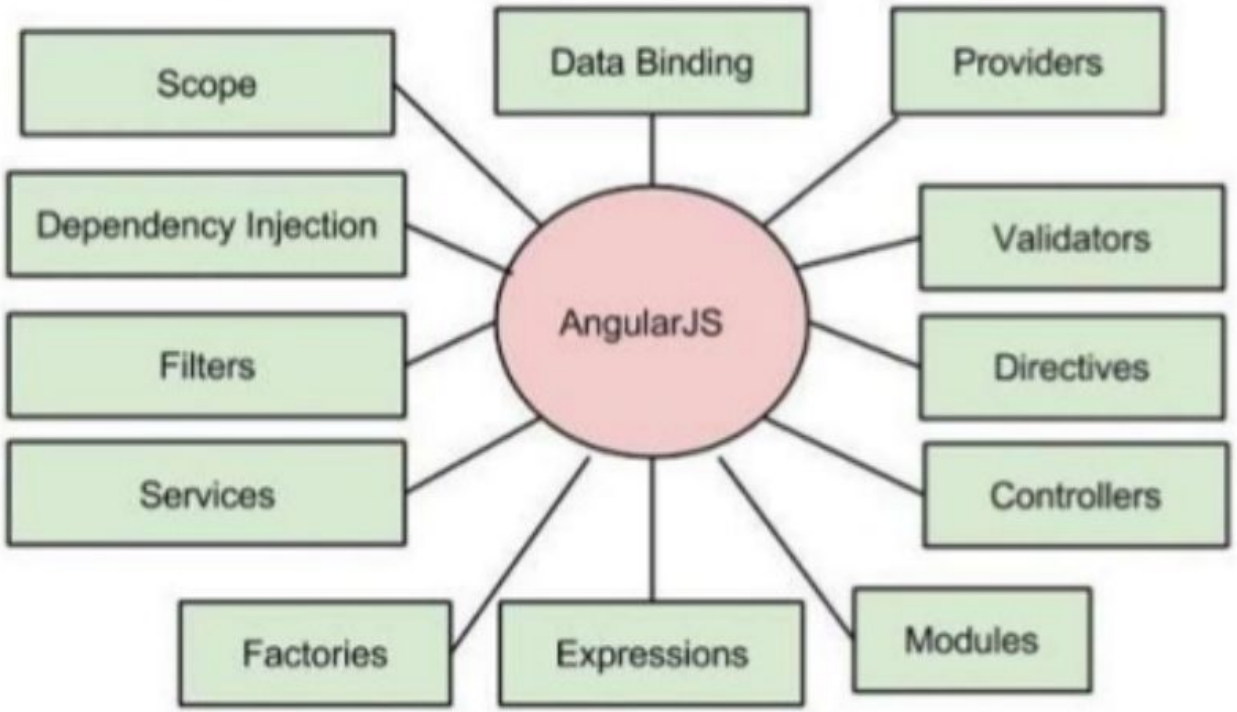
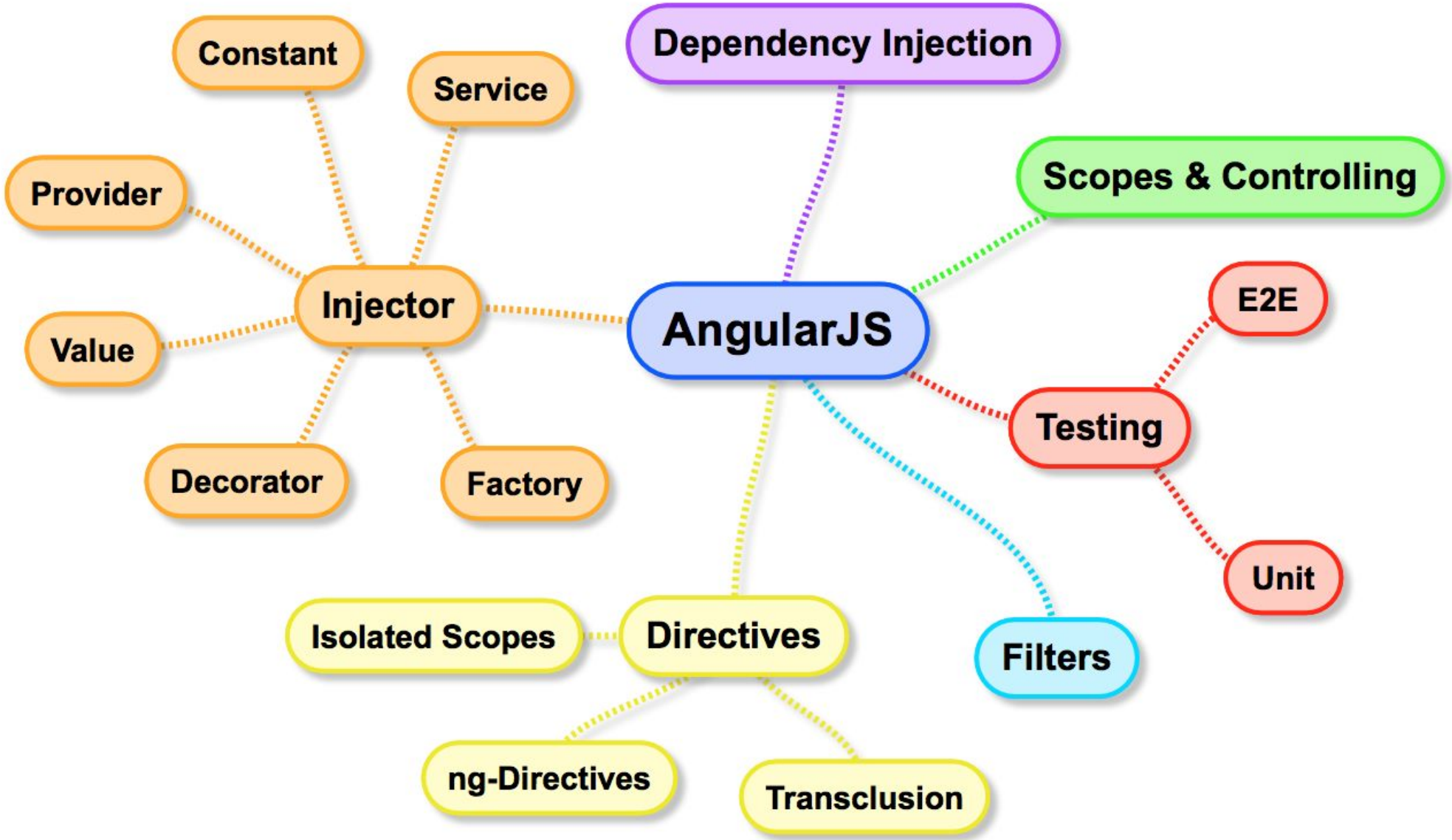
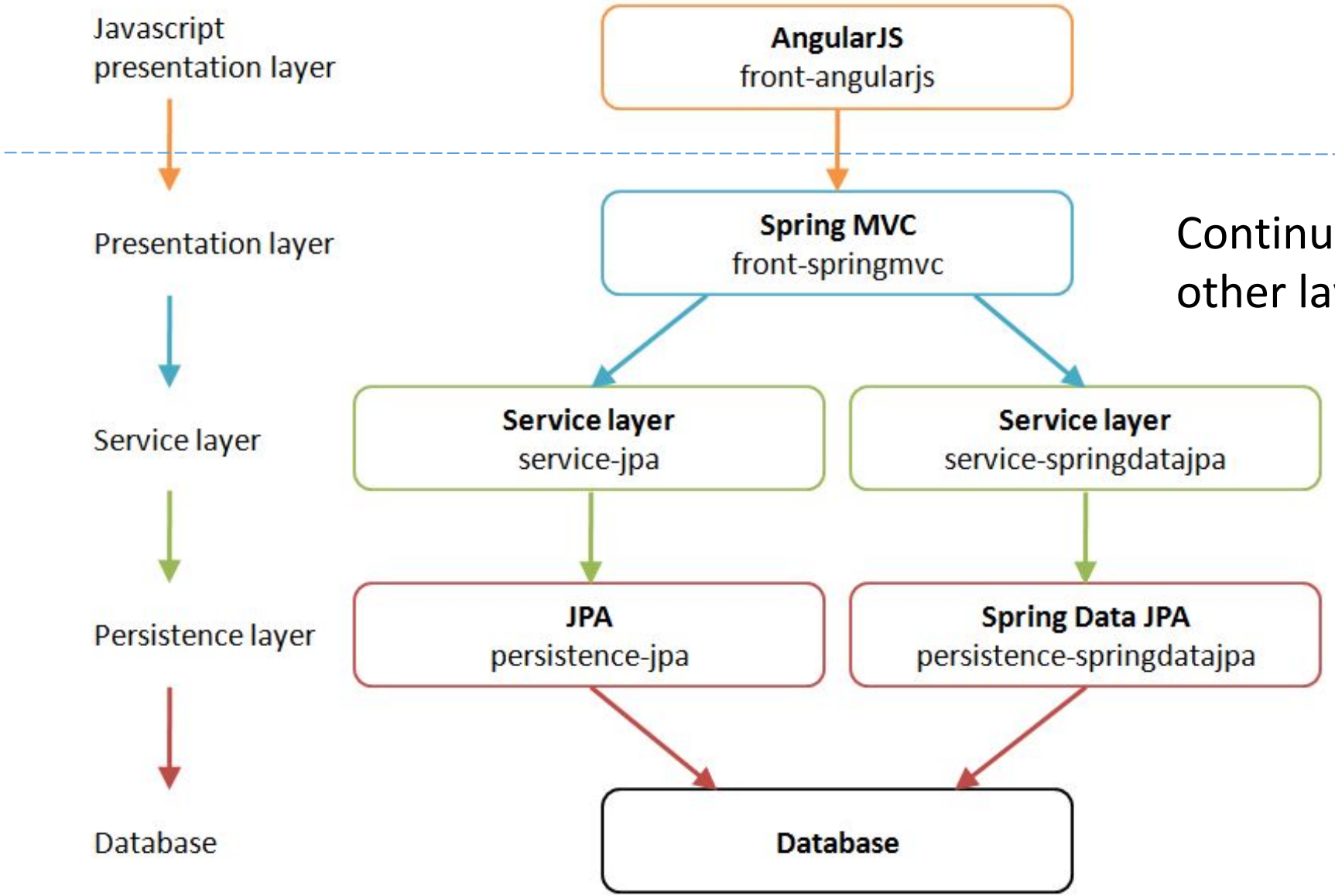


Angularjs (old) Tutorial
by: tjjenk2


Constant
Service
Dependency Injection
Provider
Scopes & Controlling
Injector
AngularJS
E2E
Value
Testing
Decorator
Factory
Unit
Isolated Scopes
Directives
Filters
ng-Directives
Transclusion

Angular is a Complete Client-Side Solution



Continue to use Spring MVC and other layers in the usual manner

JQuery – Dom Manipulation

```
// jquery
$(document).ready(function() {
  var data = [
    { "id": 1, "name": "Travis" },
    { "id": 2, "name": "Amy" },
    { "id": 3, "name": "Joshua" },
    { "id": 4, "name": "Abigail" },
    { "id": 5, "name": "Katie" },
    { "id": 6, "name": "Alexis" },
    { "id": 7, "name": "Madison" }
  ];

  // selectors
  var myFamilyEl = $('#myFamily');

  var update = function() {

    console.log(myFamilyEl);
    myFamilyEl.append('<ul>');
    data.forEach(function(element, index, array) {
      myFamilyEl.append('<li>');
      myFamilyEl.append('row:' + element.id + ' ' + element.name);
      myFamilyEl.append('<\li>');
    });
    myFamilyEl.append('</ul>');
  };

  update();
});
```

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Hello Plunker!</h1>

    <div id='myFamily'></div>

    <script src="https://code.jquery.com/jquery-1.11.3.min.js"></script>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.0/angular.js"></script>
    <script src="script.js"></script>
  </body>
</html>
```

- Imperative Style
- HTML in the javascript
- Took many lines of code
- Could use with templating framework

Angular - Bootstrap

```
(function () {
    angular.module('tjApp', [ ]).controller('tjController', [
      '$scope',     // inject the scope
      function($scope) {
        $scope.data = [
          { "id": 1, "name": "Travis" },
          { "id": 2, "name": "Amy" },
          { "id": 3, "name": "Joshua" },
          { "id": 4, "name": "Abigail" },
          { "id": 5, "name": "Katie" },
          { "id": 6, "name": "Alexis" },
          { "id": 7, "name": "Madison" }
        ];
      }
  ]);

})();
```

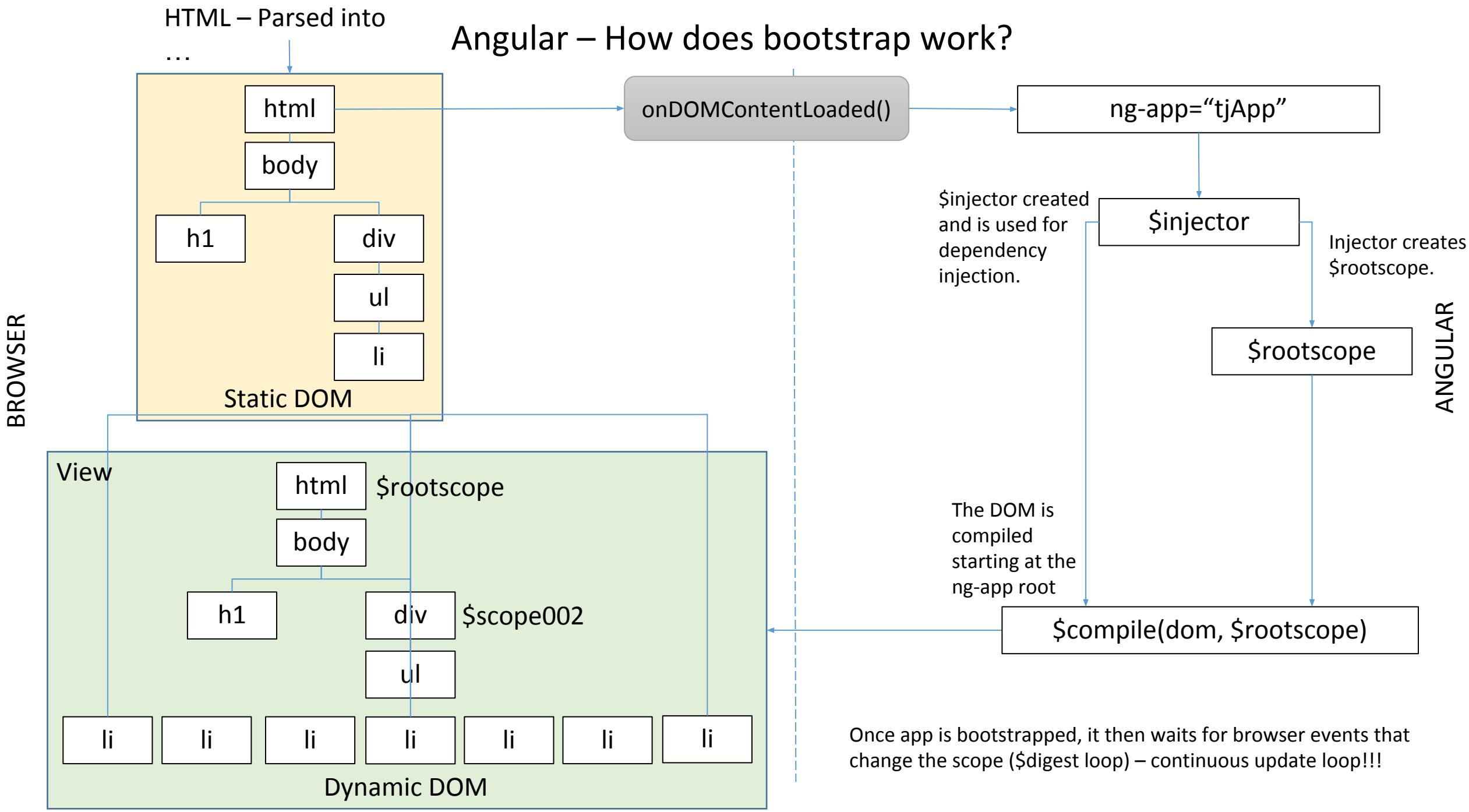
```
<!DOCTYPE html>
<html>
  <body ng-app="tjApp">   <!-- bootstrap angular -->
    <h1>Hello Plunker!</h1>

    <div ng-controller="tjController">
        <ul ng-repeat="person in data">
          <li>
            row: {{ person.id }} {{ person.name }}
          </li>
        </ul>
    </div>

    <script src="https://code.jquery.com/jquery-1.11.3.min.js"></script>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.0/angular.js"></script>
     <script src="script.js"></script>
  </body>
</html>
```

- Declarative Style
- HTML out of the javascript
- Less lines
- IOC – dependency injection
- DOM control structures for repeating, showing, disabling, hiding DOM fragments, etc.
- Grouping of HTML into reusable components

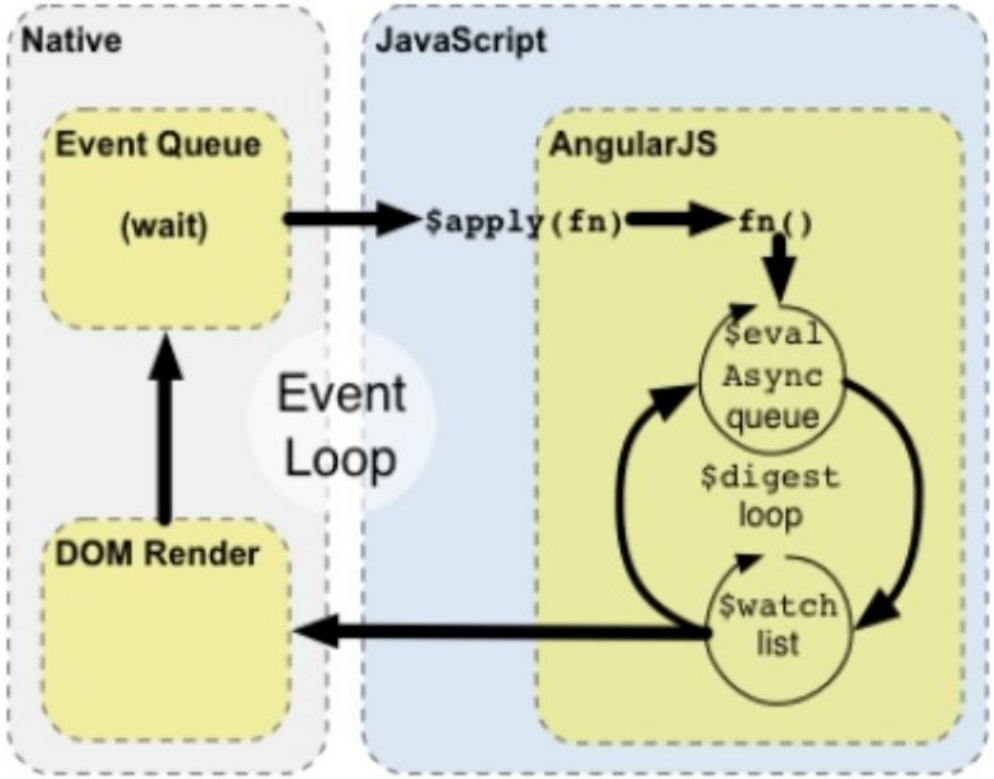
Angular – How does bootstrap work?



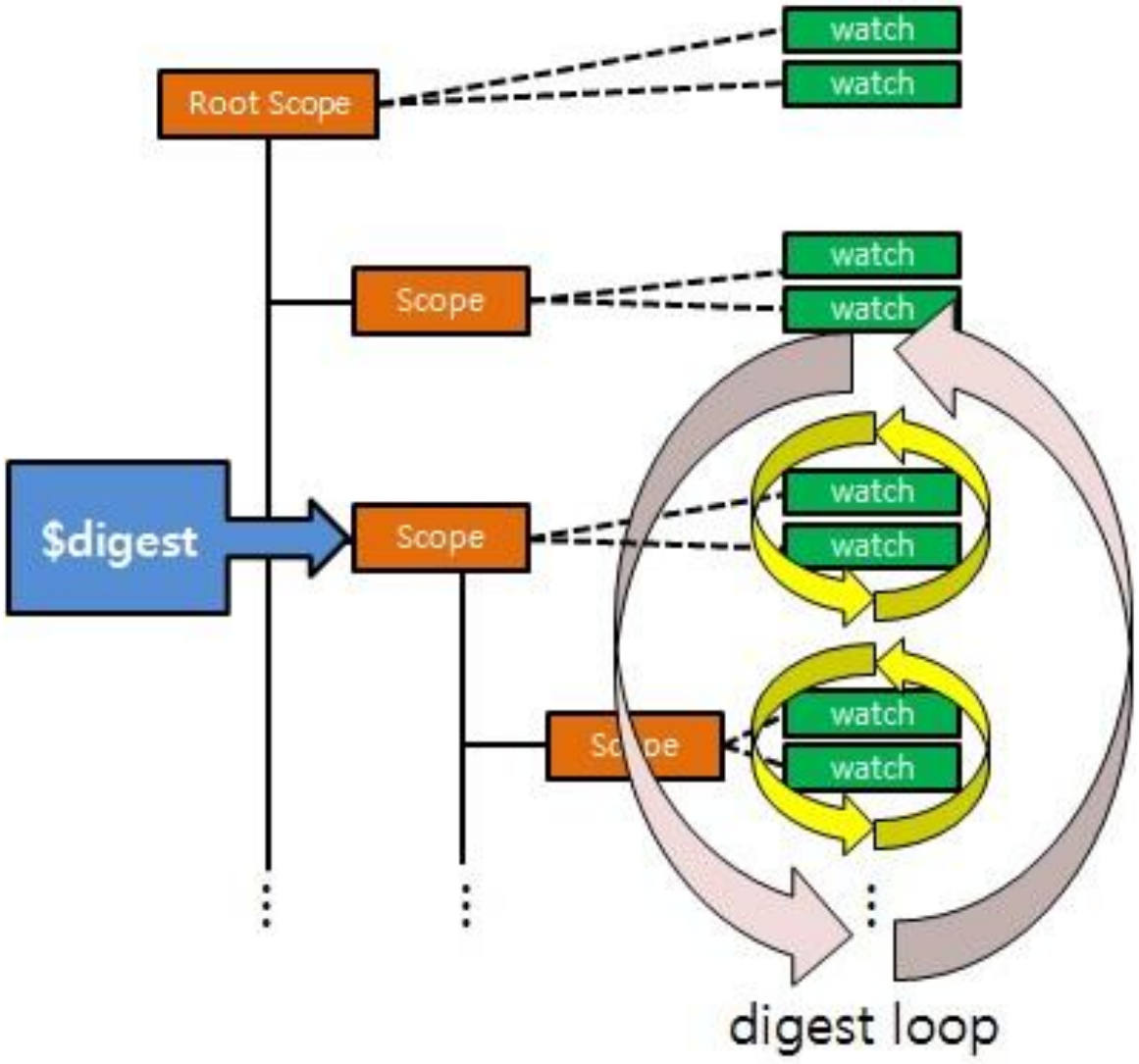
3. Framework features - Dirty checking

- Browser API's suck currently (no Observables for DOM objects to know when something changes)
- Angular needs a manual loop to know
  - Whether the DOM element changed and update the model
  - Whether the model changed and update the view
- Multiple passes by default
  - Only update when scope remains unchanged for x passes
  - Maximum 10 passes
- Pitfall
  - Dirty checking can slow down application substantially
  - Potentially uses a lot of CPU cycles
  - Keep objects/functions on your $scope as few as possible
    - Less updates === less dirty checks

$digest loop

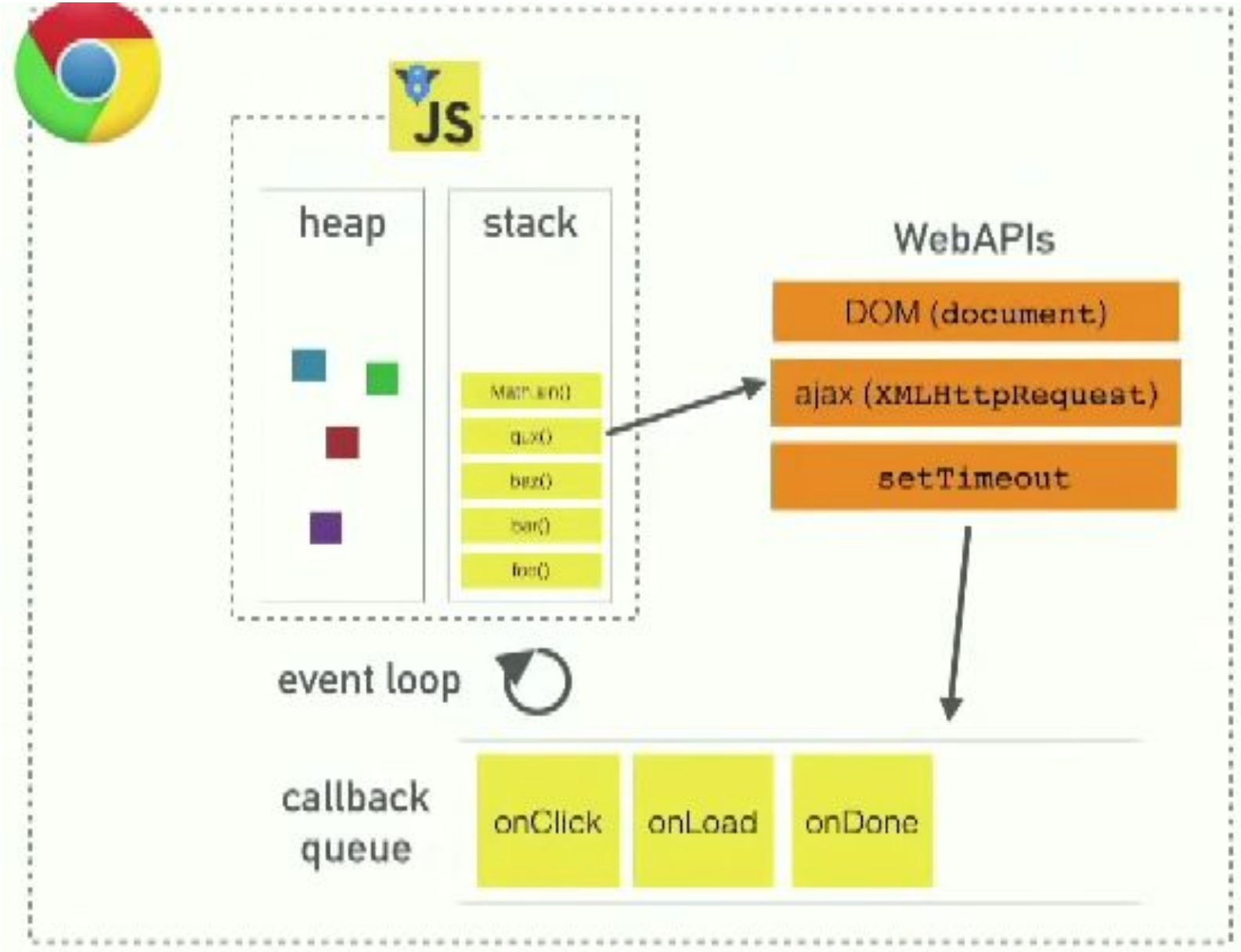


$digest loop watching



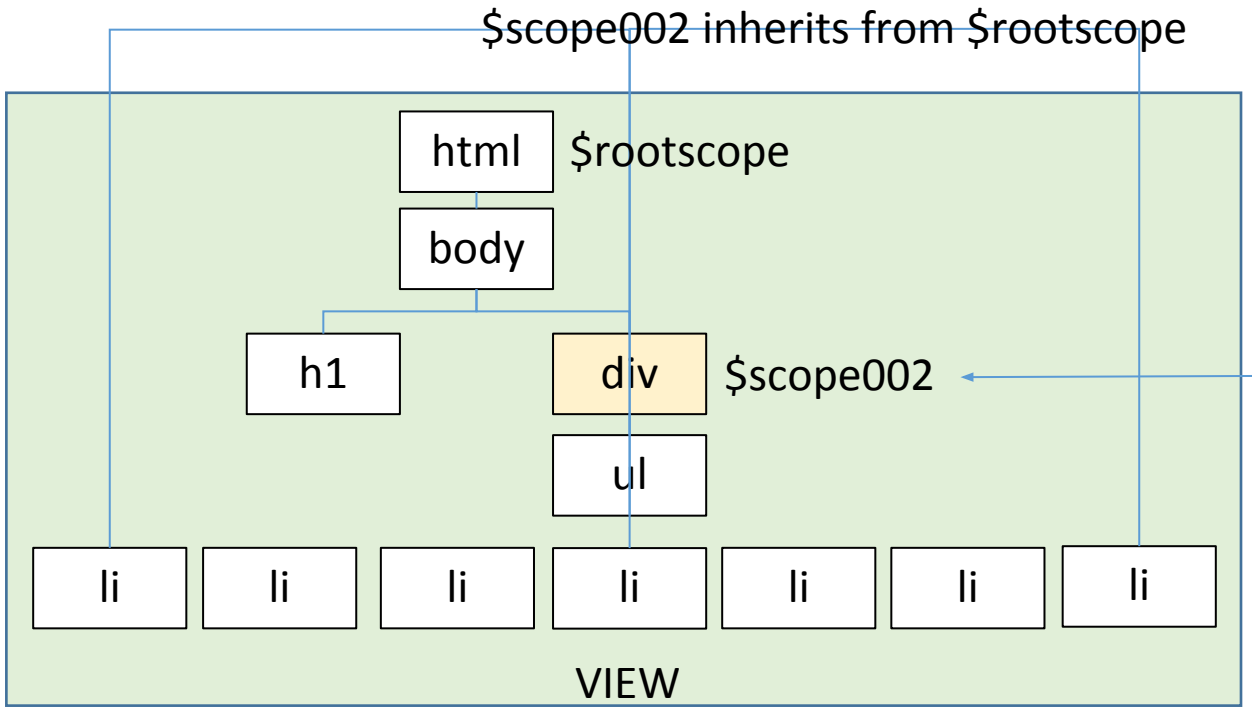
When the scope changes, the watchers will call a function. You can add your own custom watchers.

A word about the browser's native event loop



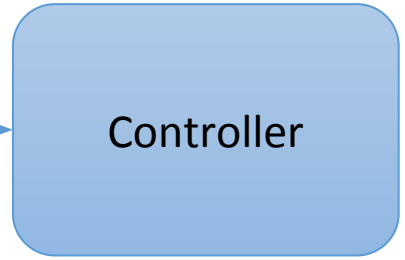
View rendering and webapi (e.g. async) calls are queued up and do not execute until the function call stack is empty!

Two-way data binding



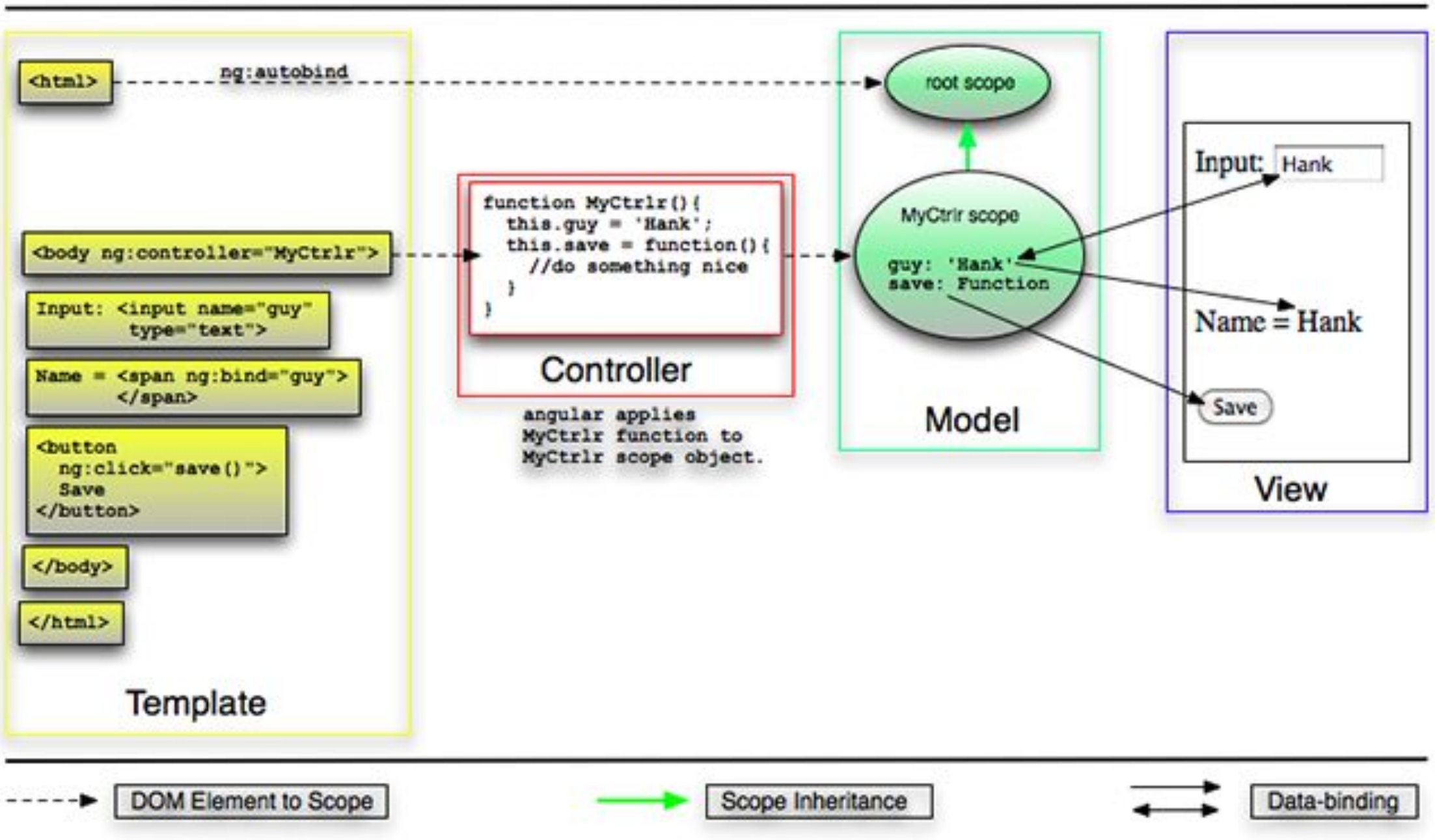
The view can manipulate the $scope using angular directives – custom or built-in

Any changes to the $scope
Is picked up by the $digest loop and reflected in both the view and the controller



The controller can manipulate the $scope

Two-way data bind example



A Module

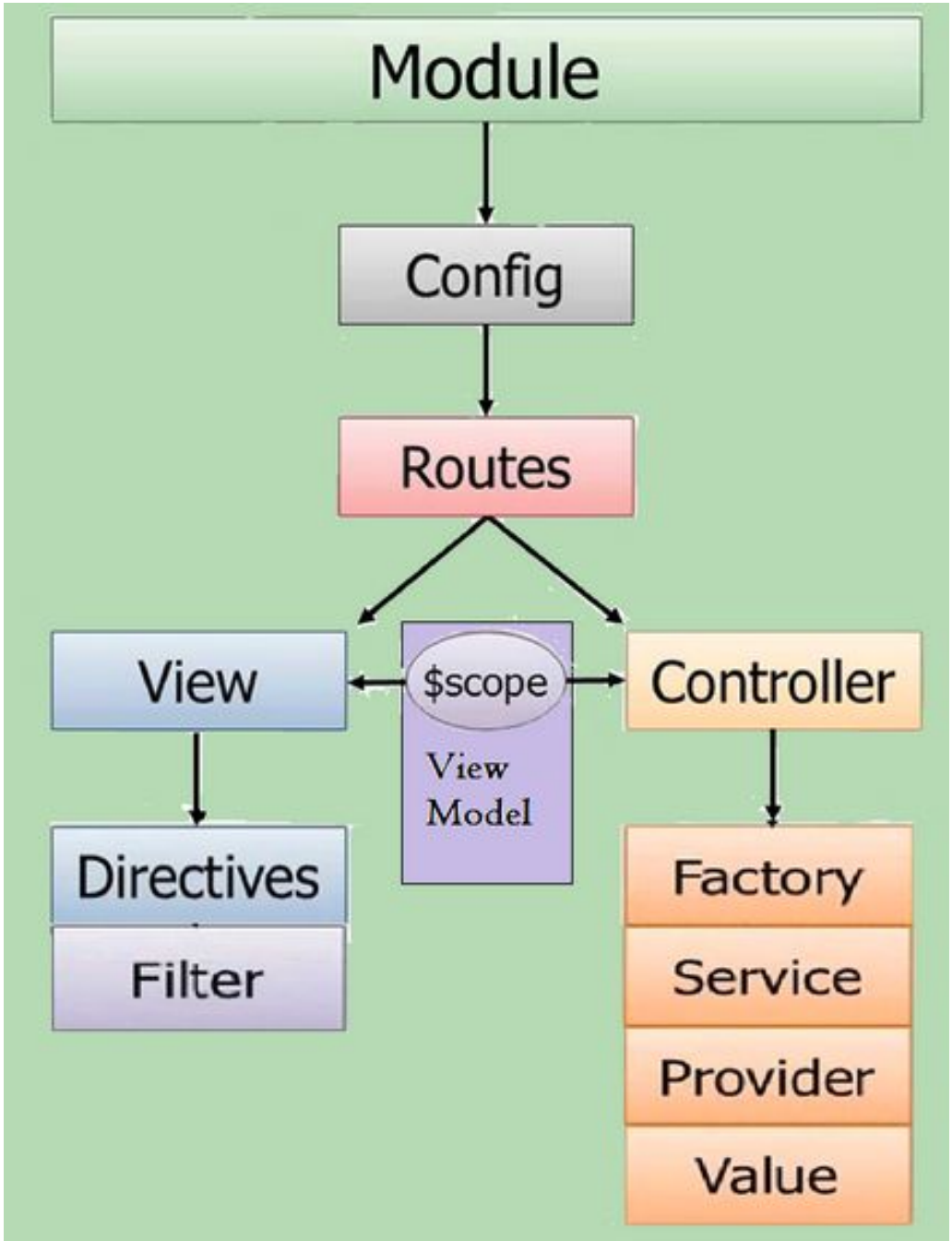
A module is a container for the different parts of the app – controllers, services, filters, directives, etc.

Note: A Word on Modules

In order for the injector to know how to create and wire together all of these objects, it needs a registry of "recipes". Each recipe has an identifier of the object and the description of how to create this object.

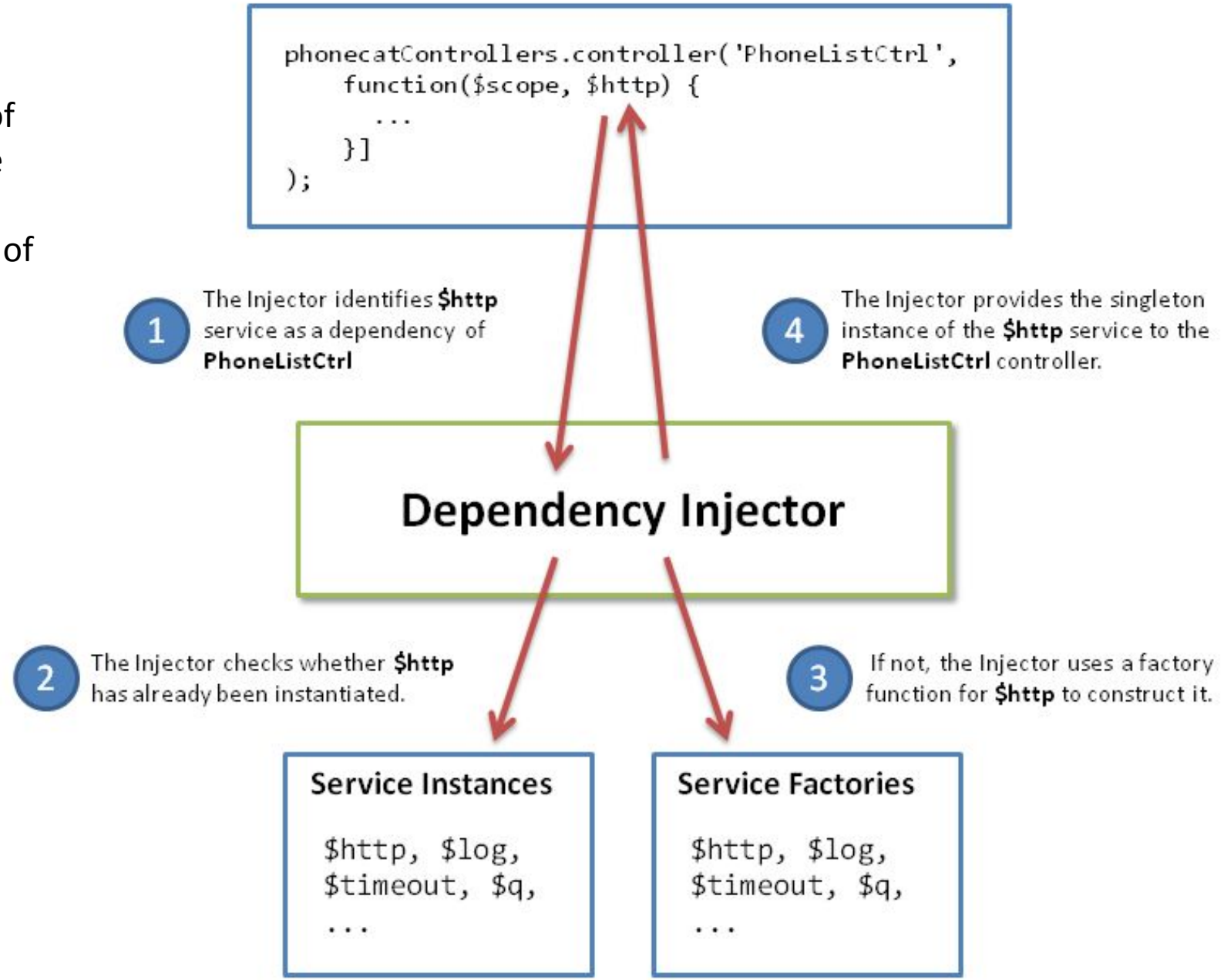
Each recipe belongs to an Angular module. An Angular module is a bag that holds one or more recipes. And since manually keeping track of module dependencies is no fun, a module can contain information about dependencies on other modules as well.

When an Angular application starts with a given application module, Angular creates a new instance of injector, which in turn creates a registry of recipes as a union of all recipes defined in the core "ng" module, application module and its dependencies. The injector then consults the recipe registry when it needs to create an object for your application.


Module
Config
Routes
View
$scope
Controller
View
Model
Directives
Filter
Factory
Service
Provider
Value

A Module – Dependency Injection

This is an example of implicit annotation of DI; it is the simplest way to get hold of the dependencies and it assumes that the function parameter names are the names of the dependencies. This is dangerous because of minification.



Dependency Injection Techniques

Inline Array Annotation

This is the preferred way to annotate application components. This is how the examples in the documentation are written.

For example:

```
someModule.controller('MyController', ['$scope', 'greeter', function($scope, greeter) {
  // ...
}]);
```

Here we pass an array whose elements consist of a list of strings (the names of the dependencies) followed by the function itself.

When using this type of annotation, take care to keep the annotation array in sync with the parameters in the function declaration.

`$inject` Property Annotation

To allow the minifiers to rename the function parameters and still be able to inject the right services, the function needs to be annotated with the `$inject` property. The `$inject` property is an array of service names to inject.

```
var MyController = function($scope, greeter) {
  // ...
}
MyController.$inject = ['$scope', 'greeter'];
someModule.controller('MyController', MyController);
```

Example Multiple Modules (using dependency Injection)

```
angular.module('xmpl.service', [])

  .value('greeter', {
    salutation: 'Hello',
    localize: function(localization) {
      this.salutation = localization.salutation;
    },
    greet: function(name) {
      return this.salutation + ' ' + name + '!';
    }
  })

  .value('user', {
    load: function(name) {
      this.name = name;
    }
  });

angular.module('xmpl.directive', []);

angular.module('xmpl.filter', []);

angular.module('xmpl', ['xmpl.service', 'xmpl.directive', 'xmpl.filter'])

  .run(function(greeter, user) {
    // This is effectively part of the main method initialization code
    greeter.localize({
      salutation: 'Bonjour'
    });
    user.load('World');
  })

  .controller('XmplController', function($scope, greeter, user){
    $scope.greeting = greeter.greet(user.name);
  });
```

Services

Providers

Improve this Doc

Each web application you build is composed of objects that collaborate to get stuff done. These objects need to be instantiated and wired together for the app to work. In Angular apps most of these objects are instantiated and wired together automatically by the injector service.

The injector creates two types of objects, **services** and **specialized objects**.

Services are objects whose API is defined by the developer writing the service.

Specialized objects conform to a specific Angular framework API. These objects are one of controllers, directives, filters or animations.

The injector needs to know how to create these objects. You tell it by registering a "recipe" for creating your object with the injector. There are five recipe types.

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.

Let's take a look at the different scenarios for creating and using services via various recipe types. We'll start with the simplest case possible where various places in your code need a shared string and we'll accomplish this via Value recipe.

Example

```
<!DOCTYPE html>
<html>
  <body ng-app="formExample">   <!-- bootstrap angular -->
    <div ng-controller="ExampleController">
      <form novalidate class="simple-form">
        Name: <input type="text" ng-model="user.name" /><br />
        E-mail: <input type="email" ng-model="user.email" /><br />
        Gender: <input type="radio" ng-model="user.gender" value="male" />male
        <input type="radio" ng-model="user.gender" value="female" />female<br />
        <input type="button" ng-click="reset()" value="Reset" />
        <input type="submit" ng-click="update(user)" value="Save" />

        <div ng-show="isTravis"><br/>IS TRAVIS</div>
      </form>
      <pre>user = {{user | json}}</pre>
      <pre>master = {{master | json}}</pre>
    </div>

    <script src="https://code.jquery.com/jquery-1.11.3.min.js"></script>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.0/angular.js"></script>
    <script src="script.js"></script>

    <script>
      angular.module('formExample', [])
        .controller('ExampleController', ['$scope', function($scope) {
          $scope.master = {};

          $scope.update = function(user) {
            $scope.master = angular.copy(user);
            $scope.isTravis = $scope.user.name === 'travis';
          };

          $scope.reset = function() {
            $scope.user = angular.copy($scope.master);
          };

          $scope.reset();
        }]);
    </script>
  </body>
</html>
```

Name: travis
E-mail:
Gender: male female
Reset Save

IS TRAVIS

```
user = {
  "name": "travis",
  "gender": "male"
}

master = {
  "name": "travis",
  "gender": "male"
}
```