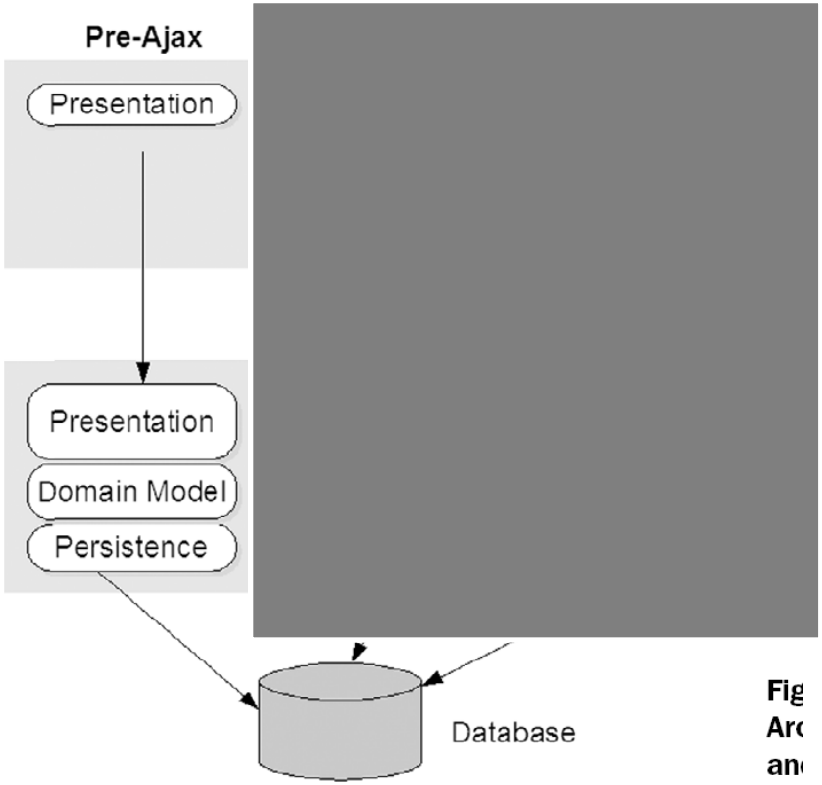
Single Page JS Applications

Antonio Servetti servetti@polito.it
Internet Media Group http://media.polito.it
Dip. di Automatica ed Informatica
Politecnico di Torino http://www.polito.it

Redefining web app. architecture

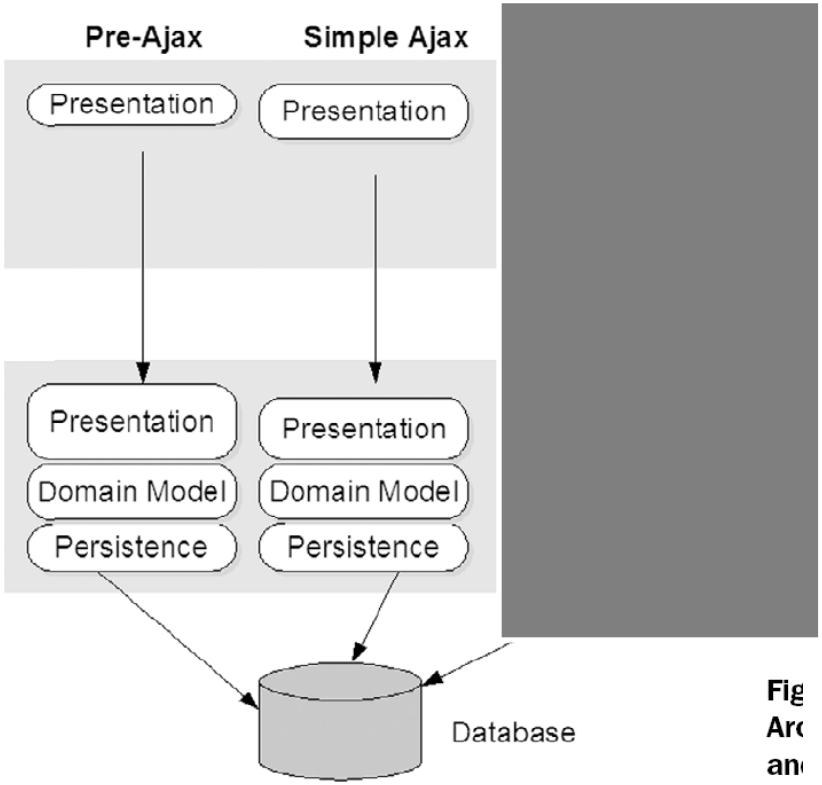
- Three tier architecture
  - ✓ Presentation-tier (View)
  - ✓ Business-tier (Controller)
  - ✓ Data-tier (Model)
- Pre Ajax
  - ✓ Action is taking place on the server
  - ✓ Browser is a dumb terminal



Reference:

Redefining web app. architecture

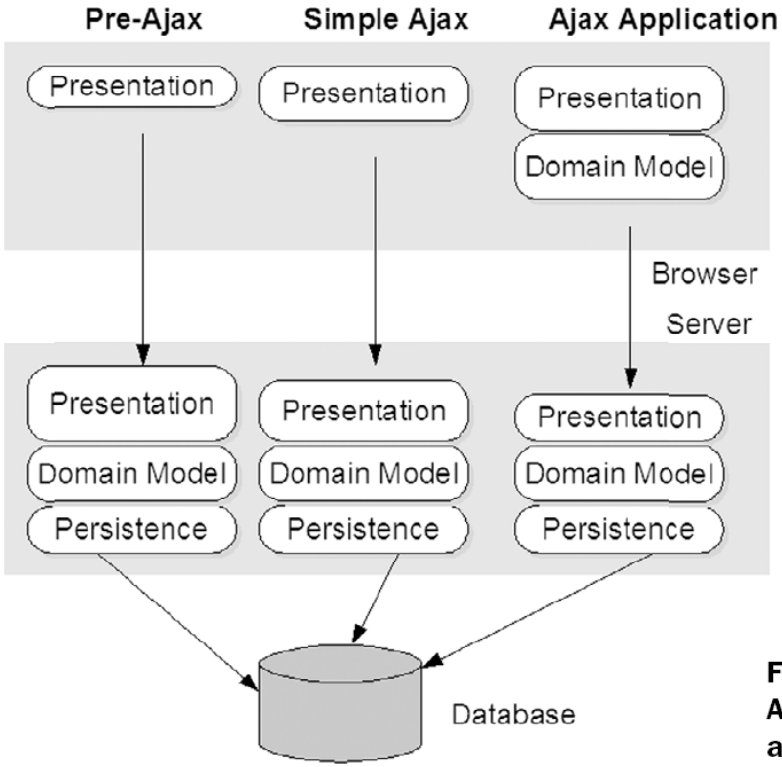
- Three tier architecture
  - ✓ Presentation-tier (View)
  - ✓ Business-tier (Controller)
  - ✓ Data-tier (Model)
- Simple Ajax
  - ✓ Server controls all aspects of the workflow
  - ✓ Browser rearranges DOM, presentation tier starts to get thicker
    (and smaller on the server)



Reference:

Full AJAX

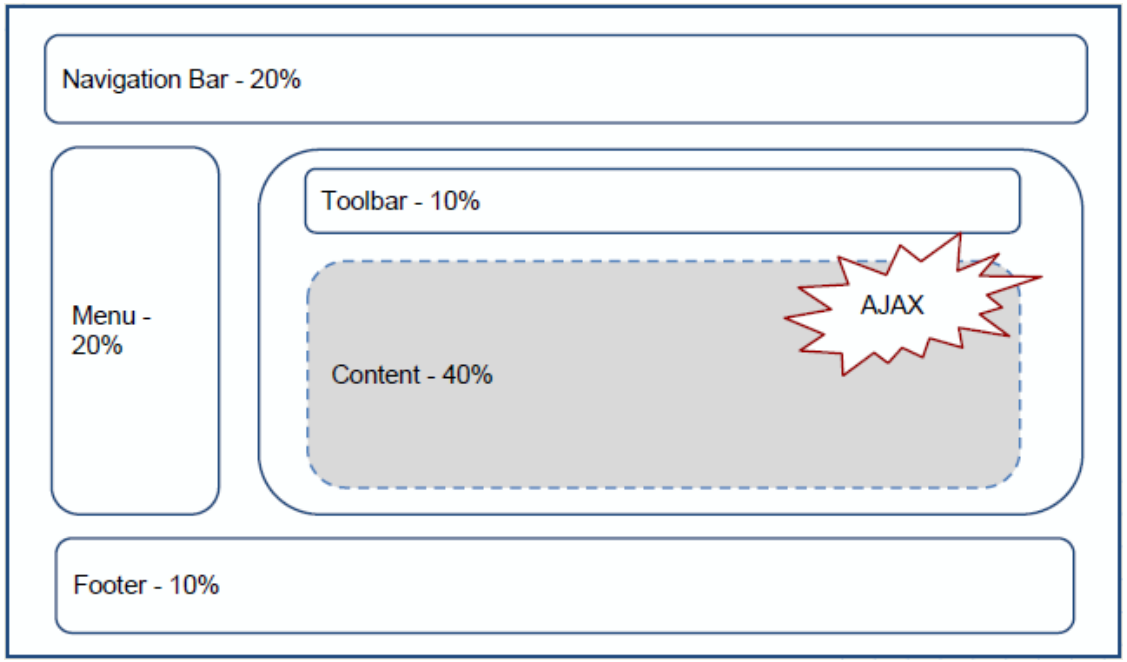
- Client-side
  - ✓ Browser JS gets complex
  - ✓ It handles part of the business model (flow control)
  - ✓ It tends to communicate to its b.m. rather than directly to the server
- Server-side
  - ✓ Its business model provides a coarser-grained facade to handle main use cases
  - ✓ Manages data persistence



Reference:

Single-page applications

- In an SPA the appropriate resources are dynamically loaded and added to the page as necessary
  - ✓ The page does not reload at any point in the process, nor does transfer control to another page



Reference:

Single-page applications

- SPAs instead of changing the page, change a view inside the same page.
  - ✓ When a user navigates to a new view, additional content is requested using an XHR to a server-side REST API
- The typical SPA consists of smaller pieces of interface representing logical entities all of which have their own **UI, business logic and data**.
  - ✓ E.g. shopping basket

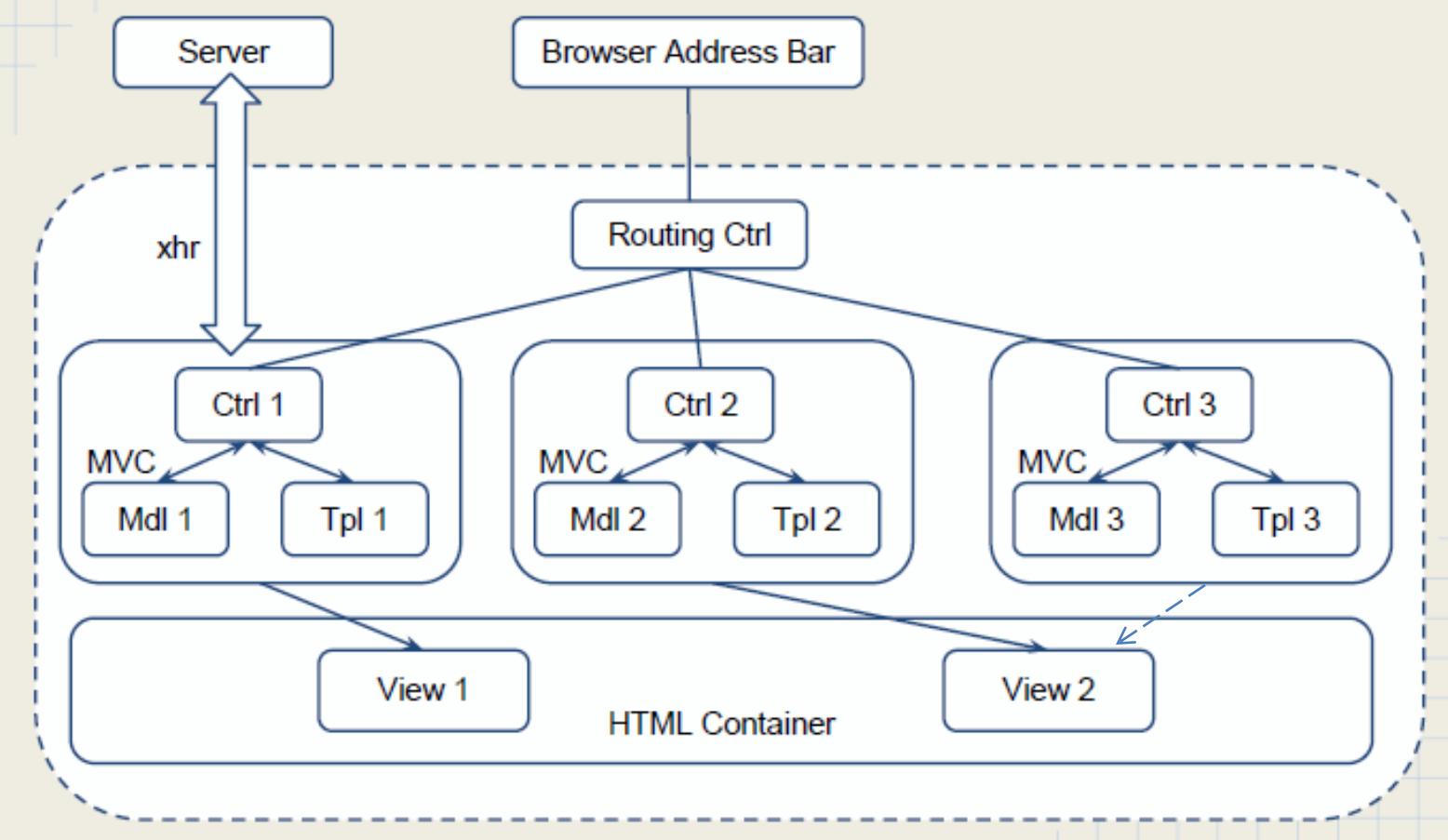
Reference: Single page apps in depth, http://singlepageappbook.com

New problems and solutions

- Problems
  - ✓ Mimic static addresses (http://mysite.com/...) and manage browser history
  - ✓ Handle Ajax callbacks
  - ✓ Mix HTML strings and Javascript
- Solutions
  - ✓ Routing (http://mysite.com/#/...)
  - ✓ Model – View – Controller paradigm (on the client-side)
  - ✓ Templating (Javascript-HTML)

Reference:

SPA architecture



Reference: /

MVC paradigm

- MVC provides clean separation of concerns
  - ✓ Data (Model)
    - Talks to the server using RESTful architecture
  - ✓ Presentation (View)
    - Specific elements of the User Interface
    - Is generated from a Template
    - One model might have multiple views
  - ✓ User input (Controller)
    - Glue between model and view
    - Handles user interactions
    - Might perform business logic role
    - *Note: the role of controller greatly varies in frameworks*

Reference:

JS MVC frameworks

- Major players (mainly MV*)
  - ✓ AngularJS
  - ✓ Backbone
  - ✓ Ember
  - ✓ Dojo
  - ✓ JavascriptMVC
  - ✓ Knockout

  - ✓ Note: MVC comes from server side architecture, when ported to client side with user interaction the *controller* element needs to be adapted.

Reference:

AngularJS

- Overview
  - The library reads in HTML that contains additional custom tag attributes;
  - It then obeys the directives in those custom attributes, and
  - Binds input or output parts of the page to a model represented by standard JavaScript variables.
  - The values of those JavaScript variables can be manually set, or retrieved from static or dynamic JSON resources.
  - The framework adapts and extends traditional HTML to better serve dynamic content through two-way data-binding that allows for the automatic synchronization of models and views.
  - Version 1.0 released December 2012

Reference: http://en.wikipedia.org/wiki/AngularJS

AngularJS - Motivation

- Developed by Google + great community
- REST easy
- MVVM to the rescue: model talks to view-model objects which listen for changes in the model
- Data binding and dependency injection: all happens auto-magically
  - ✓ You can ask for your dependencies as parameters in AngularJS service functions
- Extends HTML: you can operate your HTML like XML with the use of directives to trigger behaviors
- Makes HTML your Template.
- Enterprise-level Testing

Reference: http://net.tutsplus.com/tutorials/javascript-ajax/3-reasons-to-choose-angularjs-for-your-next-project/

AngularJS - Download

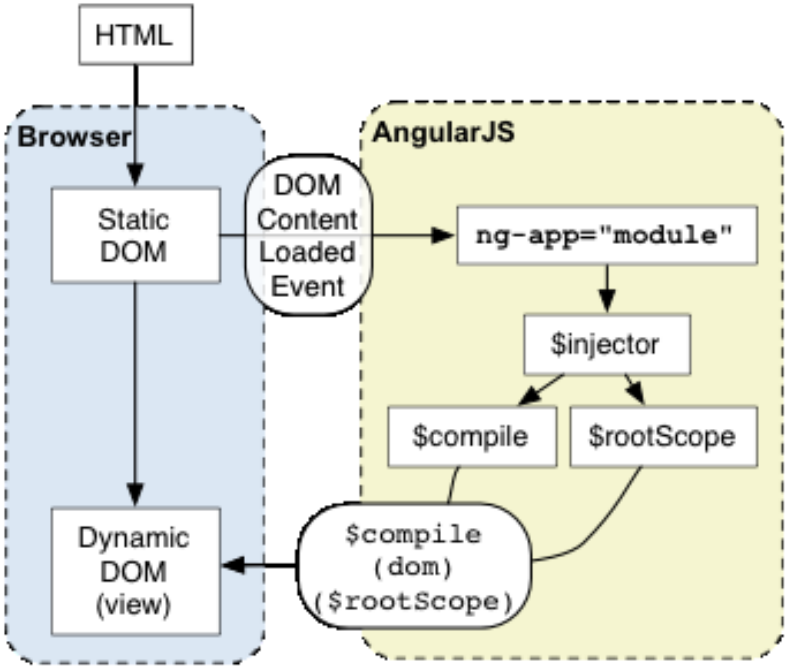
- http://angularjs.org/

- <script src="angularjs.js">
  </script>

Reference:

Bootstrap

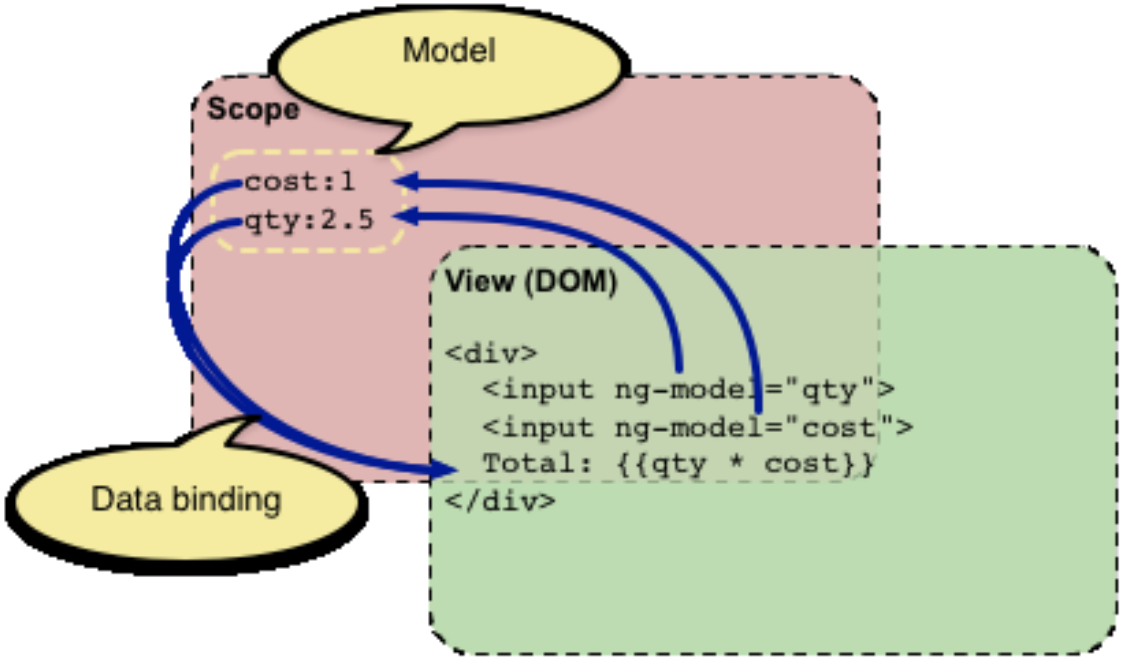
- Angular initializes automatically
- It looks for the ng-app directive which designates your application root.
- Then:
  - ✓ loads the module associated with the directive
  - ✓ creates the application injector
  - ✓ compiles the DOM



Reference: http://docs.angularjs.org/guide/bootstrap

Example

- The HTML file is called a "template".
- Angular parses and processes this new markup from the template using the so called "compiler".
- The loaded, transformed an rendered DOM then called the "view".



Reference: http://docs.angularjs.org/guide/concepts

Directive, filters, data binding

- Directive:
  - ✓ Directives are markers on a DOM element that tell AngularJS's HTML compiler to attach a specified behavior to that DOM element
  - ✓ <html **ng-app**>
- Data binding:
  - ✓ Data-binding is the automatic synchronization of data between the model and view components.
  - ✓ {{ name }}
- Filters:
  - ✓ A filter formats the value of an expression for display to the user. {{ name | uppercase }}

Reference:

Directives

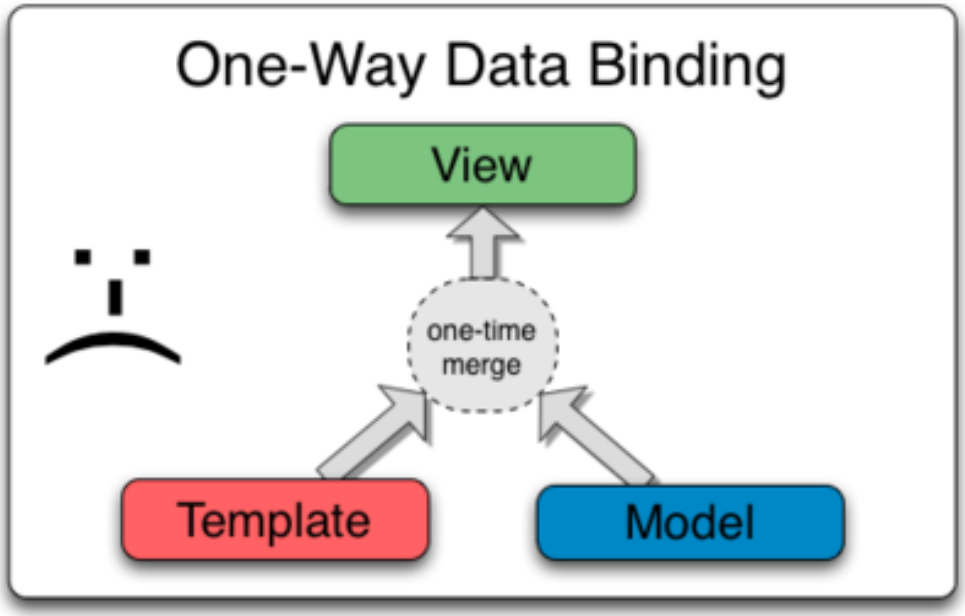
- ng-app
  - ✓ It initialized the AngularJS App (more later)
- ng-model
  - ✓ Adds a property up in the memory called “name” into what’s called “the scope”.

```
<!DOCTYPE html>
<html data-ng-app="">
<head> <title>Angular-js data binding</title> </head>
<body> Name: <br />
<input type="text" data-ng-model="name" /> {{ name }}
<script type="text/javascript" src="angular.min.js"></script>
</body> </html>
```

Reference:

The model

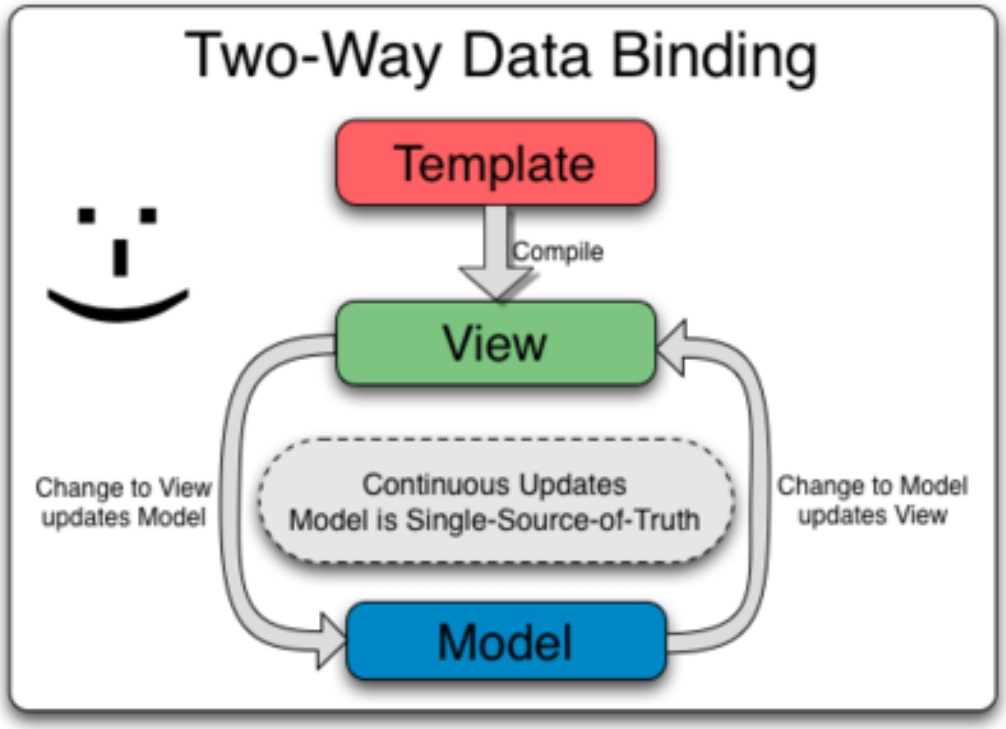
- Most templating systems bind data in only one direction: they merge template and model components together into a view



Reference: http://docs.angularjs.org/guide/databinding

Two-way data binding

- AngularJS enables "live view" because any changes to the view are immediately reflected in the model, and any changes in the model are propagated to the view



Reference: http://docs.angularjs.org/guide/databinding

More on directives

- ng-init
  - ✓ Sets initialization data that I want to bind to and display
- ng-repeat
  - ✓ instantiates a template once per item from a collection
- ng-src
  - ✓ Delays img src intepretation to get handled by angular

```
<div class="container"
data-ng-init="names=['Barbara', 'Antonio', 'Cristina']">
    <h3>Loop through names with ng-repeat</h3>
    <ul>
        <li data-ng-repeat="name in names">{{name}}</li>
    </ul>
    {{names | json}}
</div>
```

Using filters

- Keep formatting into presentation (not logic)
  - ✓ Examples: uppercase, orderBy, currency (see API)
- Syntax
  - ✓ {{ expression | filter }}
  - ✓ {{ expression | filter1 | filter2 }}
  - ✓ {{ expression | filter1:argument1 }}

```
<ul>
   <li data-ng-repeat="name in names | orderBy:'toString()'">
         {{name | uppercase}}
   </li>
</ul>
```

Reference: http://docs.angularjs.org/api/

More on filters

- The *filter* filter
  - ✓ Selects a subset of items from array and returns it as a new array
  - ✓ {{ filter_expression | filter:expression:comparator }}

```
<div class="container" data-ng-init="persons = [ {name: 'Barbara',
city: 'Cuneo'}, {name: 'Antonio', city: 'Torino'}, {name: 'Cristina',
city: 'Milano'} ]">
<h3>Loop through names with ng-repeat</h3>
Filter by: <input type="text" data-ng-model="searchText"/>
<ul>
<li data-ng-repeat="person in persons | filter:searchText |
orderBy:'name'"> {{person.name}} - {{person.city | uppercase}}
</li>
</ul>
{{names | json}}
</div>                                                    // app5.html
```

Reference: http://docs.angularjs.org/api/

View, controllers and scope

- The **Controller** drives things.
  - ✓ It's going to control ultimately what data gets bound into the **view**, i.e. prepares data for the view.
  - ✓ If the View passes up data to the controller it will handle passing off maybe to a service which then updates a back-end data store.

The view doesn't have to know about the controller, and the controller definitely doesn't want to know about the view.

$scope is the glue (ViewModel) between a controller and a view

Reference:

Create a controller

- The scope is automatically bound into the view once the view knows about its controller

```
 <!-- was data-ng-model="customer ..." -->
<div class="container" data-ng-controller="SimpleController">

<script>
// dependency injection
function SimpleController($scope) {
        $scope.persons = [
                {name: 'Barbara', city: 'Cuneo'},
                {name: 'Antonio', city: 'Torino'},
                {name: 'Cristina', city: 'Milano'}
        ];
}
</script>                                                    // app5.html
```

Reference:

The controller

- A controller is a JavaScript function
- It contains data
- It specifies the behavior
- It should contain only the business logic needed for a single view.

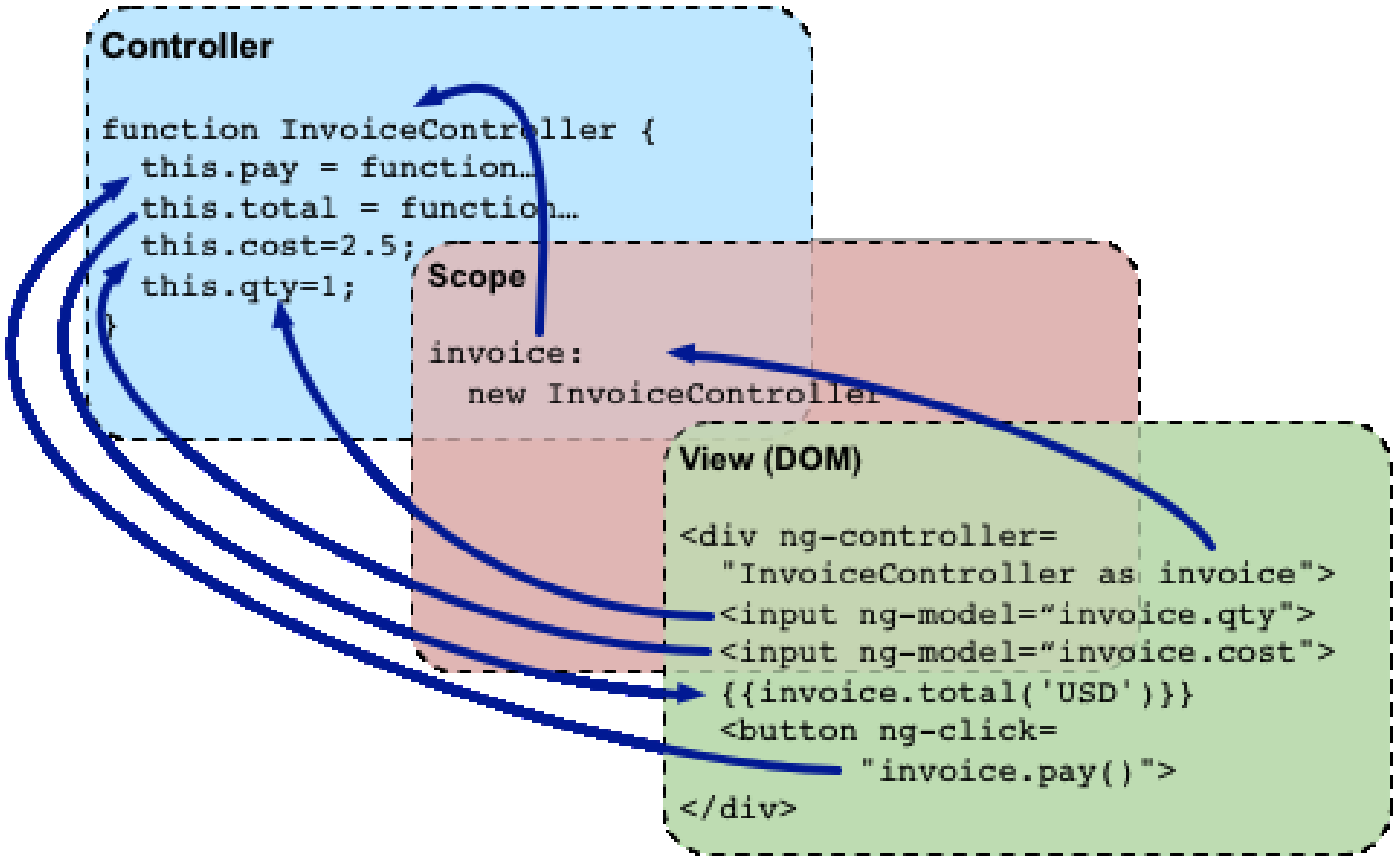
Reference:

The scope

- It's an object that refers to the application model
- It's an execution context for expressions like {{ todo.name }}
- Scopes are arranged in hierarchical structure which mimic the DOM structure of the application
- Scopes can watch expressions and propagate events
- $rootScope
  - ✓ Every application has a single root scope. All other scopes are descendant scopes of the root scope.

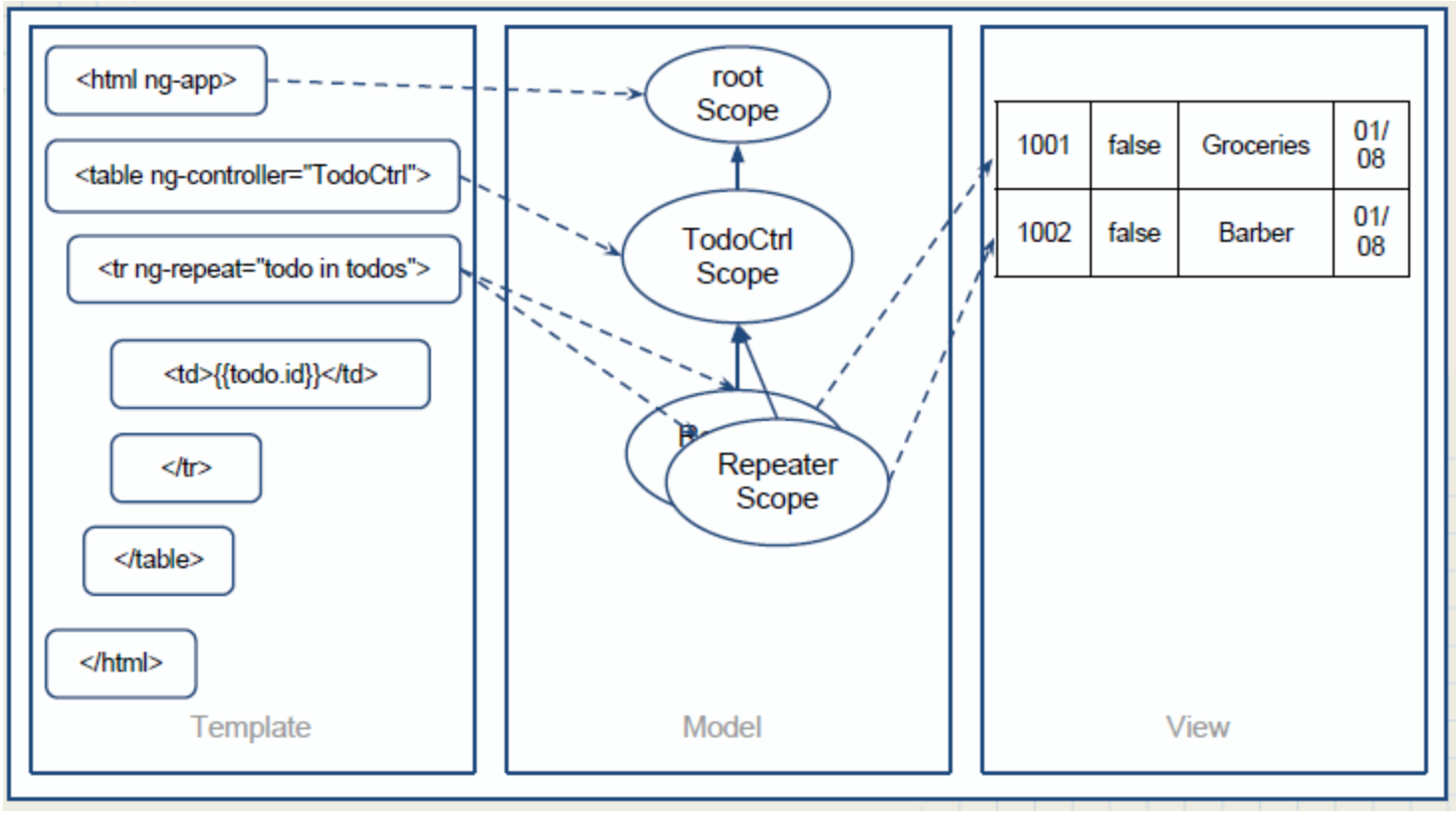
Reference:

The controller-scope-view



Reference:

The controller-scope-view

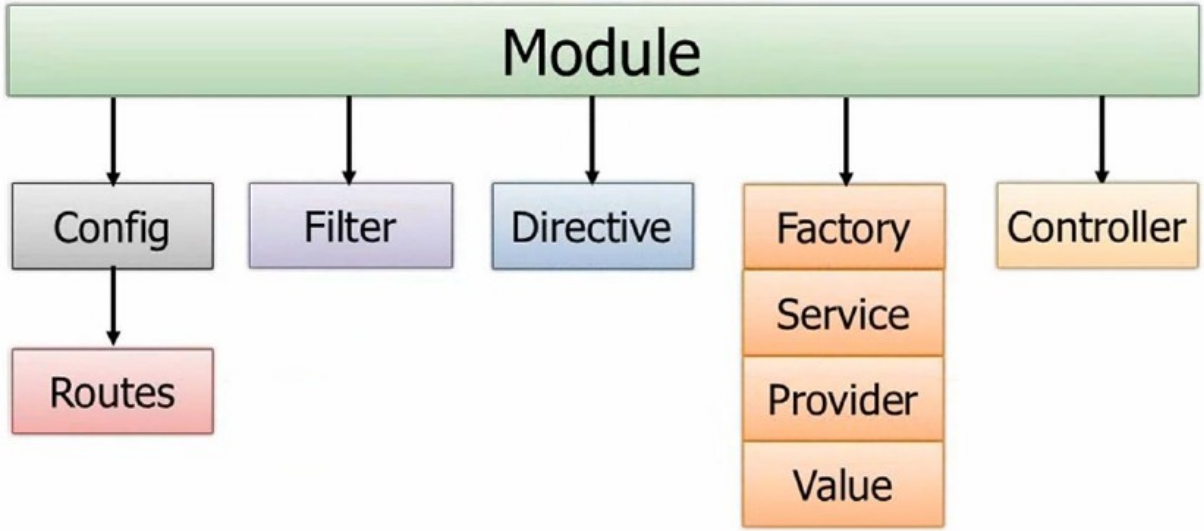


Reference:

Modules

- Module are containers

- ✓ It is a place where you can collect and organize components in a *modular* way
- ✓ A service module, a directive module, a filter module, …
- ✓ And an application level module which depends on the above modules, and which has initialization code.



Create a module

- The scope is automatically bound into the view once the view knows about its controller

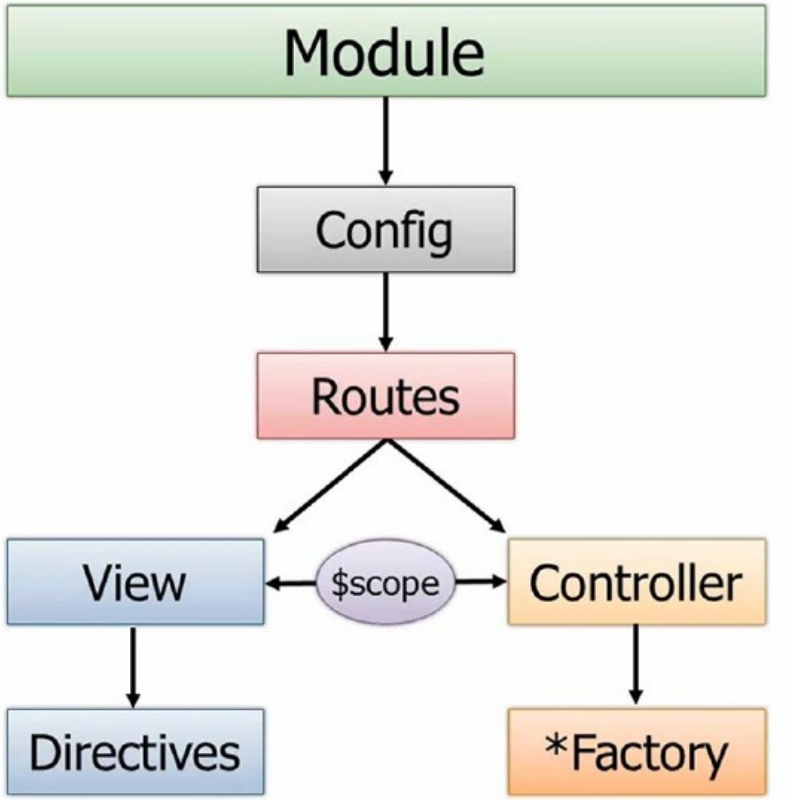
```
<html data-ng-app="myApp">
<div class="container" data-ng-controller="SimpleController">

<script>
var myApp = angular.module('myApp', []);

myApp.controller('SimpleController', function ($scope) {
        $scope.persons = [
                {name: 'Barbara', city: 'Cuneo'},
                {name: 'Antonio', city: 'Torino'},
                {name: 'Cristina', city: 'Milano'}
        ];
});
</script>                                                    // app6.html
```

Reference:

Modules, routes and factories

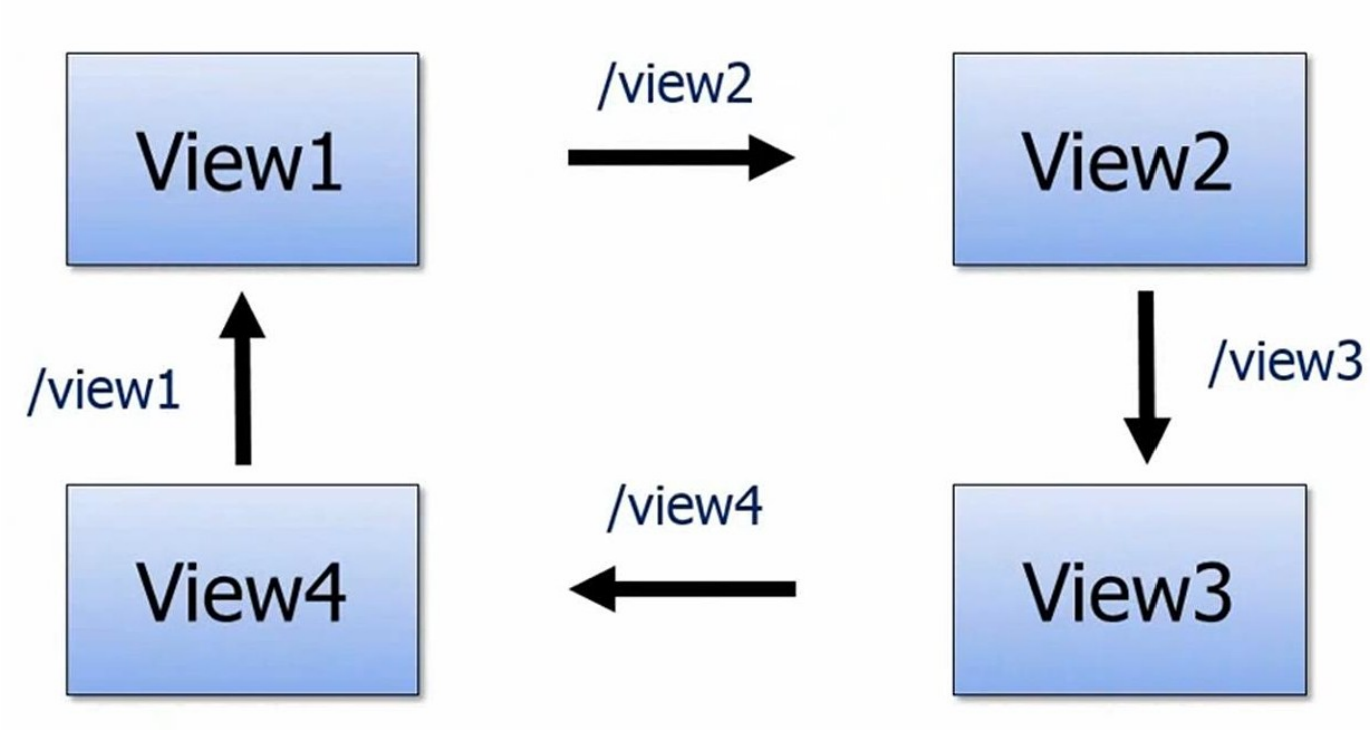
- So a module can have something off of it called a config function and it can be defined to use different routes.

If you have different views and those views need to be loaded into the shell page then we need a way to be able to track what route we're on and what view that's associated with and then what controller goes with that view and how we do all of that marrying together of these different pieces



Routing

- Load different views into the single-page-app



Reference:

Create routes

- New dependency ngRoute in angular-route.js

```
<html data-ng-app="myApp">

<!-- ng-view handles loading partials into it based upon routes -->
<div data-ng-view=""></div>
<script type="text/javascript" src="angular.js"></script>
<script type="text/javascript" src="angular-route.js"></script>

<script>
var myApp = angular.module('myApp', ['ngRoute']);
myApp.config( ['$routeProvider',
  function ($routeProvider) {
  $routeProvider
     .when('/',
        { controller: 'SimpleController', templateUrl: 'viewA7.html'})
     .when('/viewB',
        { controller: 'SimpleController', templateUrl: 'viewB7.html'})
     .otherwise( { redirectTo: '/' });
}]);
</script>                                                    // app7.html
```

Create views

- "partials" are loaded in the view

```
<div class="container">
<h3>View A</h3>
Filter by:
<input type="text" data-ng-model="searchText"/>
<ul>
<li data-ng-repeat="person in persons | filter:searchText |
orderBy:'name'">
        {{person.name}} - {{person.city | uppercase}}
</li>
</ul>
{{persons | js
</div>
```

```
<div class="container">
<h3>View B</h3>
<ul>
<li data-ng-repeat="person in persons | orderBy:'city'">
        {{person.city | uppercase}} - {{person.name}}
</li>
</ul>
{{persons | json}}
</div>
```

Providers

- Providers are objects that provide (create) instances of services and expose configuration APIs that can be used to control the creation and runtime behavior of a service.
- In case of the $route service, the $routeProvider exposes APIs that allow you to define routes for your application.
- Note: Providers can only be injected into config functions.

Reference:

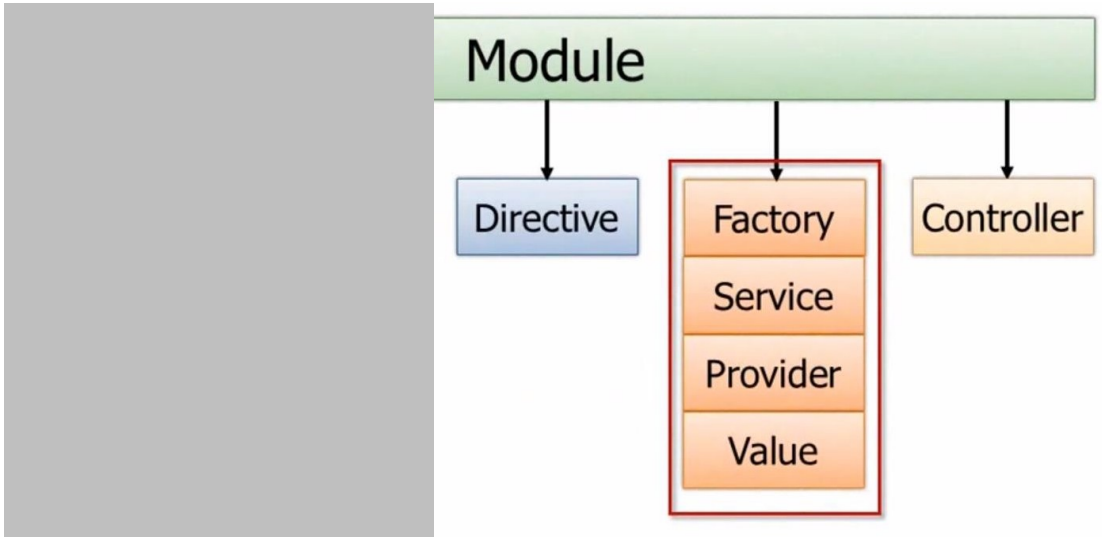
Modify the model

- The controller has to provide a method to add, delete, modify a record

```
Name: <input type="text" data-ng-model="newPerson.name" /><br />
City: <input type="text" data-ng-model="newPerson.city" /><br />
<button ng-click="addPerson()">Add</button>

myApp.controller('SimpleController', function ($scope) {
    // data
    $scope.persons = [
        // ...
    ];
    // behaviour
    $scope.addPerson = function () {
        $scope.persons.push(
        { name: $scope.newPerson.name, city: $scope.newPerson.city});
    };
});                                                    // app8.html
```

Provider, factory, service

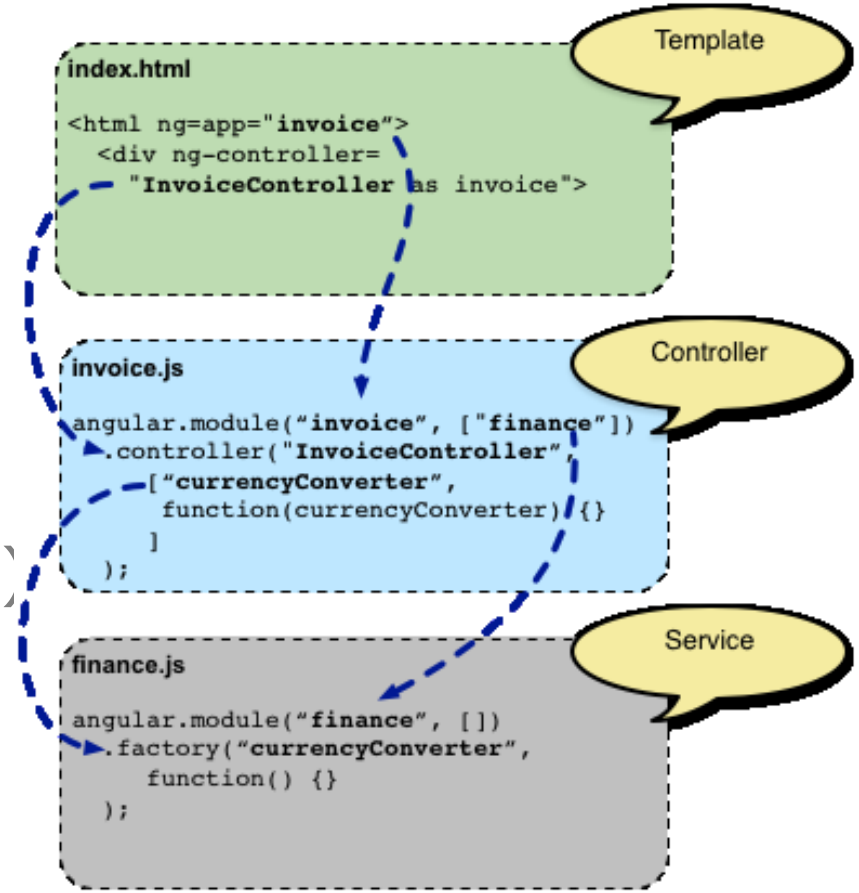
- Encapsulate data functionality into
  - ✓ Factory, Service, Provider, Value
- instead than in multiple controllers that manage user interaction (not model behaviour)
- e.g. person's get/set/check



Reference:

Provider, factory, service

- Encapsulate data functionality into
  - ✓ Factory, Service, Provider, Value
- instead than in multiple controllers that manage user interaction (not model behaviour)
- e.g. person's get/set/check



Reference:

Provider, factory, service

- Difference: the way in which they create the object that goes and gets the data.
- Factory
  - ✓ you create an **object** inside of the factory and return it
- Service
  - ✓ you have a function that uses *this* to define a function
- Provider
  - ✓ you define $get that is used to get the data
- Value
  - ✓ it is like a config value

Reference:

Create a factory

- A factory for "persons"

```
myApp.factory( 'simpleFactory', function () {
   var persons = [ /* ... */ ];

   var factory = {};
   factory.getPersons = function ()  { return persons; }
   factory.postPerson = function (p) { persons.push(p); }

   return factory;          // return the factory !
});

// dependence injection, just put the name of the factory
myApp.controller('SimpleController', function ($scope, simpleFactory) {
   $scope.persons = [];
   init();
   // private function
   function init() { $scope.persons = simpleFactory.getPersons(); };
   // ...
});                                                    // app9.html
```

Building a REST application

- Server.js (Nodejs)
  - ✓ Read collection: GET /persons
  - ✓ Read item: GET /persons/:id
  - ✓ Add item: POST /persons
  - ✓ Update item: POST /persons/:id (or PUT)

- AngularJS Resource
  - ✓ A factory which creates a resource object that lets you interact with RESTful server-side data sources.
  - ✓ It has methods with high-level behaviors, no need to interact with the low level $http service.
  - ✓ Requires the ngResource module to be installed.

```
{ 'get':    {method:'GET'},
  'save':   {method:'POST'},
  'query':  {method:'GET', isArray:true},
  'remove': {method:'DELETE'},
  'delete': {method:'DELETE'} };
```

Reference:

Create a resource

- Dependency on ngResource

```
<script type="text/javascript" src="angular-resource.js"></script>
<script>
var myApp = angular.module(  'myApp', ['ngRoute', 'ngResource']);
</script>
```

```
myApp.factory( 'simpleFactory', function ($resource) {
   return $resource('/persons');
});

myApp.controller('SimpleController', function ($scope, simpleFactory) {
   $scope.persons = [];
   (function init() {
      // the factory it is now a resource
      $scope.persons = simpleFactory.query();
   })();
   // ...
});                                                    // app10.html
```

Complete service

- Retrieve the item

```
<button ng-click="curPerson($index);">{{$index}}</button>
{{person.name}} - {{person.city | uppercase}}
```

```
myApp.factory( 'simpleFactory', function ($resource) {
   return $resource('/persons/:personId');
});

myApp.controller('SimpleController', function ($scope, simpleFactory) {
   // ...
   $scope.curPerson = function(id) {
       $scope.selPerson = simpleFactory.get({personId: id});
   };
   // ...
});                                                    // app11.html
```

Complete service

- Modify the model on the server using POST

```
Id: {{selPersonId}}<br/>
<button  ng-click="updatePerson(selPersonId);">Save</button><br/>
```

```
myApp.factory( 'simpleFactory', function ($resource) {
   return $resource('/persons/:personId');
});

myApp.controller('SimpleController', function ($scope, simpleFactory) {
   // ...
   $scope.updatePerson = function(id) {
       $scope.selPerson.$save({personId: id});        // updates the item
       $scope.persons = simpleFactory.query();
   }
   // ...
});                                                   // app12.html
```

Tools

- JS IDE: Webstorm (JetBrains)
- SPA boilerplate: angular-seed

Reference:

Bibliography

- AngularJS in 60 Minutes
  - Video: http://www.youtube.com/watch?v=i9MHigUZKEM
- Single page apps in depth
  - http://singlepageappbook.com/
- Slides
  - http://www.slideshare.net/bolshchikov
- Books
  - "AngularJS", O'Reilly, 2013
- Reference documentation
  - http://angularjs.org