The Evolution of Injecting Services in Ember

Posted On: February 26, 2015

I've been using Ember for about two years now. One of the things that initially impressed me is you don't need a lot of boilerplate code. At a bare minimum you need an `Ember.Application.create()` and an application template to get your app running. It won't do anything, but it is a perfectly functioning application. The first Ember app I wrote is still up on the Internet here: http://raytiley-msse-static-scheduler.herokuapp.com.

If you dig into the source code you'll see there are only `Routes`, `Models`, and `Controllers`. Alas, things were simpler back then. The app I develop today has `Mixins`, `Transforms`, `Helpers`, `Views`, `Adapters`, `Serializers`, and I think we just passed fifty `Components`.

If you are new to Ember then this might seem like a lot to manage, but actually it is quite the opposite. All these things have meaning. They all have their own folder in my app, and just by naming them correctly Ember CLI magically wires them all together into a working application. My favorite thing about Ember is how it can take away the pointless choices, and nothing is more pointless then deciding where to put a thing, or what to call it.

However, it is not all rainbows and unicorns. There have always been these pesky things in real apps that don't fit neatly into one of Ember's buckets. They usually need to be used from multiple parts of your app, think logging, analytics, sessions, authentication, etc. Up until Ember 1.10 you had two basic approaches for these shared objects.

Option One - Make it a Controller

This is a fine approach. Controller is one of those generic words like manager. When programming if you're not in a creative mood just call it a Manager. Unless you happen to be writing an Ember app, then just call it a Controller. You can then get access to your plethora of controllers using `needs` or `controllerFor`. This might look like the following:

```
// controllers/flash-messages.js
// This controller will be a singleton, meaning any place we use it we'll get the
// So we can push a message on from a route or a controller and the application t
// display all the messages.
```

```
export default Ember.Controller.extned({

  pushMessage(type, message) {
    // Do your sweet thing here
  },

  messages: function() {
    // Return the latest message, or all the messages, whatever
  }.property('someInternalThing')

});

// routes/posts/view.js
// In a Route we can use controllerFor to access our flash message controller.
export default Ember.Route.extend({
  actions: {
    postComment: function(trollingMessage) {
      // Do your comment stuff
      var flashManager = this.controllerFor('flash-messages');
      flashManager.pushMessage('success', 'You are a troll.')
    }
  }
}
});

// controllers/posts/view.js
// In a controller we can use the needs api to Ember know we need access to the f
// It is very common to use an alias so you don't have to type controllers a mill
export default Ember.Controller.extend({
  needs: ['flash-message'],
  flashManager: Ember.computed.alias('controllers.flash-message'),

  actions: {
    upVote: function() {
      // Do some voting
      var flashManager = this.get('flashManager');
      flashManager.pushMessage('error', 'Your vote does not count');
    }
  }
}
});
```

So beyond the obvious fact that controllers are going away in Ember 2.0 this approach has some drawbacks. First, you have two different APIs for getting the same thing depending on if you're trying to get it from a controller or a route. Second, you're limited to where you can access these objects. I may only need to use my flash manager in controllers and routes, but something like a logging service would be useful in other locations such as components.

Option Two - Dependency Injection with Initializers

Dependency injection sounds scary, but it's not. It is just a mechanism to reduce coupling between different objects in your application. The example above using `needs` and `controllerFor` is one way Ember does dependency injection. If you have ever passed an object to another object's constructor you've done constructor injection.

Tight coupling makes code harder to maintain. Imagine you are an aspiring `photographer`, but every time you want to take a photo you buy a new `iPhone` to take a picture. This is analogous to creating a `new LoggingService()` every time you want to log something in your application.

Besides being quickly broke, you the `photographer` are tightly coupled to `iPhone`. If we wanted to change the type of camera you use to 'DigitalSLR' we would need to follow you around and make sure we find every place you might take a picture and switch `iPhone` to `DigitalSLR`. This is analogous to having to change every place we use `new LoggingService()` in a big application.

Now imagine that at the start of every day some magical person put a camera on your bed side table and you used that camera all day. Having the camera provided to you makes it easy to change `iPhone` to `DigitalSLR`. We have **injected** the camera **dependency**, and now our code is easier to maintain.

In Ember that magical person that can change your `camera` in an `initializer`. Initializers are just bits of code that run before your application is booted and let you set things up. In our initializer we will `register` our dependency and then `inject` it into the types of objects where we want to use it. Coding our logging service this way would look like this:

```
// utils/logger.js
// Just a simple object that does some logging for us
export default Ember.Object.extend({

  log(type, message) {
    // Do your sweet thing here
  }

});

// initializers/logging.js
// In our initializer we import our Logger and then register it as logger:main
// Then we can inject logger:main into whatever types of objects we want access t
import Logger from '../utils/logger.js';
```

```
export default {
  name: 'logger',
  initialize: function(container, application) {
    // register our logger
    application.register('logger:main', Logger);

    // inject our logger so it is availble in controllers, routes, and components
    application.inject('controller', 'logger', 'logger:main');
    application.inject('route', 'logger', 'logger:main');
    application.inject('component', 'logger', 'logger:main');
  }
};

// controllers/posts/view.js
// In our route there is a property logger that is our logging service
export default Ember.Controller.extend({

  activate: function() {
    this.logger.log('This route was activated.');
  }

});
```

There are lots of different knobs that can be dialed in here. For instance the default behavior is we get a singleton logger, meaning that all the logger properties will be the same across routes, controllers, and components. We could change this. For more details read the api docs. So what's wrong with this approach? Nothing in particular, I actually quite like it. There are a few things to be aware of however.

Notice I didn't need to declare a `logger` property on my `PostsViewRoute`. The `logger` property is injected onto all my routes whether I like it or not. I find it good practice to declare my injected properties with `null` (`logger: null`). I spent a long time once debugging an injected property conflicting with another property. By declaring it explicitly I prevent myself from accidentally using that name for something else.

Another thing about this approach is it can be a bit tedious to limit your injections. Something like logging we want everywhere, but that's not always the case. You can configure inject a bunch, but if you're lazy it is easy just to give all the routes something that maybe only one or two of them need.

Meet Ember.inject.service and Ember.inject.controller

What's great about Ember is how it embraces paving cow paths. Pretty much every non

trivial Ember application is doing some combination of the above. In my own app I have lots of `needs` (pun intended) and several initializers injecting services. These patterns are tried, trued, and we know where all the cow shit is. So lets smooth them over using some new APIs. The controller part of our first example becomes:

```
// controllers/posts/view.js
export default Ember.Controller.extend({
  flashManager: Ember.inject.controller('flash-message'),

  actions: {
    upVote: function() {
      // Do some voting
      var flashManager = this.get('flashManager');
      flashManager.pushMessage('error', 'Your vote does not count');
    }
  }
});
```

Granted this doesn't do a whole lot for us, we only got to remove one line of code. Also if we need access to a controller from a route we still need to use `controllerFor`. What I'm really excited about is `Ember.inject.service()`. Our second example becomes:

```
// services/logger.js
// Just a simple object that does some logging for us
export default Ember.Service.extend({

  log(type, message) {
    // Do your sweet thing here
  }

});

// routes/posts/view.js
export default Ember.Controller.extend({
  logger: Ember.inject.service('logger'),

  activate: function() {
    this.get('logger').log('This route was activated.');
  }

});
```

Sweet! We got to remove a whole file, the initializer is gone. Notice also that we moved `logger.js` from `utils` to `services`. By placing our services in the `services` folder Ember CLI

will find them for us automatically. One less pointless choice. (the choice of `utils` was pointless. I just made that up in my own app.) I also like this better because I'm explicitly delcaring that I want the `logger` service available on `PostsViewRoute`. Other routes won't have a `logger` automatically. This also means that I don't have to worry about my injected property conflicting.

Now there are some rules for what can be injected where, but there is no public API for customizing that yet. For the most part you can inject services where you would think, including `routes`, `controllers`, `views`, and `components`. This isn't only for your own services. Addons can expose services and you can inject them into your apps objects. For example Ember Data has an open PR to allow you to expose the `store` as a service.

I'm a big fan of this API. It allowed me to delete a bunch of initializers in our app, and removed some silly choices from my day to day development. I think the Services API shows how Ember can constantly iterate on ideas without making me throw away all my code.

Sendage Climbing • 8 months ago

While refactoring to use the new Service API, I did find it annoying to have to change "this.logger" with "this.get('logger')". Any way to make this less painful?

Roger Adams • 9 months ago

Thanks for the write-up, found this very helpful!

jkneb • 10 months ago

Really insightful, thanks a lot. Small typos in the snippets comments though. You're mentioning the file 'view.js' which in 2 or 3 cases should be pointing to a supposed 'route.js'. Thanks again for sharing this!

codeofficer • 10 months ago

I love the evolution aspect of this article. Great work! One small correction: It's recommended to have your Service object extend Ember.Service ... The reason is that they made add functionality to that class later on.

raytiley Mod → codeofficer • 10 months ago

Nice catch. I'll update that ASAP. In practice I've been migrating things over that already extend Ember.Object. Works fine, but future proofing is always wise.

Steven Lindberg • 10 months ago

Great writeup! If you're willing to sacrifice a little explicitness, you can omit the `'logger'` param from the service injection, since it will default to looking up a service with the same name as the property:
https://github.com/emberjs/emb....

raytiley Mod → Steven Lindberg • 10 months ago