AngularJS

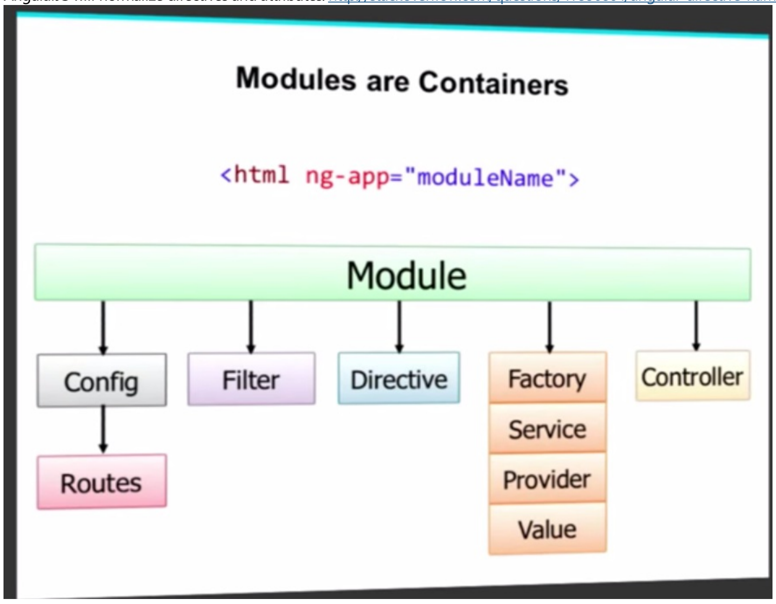
DataBinding.

$scope.value = ??, this will create $watch at the background, which will update DOM with $apply
on UI if anything change, this will be triggered by events which are handled by angularJS, which update the value on model and then in advance update DOM.
everytime the model changes will trigger $digest and then $watch.
在AngularJS内部，每当我们对ng-model绑定的name属性进行一次修改，AngularJS内部的$digest就会运行一次，并在运行结束之后检查我们使用$watch来监视的东西，如果和进行上一次$digest之前相比有了变化，则执行我们在其中绑定的处理函数。
at the background, angularJS sets up binding-concept at the background when parsing tags like ng-model.

Similar to WPF, IPropertyChange notifies the change to UI, meanwhile UI uses event to trigger the model change. Binding object is used to set up the association.

AngularJS can use $scope.$parent to access parent scope.

AngularJS will normalize directives and attributes: http://stackoverflow.com/questions/17990864/angular-directive-name-only-lower-case-letters-allowed



Can you pass parameters to an AngularJS controller on creation?

Another way is to use service.

277

Notes:

This answer is old. This is just a proof of concept on how the desired outcome can be achieved. However, it may not be the best solution as per some comments below. I don't have any documentation to support or reje
his topic.

Original Answer:

I answered this to Yes you absolutely can do so using `ng-init` and a simple init function.

Here is the example of it on plunker

HTML

```
<!DOCTYPE html><htmlng-app="angularjs-starter"><headlang="en"><scriptsrc="//ajax.googleapis.com/ajax/libs/angularjs/1.0.3/angular.min.js"></script><scriptsrc="app.js"></script></head><bodyng-controller
```

JavaScript

```
var app = angular.module('angularjs-starter',[]);

app.controller('MainCtrl',function($scope){

  $scope.init =function(name, id){//This function is sort of private constructor for controller
    $scope.id = id;
    $scope.name = name;//Based on passed argument you can make a call to resource//and initialize more objects//$resource.getMeBond(007)};});
```

shareimprove this answer	edited Oct 16 '15 at 4:29 Josh **294**15	answered Jan 25 '13 at 22:56 Jigar Patel **3,390**1815

93

I'm very late to this and I have no idea if this is a good idea, but you can include the `$attrs` injectable in the controller function allowing the controller to be initialized using "arguments" provided on an element, *e.g.*

```
app.controller('modelController',function($scope, $attrs){if(!$attrs.model)thrownewError("No model for modelController");// Initialize $scope using the value of the model attribute, e.g.,
    $scope.url ="http://example.com/fetch?model="+$attrs.model;})<div ng-controller="modelController" model="foobar"><a href="{{url}}">Click here</a></div>
```

Again, no idea if this is a good idea, but it seems to work and is another alternative.

shareimprove this answer	answered May 12 '14 at 15:13 Michael Tiller **5,278**21126

30

This also works.

Javascript:

```
var app = angular.module('angularApp',[]);

app.controller('MainCtrl',function($scope, name, id){
    $scope.id = id;
    $scope.name = name;// and more init});
```

Html:

```
<!DOCTYPE html><htmlng-app="angularApp"><headlang="en"><scriptsrc="//ajax.googleapis.com/ajax/libs/angularjs/1.0.3/angular.min.js"></script><scriptsrc="app.js"></script><script>
       app.value("name","James").value("id","007");</script></head><bodyng-controller="MainCtrl"><h1>I am  {{name}} {{id}}</h1></body></html>
```

Passing data between controllers in Angular JS?

From the description, seems as though you could be using a service. Check outhttp://egghead.io/lessons/angularjs-sharing-data-between-controllers and AngularJS Service Passing Data Between Controllers to see some examples.

You could define your product service as such:

```
app.service('productService',function(){var productList =[];var addProduct =function(newObj){
      productList.push(newObj);};var getProducts =function(){return productList;};return{
    addProduct: addProduct,
    getProducts: getProducts
  };});
```

Dependency inject the service into both controllers.

In your ProductController, define some action that adds the selected object to the array:

```
app.controller('ProductController',function($scope, productService){
    $scope.callToAddToProductList =function(currObj){
        productService.addProduct(currObj);};});
```

In your CartController, get the products from the service:

```
app.controller('CartController',function($scope, productService){
    $scope.products = productService.getProducts();});
```

shareimprove this answer	edited May 7 '15 at 1:37	answered Nov 24 '13 at 21:54 Charx **2,072**11327

38

how do pass this clicked products from first controller to second?

On click you can call method that invokes broadcast:

```
$rootScope.$broadcast('SOME_TAG','your value');
```

and the second controller will listen on this tag like:

```
$scope.$on('SOME_TAG',function(response){// ....})
```

AngularJS: Service vs provider vs factory

All Services are **singletons**; they get instantiated once per app. They can be **of any type**, whether it be a primitive, object literal, function, or even an instance of a custom type.

The `value`, `factory`, `service`, `constant`, and `provider` methods are all providers. They teach the Injector how to instantiate the Services.

> The most verbose, but also the most comprehensive one is a Provider recipe. The **remaining four** recipe types — Value, Factory, Service and Constant — **are just syntactic sugar on top of a provider recipe**.

- The **Value Recipe** is the simplest case, where you instantiate the Service yourself and provide the **instantiated value** to the injector.
- The **Factory recipe** gives the Injector a factory function that it calls when it needs to instantiate the service. When called, the **factory function** creates and returns the service instance. The dependencies of the Service are injected as the functions' arguments. So using this recipe adds the following abilities:
  - Ability to use other services (have dependencies)
  - Service initialization
  - Delayed/lazy initialization
- The **Service recipe** is almost the same as the Factory recipe, but here the Injector invokes a**constructor** with the new operator instead of a factory function.
- The **Provider recipe** is usually **overkill**. It adds one more layer of indirection by allowing you to configure the creation of the factory.

  > You should use the Provider recipe only when you want to expose an API for application-wide configuration that must be made before the application starts. This is usually interesting only for reusable services whose behavior might need to vary slightly between applications.

- The **Constant recipe** is just like the Value recipe except it allows you to define services that are available in the **config** phase. Sooner than services created using the Value recipe. Unlike Values, they cannot be decorated using `decorator`.

" Hello world " example with `factory` / `service` / `provider`:

```
var myApp = angular.module('myApp',[]);//service style, probably the simplest one
myApp.service('helloWorldFromService',function(){this.sayHello =function(){return"Hello, World!"};});//factory style, more involved but more sophisticated
myApp.factory('helloWorldFromFactory',function(){return{
        sayHello:function(){return"Hello, World!"}};});//provider style, full blown, configurable version
myApp.provider('helloWorld',function(){this.name ='Default';this.$get =function(){var name =this.name;return{
            sayHello:function(){return"Hello, "+ name +"!"}}};this.setName =function(name){this.name = name;};});//hey, we can configure a provider!
myApp.config(function(helloWorldProvider){
    helloWorldProvider.setName('World');});functionMyCtrl($scope, helloWorld, helloWorldFromFactory, helloWorldFromService){

    $scope.hellos =[
        helloWorld.sayHello(),
        helloWorldFromFactory.sayHello(),
        helloWorldFromService.sayHello()];}
```

```
<scriptsrc="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.23/angular.min.js"></script><bodyng-app="myApp"><divng-controller="MyCtrl">
    {{hellos}}
</div></body>
```

To wrap it up, let's summarize the most important points:

- The injector uses recipes to create two types of objects: services and special purpose objects
- There are five recipe types that define how to create objects: Value, Factory, Service, Provider and Constant.
- Factory and Service are the most commonly used recipes. The only difference between them is that the Service recipe works better for objects of a custom type, while the Factory can produce JavaScript primitives and functions.
- The Provider recipe is the core recipe type and all the other ones are just syntactic sugar on it.
- Provider is the most complex recipe type. You don't need it unless you are building a reusable piece of code that needs global configuration.
- All special purpose objects except for the Controller are defined via Factory recipes.

Features / Recipe type	Factory	Service	Value	Constant	Provider
can have dependencies	yes	yes	no	no	yes
uses type friendly injection	no	yes	yes*	yes*	no
object available in config phase	no	no	no	yes	yes**
can create functions	yes	yes	yes	yes	yes
can create primitives	yes	no	yes	yes	yes

* at the cost of eager initialization by using `new` operator directly

** the service object is not available during the config phase, but the provider instance is (see the `unicornLauncherProvider` example above).