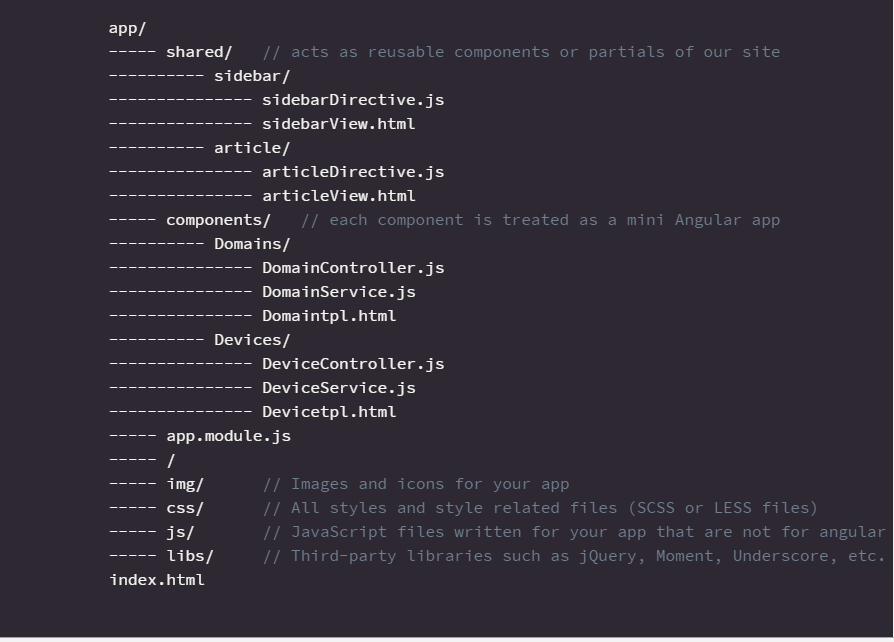
**[# AngularJS app Structure and Foundation](https://scotch.io/tutorials/angularjs-best-practices-directory-structure" \l "a-better-structure-and-foundation)**

**An ideal AngularJS app structure should be modularized into very specific functions.** We also want to take advantage of the wonderful AngularJS directives to further compartmentalize our apps. Take a look at a sample directory structure below:



Let’s dive into the directory structure above and see what’s going on here.

**INDEX.HTML**

The index.html lives at the root of front-end structure. The index.html file will primarily handle loading in all the libraries and Angular elements.

**ASSETS FOLDER**

The assets folder is also pretty standard. It will contain all the assets needed for your app that are not related your AngularJS code.

**APP FOLDER**

This is where the meat of your AngularJS app will live. We have two subfolders in here and a couple JavaScript files at the root of the folder. Theapp.module.js file will handle the setup of your app, load in AngularJS dependencies and so on. After that we have two subfolders – **components** and **shared**. Let’s dive into those next.

**COMPONENTS FOLDER**

The components folder will contain the actual sections for your Angular app. These will be the static views ,directives and services for that specific section of the site (think an admin users section, gallery creation section, etc). Each page should have it’s own subfolder with it’s own controller, services, and HTML files.

Each component here will resemble a mini-MVC application by having a view, controller and potentially services file(s). If the component has multiple related views, it may be a good idea to further separate these files into ‘views’, ‘controllers’, ‘services’ subfolders.

**So you could essentially think of this as multiple mini Angular applications inside of your giant Angular application.**

**SHARED FOLDER**

The shared folder will contain the individual features that your app will have. These features will ideally be directives that you will want to reuse on multiple pages.

Features such as article posts, user comments, sliders, and others should be crafted as AngularJS Directives. Each component here should have it’s own subfolder that contains the directive JavaScript file and the template HTML file.

In some instances, a directive may have it’s own services JavaScript file, and in the case that it does it should also go into this subfolder.

This allows us to have definitive components for our site so that a slider will be a slider across the site. You would probably want to build it so that you could pass in options to extend it. For example, you could have:

[**Benefits of the Modularized Approach**](https://scotch.io/tutorials/angularjs-best-practices-directory-structure#benefits-of-the-modularized-approach)

The example above shows a modularized approach to building AngularJS. The benefits of this approach include:

**CODE MAINTAINABILITY**

Follow the approach above will logically compartmentalize your apps and you will easily be able to locate and edit code.

**SCALABLE**

Your code will be much easier to scale. Adding new directives and pages will not add bloat to existing folders. Onboarding new developers should also be much easier once the structure is explained. Additionally, with this approach, you will be able to drop features in and out of your app with relative ease so testing new functionality or removing it should be a breeze.

**DEBUGGING**

Debugging your code will be much easier with this modularized approach to app development. It will be easier to find the offending pieces of code and fix them.

**TESTING**

Writing test scripts and testing modernized apps is a whole lot easier then non-modularized ones.

**Presentation Server ‘Grunt’ : :**

|  |  |
| --- | --- |
|  | Grunt is basically a build / task manager written on top of Node JS. I would call it the Node JS stack equivalent of ANT for Java. Here are some common scenarios you would want to use grunt under:   * You have a project with JavaScript files requiring magnification, and generally generating a front end build separately (in case you're using say JAVA for your backend). (**grunt-contrib-uglify**) * When you save code on your machine during development, you want the browser to reload your page automatically (might seem like a small thing, but believe me this has saved me lots of time). (**Live reload**) * When a developer saves code on his machine, he wants a comprehensive list of JS errors / general best practice violations to be shown. (**grunt-contrib-jshint**) * You have a project with SASS/ LESS files which need to be compiled to CSS files on the developers machine during development, For example whenever he saves a SASS file, you want it to be compiled to a CSS file automatically, for inclusion in your page. (**grunt-contrib-sass**) * You have a team of front end developers who're working on the UI, and a team of backend developers working on the backend, you want the front end devs to use the backend REST API's without having to compile & deploy code every time on their own machines. In case you were wondering, this isn't possible with a typical web server setup because XHR isn't allowed to be cross-domain by browser. Grunt can setup a proxy for you redirecting XHR requests on your own system within the grunt connect server to another system! (**grunt-contrib-proxy, grunt-contrib-connect**) .   **Responsiveness:**   * Modern, responsive and patented WEB UI foundation: HTML5/CSS3/full   JavaScript stack   * Mobile 1st approach, with multi-device support (tablet, phone, pc…) |

**Technologies** : Twiter BootStrap 3, Media quaries.

**Consuming Restful API’s :**

**Portlet**

Currently $http service is used for consuming REST API**Advantages of Restangular over $http**

* **It uses**[**promises**](http://docs.angularjs.org/api/ng.$q). Instead of doing the “magic” filling of objects like $resource, it uses promises.
* **You can use this in $routeProvider.resolve**. As Restangular returns promises, you can return any of the methods in the $routeProvider.resolve and you’ll get the real object injected into your controller if you want.
* **It supports all HTTP methods**.
* **It supports self linking elements** If you receive from the server some item that has a link to itself, you can use that to query the server instead of writing the URL manually.
* **You don’t have to create one $resource object per request**. Each time you want to do a request, you can just do it using the object that was returned by Restangular. You don’t need to create a new object for this.
* **You don’t have to write or remember ANY URL**. With $resource, you need to write the URL Template. In here, you don’t write any urls. You just write the name of the resource you want to fetch and that’s it.
* **It supports nested RESTful resources**. If you have Nested RESTful resources, Restangular can handle them for you. You don’t have to know the URL, the path, or anything to do all of the HTTP operations you want.
* **Restangular lets you create your own methods**. You can create your own methods to run the operation that you want. The sky is the limit.
* **Support for wrapped responses**. If your response for a list of element actually returns an object with some property inside which has the list, it’s very hard to use $resource. Restangular knows that and it makes it easy on you. **You can build your own URLs with Restangular objects easily**. Restangular lets you create a Restangular object for any url you want with a really nice builder.

**Restangular –** Restangular is a perfect option for complex operations on the client side. It lets you easily attach custom behaviors and interact with your data in much the same way as other model paradigms you’ve used in the past. It’s promise-based, clean, and feature-rich.

# **Debugging AngularJS App:**

[AngularJS Batarang](https://chrome.google.com/webstore/detail/angularjs-batarang/ighdmehidhipcmcojjgiloacoafjmpfk?hl=en) is a powerful Chrome extension that makes your developer tools "Angular aware".