Learn Angular

BY MAKING A QUIZ APP FROM SCRATCH

HUNGRYTURTLECODE.COM

Part 8

Check Out The Whole Course Index

Make The Factory Useful – Dependency Injection

It is no use to us to have a factory if we cannot use it inside our controller. In this part we will inject the factory into the list controller so we can use it. Then we will refactor the list controller to remove the now redundant quizActive code. Let's dive into some dependency injection.

If you want to see the app for yourself, check it out here.

The git repo can be found here.

You Know The Drill.

Watch the video, or read the article. Or both.

Click the image to go to the video on youtube or go here https://www.youtube.com/watch?v=txzLabEIP_w&index=8&list=PLqr0oBkln1FBmOjK24_B4y_VAA8736wPq

[The next part can be found here.]

First things first, injecting the factory into the controller. To do this we will make use of an Angular method called $inject. This is an explicit way for us to inject any dependencies into our controllers.

```
(function(){

    angular
        .module("turtleFacts")
        .controller("listCtrl", ListController);

    ListController.$inject = ['quizMetrics'];

    // Rest of code left out to keep this snippet clean

})();
```

As you can see, with $inject we call it and set it equal to an array. Inside this array we just list all the dependencies in standard array syntax.

```
(function(){

    angular
        .module("turtleFacts")
        .controller("listCtrl", ListController);

    ListController.$inject = ['quizMetrics'];

    function ListController(quizMetrics){
        var vm = this;

        vm.quizMetrics = quizMetrics;

        // Rest of Controller code
    }

})();
```

We are not done quite yet. We now need to pass that factory that we injected, into our controller function as an argument. This allows us to then bind to the properties on the factory like we would with any other properties.

We have given the view model (ie. the html) access to all

properties on the factory object by creating a property on the controller using the vm. Syntax and setting it equal to the factory object.

Getting Rid Of The Redundant Code

We now have access to the quizActive property on the factory but also have a quizActive property inside the controller. This is now redundant so we shall remove it.

There is also a function in the controller called activateQuiz. But isn’t this logic part of the general logic for the quiz? It is not specifically for the list controller.

This means that the list controller is doing more than one job! No separation of concerns. So we will remove that function from the controller and add that function into our factory, thus allowing all controllers access to that code.

With that code removed from the list controller, the controller looks like this:

(Notice we haven’t removed the activateQuiz function, just the logic inside it. This is because we will call the function we add to our factory from inside this function in our controller later on)

```
function ListController(){
    var vm = this;

    vm.data = turtlesData;
    vm.activeTurtle = {};
    vm.search = "";
    vm.changeActiveTurtle = changeActiveTurtle;
    vm.activateQuiz = activateQuiz;

    function changeActiveTurtle(index){
        vm.activeTurtle = index;
    }

    function activateQuiz(){
        // Fix this further down the tutorial
    }
}
```

Now let’s add that function that we removed from the controller into the factory. To keep things more general (and maybe help us down the road) we will call the function in the factory changeState.

```
(function(){

    angular
        .module("turtleFacts")
        .factory("quizMetrics", QuizMetrics);

    function QuizMetrics(){

        var quizObj = {
            quizActive: false,
            changeState: changeState
        };

        return quizObj;

        function changeState(state){
            quizObj.quizActive = state;
        }
    }
}
```

Wait! That Looks Weird!

Now, you may notice a few things with that above snippet from our factory. Number one, we are again using named functions instead of inline anonymous functions. Again, this keeps things

more explicit.

But the second thing you may notice, especially if you are used to more traditional programming languages like C, is that we are declaring our functions at the bottom, below the return statement. This would be invalid code in many languages. So why do we do it here?

First of all, JavaScript allows this syntax due to a concept called hoisting, which I won't go into now. Just know that the code is valid. Secondly, because it is valid code, we do it this way to make things easier to read. To separate out the interface and the implementation.

So when you first see this factory, you will straight away see the object that is being returned. At a quick glance you can know exactly what the code is doing at a high level, without having to dig into the actual implementation. It just makes it easier for us developers to read. The computer doesn't care.

Back In The Controller

Inside the activateQuiz function that we left blank earlier we can now call the changeState function that is on our factory and pass it in the value of true. This will set the quizActive to the true state.

```
function activateQuiz(){

    // We will create this function on the factory soon
    quizMetrics.changeState(true);
}
```

You may be wondering why we called quizMetrics.changeState and not vm.quizMetrics.changeState.

The reason for this is that we passed in quizMetrics to the controller, so we still have access to it in it raw form. We just added the used the vm. Syntax to attach all the quizMetrics properties to our view. But it is a secondary source. We should still be using the original factory that was passed in.

Back To The Future – Now With Dependency Injection

We are nearly back to the functionality we had before we started playing around with the factory. The only thing that is left to change is the ng-hide on the list controller html. This is still referencing list.quizActive, which doesn't exist anymore.

```
<div ng-controller="listCtrl as list" ng-hide="list.quizMetrics.quizActive">
```

Dependency Injection For The Quiz Controller

Now we inject the factory into our quiz controller in the exact same way as we did with the list controller. The quiz controller script will look something like this:

```
(function(){

    angular
        .module("turtleFacts")
        .controller("quizCtrl", QuizController);

    QuizController.$inject = ['quizMetrics'];

    function QuizController(quizMetrics){

        var vm = this;

        vm.quizMetrics = quizMetrics;

    }

})();
```

Now we have access to the factory properties – including the quizActive property. We can now use this in an ng-show to show the quiz controller markup whenever the start quiz button is pressed.

```
<div ng-controller="quizCtrl as quiz" ng-show="quiz.quizMetrics.quizActive">
```

Onwards To Part 9

We are now well on our way to creating this full application. In the next part we will look at how we can separate the concerns out even further using the knowledge we now have. This will lead us nicely into creating the markup to display the questions in the quiz.

I will see you in part 9.

Adrian