Angular 2.0 Router Design Document

*Status: Draft*

*Authors: bford@google.com, rob@bluespire.com, nate.abele@gmail.com, and timkindberg@gmail.com*

# Objective

There are many known limitations with ngRoute in AngularJS 1.x. The purpose of this doc is to summarize the issues with the current router, and plan the features, requirements, and overarching design for the router in Angular 2.0.

**Note:** Although the target for this is Angular 2.0, it might be possible to backport this new router to Angular 1.x.

# Background

The initial Angular router was designed to handle jst a few simple cases. As Angular grew, we've slowly added more features. whoever, the underlying design is ill-suited to be extended much further.

It's very easy for developers to write apps that ignore URLs and break the back button with Angular. Angular should make it easier for developers to create apps with deeplinking.

# Prior Art

In identifying who to improve the routing in Angular 2.0, the following routing solutions were investigated from both within the angular ecosystem and outside of it:

## ngRoute 1.x

Strengths:

* Simple API
* Easy to reason about who routes resolve
* Emits events

Weaknesses:

* Doesn't support nested nor sibling views
* "resolve" option in route config feels misplaced (this should be solved by di.js in 2.0)
* Uses hrefs to navigate
* [Can't instantiate a controller without a template (?)](https://github.com/angular/angular.js/issues/1838)

## [uiRouter](https://github.com/angular-ui/ui-router)

uiRouter is an alternative router for AngularJS v1.0. Instead of configuring a list of routes, you configure states. It's popular in the open source world, and solves a lot of the same problems we're aiming to solve in Angular 2.0's router.

The configuration API looks like this

myApp.config(function($stateProvider, $urlRouterProvider) {  
 // For any unmatched url, redirect to /state1  
   
 // Now set up the states  
 $stateProvider

States have a name and a URL. State names use [dot-delimited paths](https://github.com/angular-ui/ui-router/wiki/Nested-States-%26-Nested-Views#wiki-methods-for-nesting-states) to denote nesting ([or object based states](https://github.com/angular-ui/ui-router/wiki/Nested-States-%26-Nested-Views#wiki-object-based-states)).

Strengths:

* [Supports multiple views](https://github.com/angular-ui/ui-router/wiki/Multiple-Named-Views)
* Supports [nested states and views](https://github.com/angular-ui/ui-router/wiki/Nested-States-%26-Nested-Views), resolves and custom data that [waterfall down](https://github.com/angular-ui/ui-router/wiki/Nested-States-%26-Nested-Views#wiki-inherited-resolved-dependencies) the state hierarchy. $state.go() allows [relative](https://github.com/angular-ui/ui-router/wiki/Quick-Reference#wiki-examples-diagram) state transitioning, e.g. $state.go(‘^’) to go up one state.
* Supports url-less states
* [ui-sref](https://github.com/angular-ui/ui-router/wiki/Quick-Reference#wiki-ui-sref) replaces ng-href and uses state names instead of urls. Urls are compiled.
* [ui-sref-active](https://github.com/angular-ui/ui-router/wiki/Quick-Reference#wiki-ui-sref-active) adds class when on active route
* Attempts to offer a compatibility mode for ngRoute [(but dropping soon to reduce file size)](https://github.com/angular-ui/ui-router/issues/838)
* Emits change events
* [Abstract State](https://github.com/angular-ui/ui-router/wiki/Nested-States-%26-Nested-Views#wiki-abstract-states)s
* ui.router is divided into more single responsibility modules ($states (the largest), $urlRouter (for route matching), $view (for populating views)). In a perfect world we’d like to be able to use states with no urls at all, or views with no states at all, or urlMatching with no views… etc.
* Supports lazy state definition via $stateNotFound event
* Supports custom data in state config, for dev personal usage
* onEnter and onExit callbacks in state config, allows interesting things like having no url, template or controller, but just doing some logic or triggering an action from onEnter when the state is activated.
* Can register states across modules, even child states (it waits until the parent exists).
* Any state (even children) can have absolute URLs. URLs can be completely independent of state hierarchy.
* Accompanying filters: [isState](http://angular-ui.github.io/ui-router/site/#/api/ui.router.state.filter:isState), [includedByState](http://angular-ui.github.io/ui-router/site/#/api/ui.router.state.filter:includedByState).
* Parameter inheritance, when transitioning from one state to another if the states share params.
* Allows for default query parameters

Weaknesses:

* Conceptual overhead re: understanding your UI as a state machine
* Poor transition management means event handlers become case-specific dumping grounds
* View target has been a bit confusing:
  + Many don’t realize that by default templates populate into the unnamed parent view. Some developers think it always populates the top level view, like ngRoute. Docs solved this for the most part. Overall though I’m not sure I’d do it differently.
  + Explicit targeting of views is a bit complicated, e.g. “header@contact.detail” would be the header view in the contact.detail state’s template.
* $state contains a lot of internal… state, thus it is difficult to effectively extend via decorators
* Abstract states are conceptually confusing (both a pro and con :))
* Much micro-syntax

Wishlist/Roadmap Items:

* ~~We want parameter encoding/decoding from and to objects. So there’s be a service API to serialize the parameters to actual usable objects instead of just strings.~~ (Implemented in master now -NA)
* “Components”. Re-usable state tree branches that can be attached anywhere easily. Great for modals in various places, edit modes of pages, etc.
* Orthogonal view routing. Two views who have sub-trees of their own, whose states are automatically serialized to the url.
* Special $transition$ service that holds all info related to the currently active transition. Can inject into resolves and event handlers.

Typical User Stories:

* I want to trigger a modal on a state instead of populating a view
* I want to swho a modal for a state if its triggered internally but if coming to the state straight from a permalink I want to swho a page
* I want to authorize my users and limit state activation
* I want to be able to go back in state history easily
* I want to have default child states
* I want to change the url without reactivating the state
* I want to be able to go back to a state without reloading anything, retaining the content from when it was activated previously
* I want to be able to generate a menu automatically based on the routing information.
* I want to be able to swho/hide menu items based on the links/states that I am allowed to go to from the current state.

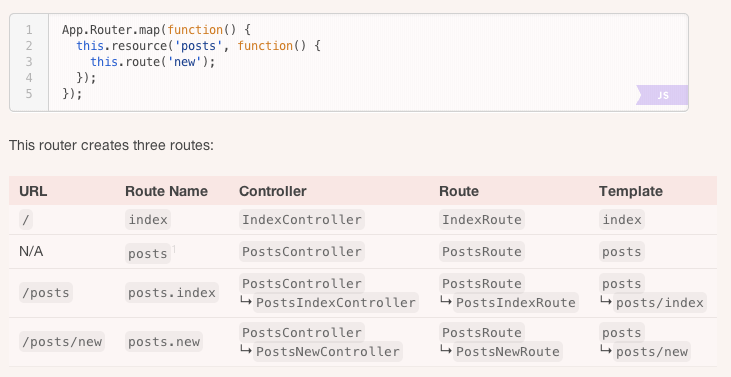
References:

* Presentation: <http://slid.es/timkindberg/ui-router#/>

## 

## [Ember.js](http://emberjs.com/guides/routing/)'s Router

Strengths:

* Very little boilerplate
* URL is a first-class citizen in the app
* Resources create CRUD routes:
* Routes automatically fetch relevant models/resources from the server.
* Uses this microlib for routing: <https://github.com/tildeio/router.js>
* And this one for parsing URLs: <https://github.com/tildeio/route-recognizer>

**Note:** Angular's routing should remain as flexible as possible, but it should be possible to build primitives that allow developers to write a mre opinionated API like Ember's on top.

Weaknesses:

* You can only nest resources. You can add other routes, but not inside of resources unless the nested route uses a resource.
* Route names are tied to Controller names (as far as I can tell). I think this is a good default, but it should not be required.
* APIs rely heavily on function context (the value of `this`)

## [Durandal](http://durandaljs.com/documentation/Using-The-Router.html)'s Router

Durandal is a JavaScript framework built on top of jQuery, Knockout and RequireJS. John Papa suggested that we take a look at it.

Interesting features:

* **Conventional Routing:** The router has a hook called mapUnknownRoutes which allows a developer to plug in custom code to govern the router's behavior when no route match is found. It can be used simply to swho a "not-found" view or a callback function can be supplied to allow the developer the capability to completely conventionally map route info to modules. You can also turn on a default convention which maps route paths to module paths (not really recommended of course.) See <http://durandaljs.com/documentation/Using-The-Router.html#handling-unknown-routes>
* **Route Guards:** At the router level there is a hook called guardRoute. Before any module is resolved to handle a route, this hook is invoked allowing a router-level security mechanism. This is often used to implement security. For example, the developer can store a list of roles as part of his route config object, then use that to allow/deny navigation at the router level before a controller is ever resolved. The guard callback can return a boolean or a redirect string. It can also return a promise for either.
* **Multi-Route:** As part of a route configuration, you can specify an array of route patterns that all map to the same module. It's mostly a convenience. ex. { route: ['', 'home'], moduleId: 'home' }
* **Navigation Model:** Each router can transform it's route config into a "navigation model" It's an array of route objects, ordered by config, augmented with proper hash info, etc. where each item knows if it is the active route or not (isActive flag). The idea being that you can easily databind the array to create a nav UI and easily style the active route. *Ex: building a menu at the top of your app.*
* **Child Routers:** Uses multiple router instances for nesting – Each router allows for complete encapsulation of a set of relative routes. The child router can generate it's own navigation model (suitable for use in a UI) and has an associated element (synonymous to ngView) where the child router's current view will be rendered.

Strengths:

* [API for reusing components when switching between routes](http://durandaljs.com/documentation/Using-The-Router.html#module-reuse); this is something ngRoute doesn't do – it instead reinstantiates controllers between states
* **Activators:** Each router maintains an internal "activator" which manages a simple state machine around the "active route". This allows each module to implement a series of callbacks which control flow into and out of states. The callbacks are:canActivate, activate, canDeactivate and deactivate. The can\* callbacks can return a boolean or promise of boolean. This allows the active module to cancel navigation away or to prevent navigation to itself. The can\* callbacks receive all the route parameter info to aid in decision making. The activate and deactivate callbacks can also return a promise to tell the router to "wait" until work is done before continuing with the navigation.
* **Composition:** The router "element" (synonymous to ngView) is just sugar on top of Durandal's view composition system. In essence, the router element simply databinds itself to the "activeRouter" property of the router. When the router changes its state, the element re-composes its view. In fact, in Durandal, you can accomplish this without any special router binding just by using and configuring the "compose" binding.
* **Declarative Config:** Routes are configured declaratively by passing an array of route objects to the router. Each route object is very simple. For example { route: 'home', moduleId: 'home/index' }. Under the covers the router uses the module system directly to obtain an instance of the module whenever the router pattern is matched. No "template" needs to be specified since Durandal can infer that from the module id. By default a module of "home/index" resolves to a module "home/index.js" and a corresponding view of "home/index.html" (In fact the underlying composition system does this which is what allows Durandal to locate a view template for "any" module...)

Weaknesses:

* The combination of child routers and activators is a bit tricky.

## [Passport](http://passportjs.org/) (middleware for Express)

Server-side middleware for express that attempts to normalize 3rd party authentication services:

app.post('/login', passport.authenticate('local', {  
 successRedirect: '/',  
 failureRedirect: '/login'  
}));

Passport provides a consistent API for accessing the data returned from an OAuth provider. There's also a plugin system for different authentication "strategies," like Twitter, G+, Facebook, email, etc.

It'd be nice to have a similar service for Angular that handles auth handshakes.

Use Cases / Features / Requirements

These cases especially highlight failings of the existing ngRoute.

## [Multiple ngViews](#heading=h.67t2io55nj34)

Right now, an Angular app can only contain a single ngView.

When views are nested, the outer view captures some segment of the input, passing the rest to the inner ng-view.

## State-based Routing

The existing routing system uses a list of routes, starting from the first one provided, and continuing until it finds a match.

Being able to control the transitions between different states. For instance, being able to display a modal prompting a user to save before navigating from route with a form that's partially filled in.

In ngRoute, the only way to do this is by listening to route change events on $scope. You can put this logic in either a run block or a controller; it's not obvious where this responsibility should live.



## Idioms for Authentication and Authorization

This isn't so much a feature ofthe router itself as it is a feature of the documentation. Developers have lots of questions about the "right" way to do this. We should have a few examples of who to do authentication and authorization. A service that abstracts oauth providers might also be useful.

## A way to preserve state of certain views

ngRoute 1.0 always destroys/recreates controllers/views when navigating between routes

See: <https://groups.google.com/forum/#!topic/angular/iWUUtTz-onE>

The discussion explicitly mentions a flag for each config, which might work. A hierarchical LRU cache for routes might also make sense.

# Detailed Design

Below is a diagram swhoing the different parts of the new routing system, followed by an explanation of the responsibilities of each part.Arrows indicate a dependency.



## URL Resolver

In 1.x, this functionality lives in `src/ng/urlUtils.js`. [This can be its own microlib/service](https://github.com/btford/url-resolver.js).

## Location

Service to wrap browser APIs for consistency. API will be approximately the same as the $location service in Angular 1.0. This service should be packaged independently from the router.

## URL Matcher

Takes some token: `/:foo/:bar` and a URL and returns the parsed segments out of the URL. One of the challenges here is dealing with wildcards: `/foo\*/bar\*`; it's not obvious who much of the URL each part should match, and whether the wildcard does a greedy or non-greedy match. You can use regular expressions

## Route Resolver

The route resolver decides what route is active, and when and who to transition between routes. It responds to changes in the URL as well as to events sent from ngLink directives.

## Route Configuration

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |







* Don’t swho a user links they’re not authorized to visit
* Swho experimental features to certain users

## Summary

Good API boundaries mean that developers can replace or decorate these services to support more exotic routing configurations and use cases.

ngRoute 2.x makes no attempt at backwards compatibility like uiRoute does for ngRoute 1.x.

# Security Considerations

Location should be treated as untrusted user input. I don't think there's much else to say.

# Performance Considerations / Test Strategy

* Routing is not performance-critical for most applications. Testing should be relatively straightforward since the services that interface with browser location APIs will be separate from the services that resolve routes.

# Work Breakdown

I expect designing and revising the API to be the most significant part of the work. With the initial design work done, implementation should be relatively straightforward.

We need to Implement:

* URL resolver – this is done; we just have to pull it out of Angular 1.x
* Location service – we can start with the current $location implementation as a base, and rewrite it in ES6.
* Routing service – this needs to be totally redone
* Implement "view" component on top of the new compiler

I expect to iterate on the API design while building our representative app. A suggestion after reading.. keep the router barebones but extendable with module import. A state would be a component that has only a template, route, and controller functionality baked in. Hooks, data wrangling, and other features get added as needed just like the other ng2 modules.