `getStudents()` method in Listing 6.5 shows you how to construct each `URL` to issue an HTTP request.

Now consider the following generalization: Specify multiple external Web pages that contain information that you want to aggregate in a Web browser, which is an example of combining Angular and microservices.

The next portion of this chapter contains examples of Angular applications that issue requests to `Express`-based applications that use `MongoDB` as a data store as well as `Jade` templates. If you are unfamiliar with `Jade`, you can replace `Jade` with `EJS` or another templating engine of your preference. Although Jade has been deprecated and replaced by Pug, you will find numerous online code samples that use Jade, and this section will help you understand those code samples. As a side point, if you understand how to use Jade, which many people feel is somewhat difficult, other templating engines will probably be much easier for you to learn.

If you are new to `MongoDB`, please read an online tutorial that discusses `MongoDB`. Then you will be ready for the next section, which shows you how to set up an `Express`-based application that connects to an instance of a `MongoDB` database.

An Express Application with MongoDB (Part 1)

This section (part 1) explains how to set up an Express application with Mongo, followed by a section (part 2) that discusses the files in the Express application.

DVD Make sure that you have installed `Mongo`, `node`, and `nodemon` on your machine before you attempt to launch the application in this section. Next, copy the directory `ExpressMongoJade` from the companion disc to a convenient location. This directory contains the server-side code that will process requests from the Angular application `NG2NodeApp`, which is discussed later in this section.

Here are the steps that you must perform to launch the existing `Express` application that accesses data in a `MongoDB` database:

1. Install Mongo on your machine.
2. Launch `mongod`.

3. Launch `cd ExpressMongoJade.`
4. Launch mongo load-newusers.js.
5. Launch npm install.
6. Launch npm start.
7. Navigate to `localhost:3000.`

After you have installed Mongo, add the `bin` directory to the `PATH` environment variable in a command shell, and then start the `mongod` daemon process:

```
mongod
```

You can also specify a different directory for the data files:

```
mongod --dbpath <full-path-to-a-directory>
```

Step 4 populates a Mongo database with seed data. Step 5 is required if you do not already have the `node_modules` directory (which exists in the sample code), and Step 6 launches the Express application (by executing the `script` bin/www).

If everything went well, you will see the following text in a browser session:

Express

Welcome to Express

The URL `localhost:3000/userlist` displays the current list of users in the Mongo database, an example of which is shown here:

List of Users

- 0 Janice | Smith
- 1 Steven | Stone
- 2 Yuki | Tanaka
- 3 Hideki | Hiura
- 4 Himiko | Yamamoto

Now let's take a look at the contents of `package.json` (displayed in Listing 6.6), wich lists all the dependencies for this application.

Listing 6.6: package.json

```
{
  "name": "expressmongojade",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "body-parser": "~1.17.1",
    "cookie-parser": "~1.4.3",
    "debug": "~2.6.3",
    "express": "~4.15.2",
    "jade": "~1.11.0",
    "mongodb": "^2.2.26",
    "monk": "^4.0.0",
    "morgan": "~1.8.1",
    "serve-favicon": "~2.4.2"
  }
}
```

As you can see, Listing 6.6 specifies various Node modules, such as `express`, `jade` (a templating engine), and `mongo`. Note that the version numbers will probably change in some cases, as well as the version of `node` and `npm` on your machine.

The Server-Side Custom Files (Part 2)

Before delving into the contents of various Angular-related files for the project that was introduced in the previous section, let's take a brief tour

of the names of some important files and their purpose.

As a starting point, the file `app.js` (in Listing 6.7) contains the `Node` application code. This code contains middleware, a reference to an instance of a `Mongo` database, a reference to the `views` directory, and route definitions in the routes subdirectory. You launch `app.js` via this command:

```
npm start
```

Whenever you issue a valid `HTTP GET` request, that request is handled via a three-step process: The first step involves the appropriate code in the Express application (defined in `app.js`), which invokes an appropriate block of code in `routes/index.js` as the second step, and then data is passed to a `Jade` template as the third step. At the end of this process, an HTML page will be generated and returned to the client that made the initial request.

By convention, the Jade templates for this application are located in the `views` subdirectory. For this example, the Jade templates include `userlist.jade` (for displaying all users) and `index.jade` (for displaying a simple message). Each of these Jade templates manipulates the data that it receives (from a route in `routes/index.js`) to populate a Jade template.

Because the Jade template and the data are *dynamically* combined to generate an HTML Web page, there are no HTML Web pages in the application that correspond to the Jade templates.

One other point: The route that matches `/userlist` is the most complex route because it first extracts the list of all users from a Mongo database and then passes that list of users to the Jade template `views/userlist.jade`.

To summarize the steps in a bullet fashion, the flow of logic works like this:

1. A client-side request arrives and is processed by `app.js`.
2. That request is "routed" to `routes/index.js`.
3. Data is passed to a Jade template in the `views` directory.
4. An HTML Web page is generated.
5. The generated HTML Web page is returned to the client.

The app.js File

Listing 6.7 displays the contents of `app.js` for a `Node`-based application for retrieving user-related data from a Mongo database.

Listing 6.7: app.js

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');

var mongo = require('mongodb');
var monk = require('monk');
var db = monk('localhost:27017/test');

var routes = require('./routes/index');
//var users = require('./routes/users');

var app = express();

// view engine set-up
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

// uncomment after placing your favicon in /public
//app.use(favicon(dirname + '/public/favicon.ico'));
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

// Make our db accessible to our router
app.use(function(req,res,next){
```

```
    req.db = db;
    next();
});

app.use('/', routes);
app.use('/userlist', routes);
//app.use('/users', users);

/// catch 404 and forwarding to error handler
app.use(function(req, res, next) {
    var err = new Error('Not Found');
    err.status = 404;
    next(err);
});

// development error handler
// will print stacktrace
if (app.get('env') === 'development') {
    app.use(function(err, req, res, next) {
        res.status(err.status || 500);
        res.render('error', {
            message: err.message,
            error: err
        });
    });
}

// production error handler
// no stacktraces leaked to user
app.use(function(err, req, res, next) {
    res.status(err.status || 500);
    res.render('error', {
        message: err.message,
        error: {}
    });
});

module.exports = app;
```

Listing 6.7 contains code that tends to be particularly confusing to beginners, partly because of limited documentation (and perhaps the nonintuitiveness of the code). After enough repetition, you will become comfortable with this code, which appears frequently in `Node`-based applications (at least there is some good news for you).

The first part of Listing 6.7 contains standard `require` statements, initializes routes (`routes/index.js`) and users (`routes/users.js`), and then initializes `app` as an `Express` application. Another section of Listing 6.7 sets up the middleware, followed by Mongo-related variables such as `url` (with a default value).

The index.js File

Listing 6.8 displays the contents of `routes/index.js` that defines routes for user-related actions, such as retrieving a list of users from a Mongo instance and saving a new user.

Listing 6.8: index.js

```
var express = require('express');
var router = express.Router();

// GET request for home page
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

// GET the userlist page
router.get('/userlist', function(req, res) {
    var db = req.db;
    var collection = db.get('users');
    collection.find({},{},function(e,docs){
        res.render('userlist', {
```

```
            "userlist" : docs
        });
    });
});

module.exports = router;
```

Listing 6.8 contains the `router` variable, which manages the default route and two other routes. However, the routes in Listing 6.8 pass JSON-based data to Jade-based templates, which are in the `views` subdirectory. The naming convention makes it easy: The default route passes the value for `title` to the template `views/index.jade` and the second route passes the value for `title` (a different value) to the template `views/helloworld.jade`.

The code in the second route performs the following sequence of steps:

1. First, initialize the variable collection as a reference to the users database.
2. Initialize `userlist` as a reference to the documents in the users database.
3. Pass the `userlist` variable to the `views/userlist.jade` template.

As you can see, a lot of work is being performed on your behalf in the third route.

Note that the Jade templates are populated with various bits of information during the process of dynamically generating HTML Web pages that are send back to the browser.

Go back to Listing 6.3 and you will see the following code snippet, which specifies the `Jade` template engine:

```
app.set('view engine', 'jade');
```

One advantage of defining the file `index.js` with the code in Listing 6.4 is that you avoid cluttering the contents of `app.js`, which contains primarily initialization code.

For your convenience, this application contains the optional file `load-newusers.js`, which you can launch from the command line to populate a database with a set of users.

The userlist.jade Template

Listing 6.9 displays the contents of `routes/userlist.jade` that takes the list of users as input to populate a template with user-based data.

Listing 6.9: userlist.jade

```
extends layout

block content
    h1.
        List of Users
    ul
        each user, i in userlist
            li #{i} #{user.fname} | #{user.lname}
```

If you are new to Jade, you might be surprised by the compactness of the code in Listing 6.9. The variable `userlist` (which contains an array of users) is passed in from the route in `routes/index.js` that matches the route `/userlist`. Each `user` in `userlist` is a JSON-based string that contains an `fname` property and an `lname` property.

For your convenience, the following handy website converts HTML to `Jade`, thereby increasing your productivity when you create Jade templates:

http://html2jade.org/

Launching the Server-Side Application

Just to review the necessary steps for launching the application, you need to start the `Mongo` daemon process and launch the `Express` application. Specifically, you need to perform the following two steps:

Step 1: Launch the `Mongo` daemon process: `mongod.`

Step 2: Launch the `Express` application: `npm start.`

One other point: The code sample in this section was created with the `express` command. If you want to create your own Express-based applications, you need to execute the following two commands:

```
sudo npm install -g express-generator
express myexpressappname
```

You can use the files in this section to populate your new Express application, or you can provide your own custom content.

Now that you have completed this section, you are ready to work with an Angular application that accesses data from a Mongo database, which is the topic of the next section.

An Angular Application with Express

This section contains an Angular application that sends data requests to an `Express` application, which in turn sends data from a hard-coded array of values. If you want to retrieve data from a Mongo database, you can incorporate some of the code in the `ExpressMongoJade` application.

Another point: The code in this code sample does not illustrate "best practices," but instead shows you some functionality (such as the code in the `userInfo()` method in Listing 6.10) that might be useful for your own application.

DVD Now copy the directory `NG2NodeApp` from the companion disc to a convenient location. The file `app.component.ts` in Listing 6.10 is very similar to the Angular application in Chapter 4 involving an `HTTP GET` request. We need to make very minor changes to the new version of `app.component.ts`, which includes specifying a URL that matches the code in `app.js` in the previous section.

Listing 6.10: app.component.ts

```
import { Component }    from '@angular/core';
import {Inject}         from '@angular/core';
import {Http}           from '@angular/http';
import 'rxjs/add/operator/map';

@Component({
   selector: 'app-root',
   template: '<button (click)="httpRequest()">Student
                                          Details</button>
              <div>
                <li *ngFor="let fname of firstNames; let i =
                                                   index;">
                   {{fname}} * {{lastNames[i]}}
                </li>
              </div>
             '
})
export class AppComponent {
  userData:any;
  firstNames:Array<string> = [];
  lastNames:Array<string> = [];
  studentIDs:Array<string> = [];

  constructor(@Inject(Http) public http:Http) { }

  httpRequest() {
    this.http.get('http://localhost:3000/students')
      .map(res => res.json())
      .subscribe(
        data => this.userData = data,
        err => console.log('Error: '+err),
        () => this.userInfo()
      );
  }

  userInfo() {
     this.userData = JSON.stringify(this.userData);

     // populate several JavaScript arrays with data
     JSON.parse(this.userData, (key, value) => {
       if(key == "fname") {
          this.firstNames.push(value);
       } else if (key == "lname") {
          this.lastNames.push(value);
       } else {
          this.studentIDs.push(value);
```

```
        }
      });
  }
}
```

Listing 6.10 illustrates how easily you can separate the code in an Angular application from the server-side code in a NodeJS application. Notice how the `*ngFor` code block uses the variable `i` to keep track of the current index in the `firstNames` array, and then uses this same variable to find the corresponding last name in the `lastNames` array.

When users click the `<button>` element in the `template` property, the `httpRequest()` method is invoked to retrieve student-related data from an Express application. After the `httpRequest()` method has completed, the `firstNames` array and the `lastNames` array are populated in the `userInfo()` method. The output looks like this:

- John ✡ Smith
- Jane ✡ Jones
- Dave ✡ Stone
- Miko ✡ Mason
- Yuki ✡ Smith

Recall that the `Express` application needs to bypass the `Jade` templates and return the `JSON`-based data to the browser. Listing 6.11 displays the contents of `app.js` (located in the p`NG2NodeApp/server` subdirectory) that returns a hard-coded array of student information.

Listing 6.11: app.js

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

var students = {
                  "1100": {"fname": "John", "lname":"Smith"},
                  "1200": {"fname": "Jane", "lname":"Jones"},
                  "1300": {"fname": "Dave", "lname":"Stone"},
                  "1400": {"fname": "Miko", "lname":"Mason"},
                  "1500": {"fname": "Yuki", "lname":"Smith"}
                }

// add CORS support to allow cross-domain requests
app.use(function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*");
  res.header("Access-Control-Allow-Headers", "Origin,
                    X-Requested-With, Content-Type, Accept");
  next();
});

// send a message for the root path ("/"):
app.get('/', function(req, res){
    res.send('hello world');
});

// get all students:
app.get('/students', function(req, res){
    res.send(students);
});

console.log("Listening on port 3000");
app.listen(3000);
```

Listing 6.11 starts with a hard-coded list of users in the `students` array, along with the `/students` route that returns the contents of the `students` array.

Navigate to the `NG2NodeApp/server` subdirectory and invoke the following command:

```
node server.js
```

Next, navigate to the `NG2NodeApp/src` subdirectory and launch the Angular application with this command:

```
ng serve
```

Click the <button> element; if everything was set up correctly, you will see the following list of students:

- John ✡ Smith
- Jane ✡ Jones
- Dave ✡ Stone
- Miko ✡ Mason
- Yuki ✡ Smith

The next section outlines the steps for retrieving Japanese text that is stored in a Mongo database.

Angular and Japanese Text (Optional)

You can easily modify the code sample ExpressMongoJade in the previous section to work with Japanese text. There are two modifications that you need to make for this code sample:

1. Populate a collection with Japanese text (mongo load-japanese.js).
2. Define a new route /japanesejson in index.js.

Listing 6.12 displays the contents of load-japanese.js that populates a Mongo collection with a simple dictionary of English words and their counterparts in Japanese.

Listing 6.12: load-japanese.js

```
// you can invoke either of these commands:
// mongo localhost:27017/japanesedb load-japanese.js
// mongo localhost:27017/japanesedb --quiet load-japanese.js

// drop the current database
db.dropDatabase()

// insert data
db.dictionary.insert({ english: 'eat',
                       japanese: 'たべる'});
db.dictionary.insert({ english: 'drink',
                       japanese: 'のむ'});
db.dictionary.insert({ english: 'return',
                       japanese: 'かえる'});
db.dictionary.insert({ english: 'dance',
                       japanese: 'おどる'});
db.dictionary.insert({ english: 'guess',
                                 'アングルにわとてもかっこいいで
                       japanese:                    すよ！'});});
```

Now add the following new route to index.js:

```
router.get('/japanesejson', function(req, res) {
    var db = req.db;
    var collection = db.get('dictionary');

    collection.find({},{},function(e,docs){
        res.json(docs);
    });
});
```

The preceding code block is very similar to the modified route in the previous section: Instead of passing data to a Jade template, simply return the data via the code snippet shown in bold.

Angular Universal

Universal Angular supports server-side rendering for Angular, and the original home page is located here (but not used in this section):

https://github.com/angular/universal

However, the Angular core team has made very extensive modifications to integrate the code in the preceding link into Angular. The Universal

Angular code is integrated in `@angular/platform-server`.

A very rudimentary sample application involving Universal Angular is downloadable here:

https://github.com/robwormald/ng-universal-demo/

Download and uncompress the code from the preceding link in a convenient location and then type the following command:

```
npm install
```

After the preceding command has completed, launch the application:

```
npm start
```

Navigate to the URL `localhost:8000` and you will see the following simple interface:

Universal Demo

Home Lazy

Home View

Click either of the preceding links and you will see some basic text displayed.

The Server File main.server.ts

Listing 6.13 displays the contents of the file `main.server.js` that is executed when you launch the Angular Universal application in the previous section.

Listing 6.13: main.server.js

```
import 'zone.js/dist/zone-node';
import { platformServer, renderModuleFactory }
         from '@angular/platform-server'
import { enableProdMode } from '@angular/core'
import { AppServerModule } from './app.server'
import { AppServerModuleNgFactory }
         from './ngfactory/src/app.server.ngfactory'
import * as express from 'express';
import {ngExpressEngine} from './express-engine'

enableProdMode();

const app = express();

app.engine('html', ngExpressEngine({
        baseUrl: 'http://localhost:4200',
        bootstrap: [AppServerModuleNgFactory]
}));

app.set('view engine', 'html');
app.set('views', 'src')

app.get('/', (req, res) => {
        res.render('index', {req});
});

app.get('/lazy', (req, res) => {
        res.render('index', {req});
});

app.listen(8000,() => {
        console.log('listening...')
});
```

Listing 6.13 contains `import` statements for Angular code and for Express-related code that is familiar from code samples in the first part of this chapter. In particular, the `app` variable is initialized as an Express application, followed by some middleware setup steps. Two simple routes—"`/`" and "`/lazy`"— are defined to demonstrate how to handle route-based requests. Note that an application with multiple routes would probably place its definitions in a separate file (located in a separate subdirectory).

Notice that Listing 6.13 specifies the `src` directory as the directory that contains view-related files. In particular, the `/lazy` route is "mapped"

to the file `index.html`, which is collocated in the `src` subdirectory.

The Web Page index.html

Listing 6.14 displays the contents of the Web page `index.html` that contains the custom element `<demo-app>`.

Listing 6.14: index.html

```
<html>
<head>
  <meta charset="UTF-8">
  <title>Universal Test</title>
</head>
<body>
  <demo-app></demo-app>
</body>
</html>
```

Listing 6.14 is straightforward: The `<body>` element contains the `<demo-app>` custom element, which is the top-level element for the Angular application.

The TypeScript File app.ts

The TypeScript file `app.ts` contains route-related code, which was discussed in greater detail in Chapter 5.

Listing 6.15 displays the contents of the `app.ts` file, which is also located in the `src` subdirectory.

Listing 6.15: app.ts

```
import { NgModule, Component } from '@angular/core'
import { BrowserModule }       from '@angular/
                                        platform-browser'
import { RouterModule }        from '@angular/router'

@Component({
   selector: 'home-view',
   template: '<h3>Home View</h3>'
})
export class HomeView {}

@Component({
   selector: 'demo-app',
   template: '
     <h1>Universal Demo</h1>
     <a routerLink="/">Home</a>
     <a routerLink="/lazy">Lazy</a>
     <router-outlet></router-outlet>
   '
})
export class AppComponent {}

@NgModule({
   imports: [
      BrowserModule.withServerTransition({
        appId: 'universal-demo-app'
      }),
      RouterModule.forRoot([
         { path: '', component: HomeView, pathMatch: 'full'},
         { path: 'lazy', loadChildren: './lazy.
                                        module#LazyModule'}
      ])
   ],
   declarations: [ AppComponent, HomeView ],
   bootstrap: [ AppComponent ]
})
export class AppModule {}
```

Listing 6.15 contains three main parts: the `HomeView` class, the `AppComponent` class, and the `AppModule` class (all of which are exported).

The `HomeView` class is associated with the "home" route, which is the first of the two routes in this sample application.

The `AppComponent` class is preceded by an `@Component` decorator, whose `selector` property specifies the `<demo-app>` custom element. In addition, the `template` property contains the definitions of the two routes.

Finally, the `AppModule` class is preceded with an `@NgModule` decorator, whose contents are typically located in a separate file such as `app.module.ts`.

In essence, Universal Angular applications involve a two-step process: Use `npm` to launch an Express-based application that contains a route to an `index.html` Web page. This Web page contains an element that is the root of an Angular application, and the latter is defined in TypeScript files.

Working with Microservices (Optional)

This section contains a condensed and simplified overview of microservices. You will learn about some of the advantages and the disadvantages of microservices, followed by an example of an Angular application that uses microservices. Although Angular supports Microservices (so it's a relevant topic for this chapter), you can skip this section with no loss of continuity.

Advantages of Microservices

The microservices architecture decomposes monolithic applications into a set of services without reducing the functionality of the original applications. Each service is accessible via its exposed application programming interface (API). The use of microservices simplifies the process of developing, maintaining, and enhancing "one-purpose" services.

Second, microservices allows for the development of services in an independent manner. Each new service can be written using the technology that is most suitable for that service (which means current technology is not mandatory).

A third advantage is that a microservice can be deployed independently of other microservices. Fourth, services can be scaled independently of each other.

Disadvantages of Microservices

A collection of many independent microservices can create complexity. Communication between microservices requires a decision regarding interprocess communication. Moreover, the distributed nature of microservices necessitates handling failures that can arise due to timeouts and other causes.

For example, if transaction-oriented microservices interact with multiple databases, then processing and coordinating those transactions can be complicated. Keep in mind that one of the primary goals of adopting microservices is to reduce the complexity of the interdependencies that can occur in large monolithic applications.

Some people think of microservices as a "fine-grained service-oriented architecture (SOA)." However, SOA requires the Web Services Definition Language, which defines service endpoints and is strongly typed, whereas microservices have very simple connections and smart endpoints. Another important difference is that SOA is often stateless, whereas microservices are stateful and keep data and logic together. In general, SOA is complex and heavyweight, whereas microservices are simpler and lightweight independent processes.

Summary

This chapter showed you how to create simple `Express`-based Node applications that can serve requests from Angular applications. You saw how to use the NoSQL database `MongoDB` to store and retrieve data for Web applications. You also learned how to launch multiple Express applications on different port numbers to serve data to an Angular application. You then saw an overview of microservices, which can work seamlessly with Angular. Finally, you learned some basic concepts about Angular Universal, which involves server-side rendering of Angular code.