Front End Workflow for Large Enterprise Web apps

Introducing

KeplerJS

Rubén D. Restrepo
Software Engineer
<*rrestrepo@metacode.com.co*>

November 25, 2016

INTRODUCTION

In front end development, there is a lot of great tools and frameworks available.
During last year, we had been working with some of them.

In the beginning we choose one of the hottest framework in 2015 (AngularJS) but right now we have a huge web app with more than 200 controllers (and counting) none of them have a single unit test (our fault).

We're worried about the front end code size which is right now in one single file as globals modules.

We would like to have a dynamic component loading strategy (only the views are loaded on demand when using the dialog component).

The software is working great, we support a local airline daily operations, the team is now mature in angular knowledge, everyone knows how the code is structured and we have a working continuous integration strategy.

But, we feel that we can be doing a better job, we found many tasks that are candidates for automation and we created some useful tasks using gulp , which is a great tool.

We use gulp for our development, and we have some extra tasks for our CI tool (Jenkins)

We built our software modular, using angular directives, but we wanted to use a more simpler approach: ***components***; They're available since angular 1.5 with a more simpler syntax.

You can read more about Angular components here:

<https://docs.angularjs.org/guide/component>

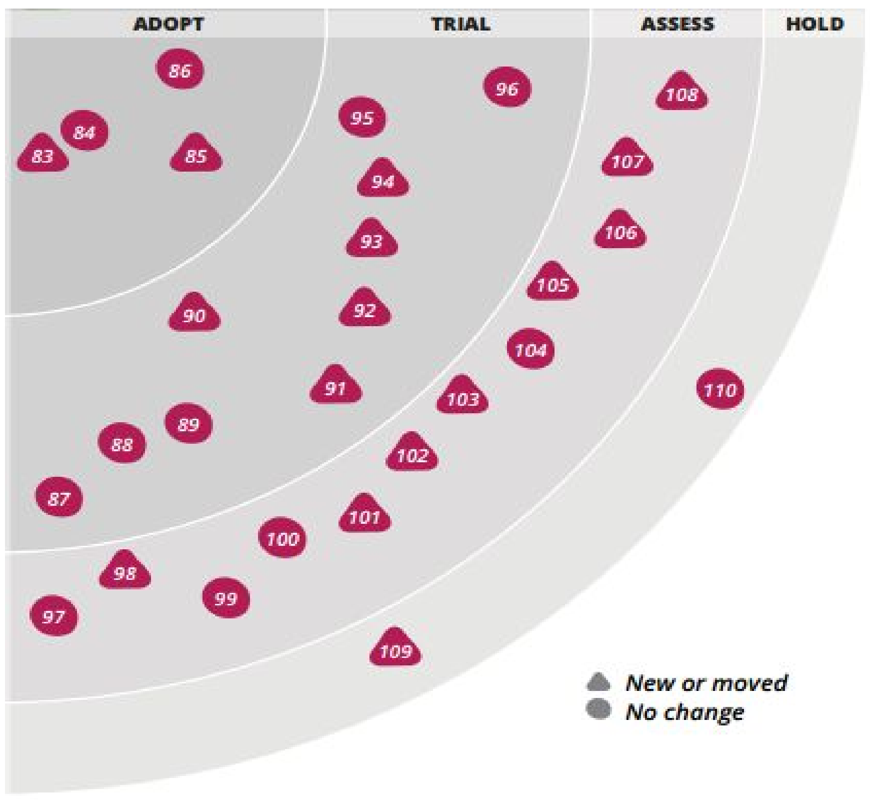
Advantages of Components:

- simpler configuration than plain directives
- promote sane defaults and best practices
- optimized for component-based architecture
- writing component directives will make it easier to upgrade to Angular 2

We don't use the component communication system in angular components (inputs and outputs), you will find our approach in the component communication section.

Despite the improvements, we feel that we're learning angular specific things, and for the newcomers the learning curve was very hard at the beginning.
We created many components that help us to improve this part and now everyone works the same way.

Angular 2
Sometime ago, we started to hear about angular 2, but we don't feel we're ready for that yet.
Checkout the **thoughtworks** technology radar and you'll find something interesting, in the language and tools, angular is on "hold" radar, that means, "proceed with caution".
To me, this is very important, **thoughtworks** have many smart people and I trust them.



80. Repsheet
81. Scikit-learn new

HOLD
82. Jenkins as a deployment pipeline

LANGUAGES & FRAN

ADOPT
83. Ember.js
84. React.js
85. Redux
86. Spring Boot

TRIAL
87. Butterknife
88. Dagger
89. Dapper
90. Elixir
91. Enzyme new
92. Immutable.js
93. Phoenix new
94. Quick and Nimble new
95. React Native
96. Robolectric

ASSESS
97. Aurelia
98. ECMAScript 2017 new
99. Elm
100. GraphQL
101. JuMP new
102. Physical Web new
103. Rapidoid new
104. Recharts
105. ReSwift new
106. Three.js new
107. Vue.js new
108. WebRTC new

HOLD
109. AngularJS
110. JSPatch

Image from <https://www.thoughtworks.com/radar/languages-and-frameworks>
All rights reserved to thoughtworks

“**AngularJS** helped revolutionize the world of singlepage JavaScript applications, and we have delivered many projects successfully with it over the years.
However, we are no longer recommending it (v1) for teams starting fresh projects. We prefer the ramp-up speed and more maintainable codebases we are seeing with Ember and React, particularly in conjunction with Redux.”

THE WORKFLOW JOURNEY

We would like to give a try to **EmberJS and React** in the near future, but right now we have to support a software we created one year ago in Angular 1.x.

To be honest, we don't have migration plans to Angular 2.

For this reason we created a ***Front end workflow*** that we call ***KeplerJS***, this isn't a new framework or library, this is only our approach following some key principles at the beginning:

Key principles

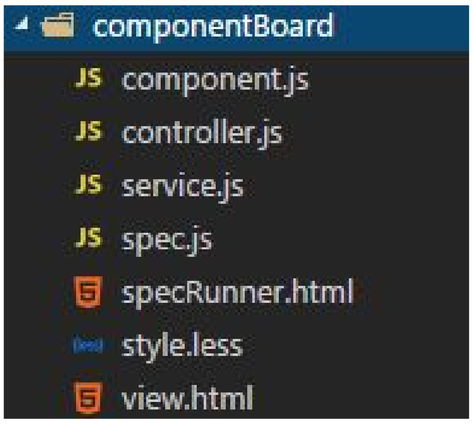
1) Component oriented architecture
2) Dynamic component loading
3) Dynamic injection
4) Component communication
5) Help companies using Angular 1.x to have a better and strong workflow
6) Automation tasks
7) Use ES2015 or future versions
8) TDD strong focus
9) **Future plans: Adapt this architecture to other popular javascript frameworks**

INTRODUCING KEPLER WORKFLOW

KeplerJS is the name of the front end workflow we're working on, we would like to share this project with the community and improve it every day.

1. COMPONENT ORIENTED ARCHITECTURE

Following our key principles, everything starts with a component, right now many of the logic is built using angular, but we think this can be adapted to other frameworks and libraries.
Each component should live in his own folder, just inside the components folder, we follow a convention over configuration philosophy, with this we can get the work done more quickly in the setup.



Folder structure of a component

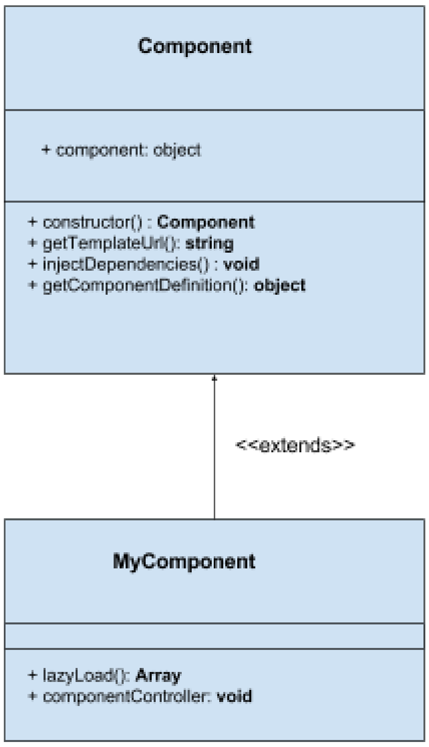
From the previous image we can find:

- A unique folder with the component name
- **component.js :** It contains the angular specific code for the controller, the component manifest and the dependencies.
- **controller.js:** Framework agnostic code, representing the controller logic, this is imported in the component.js file.
- **service.js:** a factory that represents the component service calls.
- **spec.js:** It contains the component controller unit tests.
- **specRunner.js**: HTML reporter for the unit test (using jasmine)
- **style.less:** CSS file for the component
- **view.html:** Represents the component view.

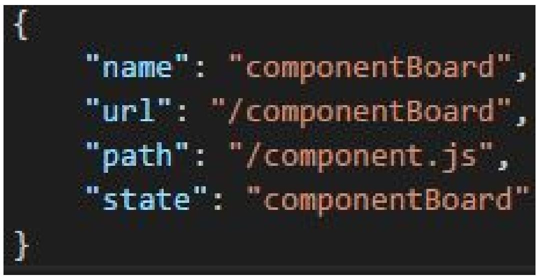
The style.less and service.js files are optional.

1.1 Component definition

Each component should extends Component class


Component
+ component: object
+ constructor() : Component
+ getTemplateUrl(): string
+ injectDependencies() : void
+ getComponentDefinition(): object
<<extends>>
MyComponent
+ lazyLoad(): Array
+ componentController: void

Each component is directly associated to a route, this route can be nested or isolated.
A component can be rendered directly inside another component view.
By default, each component route is registered in routes.json, you can get this routes from a database too.
Each route is an object with the following properties: name, url, path, state



```
{
    "name": "componentBoard",
    "url": "/componentBoard",
    "path": "/component.js",
    "state": "componentBoard"
}
```

If you need a component to render inside another component view from the routes, you can add it to the state property, this is achieved through the state name of the "parent" component followed by the component own state,
example:

state for component 1: state1
state for component 2: state1.state2

To load component 2 you navigate to /state1/state2

Now you will see both components on the page.

2. DYNAMIC COMPONENT LOADING

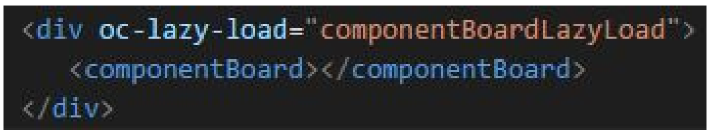
You can load components in two ways:

- Component tag
- URL matching

Component tag

You can load any component inside another component view using ***oclazyload***:

The component task will create for you the code inside the component.js file if you want to use this mechanism:



```
<div oc-lazy-load="componentBoardLazyLoad">
   <componentBoard></componentBoard>
</div>
```

*In this example **componentBoard** is the component name you want to load.*
< Read more at: https://oclazyload.readme.io/>

Url matching

You can load any component navigating to the component route, kepler will resolve all the dependencies and register the component in your angular app.

We're using ui-router project here.
<https://ui-router.github.io/ng1/>

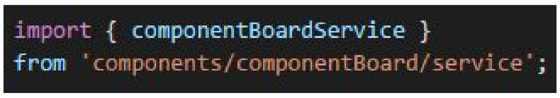
3. DYNAMIC INJECTION

You can inject any dependency to any components dynamically, this dependency should extend the Factory class in order to be injected inside the component controller.

You should create a new instance of this factory inside the ***injectDependencies*** method created in the **component.js** file.

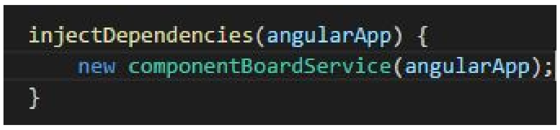
Service injection example:

1) Import the service:



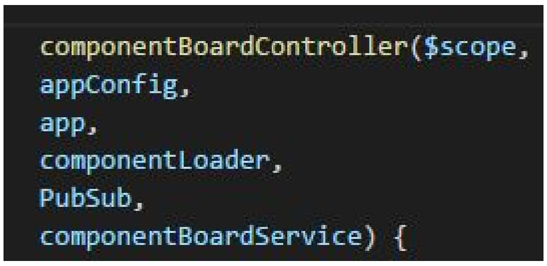
```
import { componentBoardService }
from 'components/componentBoard/service';
```

2) Create service instance inside injectDependecies method



```
injectDependencies(angularApp) {
    new componentBoardService(angularApp);
}
```

3) Inside your component controller inject the service:



```
componentBoardController($scope,
appConfig,
app,
componentLoader,
PubSub,
componentBoardService) {
```

You can use the factory now inside your controller.

4. COMPONENT COMMUNICATION

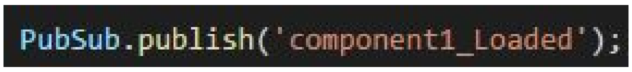
We choose the PubSub pattern for communication between components, it's more easy and less angular specific.

"publish–subscribe is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead characterize published messages into classes without knowledge of which subscribers, if any, there may be. Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are."

Source:
<https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern>

We use PubSub library from georapbox
<https://github.com/georapbox/angular-PubSub>

Each component controller injects the PubSub module, and by default publish the Loaded event.



```
PubSub.publish('component1_Loaded');
```

Other components can "listen" that event using the subscribe method.

You can send and receive data in the publish and subscribe methods.

5. Better and Strong workflow for companies using angular 1.x

You should read this great article <https://daveceddia.com/angular-2-should-you-upgrade/>

Most companies can't just upgrade to Angular 2, it's like a new learning curve, the docs are mainly in typescript, and you have to re-learn new things, I love learning new things, but that doesn't mean you have to upgrade always to the newest version (if that means you have to learn again and you know that your current code will not work), we would like to try another libraries and frameworks, but the business should continue improving our current software, we have to develop and maintain current modules, that's why we wanted to create this workflow and do the best we can with Angular 1.x

We think other companies using Angular 1.x may find this workflow helpful and why not, improve it!

For now (2016 year), don't use Angular for fresh projects yet. Let's wait what happens with Angular 2 future, give a try to other libraries like React + Flux or EmberJS <https://www.thoughtworks.com/radar/languages-and-frameworks/react-js>

We hope to have Kepler adapted to other libraries like React and Ember, it's a matter of time. (Even better, a day we don't need something like this and dealing with just another framework hell)

6. AUTOMATION TASKS

We want developers creating components faster, we know there's a great tool called yeoman, with angular generators, but since, we have a more complete workflow in order to get things done, we have a simple gulp task that allow us to create components quickly:

```
>> gulp component
--name=componentName --style --service
```

This will create a new component with a custom service and style in less
--style and --service arguments are optional.

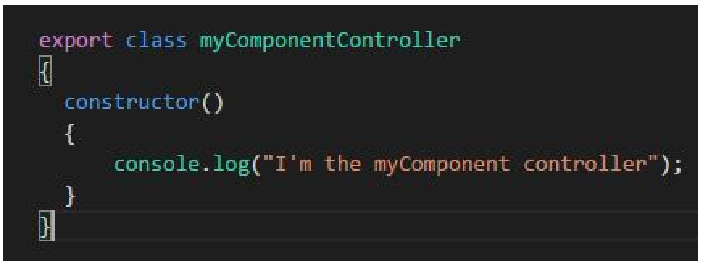
By default the component route will be the component name, you can change it using --state argument.

7. USE ES2015 OR LATER

We always wanted to create javascript code in ES2015, kepler will allow you to this and use any module loader syntax like AMD, CommonJS.

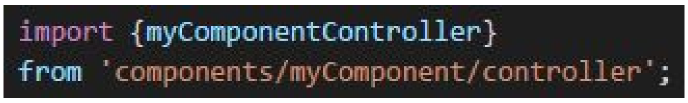
You can use class, extends, arrow function, string interpolation powered by babelJS <https://babeljs.io/>

A kepler controller looks like:



```
export class myComponentController
{
  constructor()
  {
      console.log("I'm the myComponent controller");
  }
}
```

You can load the component and css using import or require syntax:



```
import {myComponentController}
from 'components/myComponent/controller';
```

Load CSS

```
import 'components/myComponent/style.css
```

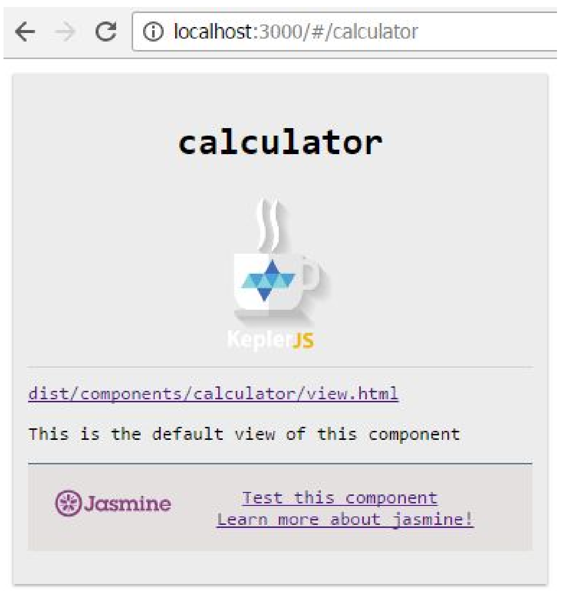
This is great, now you can load specific css for your module too.

8. TDD STRONG FOCUS

We had seen TDD benefits on our service layer using c#.

We wanted to have the same on javascript.
For now, we support Jasmine, a Behaviour Driven Javascript library
<https://jasmine.github.io/>

Each component controller have a default spec file with two test functions.
Once your component is created you can navigate to the components view (Kepler will create a default view for the component like this one)

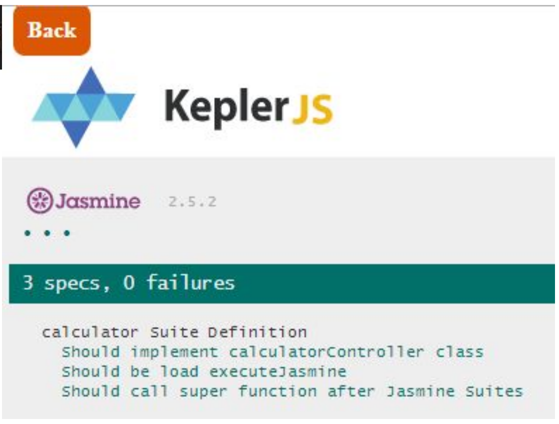


calculator

dist/components/calculator/view.html

This is the default view of this component

You can execute the tests navigating to specs/[ComponentName]/specRunner.html

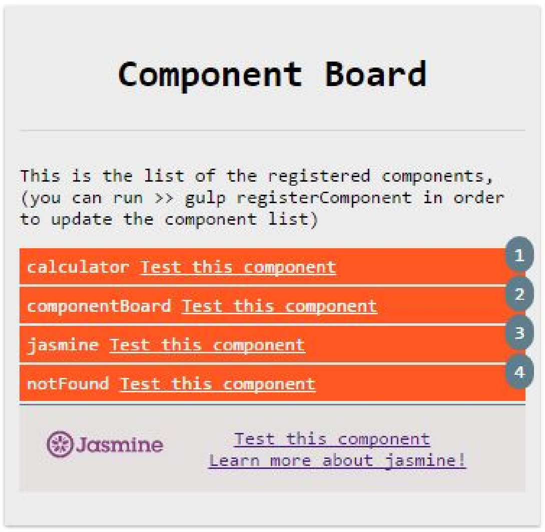


```
3 specs, 0 failures

calculator Suite Definition
  Should implement calculatorController class
  Should be load executeJasmine
  Should call super function after Jasmine Suites
```

You will see the test results from the jasmine report.

Indeed, we have a component called **ComponentBoard**, this list all the kepler components you have on your project:



Component Board

This is the list of the registered components,
(you can run >> gulp registerComponent in order
to update the component list)

```
calculator Test this component        1
componentBoard Test this component    2
jasmine Test this component           3
notFound Test this component          4
```

9. FUTURE PLANS

I think this workflow shouldn't be Angular specific, we can adapt this approach to React and other libraries.

We invite you to contribute on github:

<https://keplerjsworkflow.github.io/>

I hope to talk further about the Kepler source code soon, I would like to improve some things in the near future.

I was inspired to open source this "workflow" by this blog post from Christian Heilmann and Rita Zhang, thank you!

https://www.sitepoint.com/open-sourcing-javascript-code/