The AngularJS Enterprise

Due to the varsity that Angular framework possesses, it becomes extremely difficult for a Designer to use Angular at its optimum level. Sometimes the architecture got miss-leaded due to lack of adherence to Enterprise Patterns an Application would preserve.

To understand the things further, lets callout the features of a high scalable enterprise architecture could use:

1. ***Atomicity*** : even a smallest piece of arch is testable with data , load & can scale horizontally
2. ***Decoupling***: the pieces constituted the Whole ecosystem, can be coupled on Demand
3. ***Inheritable***: what ever written to build the arch, can be reused without copying it
4. ***Polymorphic***: the app behaves differently based on the context
5. ***Multi-tenant***: should have a logical & physical isolation within a captive business-case

To create a core / platform for rapidly develop business applications / features; many challenges to be faced. The primary would be of how to create an extendable code-base; that can be shared all across the applications as PARTs or as a WHOLE.

Preferably any front-end oriented framework (FOF), should be hosted as a stack of files either in a Webserver or CDN. Here comes the important question, is it good to use CDN to host the web-application?

Of course, YES.. a CDN is nothing but a web-server and meant to host the assets of the Portal. FOF like a plain-vanilla HTML file stack; that driven by couple of JS files. So the expenses of hosting an enterprise class FOF is no more a nightmare; rather a simple budgetary affair of selecting an economically viable Provider & the job is done. Let's freeze that a mere www folder structure is fine to start with and scale as you grow deeper to that track.

We will now focus some of the Enterprise feature that AngularJS possess to make an optimistic application:

1. ***Sign of inheritance***: In angular world this is achieved by the injection at Module, Controller, State and View level.

   Module inheritance:

   ```
   angular.module("edpApp.UserProfile", ['edpApp']);
   ```

   When we write this, actually we are inheriting all the features of 'edpApp' to edpApp.UserProfile. This means all the definitions at edpApp are also visible to the derived code. You have the option to define things over again, considering few like State definition can't be overridden.

Controller inheritance:

```
app.controller("DefaultController", function($scope,$log) {   });
```

Once you derive a Module as above, all the parent controllers are visible and say DefaultController as you could be defined in the edpApp can be used as-is or get overridden as above freshly.

State inheritance:

```
$stateProvider
                .state('home', {
                    url: '',
                    cache: false,
                    views: {
                        'mainHeader@': {
                            templateUrl: coreTplPrefix + '/Core/partials/header.htm',
                            controller: 'HeaderController'
                        },
                        'mainFooter@': {
                            templateUrl: coreTplPrefix + '/Core/partials/footer.htm',
                            controller: 'FooterController'
                        }
                    }
                });
```

Define a State is simple; typically say ‘home’ state is defined in edpApp module. When we do edpApp.UserProfile and subsequently inherit ‘home’ state; we have option to extend it too as below:

```
$stateProvider
                .state('home.dashboard', {
                    url: '/dashboard',
                    cache: false,
                    views: {
                        'container@': {
                            templateUrl:  './partials/contentLayout.htm',
                            controller: 'DefaultController'
                        }
                    }
                });
```

Here ‘home.dashboard’ is derived from its parent and a direct instantiation of dashboard state will also ensure that ‘home’ is getting instantiated.

The way an enterprise class application due process login-authentication, authorization (ACL) loading, user profile loading; an equivalent state load can be grafted as below:

```
auth -> auth.profile -> auth.profile.home
```

Where auth , auth.profile could be abstract, i.e. when you invoke state auth.profile.home it ensures that preprocessing is performed implicitly..

This opens a technique to code that reduces lots of headache, to explicitly define & write logics for validating access on each State binding.

Common stack:

When you decide to write a framework for your application; the first thing comes to the mind is how we keep core processing intact as well as pluggable & mandate to the derived structure.

As we have seen during Module inheritance, anything defined at core module level gets pushed to the child by default. So keep defining common things at base is always profitable..

```
// common exception handlers
$provide.factory('$exceptionHandler', function () {
//common log mechanism
$provide.decorator('$log', ['$delegate', function ($delegate) {
// global constants
app.constant('cfg'
```

Above few ones can defined at base level, so that rest of the implantation can adhere to it.

Now comes the most interesting part of presenting data to the browser; the View.. do we require to inherit a View ? of course Yes, when you have a big Enterprise application, and your company has a web-standards to follow ; you need to take help of the inheritance..

Practically the Page Header , Footer of an enterprise application maintains a standard layout; which to be followed across the application; what changes is the data, that gets fed to the common layout.. now I'm going to DEMO that too well here:

View inheritance

```
$stateProvider
                .state('home', {
                    url: '',
                    cache: false,
                    views: {
```

```
                    'mainHeader@': {
                        templateUrl: coreTplPrefix + '/Core/partials/header.htm',
                        controller: 'HeaderController'
                    },
                    'mainFooter@': {
                        templateUrl: coreTplPrefix + '/Core/partials/footer.htm',
                        controller: 'FooterController'
                    }
                }
            });
```

Follow the above example; the views for the header & footer section is defined in the Core module; leaving an option to the respective controllers of how to deal with.

The templates used are from core path, so that a new application can leverage it..

Now in a derive module edpApp.UserProfile, we defined the content:

```
$stateProvider
            .state('home.dashboard', {
                url: '/dashboard',
                cache: false,
                views: {
                    'container@': {
                        templateUrl:  './partials/contentLayout.htm',
                        controller: 'DefaultController'
                    }
                }
            });
```

As you grow to multiple module, you decide what to be rendered at the content area & keep implementing this.. You push a relative template instead from a core location.

These are those most advanced & challenging features that opens a complete new era of Enterprising the Single Page application and correctly framed it to work!!!

I have an working example to be posted soon…