v1.5.6-build.4825 (snapshot

/ Tutorial (tutorial) / 4 - Directory and File Organization (tutorial/step_04)

tutorial/step_04.ngdoc?message=docs(tutorial%2F4 - Directory and File Organization)%3A%20describe%20your%20change...)

Previous (tutorial/step_03) Live Demo (http://angular.github.io/angular-phonecat/step-4/app) Code Diff (https://github.com/angular/angular-phonecat/compare/step-3...step-4)
Next (tutorial/step_05)

In this step, we will not be adding any new functionality to our application. Instead, we are going to take a step back, refactor our codebase and move files and code around, in order to make our application more easily expandable and maintainable.

In the previous step, we saw how to architect our application to be modular and testable. What's equally important though, is organizing our codebase in a way that makes it easy (both for us and other developers on our team) to navigate through the code and quickly locate the pieces that are relevant to a specific feature or section of the application.

To that end, we will explain why and how we:

- Put each entity in its **own file**.
- Organize our code by **feature area**, instead of by function.
- Split our code into **modules** that other modules can depend on.

We will keep it short, not going into great detail on every good

practice and convention. These principles are explained in great detail in the Angular Style Guide (https://github.com/johnpapa/angular-styleguide/blob/master/a1/README.md), which also contains many more techniques for effectively organizing Angular codebases.

Workspace Reset Instructions ➤

The most important changes are listed below. You can see the full diff on GitHub (https://github.com/angular/angular-phonecat/compare/step-3...step-4).

One Feature per File

It might be tempting, for the sake of simplicity, to put everything in one file, or have one file per type; e.g. all controllers is one file, all components in another file, all services in a third file, and so on. This might seem to work well in the beginning, but as our application grows it becomes a burden to maintain. As we add more and more features, our files will get bigger and bigger and it will be difficult to navigate and find the code we are looking for.

Instead we should put each feature/entity in its own file. Each stand-alone controller will be defined in its own file, each component will be defined in each own file, etc.

Luckily, we don't need to change anything with respect to that guideline in our code, since we have already defined our `phoneList` component in its own `phone-list.component.js` file. Good job!

We will keep this in mind though, as we add more features.

Organizing by Feature

So, now that we learned we should put everything in its own file, our `app/` directory will soon be full with dozens of files and specs (remember we keep our unit test files next to the corresponding source code files). What's more important, logically related files will not be grouped together; it will be really difficult of locate all files related to a specific section of the application and make a change or fix a bug.

So, what shall we do?

Well, we are going to group our files into directories *by feature*. For example, since we have a section in our application that lists phones, we will put all related files into a `phone-list/` directory under `app/`. We are soon to find out that certain features are used across different parts of the application. We will put those inside `app/core/`.

> Other typical names for our `core` directory are `shared`, `common` and `components`. The last one is kind of misleading though, as it will contain other things than components as well.
>
> (This is mostly a relic of the past, when "components" just meant the generic building blocks of an application.)

Based on what we have discussed so far, here is our directory/file layout for the `phoneList` "feature":

```
app/
  phone-list/
    phone-list.component.js
    phone-list.component.spec.js
  app.js
```

Using Modules

As previously mentioned, one of the benefits of having a modular architecture is code reuse — not only inside the same application, but across applications too. There is one final step in making this code reuse frictionless:

- Each feature/section should declare its own module and all related entities should register themselves on that module.

Let's take the `phoneList` feature as an example. Previously, the `phoneList` component would register itself on the `phonecatApp` module:

```
angular.
  module('phonecatApp').
  component('phoneList', ...);
```

Similarly, the accompanying spec file loads the `phonecatApp` module before each test (because that's where our component is registered). Now, imagine that we need a list of phones on another project that we are working on. Thanks to our modular architecture, we don't have to reinvent the wheel; we simply copy the `phone-list/` directory on our other project and add the necessary script tags in our `index.html` file and we are done, right ?

Well, not so fast. The new project doesn't know anything about a `phonecatApp` module. So, we would have to replace all references to `phonecatApp` with the name of this project's main module. As you can imagine this is both laborious and error-prone.

Yeah, you guessed it: There is a better way !

Each feature/section, will declare its own module and have all related entities registered there. The main module (`phonecatApp`) will declare a dependency on each feature/section module. Now, all it takes to reuse the same code on new project is copying the feature directory over and adding the feature module as a dependency in the new project's main module.

Here is what our `phoneList` feature will look like after this change:

`/`:

```
app/
  phone-list/
    phone-list.module.js
    phone-list.component.js
    phone-list.component.spec.js
  app.module.js
```

`app/phone-list/phone-list.module.js`:

```
// Define the `phoneList` module
angular.module('phoneList', []);
```

`app/phone-list/phone-list.component.js`:

```
// Register the `phoneList` component on the `phoneList`
module,
angular.
  module('phoneList').
  component('phoneList', {...});
```

app/app.module.js :
(since `app/app.js` now only contains the main module declaration, we gave it a `.module` suffix)

```
// Define the `phonecatApp` module
angular.module('phonecatApp', [
  // ...which depends on the `phoneList` module
  'phoneList'
]);
```

By passing `phoneList` inside the dependencies array when defining the `phonecatApp` module, Angular will make all entities registered on `phoneList` available on `phonecatApp` as well.

Don't forget to also update your `index.html` adding a `<script>` tag for each JacaScript file we have created. This might seem tedious, but is totally worth it.

In a production-ready application, you would concatenate and minify all your JavaScript files anyway (for performance reasons), so this won't be an issue any more.

Note that files defining a module (i.e. `.module.js`) need to be included before other files that add features (e.g. components, controllers, services, filters) to that module.

External Templates

Since we are at refactoring, let's do one more thing. As we learned, components have templates, which are basically fragments of HTML code that dictate how our data is laid out and presented to the user. In step 3 (tutorial/step_03), we saw how we can specify the template for a component as a string using the **`template`** property of the CDO (Component Definition Object). Having HTML code in a string isn't ideal, especially for bigger templates. It would be much better, if we could have our HTML code in `.html` files. This way, we would get all the support our IDE/editor has to offer (e.g. HTML-specific color-highlighting and auto-completion) and also keep our component definitions cleaner.

So, while it's perfectly fine to keep our component templates inline (using the **`template`** property of the CDO), we are going to use an external template for our `phoneList` component. In order to denote that we are using an external template, we use the `templateUrl` property and specify the URL that our template will be loaded from. Since we want to keep our template close to where the component is defined, we place it inside `app/phone-list/` .

We copied the contents of the **`template`** property (the HTML code) into `app/phone-list/phone-list.`**`template`**`.html` and modified our CDO like this:

`app/phone-list/phone-list.component.js`:

```
angular.
module('phoneList').
component('phoneList', {
  // Note: The URL is relative to our `index.html` file
  templateUrl: 'phone-list/phone-list.template.html',
  controller: ...
});
```

At runtime, when Angular needs to create an instance of the `phoneList` component, it will make an HTTP request to get the template from `app/phone-list/phone-list.template.html`.

Keeping inline with our convention, we will be using the `.template` suffix for external templates. Another common convention is to just have the `.html` extension (e.g. `phone-list.html`).

Using an external template like this, will result in more HTTP requests to the server (one for each external template). Although Angular takes care not to make extraneous requests (e.g. fetching the templates lazily, caching the results, etc), additional requests do have a cost (especially on mobile devices and data-plan connections).

Luckily, there are ways to avoid the extra costs (while still keeping your templates external). A detailed discussion of the subject is outside the scope of this tutorial, but you can take a look at the $templateRequest (api/ng/service/$templateRequest) and $templateCache (api/ng/service/$templateCache) services for more info on how Angular manages external templates.

Final Directory/File Layout

After all the refactorings that took place, this is how our application looks from the outside:

`/`:

```
app/
  phone-list/
    phone-list.component.js
    phone-list.component.spec.js
    phone-list.module.js
    phone-list.template.html
  app.css
  app.module.js
  index.html
```

Testing

Since this was just a refactoring step (no actual code addition/deletions), we shouldn't need to change much (if anything) as far as our specs are concerned.

One thing that we can (and should) change is the name of the module to be loaded before each test in `app/phone-list/phone-list.component.spec.js`. We don't need to pull in the whole `phonecatApp` module (which will soon grow to depend on more stuff). All we want to test is already included in the much smaller `phoneList` module, so it suffices to just load that. This is one extra benefit that we get out of our modular architecture for free.

app/phone-list/phone-list.component.spec.js :

```
describe('phoneList', function() {

  // Load the module that contains the `phoneList` component
before each test
  beforeEach(module('phoneList'));

  ...

});
```

If not already done so, run the tests (using the `npm test` command) and verify that they still pass.

> One of the great things about tests is the confidence they provide, when refactoring your application. It's easy to break something as you start moving files around and re-arranging modules. Having good test coverage is the quickest, easiest and most reliable way of knowing that your application will continue to work as expected.

Summary

Even if we didn't add any new and exciting functionality to our application, we have made a great step towards a well-architected and maintainable application. Time to spice things up. Let's go to step 5 (tutorial/step_05) to learn how to add full-text search to the application.

Previous (tutorial/step_03) **Live Demo** (http://angular.github.io/angular-phonecat/step-4/app) **Code Diff**

(https://github.com/angular/angular-phonecat/compare/step-3...step-4)