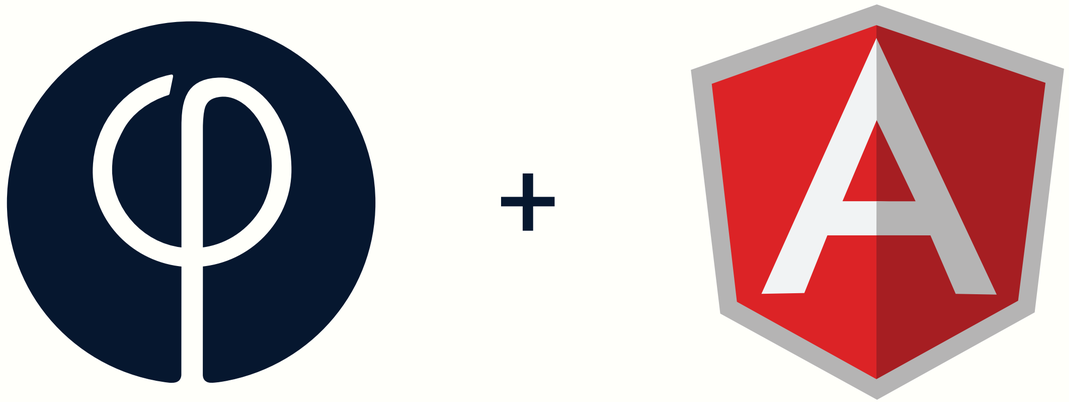
Famo.us/Angular

What is it?



It’s an MVC framework for creating Famo.us apps.

It’s powered by AngularJS and it integrates seamlessly
with both Famo.us and Angular apps.

It’s not a replacement for Famo.us, and it’s not a replacement for AngularJS:
It’s a way to bring MVC structure to Famo.us apps.

Famo.us Scene Graph

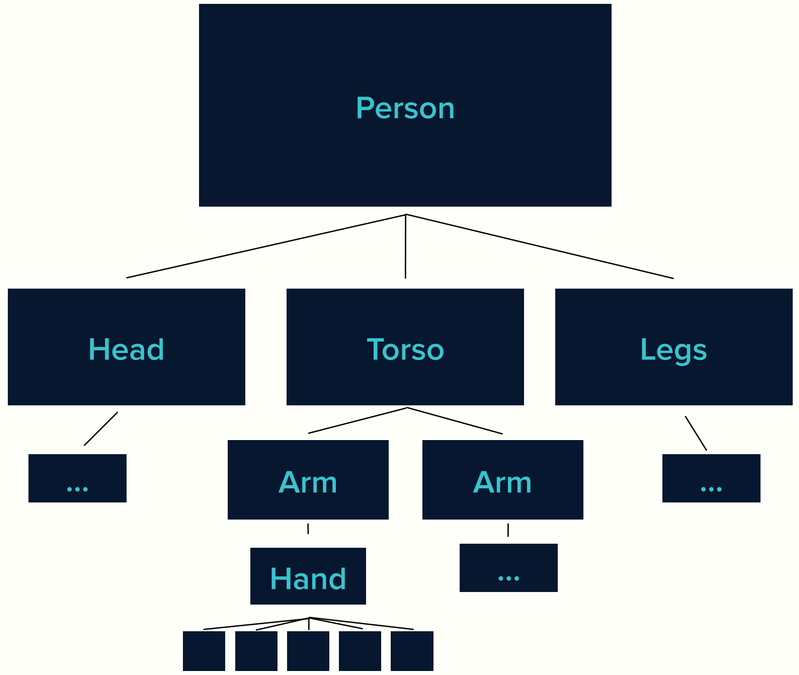
What is a scene graph?

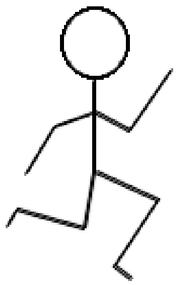
In short, it's a tree.

It's the data structure that Famo.us uses to manage nested UI components. It's often used in 3D graphics and game development.

It's an elegant way to manage hierarchical transformations: child nodes can be transformed independently, but transformations on parent nodes affect all descendants.

(rotate the whole **body** and everything else moves with it.
rotate a **finger** and it moves on its own.)





desktoppub.about.com

Famo.us Scene Graph

How do you create a scene graph in Famo.us?

You create components, then you add children to parents.

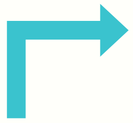
The most primitive element of the Famo.us scene graph is the RenderNode. Most core components subclass or use RenderNodes internally.

When you add a Surface to a View, you're adding one RenderNode to another, and you're creating a tree.

This is an **imperative** style of authoring a UI.
(you are **commanding** the program to put pieces together.)

```
5
6  var mainContext = Engine.createContext();
7
8  var stateModifier = new StateModifier({
9    transform: Transform.translate(150, 100, 0)
10 });
11
12 var surface = new Surface({
13   size: [100, 100],
14   properties: {
15     backgroundColor: '#FA5C4F'
16   }
17 });
18
19 mainContext.add(stateModifier).add(surface);
```

famo.us/university



Assembling a tree by adding children to parents

DOM

What is the DOM?

In short, it's a tree.

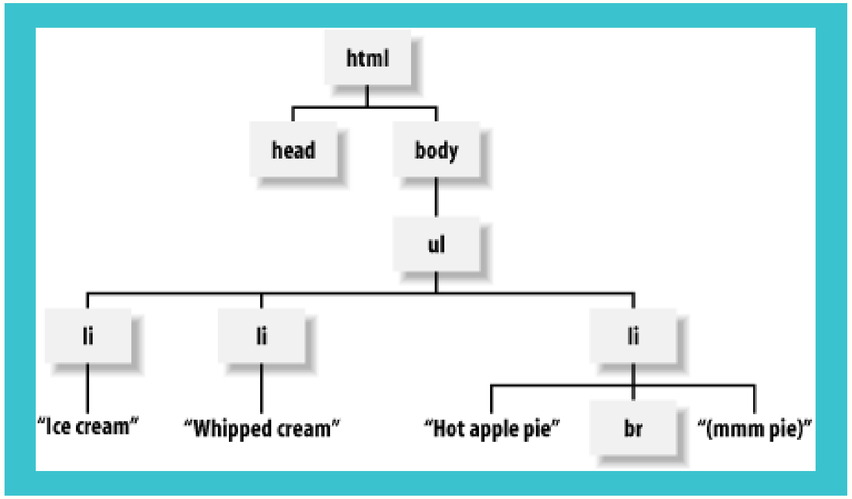
It's the data structure that browsers natively use to manage nested UI components.

It's not as elegant at positioning and styling things (CSS...)

But it's an elegant way to manage hierarchical **content**.

HTML (the language of the DOM) is a **declarative** style of authoring a UI.

(you are **describing** the content of the UI.)



http://lwp.interglacial.com/ch09_01.htm

DOM

But with Famo.us, isn't the DOM obsolete?

No.

Famo.us doesn't yet have its own way of separating **content** from **how it's presented**.

In larger scale apps (high complexity or large teams,) this architectural separation of concerns is a must.

Wouldn't it be great if there were a way to use the DOM to declare content, while letting Famo.us maintain full control of rendering?

(**drum roll...**)

ufunk.net

```
mySurface.setContent(
  "<div style='" + myStyle + "'>" +
  "  <p class='" + myP1Class + "'>" + myP1Text + "</p>" +
  "  <p class='" + myP2Class + "'>" + myP2Text + "</p>" +
  "  <div class='" + myDiv1Class + "'>" + myDiv1Text + "</div>" +
  "</div>"
)
```

Not okay.

(**and this is a pretty example**)

AngularJS

Enter: AngularJS

AngularJS is a full-powered MVC, letting you build apps with structure, modularity, and handy features like two-way data-binding.

It allows you to attach arbitrary compilation behavior to standard DOM nodes.

This compilation behavior (as well as the intrinsic hierarchy in the DOM) is what Famo.us/Angular uses to **compile** the DOM into a Famo.us scene graph.

(**wut?**)

Famo.us/ Angular

The Famo.us/Angular Jump: Compiling the DOM

Famo.us/Angular lets you use the DOM to describe the content of your app.

It crawls through that DOM and passes along the tree that it finds to Famo.us, to render as its scene graph. Then the original Angular DOM gets `display: none`'d (hidden from screen)

Angular's two-way data-binding and custom directives (or any normal HTML) remain intact.

Performance is pure Famo.us.

```
<fa-app ng-controller="MyCtrl">
  <fa-modifier fa-transform="myTransformFn">
    <fa-surface>
      <div>{{dataBoundVar}} + Normal HTML!</div>
    </fa-surface>
  </fa-modifier>
</fa-app>
```

Famo.us/Angular

Vanilla Famo.us

```
var Engine = require('famous/core/Engine');
var Modifier = require('famous/core/Modifier');
var Surface = require('famous/core/Surface');

var myContext = Engine.createContext();

var myModifier = new Modifier();
myModifier.transformFrom(myTransformFn);

var mySurface = new Surface();
mySurface.setContent('<div>[???] + Normal HTML!</div>')

myContext.add(myModifier).add(mySurface);
```

Famo.us/ Angular

Declare, data-bind, and mutate

Famo.us/Angular lets you use the Angular style of declaring a UI, binding values to controller variables, and then changing those values from the controller.

This keeps **declarative** and **imperative** concerns cleanly separated. (**MVC, yo**)

For example: **declare** a fa-modifier. **Bind** its opacity value to a variable on your controller. Now, **imperatively** update the variable in your controller. The UI updates accordingly.

Declare

```
<fa-modifier fa-opacity="myOpacity">
```

Data-bind

Mutate

```
//myOpacity is a Transitionable
$scope.myOpacity.set(0, {
  duration: 1000, curve: 'linear'
})
```

Clean separation of concerns.

Famo.us/ Angular

So is Famo.us/Angular “better” than vanilla Famo.us?

Wrong question.

Famo.us/Angular ***is*** Famo.us; it’s just an optional design pattern. And vanilla Famo.us and F/A are fully compatible: you can build a single app using both.

F/A brings some powerful features to Famo.us (e.g. data-binding, modularity) and solves some problems that you’ll run into with organizing large apps, but every technical decision comes with trade-offs.

elsagedesigns.blogspot.com

Famo.us/ Angular

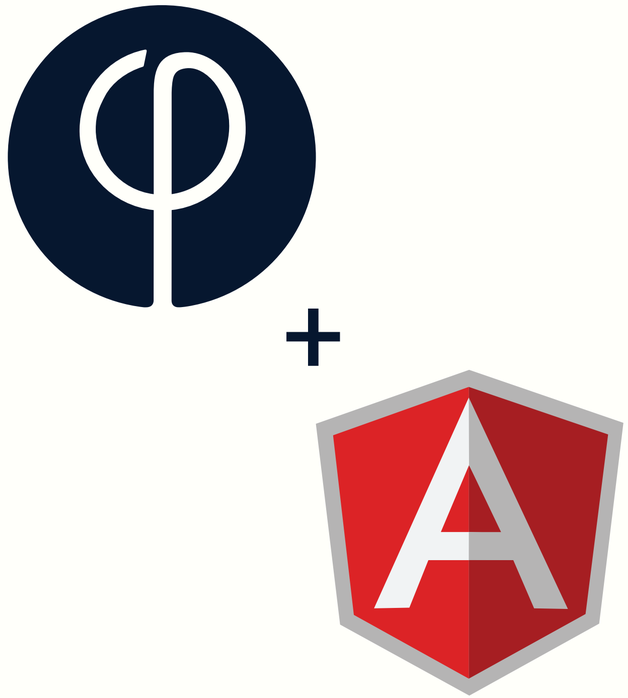
Handy-dandy comparison chart

	Pros	Cons
Vanilla Famo.us	• Doesn't require knowledge of additional frameworks • Will always be on the 'cutting edge' of Famo.us	• Lacks structure for larger apps • Requires imperative UI creation • Requires mixing markup with JS • Not easily compatible with existing codebases
Famo.us/ Angular	• Promotes clean, scalable, maintainable architecture • Allows for easy visualization of your UI (declarative, HTML!) + easier conceptualization of Famo.us • Allows use of Angular features like two-way data binding, services, and routing • Easily compatible with existing Angular apps, plus third-party libs • Is fully compatible with vanilla Famo.us components/apps, so there are no dead ends	• Requires an additional dependency • Requires AngularJS familiarity • Can be additional overhead(boilerplate) for simple apps • Has some lead time before new Famo.us features or API changes are fully supported (separate development effort on the library)

Resources

Jump right in!

If you like Angular and you're interested in Famo.us (or the other way around,) you're going to like F/A.



It brings the best of both worlds to the table.

There are handy resources for getting started, including:

famous-angular-starter: github.com/thomasstreet/famous-angular-starter
famous-angular-examples: github.com/thomasstreet/famous-angular-examples
Docs: http://famo.us/integrations/angular/docs/api/
Project Page: http://famo.us/angular

Ngus (the Famo.us/Angular cow) says:
“Ngthank you!”