

Converting an AngularJS 1.x App to Angular 4

If you've been using Angular for a while, then you probably already have production AngularJS 1 apps. Angular 4 is great, but there's no way we can drop everything and rewrite our entire production apps in Angular 4. What we need is a way to *incrementally* upgrade our AngularJS 1 app. Thankfully, Angular 4 has a fantastic way to do that.

The interoperability of AngularJS 1 (ng1) and Angular 4 (ng2) works really well. In this chapter, we're going to talk about how to upgrade your ng1 app to ng2 by writing a *hybrid* app. A hybrid app is running ng1 and ng2 simultaneously (and we can exchange data between them).

Peripheral Concepts

When we talk about interoperability between AngularJS 1 and Angular 4, there's a lot of peripheral concepts. For instance:

Mapping AngularJS 1 Concepts to Angular 4: At a high level, ng2 Components are ng1 directives. We also use Services in both. However, this chapter is about using both ng1 and ng2, so we're going to assume you have basic knowledge of both. If you haven't used ng2 much, checkout the chapter on [How Angular Works](#) before reading this chapter.

Preparing ng1 apps for ng2: AngularJS 1.5 provides a new `.component` method to make "component-directives". `.component` is a great way to start preparing your ng1 app for ng2. Furthermore, creating thin controllers (or [banning them altogether](#)¹⁴¹) is a great way to refactor your ng1 app such that it's easier to integrate with ng2.

Another way to prepare your ng1 app is to reduce or eliminate your use of two-way data-binding in favor of a one-way data flow. In-part, you'd do this by reducing `$scope` changes that pass data between directives and instead use services to pass your data around.

These ideas are important and warrant further exploration. However, we're not going to extensively cover best-practices for pre-upgrade refactoring in this chapter.

Instead, here's what we **are** going to talk about:

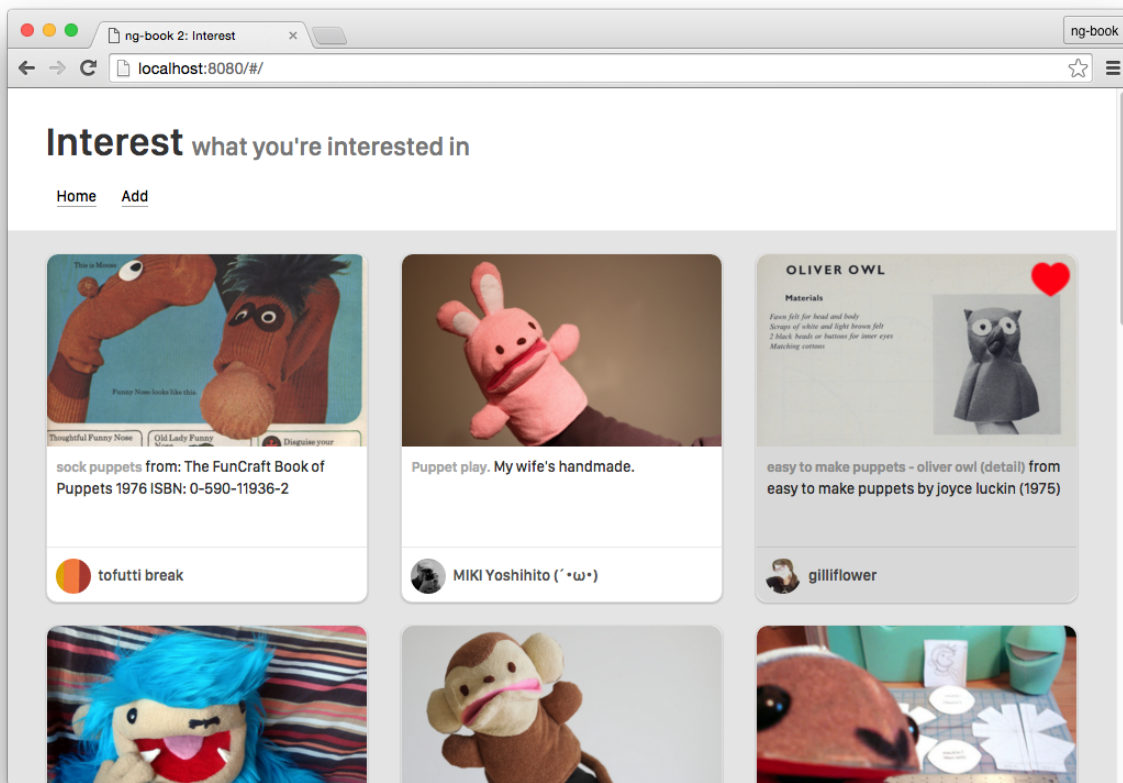
Writing hybrid ng1/ng2 apps: ng2 provides a way to bootstrap your ng1 app and then write ng2 components and services. You can write ng2 components that will mix with ng1 components and it "just works". Furthermore, the dependency injection system supports passing between ng1 and ng2 (both directions), so you can write services which will run in either ng1 or ng2.

¹⁴¹<http://teropa.info/blog/2014/10/24/how-ive-improved-my-angular-apps-by-banning-ng-controller.html>

The best part? Change detection runs within Zones, so you don't need to call `$scope.apply` or worry much about change-detection at all.

What We're Building

In this chapter, we're going to be converting an app called "Interest" - it's a Pinterest-like clone. The idea is that you can save a "Pin" which is a link with an image. The Pins will be shown in a list and you can "fav" (or unfav) a pin.



Our completed Pinterest-like app



You can find the completed code for both the ng1 version and the completed hybrid version in the sample code download under `code/upgrade/ng1` and `code/conversion/hybrid`

The hybrid app is written using Angular CLI. In order to run it, change into the directory and type:

```
1 npm install
2 npm start
```

Before we dive in, let's set the stage for interoperability between ng1 and ng2

Mapping AngularJS 1 to Angular 4

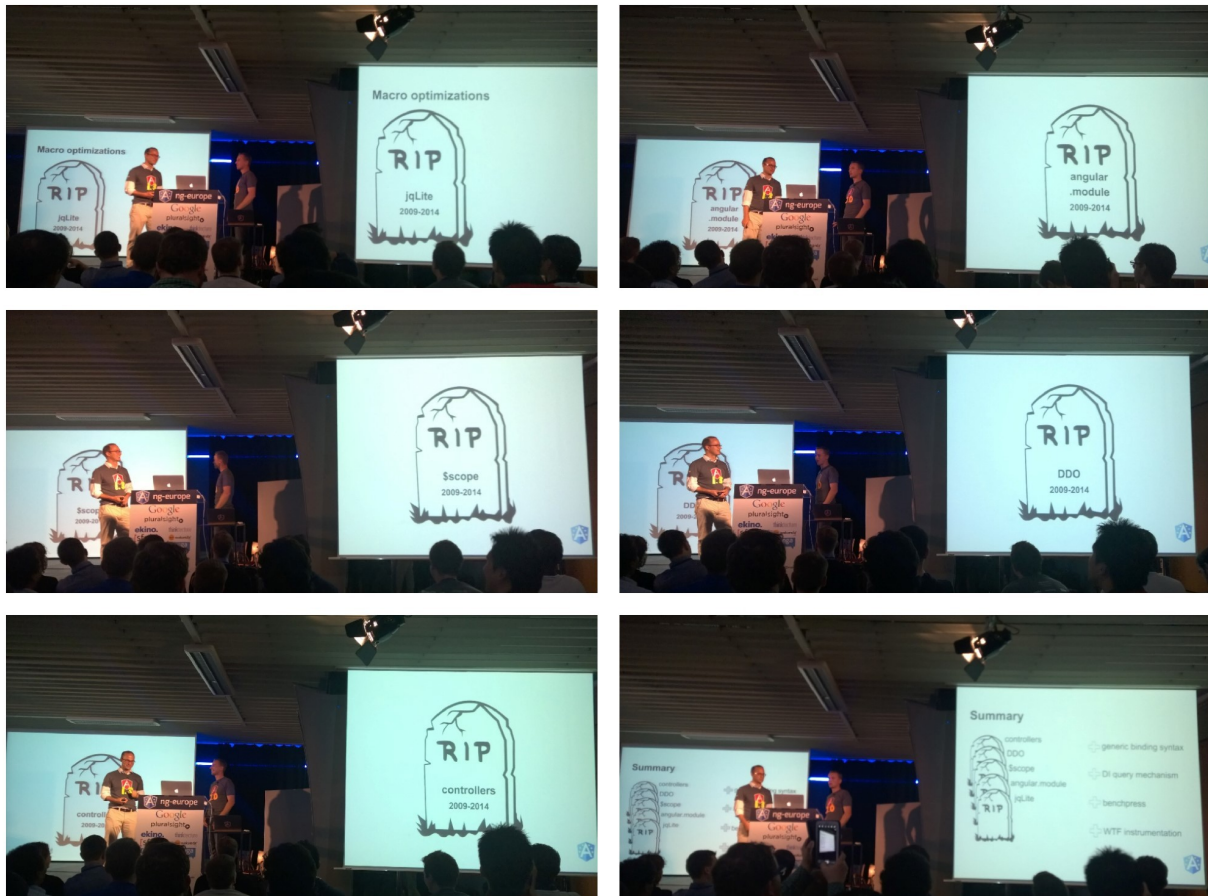
From a high level, the five main parts of AngularJS 1 are:

- Directives
- Controllers
- Scopes
- Services
- Dependency Injection

Angular 4 changes this list significantly. You might have heard that at ngEurope 2014 Igor and Tobias from the Angular core team announced that they were killing off several “core” ideas in AngularJS 1 ([video here](https://www.youtube.com/watch?v=gNmWybAyBHI)¹⁴²). Specifically, they announced that Angular 4 was killing off:

- `$scope` (& two-way binding by default)
- Directive Definition Objects
- Controllers
- `angular.module`

¹⁴²<https://www.youtube.com/watch?v=gNmWybAyBHI>



Igor and Tobias killing off many APIs from 1.x. at ngEurope 2014. Photo Credit: Michael Bromley (used with permission)

As someone who's built AngularJS 1 apps and is used to thinking in ng1, we might ask: if we take those things away, what is left? How can you build Angular apps without Controllers and \$scope?

Well, as much as people like to dramatize how **different** Angular 4 is, it turns out, a lot of the same ideas are still with us and, in fact, Angular 4 provides just as much functionality but with a **much simpler model**.

At a high-level Angular 4 core is made up of:

- Components (think “directives”) and
- Services

Of course there's tons of infrastructure required to make those things work. For instance, you need Dependency Injection to manage your Services. And you need a strong change detection library to efficiently propagate data changes to your app. And you need an efficient rendering layer to handle rendering the DOM at the right time.

Requirements for Interoperability

So given these two different systems, what features do we need for easy interoperability?

- **Use Angular 4 Components in AngularJS 1:** The first thing that comes to mind is that we need to be able to write new ng2 components, but use them within our ng1 app.
- **Use AngularJS 1 Components in Angular 4:** It's likely that we won't replace a whole branch of our component-tree with all ng2 components. We want to be able to re-use any ng1 components we have *within* a ng2 component.
- **Service Sharing:** If we have, say, a `UserService` we want to share that service between both ng1 and ng2. Services are normally plain JavaScript objects so, more generally, what we need is an interoperable **dependency injection** system.
- **Change Detection:** If we make changes in one side, we want those changes to propagate to the other.

Angular 4 provides solutions for all of these situations and we'll cover them in this chapter.

In this chapter we're going to do the following:

- Describe the ng1 app we'll be converting
- Explain how to setup your hybrid app by using ng2's `UpgradeAdapter`
- Explain step-by-step how to share components (directives) and services between ng1 and ng2 by converting the ng1 app to a hybrid app

The AngularJS 1 App

To set the stage, let's go over the AngularJS 1 version of our app.



This chapter assumes some knowledge of AngularJS 1 and [ui-router](https://github.com/angular-ui/ui-router)¹⁴³. If you're not comfortable with AngularJS 1 yet, check out [ng-book 1](http://ng-book.com)¹⁴⁴.

We won't be diving too deeply into explaining each AngularJS 1 concept. Instead, we're going to review the structure of the app to prepare for our upgrade to a ng2/hybrid app.

To run the ng1 app, cd into `conversion/ng1` in the code samples, install the dependencies, and run the app.

¹⁴³<https://github.com/angular-ui/ui-router>

¹⁴⁴<http://ng-book.com>

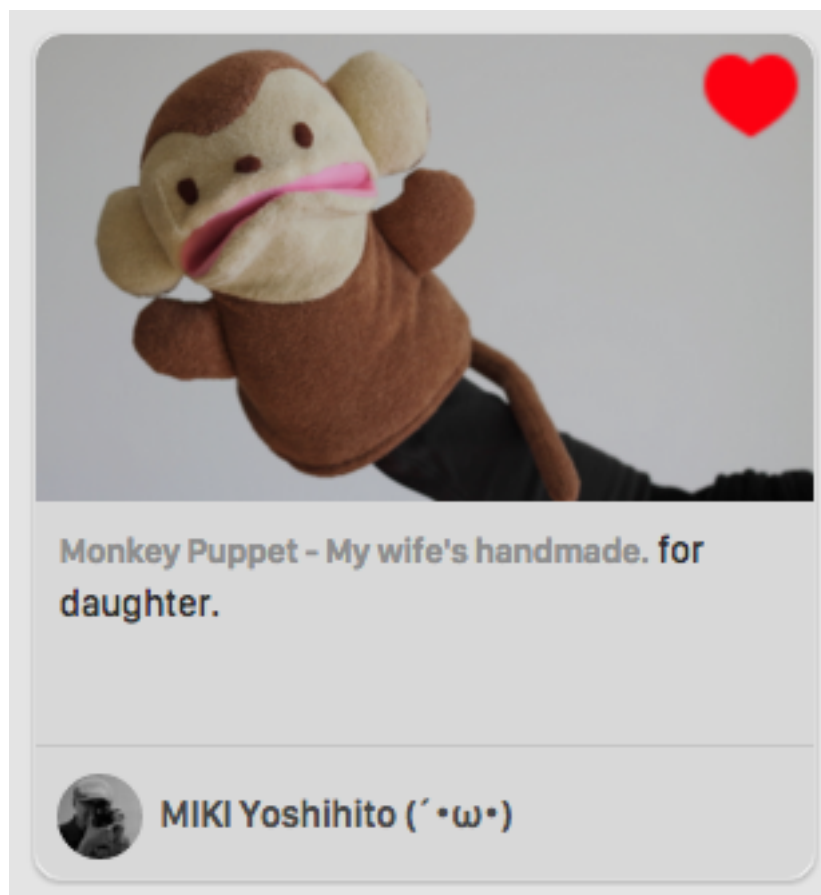
```
1 cd code/upgrade/ng1 # change directories
2 npm install          # install dependencies
3 npm run go           # run the app
```

If your browser doesn't open automatically, open the url: <http://localhost:8080>¹⁴⁵.



Note that the AngularJS 1 app in ng1 will run on port 8080 whereas the hybrid app (discussed below) will run on port 4200.

In this app, you can see that our user is collecting puppets. We can hover over an item and click the heart to “fav” a pin.



Red heart indicates a faved pin

We can also go to the /add page and add a new pin. Try submitting the default form.



Handling image uploads is more complex than we want to handle in this demo. For now, just paste the full URL to an image if you want to try a different image.

¹⁴⁵<http://localhost:8080>

The ng1-app HTML

The `index.html` in our ng1 app uses a common structure:

`code/upgrade/ng1/index.html`

```
1  <!DOCTYPE html>
2  <html ng-app='interestApp'>
3  <head>
4    <meta charset="utf-8">
5    <title>Interest</title>
6    <link rel="stylesheet" href="css/bootstrap.min.css">
7    <link rel="stylesheet" href="css/sf.css">
8    <link rel="stylesheet" href="css/interest.css">
9  </head>
10 <body class="container-fullwidth">
11
12   <div class="page-header">
13     <div class="container">
14       <h1>Interest <small>what you're interested in</small></h1>
15
16       <div class="navLinks">
17         <a ui-sref='home' id="navLinkHome">Home</a>
18         <a ui-sref='add' id="navLinkAdd">Add</a>
19       </div>
20     </div>
21   </div>
22
23   <div id="content">
24     <div ui-view=''></div>
25   </div>
26
27   <script src="js/vendor/lodash.js"></script>
28   <script src="js/vendor/angular.js"></script>
29   <script src="js/vendor/angular-ui-router.js"></script>
30   <script src="js/app.js"></script>
31 </body>
32 </html>
```

- Notice that we're using `ng-app` in the `html` tag to specify that this app uses the module `interestApp`.
- We load our javascript with `script` tags at the bottom of the body.
- The template contains a `page-header` which stores our navigation

- We're using `ui-router` which means we:
 - Use `ui-sref` for our links (Home and Add) and
 - We use `ui-view` where we want the router to populate our content.

Code Overview

We'll look at each section in code, but first, let's briefly describe the moving parts.

In our app, we have two routes:

- `/` uses the `HomeController`
- `/add` uses the `AddController`

We use a `PinsService` to hold an array of all of the current pins. `HomeController` renders the list of pins and `AddController` adds a new element to that list.

Our root-level route uses our `HomeController` to render pins. We have a `pin` directive that renders each pin.

The `PinsService` stores the data in our app, so let's look at the `PinsService` first.

ng1: PinsService

code/upgrade/ng1/js/app.js

```
1 angular.module('interestApp', ['ui.router'])
2 .service('PinsService', function($http, $q) {
3   this._pins = null;
4
5   this.pins = function() {
6     var self = this;
7     if(self._pins == null) {
8       // initialize with sample data
9       return $http.get("/js/data/sample-data.json").then(
10         function(response) {
11           self._pins = response.data;
12           return self._pins;
13         })
14     } else {
15       return $q.when(self._pins);
16     }
17   }
18 }
```



```

19  this.addPin = function(newPin) {
20    // adding would normally be an API request so lets mock async
21    return $q.when(
22      this._pins.unshift(newPin)
23    );
24  }
25  })

```

The PinsService is a .service that stores an array of pins in the property `_pins`.

The method `.pins` returns a promise that resolves to the list of pins. If `_pins` is null (i.e. the first time), then we will load sample data from `/js/data/sample-data.json`.

code/upgrade/ng1/js/data/sample-data.json

```

1  [
2    {
3      "title": "sock puppets",
4      "description": "from:\nThe FunCraft Book of Puppets\n1976\nISBN: 0-590-11936\
5      -2",
6      "user_name": "tofutti break",
7      "avatar_src": "images/avatars/42826303@N00.jpg",
8      "src": "images/pins/106033588_167d811702_o.jpg",
9      "url": "https://www.flickr.com/photos/tofuttibreak/106033588/",
10     "faved": false,
11     "id": "106033588"
12   },
13   {
14     "title": "Puppet play.",
15     "description": "My wife's handmade.",
16     "user_name": "MIKI Yoshihito (ミキ)",
17     "avatar_src": "images/avatars/7940758@N07.jpg",
18     "src": "images/pins/4422575066_7d5c4c41e7_o.jpg",
19     "url": "https://www.flickr.com/photos/mujitra/4422575066/",
20     "faved": false,
21     "id": "4422575066"
22   },
23   {
24     "title": "easy to make puppets - oliver owl (detail)",
25     "description": "from easy to make puppets by joyce luckin (1975)",
26     "user_name": "gilliflower",
27     "avatar_src": "images/avatars/26265986@N00.jpg",
28     "src": "images/pins/6819859061_25d05ef2e1_o.jpg",

```

```

29     "url": "https://www.flickr.com/photos/gilliflower/6819859061/",
30     "faved": false,
31     "id": "6819859061"
32   },

```

Snippet from Sample Data

The method `.addPin` simply adds the new pin to the array of pins. We use `$q.when` here to return a promise, which is likely what would happen if we were doing a real async call to a server.

ng1: Configuring Routes

We're going to configure our routes with `ui-router`.



If you're unfamiliar with `ui-router` you can [read the docs here](https://github.com/angular-ui/ui-router/wiki)¹⁴⁶.

As we mentioned, we're going to have two routes:

`code/upgrade/ng1/js/app.js`

```

26 .config(function($stateProvider, $urlRouterProvider) {
27   $stateProvider
28     .state('home', {
29       templateUrl: '/templates/home.html',
30       controller: 'HomeController as ctrl',
31       url: '/',
32       resolve: {
33         'pins': function(PinsService) {
34           return PinsService.pins();
35         }
36       }
37     })
38     .state('add', {
39       templateUrl: '/templates/add.html',
40       controller: 'AddController as ctrl',
41       url: '/add',
42       resolve: {
43         'pins': function(PinsService) {
44           return PinsService.pins();
45         }

```

¹⁴⁶<https://github.com/angular-ui/ui-router/wiki>

```

46     }
47   })
48
49   $urlRouterProvider.when('', '/') ;
50 })

```

The first route `/` maps to the `HomeController`. It has a template, which we'll look at in a minute. Notice that we also are using the `resolve` functionality of `ui-router`. This says that before we load this route for the user, we want to call `PinsService.pins()` and inject the result (the list of pins) into the controller (`HomeController`).

The `/add` route as similarly, except that it has a different template and a different controller.

Let's first look at our `HomeController`.

ng1: HomeController

Our `HomeController` is straightforward. We save `pins`, which is injected because of our `resolve`, to `$scope.pins`.

code/upgrade/ng1/js/app.js

```

60 .controller('HomeController', function(pins) {
61   this.pins = pins;
62 })

```

ng1: / HomeController template

Our home template is small: we use an `ng-repeat` to repeat over the pins in `$scope.pins`. Then we render each pin with the `pin` directive.

code/upgrade/ng1/templates/home.html

```

1 <div class="container">
2   <div class="row">
3     <pin item="pin" ng-repeat="pin in ctrl.pins">
4       </pin>
5     </div>
6 </div>

```

Let's dive deeper and look at this `pin` directive.

ng1: pin Directive

The pin directive is restricted to matching an element (E) and has a template.

We can input our pin via the item attribute, as we did in the home.html template.

Our link function, defines a function on the scope called toggleFav which toggles the pin's faved property.

code/upgrade/ng1/js/app.js

```
92  })
93  .directive('pin', function() {
94    return {
95      restrict: 'E',
96      templateUrl: '/templates/pin.html',
97      scope: {
98        'pin': "=item"
99      },
100     link: function(scope, elem, attrs) {
101       scope.toggleFav = function() {
102         scope.pin.faved = !scope.pin.faved;
103       }
104     }
105   }
106 })
```



This directive shouldn't be taken as an example of directive using the current best-practices. For instance, if I was writing this component anew (in ng1) I would probably use the new `.component` directive available in AngularJS 1.5+. At the very least, I'd probably use `controllerAs` instead of `link` here.

But this section is less about how to write ng1 code, as much as **how to work with the ng1 code you already have**.

ng1: pin Directive template

The template `templates/pin.html` renders an individual pin on our page.

code/upgrade/ng1/templates/pin.html

```

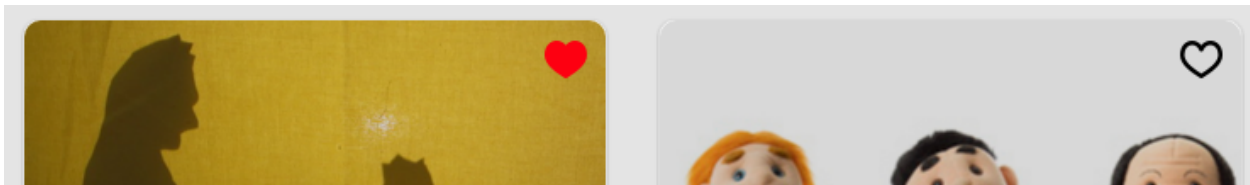
1 <div class="col-sm-6 col-md-4">
2   <div class="thumbnail">
3     <div class="content">
4       
5       <div class="caption">
6         <h3>{{pin.title}}</h3>
7         <p>{{pin.description | truncate:100}}</p>
8       </div>
9       <div class="attribution">
10        
11        <h4>{{pin.user_name}}</h4>
12      </div>
13    </div>
14    <div class="overlay">
15      <div class="controls">
16        <div class="heart">
17          <a ng-click="toggleFav()">
18            </img>
19            </img>
20          </a>
21        </div>
22      </div>
23    </div>
24  </div>
25 </div>

```

The directives we use here are ng1 built-ins:

- We use `ng-src` to render the `img`.
- Next we show the `pin.title` and `pin.description`.
- We use `ng-if` to show either the red or empty heart

The most interesting thing here is the `ng-click` that will call `toggleFav`. `toggleFav` changes the `pin.faved` property and thus the red or empty heart will be shown accordingly.



Red vs. Black Heart

Now let's turn our attention to the AddController.

ng1: AddController

Our AddController has a bit more code than the HomeController. We open by defining the controller and specifying the services it will inject:

code/upgrade/ng1/js/app.js

```
63 .controller('AddController', function($state, PinsService, $timeout) {  
64     var ctrl = this;  
65     ctrl.saving = false;
```

We're using `controllerAs` syntax in our router and template, which means we set properties on this instead of on `$scope`. Scoping this in ES5 JavaScript can be tricky, so we assign `var ctrl = this;` which helps disambiguate when we're referencing the controller in nested functions.

code/upgrade/ng1/js/app.js

```
67     var makeNewPin = function() {  
68         return {  
69             "title": "Steampunk Cat",  
70             "description": "A cat wearing goggles",  
71             "user_name": "me",  
72             "avatar_src": "images/avatars/me.jpg",  
73             "src": "/images/pins/cat.jpg",  
74             "url": "http://cats.com",  
75             "faved": false,  
76             "id": Math.floor(Math.random() * 10000).toString()  
77         }  
78     }  
79  
80     ctrl.newPin = makeNewPin();
```

We create a function `makeNewPin` that contains the default structure and data for a pin.

We also initialize this controller by setting `ctrl.newPin` to the value of calling this function.

The last thing we need to do is define the function to submit a new pin:

code/upgrade/ng1/js/app.js

```
82  ctrl.submitPin = function() {
83    ctrl.saving = true;
84    $timeout(function() {
85      PinsService.addPin(ctrl.newPin).then(function() {
86        ctrl.newPin = makeNewPin();
87        ctrl.saving = false;
88        $state.go('home');
89      });
90    }, 2000);
91  }
92  })
```

Essentially, this article is calling out to `PinsService.addPin` and creating a new pin. But there's a few other things going on here.

In a real application, this would almost certainly call back to a server. We're mimicking that effect by using `$timeout`. (That is, you could remove the `$timeout` function and this would still work. It's just here to deliberately slow down the app to give us a chance to see the "Saving" indicator.)

We want to give some indication to the user that their pin is saving, so we set the `ctrl.saving = true`.

We call `PinsService.addPin` giving it our `ctrl.newPin`. `addPin` returns a promise, so in our promise function we

1. revert `ctrl.newPin` to the original value
2. we set `ctrl.saving` to false, because we're done saving the pin
3. we use the `$state` service to redirect the user to the homepage where we can see our new pin

Here's the whole code of the `AddController`:

code/upgrade/ng1/js/app.js

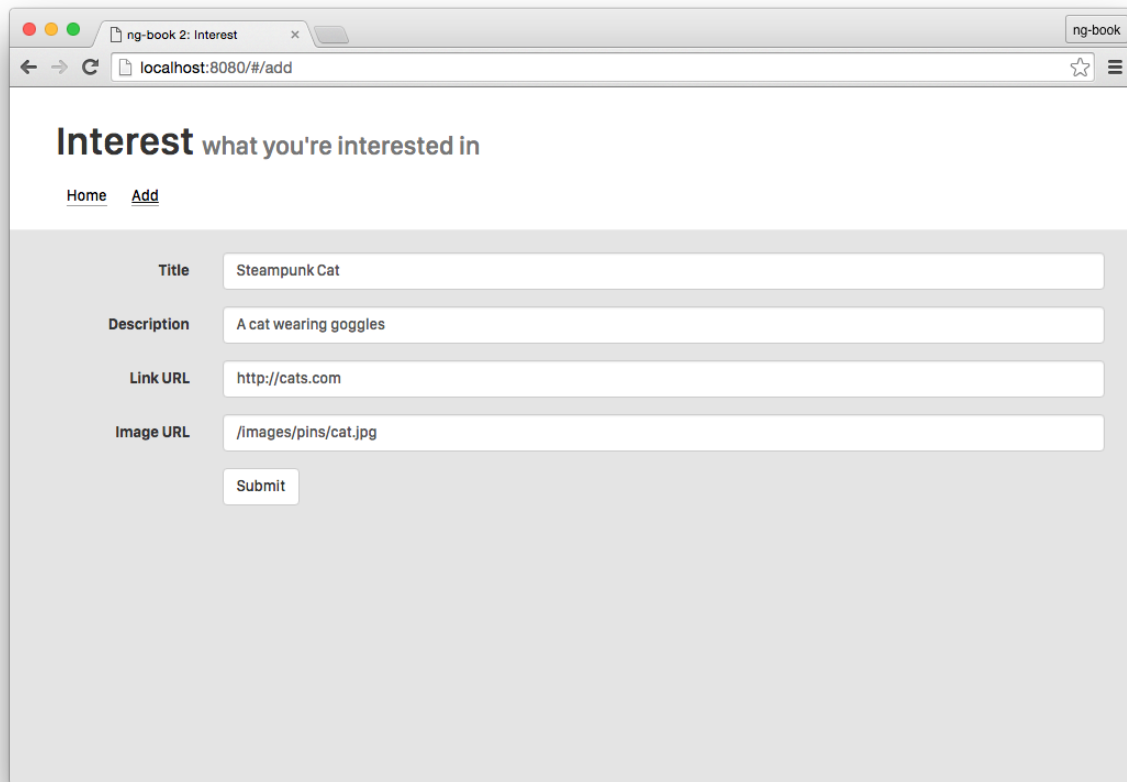
```
63  .controller('AddController', function($state, PinsService, $timeout) {
64    var ctrl = this;
65    ctrl.saving = false;
66
67    var makeNewPin = function() {
68      return {
69        "title": "Steampunk Cat",
70        "description": "A cat wearing goggles",
71        "user_name": "me",
```



```
72     "avatar_src": "images/avatars/me.jpg",
73     "src": "/images/pins/cat.jpg",
74     "url": "http://cats.com",
75     "faved": false,
76     "id": Math.floor(Math.random() * 10000).toString()
77   }
78 }
79
80 ctrl.newPin = makeNewPin();
81
82 ctrl.submitPin = function() {
83   ctrl.saving = true;
84   $timeout(function() {
85     PinsService.addPin(ctrl.newPin).then(function() {
86       ctrl.newPin = makeNewPin();
87       ctrl.saving = false;
88       $state.go('home');
89     });
90   }, 2000);
91 }
92 })
```

ng1: AddController template

Our /add route renders the add.html template.



Adding a New Pin Form

The template uses `ng-model` to bind the input tags to the properties of the `newPin` on the controller. The interesting things here are that:

- We use `ng-click` on the submit button to call `ctrl.submitPin` and
- We show a “Saving...” message if `ctrl.saving` is truthy

code/upgrade/ng1/templates/add.html

```
1 <div class="container">
2   <div class="row">
3
4     <form class="form-horizontal">
5
6       <div class="form-group">
7         <label for="title"
8           class="col-sm-2 control-label">Title</label>
```

```
9      <div class="col-sm-10">
10        <input type="text"
11              class="form-control"
12              id="title"
13              placeholder="Title"
14              ng-model="ctrl.newPin.title">
15      </div>
16    </div>
17
18    <div class="form-group">
19      <label for="description"
20            class="col-sm-2 control-label">Description</label>
21      <div class="col-sm-10">
22        <input type="text"
23              class="form-control"
24              id="description"
25              placeholder="Description"
26              ng-model="ctrl.newPin.description">
27      </div>
28    </div>
29
30    <div class="form-group">
31      <label for="url"
32            class="col-sm-2 control-label">Link URL</label>
33      <div class="col-sm-10">
34        <input type="text"
35              class="form-control"
36              id="url"
37              placeholder="Link URL"
38              ng-model="ctrl.newPin.url">
39      </div>
40    </div>
41
42    <div class="form-group">
43      <label for="url"
44            class="col-sm-2 control-label">Image URL</label>
45      <div class="col-sm-10">
46        <input type="text"
47              class="form-control"
48              id="url"
49              placeholder="Image URL"
50              ng-model="ctrl.newPin.src">
```

```
51     </div>
52 </div>
53
54 <div class="form-group">
55     <div class="col-sm-offset-2 col-sm-10">
56         <button type="submit"
57             class="btn btn-default"
58             ng-click="ctrl.submitPin()">Submit</button>
59     </div>
60 </div>
61 <div ng-if="ctrl.saving">
62     Saving...
63 </div>
64 </form>
65
66 </div>
67 </div>
```

ng1: Summary

There we have it. This app has just the right amount of complexity that we can start porting it to Angular 4.

Building A Hybrid

Now we're ready to start putting some Angular 4 in our AngularJS 1 app.

Before we start using Angular 4 in our browser, we're going to need to make some modifications to our project structure.



You can find the code for this example in `code/conversion/hybrid`.

To run it, run:

- 1 `npm install`
- 2 `npm start`

Then open your browser to `http://localhost:4200` – note that this is a **different URL** than the pure-AngularJS 1 app above.

Hybrid Project Structure

The first step to creating a hybrid app is to make sure you have both ng1 and ng2 loaded as dependencies. Everyone's situation is going to be slightly different.

In this example we've **vendored** the AngularJS 1 libraries (in `js/vendor`) and we're loading the Angular 4 libraries from `npm`.

In your project, you might want to vendor them both, use [bower](http://bower.io/)¹⁴⁷, etc. However, using `npm` is very convenient for Angular 4, and so we suggest using `npm` to install Angular 4.

One of the first challenges we face when making a hybrid app is ensuring our build-process can support both JavaScript and TypeScript files, as well as resolving our assets, type-definitions, and so on.

Here we're using Angular CLI (which is based on Webpack) in order to build this app. We'll describe the specific steps necessary to get our app running within Angular CLI, but if you have an existing build process, it might take some additional work to get it in order.

Dependencies with `package.json`

You install dependencies with `npm` using the `package.json` file. Here's our `package.json` for the hybrid example:

`code/upgrade/hybrid/package.json`

```
1 {
2   "name": "hybrid",
3   "version": "0.0.0",
4   "license": "MIT",
5   "scripts": {
6     "ng": "ng",
7     "start": "ng serve",
8     "build": "ng build",
9     "test": "ng test",
10    "lint": "ng lint",
11    "e2e": "ng e2e"
12  },
13  "private": true,
14  "dependencies": {
15    "@angular/common": "4.1.0",
16    "@angular/compiler": "4.1.0",
17    "@angular/core": "4.1.0",
18    "@angular/forms": "4.1.0",
```

¹⁴⁷<http://bower.io/>

```
19     "@angular/http": "4.1.0",
20     "@angular/platform-browser": "4.1.0",
21     "@angular/platform-browser-dynamic": "4.1.0",
22     "@angular/router": "4.1.0",
23     "@angular/upgrade": "4.0.0",
24     "core-js": "2.4.1",
25     "rxjs": "5.0.1",
26     "zone.js": "0.8.5",
27     "reflect-metadata": "0.1.3",
28     "@types/jasmine": "2.5.40"
29   },
30   "devDependencies": {
31     "@angular/cli": "1.0.1",
32     "@angular/compiler-cli": "4.1.0",
33     "@types/angular-ui-router": "1.1.36",
34     "@types/jasmine": "2.5.38",
35     "@types/node": "~6.0.60",
36     "codemlizer": "~2.0.0",
37     "jasmine-core": "~2.5.2",
38     "jasmine-spec-reporter": "~3.2.0",
39     "karma": "~1.4.1",
40     "karma-chrome-launcher": "~2.0.0",
41     "karma-cli": "~1.0.1",
42     "karma-coverage-istanbul-reporter": "0.2.0",
43     "karma-jasmine": "~1.1.0",
44     "karma-jasmine-html-reporter": "0.2.2",
45     "protractor": "~5.1.0",
46     "ts-node": "~2.0.0",
47     "tslint": "~4.4.2",
48     "typescript": "2.1.5"
49   }
50 }
```



If you're unfamiliar with what one of these packages does, it's a good idea to find out. `rxjs`, for example, is the library that provides our observables.

Notice that we've included the `@angular/upgrade` package. This module contains the tools necessary for booting a hybrid app.

Compiling our code

We're going to be using TypeScript in this example alongside our JavaScript AngularJS 1 code. To do this, we're going to put all of our "old" JavaScript code in the folder `js/`.

We also want to load AngularJS, as well as `angular-ui-router` and our AngularJS 1 app. Here, to do this we're going to include them in the `scripts` tag of our `.angular-cli.json`

```
1 {  
2   "apps": [  
3     {  
4       // ...  
5       "scripts": [  
6         "js/vendor/angular.js",  
7         "js/vendor/angular-ui-router.js",  
8         "js/app.js"  
9       ],  
10    }  
11  ]  
12 }
```



This step may vary depending on your build process. For instance, if you have an existing AngularJS app you may have an existing build process that builds that app into one or a few files (e.g. using Gulp or another build system). In that case, if you want to bring that build into your Angular CLI project, you could have a separate step that would build those files and import them into "scripts" here.

In the case that you want a more unified workflow, you'll need to run `ng eject` and modify the generated Webpack file from there.

That said, building custom Webpack configurations is beyond the scope of this book.

When we write hybrid ng2 apps **the Angular 4 code becomes the entry point**. This makes sense because **it's Angular 4 that's providing the backwards compatibility with AngularJS 1**. Let's take a closer look at the bootstrapping process.

Bootstrapping our Hybrid App

Now that we have our project structure in place, let's bootstrap the app.

If you recall, with AngularJS 1 you can bootstrap the app in 1 of two ways:

1. You can use the `ng-app` directive, such as `ng-app='interestApp'`, in your HTML or

2. You can use `angular.bootstrap` in JavaScript

In hybrid apps we use a **new bootstrap** method that comes from an `UpgradeAdapter`.

Since we'll be bootstrapping the app in code, **make sure you remove the `ng-app` from your `index.html`.**

Here's what a minimal bootstrapping of our code would look like:

```
1  // code/upgrade/hybrid/src/app/app.module.ts
2  import {
3    NgModule,
4    forwardRef
5  } from '@angular/core';
6  import { CommonModule } from '@angular/common';
7  import { BrowserModule } from '@angular/platform-browser';
8
9  import { UpgradeAdapter } from '@angular/upgrade';
10 declare var angular: any;
11
12 /*
13  * Create our upgradeAdapter
14  */
15 const upgradeAdapter: UpgradeAdapter = new UpgradeAdapter(
16   forwardRef(() => MyAppModule)); // <-- notice forward reference
17
18 // ...
19 // upgrade and downgrade components in here
20 // ...
21
22 /*
23  * Create our app's entry NgModule
24  */
25 @NgModule({
26   declarations: [ MyNg2Component, ... ],
27   imports: [
28     CommonModule,
29     BrowserModule
30   ],
31   providers: [ MyNg2Services, ... ]
32 })
33 class MyAppModule { }
34
```

```
35  /*
36   * Bootstrap the App
37   */
38  upgradeAdapter.bootstrap(document.body, ['interestApp']);
```

We start by importing the `UpgradeAdapter` and then we create an instance of it: `upgradeAdapter`.

However, the constructor of `UpgradeAdapter` requires an `NgModule` that we'll be using for our Angular 4 app - but we haven't defined it yet! To get around this we use the `forwardRef` function which allows us to take a 'forward reference' to our `NgModule` which we declare below.

When we define our `NgModule` `MyAppModule` (or specifically in this app it will be `InterestAppModule`), we define it like we would any other Angular 4 `NgModule`: we put in our declarations, imports, providers, etc.

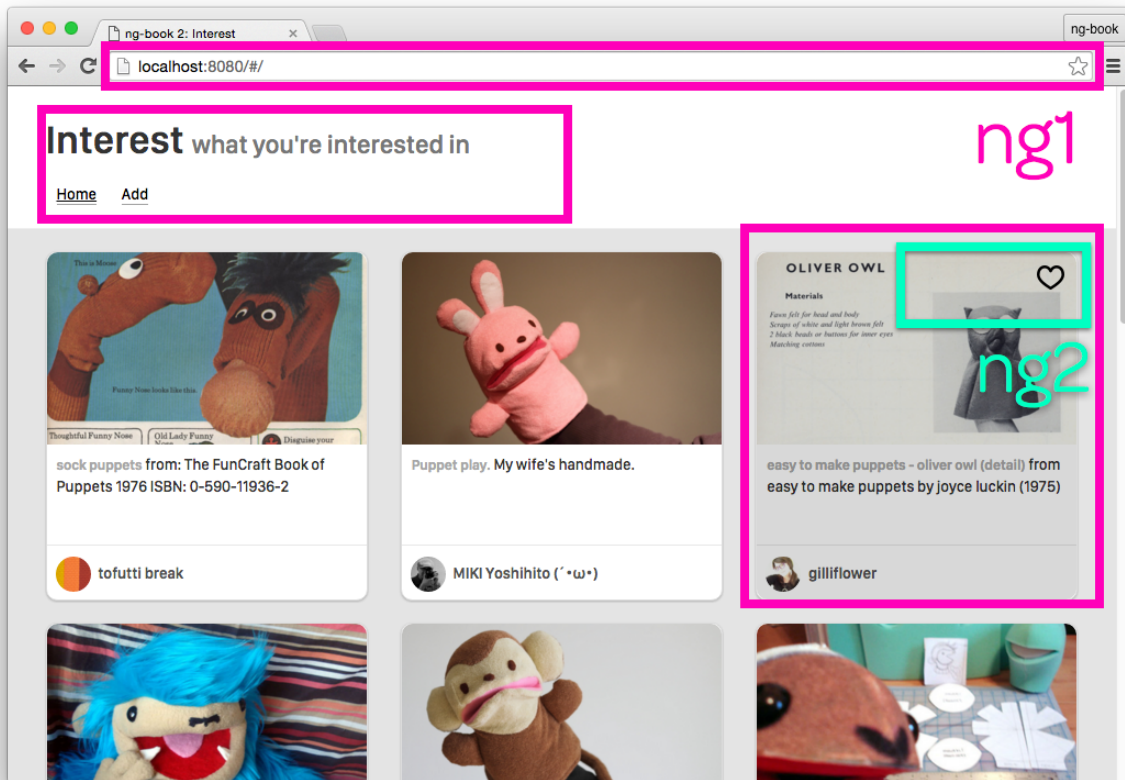
Lastly, we tell the `upgradeAdapter` to bootstrap our app on the element `document.body` and we specify the module name of our **AngularJS 1 app**.

This will bootstrap our AngularJS 1 app within our Angular 4 app! Now we can start replacing pieces with Angular 4.

What We'll Upgrade

Let's discuss what we're going to port to ng2 in this example and what will stay in ng1.

The Homepage



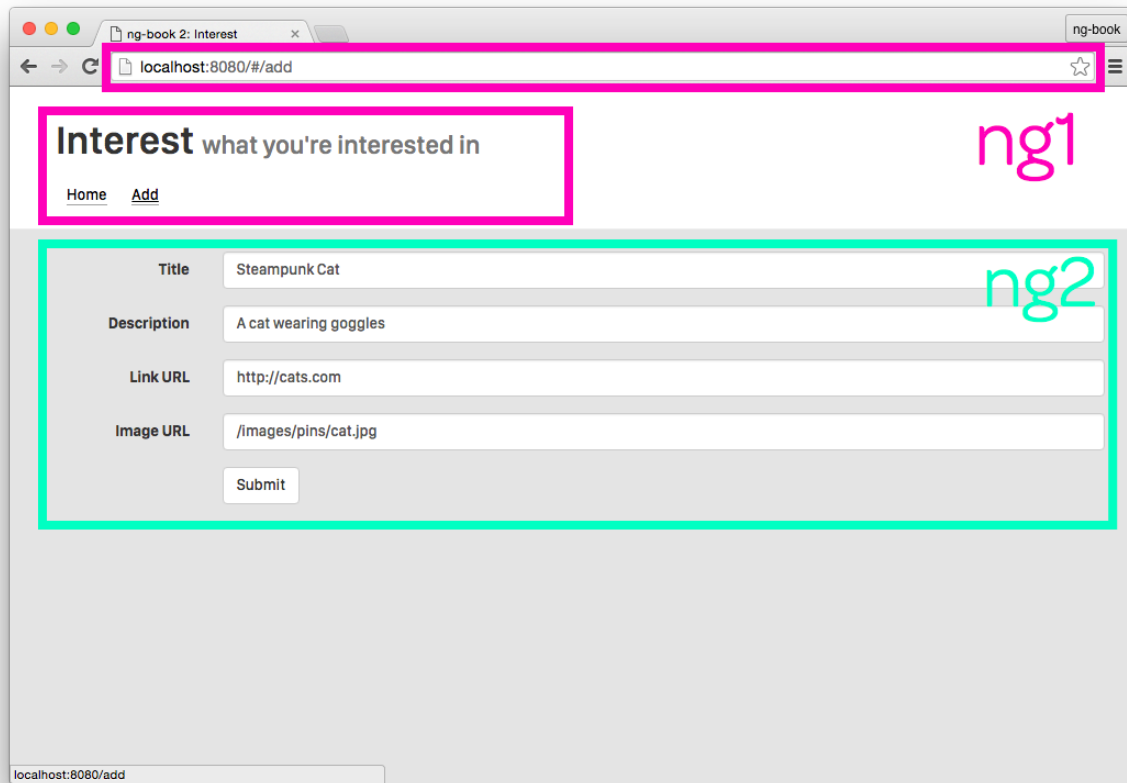
Homepage ng1 and ng2 Components

The first thing to notice is that we're going to continue to manage routing with ng1. Of course, Angular 4 has its own routing, which you can read about in [our routing chapter](#). But if you're building a hybrid app, you probably have lots of routes configured with AngularJS 1 and so in this example we'll continue to use ui-router for the routing.

On the homepage, we're going to nest a ng2 component within an ng1 directive. In this case, we're going to convert the "pin controls" to a ng2 component. That is, our ng1 pin directive, will call out to the ng2 pin-controls component and pin-controls will render the fav heart.

It's a small example that shows a powerful idea: how to seamlessly exchange data between ng versions.

The About Page



About Page ng1 and ng2 Components

We're going to use ng1 for the router and header on the about page as well. However on the about page, we're going to replace the whole form with a ng2 component: AddPinComponent.

If you recall, the form will add a new pin to the PinsService, and so in this example we're going to need to somehow make the (ng1) PinsService accessible to the (ng2) AddPinComponent.

Also, remember that when a new pin is added, the app should be redirected to the homepage. However, to change routes we need to use the ui-router \$state service (ng1) in the AddPinComponent (ng2). So we also need to make sure the \$state service can be used in AddPinComponent as well.

Services

So far we've talked about two ng1 services that will be *upgraded* to ng2:

- PinsService and
- \$state

We also want to explore “downgrading” a ng2 service to be used by ng1. For this, later on in the chapter, we’ll create an `AnalyticsService` in `TypeScript/ng2` that we share with ng1.

Taking Inventory

So to recap we’re going to “cross-expose” the following:

- Downgrade the ng2 `PinControlsComponent` to ng1 (for the fav buttons)
- Downgrade the ng2 `AddPinComponent` to ng1 (for the add pin page)
- Downgrade the ng2 `AnalyticsService` to ng1 (for recording events)
- Upgrade the ng1 `PinsService` to ng2 (for adding new pins)
- Upgrade the ng1 `$state` service to ng2 (for controlling routes)

A Minor Detour: Typing Files

One of the great things about TypeScript is the compile-time typing. However, if you’re building a hybrid app, I suspect that you’ve got a lot of untyped JavaScript code that you’re going to be integrating into this project.

When you try to use your JavaScript code from TypeScript you may get compiler errors because the compiler doesn’t know the structure of your JavaScript objects. You could try casting everything to `<any>` but that is ugly and error prone.

The better solution is to, instead, provide your TypeScript compiler with custom *type decorators*. Then the compiler will be able to enforce the types of your JavaScript code.

For instance, remember how in our ng1 app we created a pin object in `makeNewPin`?

`code/upgrade/ng1/js/app.js`

```
67  var makeNewPin = function() {
68    return {
69      "title": "Steampunk Cat",
70      "description": "A cat wearing goggles",
71      "user_name": "me",
72      "avatar_src": "images/avatars/me.jpg",
73      "src": "/images/pins/cat.jpg",
74      "url": "http://cats.com",
75      "faved": false,
76      "id": Math.floor(Math.random() * 10000).toString()
77    }
78  }
79
80  ctrl.newPin = makeNewPin();
```

It would be nice if we could tell the compiler about the structure of these objects and not resort to using `any` everywhere.

Furthermore, we're going to be using the `ui-router $state` service in Angular 4 / TypeScript, and we need to tell the compiler what functions are available there, too.

So while providing TypeScript custom type definitions is a TypeScript (and not an Angular-specific) chore, it's a chore we need to do nonetheless. And it's something that many people haven't done yet because TypeScript is, at time of publishing, relatively new.

So in this section I want to walk through how you deal with custom typings in TypeScript.



If you're already familiar with how to create and use TypeScript type definition files, you can safely skim this section.

Typing Files

In TypeScript we can describe the structure of our code by writing *typing definition files*. Typing definition files generally end in the extension `.d.ts`.

Generally, when you write TypeScript code, you don't need to write a `.d.ts` because your TypeScript code itself contains types. We write `.d.ts` files when we have some external JavaScript code that we want to add typing to after the fact.

For instance, in describing our pin object, we could write an interface for it like so:

`code/upgrade/hybrid/src/js/app.d.ts`

```
1 interface Pin {  
2   title: string;  
3   description: string;  
4   user_name: string;  
5   avatar_src: string;  
6   src: string;  
7   url: string;  
8   faved: boolean;  
9   id: string;  
10 }
```

Notice that we're not declaring a class, and we're not creating an instance. Instead, we're defining the shape (types) of an interface.

In order to use `.d.ts` files, you need to tell the TypeScript compiler where they are. The easiest way to do this is by adding a reference to `typings.d.ts`. For instance in `typings.d.ts` we'll add this:

```
1  /// <reference path="./js/app.d.ts"/>
```

We'll write `app.d.ts` in a little bit. First, let's explore a tool that exists to help us with third-party TypeScript definition files: typings.

Third-party libraries with @types

Typescript allows for loading third-party types via NPM.

We're going to use `angular-ui-router` with our app, so let's install the typings for `angular-ui-router`. To get this setup, all we have to do is install the `@types/angular-ui-router` package.

```
1  npm install @types/angular-ui-router --save
```

Now, by default, TypeScript will read types from the `node_modules/@types/` directory. We'll look at how we use these types in our code in a moment.

Custom Typing Files

Being able to use third-party typing files is great, but there are going to be situations where typing files don't already exist: especially in the case of our own code.

Generally, when we write custom typing files we co-locate the file alongside its respective JavaScript code. So let's create the file `js/app.d.ts`:

`code/upgrade/hybrid/src/js/app.d.ts`

```
1  interface Pin {
2    title: string;
3    description: string;
4    user_name: string;
5    avatar_src: string;
6    src: string;
7    url: string;
8    faved: boolean;
9    id: string;
10 }
11
12 interface PinsService {
13   pins(): Promise<Pin[]>;
14   addPin(pin: Pin): Promise<any>;
15 }
```

Here we're making an "ambient declaration" and the idea is that we're defining a variable that didn't originate from a TypeScript file. In this case, we're defining two interfaces:

1. `Pin`
2. `PinsService`

The `Pin` interface describes the keys and value-types of a pin object.

The `PinsService` interface describes the types of our two methods on our `PinsService`.

- `pins()` returns a `Promise` of an array of `Pins`
- `addPin()` takes a `Pin` as an argument and returns a `Promise`



Learn More about Writing Type Definition Files

If you'd like to learn more about writing `.d.ts` files, checkout these helpful links:

- [TypeScript Handbook: Working with other JavaScript Libraries](http://www.typescriptlang.org/Handbook#modules-working-with-other-javascript-libraries)¹⁴⁸
- [TypeScript Handbook: Writing definition files](https://github.com/Microsoft/TypeScript-Handbook/blob/master/pages/Writing%20Definition%20Files.md)¹⁴⁹
- [Quick tip: Typescript declare keyword](http://blogs.microsoft.co.il/gilf/2013/07/22/quick-tip-typescript-declare-keyword/)¹⁵⁰

Now that we have this file setup, TypeScript will know about the `Pin` and `PinsService` types in our code.

Writing `ng2 PinControlsComponent`

Now that we have the typings figured out, let's turn our attention back to the hybrid app.

The first thing we're going to do is write the `ng2 PinControlsComponent`. This will be an `ng2` component nested within an `ng1` directive. The `PinControlsComponent` displays the fav hearts and toggles fav'ing a pin.

Next, let's write our component:

¹⁴⁸<http://www.typescriptlang.org/Handbook#modules-working-with-other-javascript-libraries>

¹⁴⁹<https://github.com/Microsoft/TypeScript-Handbook/blob/master/pages/Writing%20Definition%20Files.md>

¹⁵⁰<http://blogs.microsoft.co.il/gilf/2013/07/22/quick-tip-typescript-declare-keyword/>

code/upgrade/hybrid/src/app/pin-controls/pin-controls.component.ts

```

1  import {
2    Component,
3    Input,
4    Output,
5    EventEmitter
6  } from '@angular/core';
7
8  @Component({
9    selector: 'pin-controls',
10   templateUrl: './pin-controls.component.html',
11   styleUrls: ['./pin-controls.component.css']
12 })
13 export class PinControlsComponent {
14   @Input() pin: Pin;
15   @Output() faved: EventEmitter<Pin> = new EventEmitter<Pin>();
16
17   toggleFav(): void {
18     this.faved.emit(this.pin);
19   }
20 }

```

Notice here that we'll match the element pin-controls.

Our template looks very similar to the ng1 version except we're using the ng2 template syntax for (click) and *ngIf.

Now the component definition class:

code/upgrade/hybrid/src/app/pin-controls/pin-controls.component.html

```

1  <div class="controls">
2    <div class="heart">
3      <a (click)="toggleFav()">
4        
5        
6      </a>
7    </div>
8  </div>

```

Notice that instead of specifying inputs and outputs in the @Component decorator, in this case we're annotating the properties on the class directly with the @Input and @Output decorators. This is a convenient way to us to provide typings to these properties.

This component will take an input of `pin`, which is the `Pin` object we're controlling.

This component specifies an output of `faved`. This is a little bit different than how we did it in the `ng1` app. If you look at `toggleFav` all we're doing is emitting (on the `EventEmitter`) the current `pin`.

The idea here is that we've already implemented how to change the `faved` state in `ng1` and we may not want to re-implement that functionality `ng2` (you may want to, it just depends on your team conventions).

Using `ng2` `PinControlsComponent`

Now that we have an `ng2` `pin-controls` component, we can now use it in a **AngularJS 1** template. Here's what our `pin.html` template looks like now:

code/upgrade/hybrid/src/assets/templates/pin.html

```

1 <div class="col-sm-6 col-md-4">
2   <div class="thumbnail">
3     <div class="content">
4       
5       <div class="caption">
6         <h3>{{pin.title}}</h3>
7         <p>{{pin.description | truncate:100}}</p>
8       </div>
9       <div class="attribution">
10        
11        <h4>{{pin.user_name}}</h4>
12      </div>
13    </div>
14    <div class="overlay">
15      <pin-controls [pin]="pin"
16                    (faved)="toggleFav($event)"></pin-controls>
17    </div>
18  </div>
19 </div>

```

This template is for an `ng1` directive, and we can use `ng1` directives such as `ng-src`. However, notice the line where we use our `ng2` `pin-controls` component:

```

1 <pin-controls [pin]="pin"
2             (faved)="toggleFav($event)"></pin-controls>

```

What's interesting here is that we're using the ng2 input bracket syntax [pin] and the ng2 output parentheses syntax (faved).

In a hybrid app **when you use ng2 directives in ng1, you still use the ng2 syntax.**

With our input [pin] we're passing the pin which comes from the scope of the ng1 directive.

With our output (faved) we're calling the toggleFav function on the scope of the ng1 directive. Notice what we did here: we didn't modify the pin.faved state within the ng2 directive (although, we could have). Instead, we asked the ng2 PinControlsComponent to simply emit the pin when toggleFav is called there. (If this is confusing, take a second look at toggleFav of PinControlsComponent.)

Again, the reason we do this is because we're showing how you can keep your existing functionality (scope.toggleFav) in ng1, but start porting over components to ng2. In this case, the ng1 pin directive listens for the faved event on the ng2 PinControlsComponent.

If you refresh your page now, you'll notice that it doesn't work. That's because there's one more thing we need to do: downgrade PinControlsComponent to ng1.

Downgrading ng2 PinControlsComponent to ng1

The final step to using our components across ng2/ng1 borders is to use our UpgradeAdapter to downgrade our components (or upgrade, as we'll see in a bit).

We perform this downgrade in our app.module.ts file

First we need to import the necessary libraries and declare the angular variable:

code/upgrade/hybrid/src/app/app.module.ts

```
1  import {
2    NgModule,
3    forwardRef
4  } from '@angular/core';
5  import { UpgradeAdapter } from '@angular/upgrade';
6  import { BrowserModule } from '@angular/platform-browser';
7
8  import { FormsModule } from '@angular/forms';
9  import { HttpClientModule } from '@angular/http';
10
11 import { AppComponent } from './app.component';
12 import { AddPinComponent } from './add-pin/add-pin.component';
13 import { PinControlsComponent } from './pin-controls/pin-controls.component';
14 import { AnalyticsService } from './analytics.service';
15
16 declare var angular: any;
```

Then we create a `.directive` in (almost) the normal ng1 way:

code/upgrade/hybrid/src/app/app.module.ts

```

16 declare var angular: any;
17
18 /*
19  * Create our upgradeAdapter
20  */
21 export const upgradeAdapter: UpgradeAdapter = new UpgradeAdapter(
22   forwardRef(() => AppModule));
23
24 /*
25  * Expose our ng2 content to ng1
26  */
27 angular.module('interestApp')
28   .directive('pinControls',
29     upgradeAdapter.downgradeNg2Component(PinControlsComponent))

```

Remember that our ng1 app calls `angular.module('interestApp', [])`. That is, our ng1 app has already registered the `interestApp` module with `angular`.

Now we want to look up that module by calling `angular.module('interestApp')` and then add directives to it, just like we do in ng1 normally.



angular.module getter and setter syntax

If you recall, when we pass an array as the second argument to `angular.module`, we are *creating* a module. That is, `angular.module('foo', [])` will *create* the module `foo`. Informally, we call this the “setter” syntax.

Similarly, if we omit the array we are *getting* a module (that is assumed to already exist). That is, `angular.module('foo')` will *get* the module `foo`. We call this the “getter” syntax.

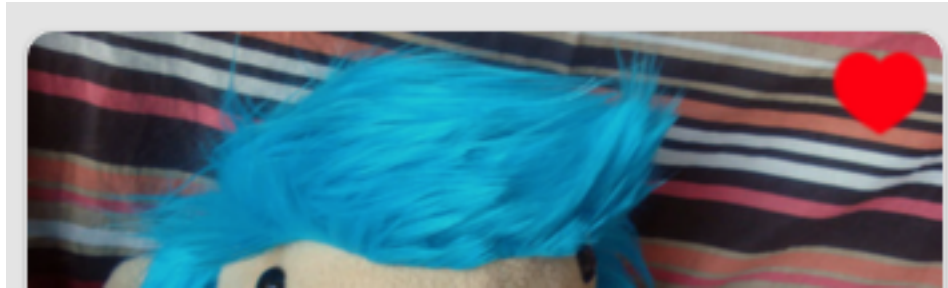


In this example, if you forget this distinction and call `angular.module('interestApp', [])` in `app.ts` (ng2) then you will accidentally overwrite your existing `interestApp` module and your app won't work. Careful!

We're calling `.directive` and creating a directive called `'pinControls'`. This is standard ng1 practice. For the second argument, the directive definition object (DDO), we don't create the DDO manually. Instead, we call `upgradeAdapter.downgradeNg2Component`.

`downgradeNg2Component` will convert our `PinControlsComponent` into an ng1-compatible directive. Pretty neat.

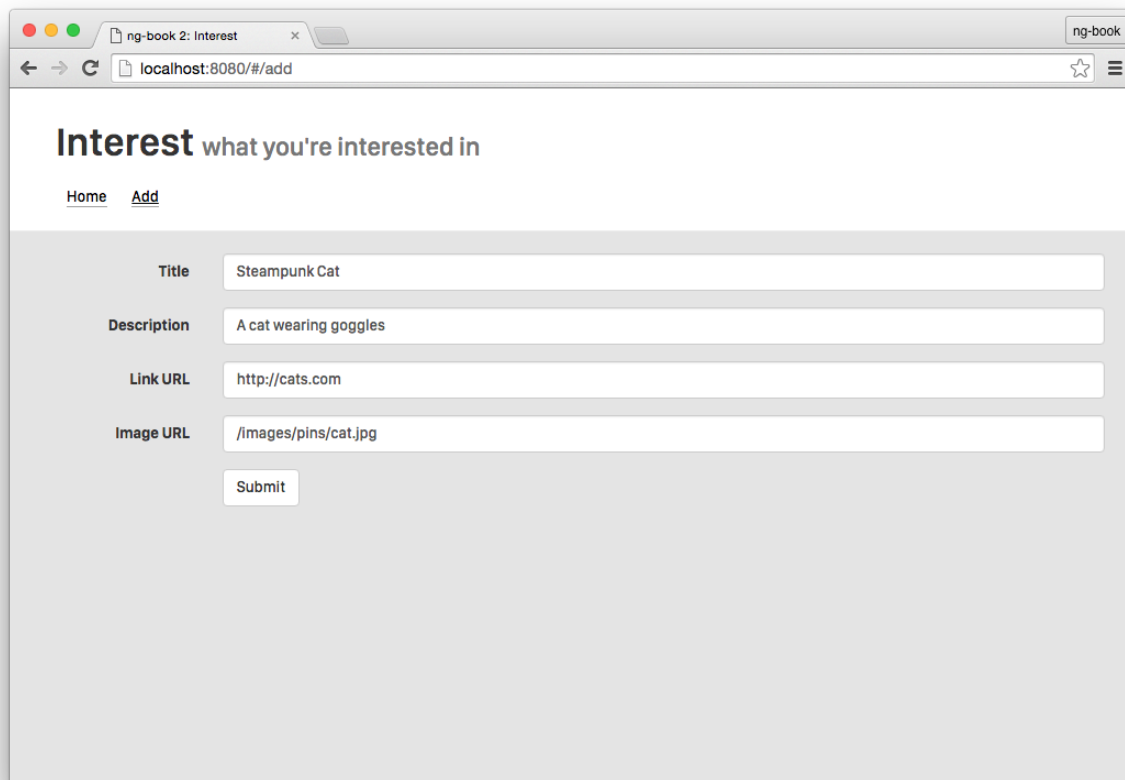
Now if you try refreshing, you'll notice that our faving works just like before, only now we're using ng2 embedded in ng1!



Faving works like a charm

Adding Pins with ng2

The next thing we want to do is upgrade the add pins page with an ng2 component.



The screenshot shows a web browser window with the title 'ng-book 2: Interest' and the URL 'localhost:8080/#/add'. The page content includes a header 'Interest what you're interested in' with links for 'Home' and 'Add'. Below the header is a form with the following fields:

Title	Steampunk Cat
Description	A cat wearing goggles
Link URL	http://cats.com
Image URL	/images/pins/cat.jpg

Below the form is a 'Submit' button.

Adding a New Pin Form

If you recall, this page does three things:

1. Present a form to the user for describing the pin
2. Use the `PinsService` to add the new pin to the list of pins
3. Redirect the user to the homepage

Let's think through how we're going to do these things from ng2.

Angular 4 provides a robust forms library. So there's no complication here. We're going to write a straight ng2 form.

However the `PinsService` comes from ng1. Often we have many existing services in ng1 and we don't have time to upgrade them all. So for this example, we're going to keep `PinsService` as an ng1 object, and *inject it into ng2*.

Similarly, we're using `ui-router` in ng1 for our routing. To change pages in `ui-router` we have to use the `$state` service, which is an ng1 service.

So what we're going to do is **upgrade** the `PinsService` and the `$state` service from ng1 to ng2. And this couldn't be any easier.

Upgrading ng1 `PinsService` and `$state` to ng2

To upgrade ng1 services we call `upgradeAdapter.upgradeNg1Provider`:

`code/upgrade/hybrid/src/app/app.module.ts`

```

37  /*
38   * Expose our ng1 content to ng2
39   */
40  upgradeAdapter.upgradeNg1Provider('PinsService');
41  upgradeAdapter.upgradeNg1Provider('$state');
```

And that's it. Now we can `@Inject` our ng1 services into ng2 components like so:

```

1  // angular.ui.IStateService is available because we've
2  // installed @types/angular-ui-router in our package.json
3  type IStateService = angular.ui.IStateService;
4
5  class AddPinComponent {
6    constructor(@Inject('PinsService') public pinsService: PinsService,
7               @Inject('$state') public uiState: IStateService) {
8    }
9    // ...
10   // now you can use this.pinsService
11   // or this.uiState
12   // ...
13 }
```


In this constructor, there's a few things to look at:

The `@Inject` decorator, says that we want the next variable to be assigned the value of what the injection will resolve to. In the first case, that would be our `ng1 PinsService`.

In TypeScript, in a constructor when you use the `public` keyword, it is a shorthand for assigning that variable to `this`. That is, here when we say `public pinsService` what we're saying is, 1. declare a property `pinsService` on instances of this class and 2. assign the constructor argument `pinsService` to `this.pinsService`.

The result is that we can access `this.pinsService` throughout our class.

Lastly we define the type of both services we're injecting: `PinsService` and `IStateService`.

`PinsService` comes from the `app.d.ts` we defined previously:

`code/upgrade/hybrid/src/js/app.d.ts`

```

12 interface PinsService {
13   pins(): Promise<Pin[]>;
14   addPin(pin: Pin): Promise<any>;
15 }

```

And `IStateService` comes from the typings for `ui-router`, which we installed with typings.

By telling TypeScript the types of these services we can enjoy type-checking as we write our code.

Let's write the rest of our `AddPinComponent`.

Writing ng2 `AddPinComponent`

We start by importing the types we need:

`code/upgrade/hybrid/src/app/add-pin/add-pin.component.ts`

```

1 declare var angular: any;
2 import {
3   Component,
4   Inject
5 } from '@angular/core';
6 // angular.ui.IStateService is available because we've
7 // installed @types/angular-ui-router in our package.json
8 type IStateService = angular.ui.IStateService;

```

Again, notice that we're importing our custom types `Pin` and `PinsService`. And we're also importing `IStateService` from `angular-ui-router`.

`AddPinComponent @Component`

Our `@Component` is straightforward:

code/upgrade/hybrid/src/app/add-pin/add-pin.component.ts

```

10 @Component({
11   selector: 'add-pin',
12   templateUrl: './add-pin.component.html',
13   styleUrls: ['./add-pin.component.css']
14 })

```

AddPinComponent **template**

We're loading our template using a `templateUrl`. In that template, we setup our form much like the ng1 form, only we're using ng2 form directives.



We're not going to describe `ngModel` / `ngSubmit` deeply here. If you'd like to know more about how Angular 4 forms work, checkout [the forms chapter](#), where we describe forms in depth.

code/upgrade/hybrid/src/app/add-pin/add-pin.component.html

```

1 <div class="container">
2   <div class="row">
3
4     <form (ngSubmit)="onSubmit()"
5       class="form-horizontal">
6
7       <div class="form-group">
8         <label for="title"
9           class="col-sm-2 control-label">Title</label>
10        <div class="col-sm-10">
11          <input type="text"
12            class="form-control"
13            id="title"
14            name="title"
15            placeholder="Title"
16            [(ngModel)]="newPin.title">
17        </div>

```

We're using two directives here: `ngSubmit` and `ngModel`.

We use `(ngSubmit)` on the form to call the `onSubmit` function when the form is submitted. (We'll define `onSubmit` on the `AddPinComponent` controller below.)

We use `[(ngModel)]` to bind the value of the `title` input tag to the value of `newPin.title` on the controller.

Here's the full listing of the template:

code/upgrade/hybrid/src/app/add-pin/add-pin.component.html

```
1 <div class="container">
2   <div class="row">
3
4     <form (ngSubmit)="onSubmit()"
5       class="form-horizontal">
6
7       <div class="form-group">
8         <label for="title"
9           class="col-sm-2 control-label">Title</label>
10        <div class="col-sm-10">
11          <input type="text"
12            class="form-control"
13            id="title"
14            name="title"
15            placeholder="Title"
16            [(ngModel)]="newPin.title">
17        </div>
18      </div>
19
20      <div class="form-group">
21        <label for="description"
22          class="col-sm-2 control-label">Description</label>
23        <div class="col-sm-10">
24          <input type="text"
25            class="form-control"
26            id="description"
27            name="description"
28            placeholder="Description"
29            [(ngModel)]="newPin.description">
30        </div>
31      </div>
32
33      <div class="form-group">
34        <label for="url"
35          class="col-sm-2 control-label">Link URL</label>
36        <div class="col-sm-10">
37          <input type="text"
38            class="form-control"
39            id="url"
40            name="url"
41            placeholder="Link URL">
```

```

42         [(ngModel)]="newPin.url">
43     </div>
44 </div>
45
46 <div class="form-group">
47     <label for="url"
48         class="col-sm-2 control-label">Image URL</label>
49     <div class="col-sm-10">
50         <input type="text"
51             class="form-control"
52             id="url"
53             name="url"
54             placeholder="Image URL"
55             [(ngModel)]="newPin.src">
56     </div>
57 </div>
58
59 <div class="form-group">
60     <div class="col-sm-offset-2 col-sm-10">
61         <button type="submit"
62             class="btn btn-default"
63             >Submit</button>
64     </div>
65 </div>
66 <div *ngIf="saving">
67     Saving...
68 </div>
69 </form>

```

AddPinComponent Controller

Now we can define AddPinComponent. We start by setting up two instance variables:

code/upgrade/hybrid/src/app/add-pin/add-pin.component.ts

```

15 export class AddPinComponent {
16     saving = false;
17     newPin: Pin;

```

We use saving to indicate to the user that the save is in progress and we use newPin to store the Pin we're working with.

code/upgrade/hybrid/src/app/add-pin/add-pin.component.ts

```

19   constructor(@Inject('PinsService') private pinsService: PinsService,
20               @Inject('$state') private uiState: IStateService) {
21       this.newPin = this.makeNewPin();
22   }

```

In our constructor we Inject the services, as we discussed above. We also set `this.newPin` to the value of `makeNewPin`, which we'll define now:

code/upgrade/hybrid/src/app/add-pin/add-pin.component.ts

```

24   makeNewPin(): Pin {
25       return {
26           title: 'Steampunk Cat',
27           description: 'A cat wearing goggles',
28           user_name: 'me',
29           avatar_src: '/assets/images/avatars/me.jpg',
30           src: '/assets/images/pins/cat.jpg',
31           url: 'http://cats.com',
32           faved: false,
33           id: Math.floor(Math.random() * 10000).toString()
34       };
35   }

```

This looks a lot like how we defined it in ng1, only now we have the benefit of it being typed.

When the form is submitted, we call `onSubmit`. Let's define that:

code/upgrade/hybrid/src/app/add-pin/add-pin.component.ts

```

37   onSubmit(): void {
38       this.saving = true;
39       console.log('submitted', this.newPin);
40       setTimeout(() => {
41           this.pinsService.addPin(this.newPin).then(() => {
42               this.newPin = this.makeNewPin();
43               this.saving = false;
44               this.uiState.go('home');
45           });
46       }, 2000);
47   }

```

Again, we're using a timeout to *simulate* the effect of what would happen if we had to call out to a server to save this pin. Here, we're using `setTimeout`. Compare that to how we defined this function in ng1:

code/upgrade/ng1/js/app.js

```

82  ctrl.submitPin = function() {
83      ctrl.saving = true;
84      $timeout(function() {
85          PinsService.addPin(ctrl.newPin).then(function() {
86              ctrl.newPin = makeNewPin();
87              ctrl.saving = false;
88              $state.go('home');
89          });
90      }, 2000);
91  }

```

Notice that in ng1 we had to use the `$timeout` service. Why is that? Because ng1 is based around the digest loop. If you use `setTimeout` in ng1, then when the callback function is called, it's "outside" of angular and so your changes aren't propagated unless something kicks off a digest loop (e.g. using `$scope.apply`).

However in ng2, we can use `setTimeout` directly because change detection in ng2 uses Zones and is therefore, more or less automatic. We don't need to worry about the digest loop in the same way, which is really nice.

In `onSubmit` we're calling out to the `PinsService` by:

```

1  this.pinsService.addPin(this.newPin).then(() => {
2      // ...
3  });

```

Again, the `PinsService` is accessible via `this.pinsService` because of how we defined the constructor. The compiler doesn't complain because we said that `addPin` takes a `Pin` as the first argument in our `app.d.ts`:

code/upgrade/hybrid/src/js/app.d.ts

```

13  pins(): Promise<Pin[]>;
14  addPin(pin: Pin): Promise<any>;
15  }

```

And we defined `this.newPin` to be a `Pin`.

After `addPin` resolves, we reset the pin using `makeNewPin` and set `this.saving = false`.

To go back to the homepage, we use the `ui-router` `$state` service, which we stored as `this.uiState`. So we can change states by calling `this.uiState.go('home')`.

Using AddPinComponent

Now let's use the AddPinComponent.

Downgrade ng2 AddPinComponent

To use AddPinComponent we need to downgrade it:

code/upgrade/hybrid/src/app/app.module.ts

```
27 angular.module('interestApp')
28   .directive('pinControls',
29     upgradeAdapter.downgradeNg2Component(PinControlsComponent))
30   .directive('addPin',
31     upgradeAdapter.downgradeNg2Component(AddPinComponent));
```

This will create the addPin directive in ng1, which will match the tag <add-pin>.

Routing to add-pin

In order to use our new AddPinComponent page, we need to place it somewhere within our ng1 app. What we're going to do is take the add state in our router and just set the <add-pin> directive to be the template:

code/upgrade/hybrid/src/js/app.js

```
39 .state('add', {
40   template: "<add-pin></add-pin>",
41   url: '/add',
42   resolve: {
43     'pins': function(PinsService) {
44       return PinsService.pins();
45     }
46   }
47 })
```

Exposing an ng2 service to ng1

So far we've downgraded ng2 components to be used in ng2, and upgraded ng1 services to be used in ng2. But as our application start converting over to ng2, we'll probably start writing services in Typescript/ng2 that we'll want to expose to our ng1 code.

Let's create a simple service in ng2: an "analytics" service that will record events.

The idea is that we have an AnalyticsService in our app that we use to recordEvents. In reality, we're just going to console.log the event and store it in an array. But it gives us a chance to focus on what's important: describing how we share a ng2 service with ng1.

Writing the AnalyticsService

Let's take a look at the AnalyticsService implementation:

code/upgrade/hybrid/src/app/analytics.service.ts

```
1 import { Injectable } from '@angular/core';
2
3 /**
4  * Analytics Service records metrics about what the user is doing
5  */
6 @Injectable()
7 export class AnalyticsService {
8   events: string[] = [];
9
10  public recordEvent(event: string): void {
11    console.log(`Event: ${event}`);
12    this.events.push(event);
13  }
14 }
```

There are two things to note here: 1. recordEvent and 2. being Injectable

recordEvent is straightforward: we take an event: string, log it, and store it in events. In your application you would probably send the event to an external service like Google Analytics or Mixpanel.

To make this service injectable, we do two things: 1. Annotate the class with @Injectable and 2. bind the token AnalyticsService to this class.



The @Injectable decorator really means that other dependencies can be injected into this service, but it's recommended to add it to all services, even those that don't have dependencies. Read more about @Injectable in the [chapter on dependency injection](#)

Now Angular will manage a singleton of this service and we will be able to inject it where we need it.

Downgrade ng2 AnalyticsService to ng1

Before we can use the AnalyticsService in ng1, we need to downgrade it.

The process of downgrading an ng2 service to ng1 is similar to the process of downgrading a directive, but there is one extra step: we need to make sure AnalyticsService is in the list of providers for our NgModule:

code/upgrade/hybrid/src/app/app.module.ts

```
43 @NgModule({
44   declarations: [
45     AppComponent,
46     AddPinComponent,
47     PinControlsComponent
48   ],
49   imports: [
50     BrowserModule,
51     FormsModule,
52     HttpClientModule
53   ],
54   providers: [
55     AnalyticsService
56   ]
57 })
58 export class AppModule { }
```

Then we can use `downgradeNg2Provider`:

code/upgrade/hybrid/src/app/app.module.ts

```
33 angular.module('interestApp')
34   .factory('AnalyticsService',
35     upgradeAdapter.downgradeNg2Provider(AnalyticsService));
```

We call `angular.module('interestApp')` to get our ng1 module and then call `.factory` like we would in ng1. To downgrade the service, we call

`upgradeAdapter.downgradeNg2Provider(AnalyticsService)`, which wraps our `AnalyticsService` in a function that adapts it to an ng1 factory.

Using AnalyticsService in ng1

Now we can inject our ng2 `AnalyticsService` into ng1. Let's say we want to record whenever the `HomeController` is visited. We could record this event like so:

code/upgrade/hybrid/src/js/app.js

```
60 .controller('HomeController', function(pins, AnalyticsService) {
61     AnalyticsService.recordEvent('HomeControllerVisited');
62     this.pins = pins;
63 })
```

Here we inject AnalyticsService as if it was a normal ng1 service we call recordEvent. Fantastic!

We can use this service anywhere we would use injection in ng1. For instance, we can also inject the AnalyticsService into our ng1 pin directive:

code/upgrade/hybrid/src/js/app.js

```
64 .directive('pin', function(AnalyticsService) {
65     return {
66         restrict: 'E',
67         templateUrl: '/assets/templates/pin.html',
68         scope: {
69             'pin': "=item"
70         },
71         link: function(scope, elem, attrs) {
72             scope.toggleFav = function() {
73                 AnalyticsService.recordEvent('PinFaved');
74                 scope.pin.faved = !scope.pin.faved;
75             }
76         }
77     }
78 })
```

Summary

Now you have all the tools you need to start upgrading your ng1 app to a hybrid ng1/ng2 app. The interoperability between ng1 and ng2 works very well and we owe a lot to the Angular team for making this so easy.

Being able to exchange directives and services between ng1 and ng2 make it super easy to start upgrading your apps. We can't always upgrade our apps to ng2 overnight, but the UpgradeAdapter lets us start using ng2 - without having to throw our old code away.

References

If you're looking to learn more about hybrid Angular apps, here are a few resources:

- [The Official Angular Upgrade Guide](https://angular.io/docs/ts/latest/guide/upgrade.html)¹⁵¹
- [The Angular2 Upgrade Spec Test](https://github.com/angular/angular/blob/master/modules/angular2/test/upgrade/upgrade_spec.ts)¹⁵²
- [The Angular2 Source for DowngradeNg2ComponentAdapter](https://github.com/angular/angular/blob/master/modules/angular2/src/upgrade/downgrade_ng2_adapter.ts)¹⁵³

¹⁵¹<https://angular.io/docs/ts/latest/guide/upgrade.html>

¹⁵²https://github.com/angular/angular/blob/master/modules/angular2/test/upgrade/upgrade_spec.ts

¹⁵³https://github.com/angular/angular/blob/master/modules/angular2/src/upgrade/downgrade_ng2_adapter.ts