Practical Development of Web Applications with JavaScript and AngularJS

Unit 3. AngularJS in Depth. Dependency Injection. Scopes.

AngularJS Dependency Injection

AngularJS Dependency Injection

- The Dependency Injection design pattern is discussed here

- DI helps to avoid explicitly creating object instances - a framework container creates and inject them into your code.

- DI Injects objects, doesn’t help with deferred modules loading.

- Angular registers and injects objects **by name**

AngularJS DI: Injecting Dependencies

1. Short form, doesn’t work after minifying:

```
angular.module('auction')
    .controller('SearchController', function ($scope) {
          //...
    });
```

Renamed while minifying

$ prefix reserved for AngularJS services

2. Long form, works well in production:

```
var SearchController = function ($scope) {
    //..
};
SearchController['$inject'] = ['$scope'];
angular.module('auction').controller('SearchController', SearchController);
```

Constructor function’s attribute (*not* on the instance)

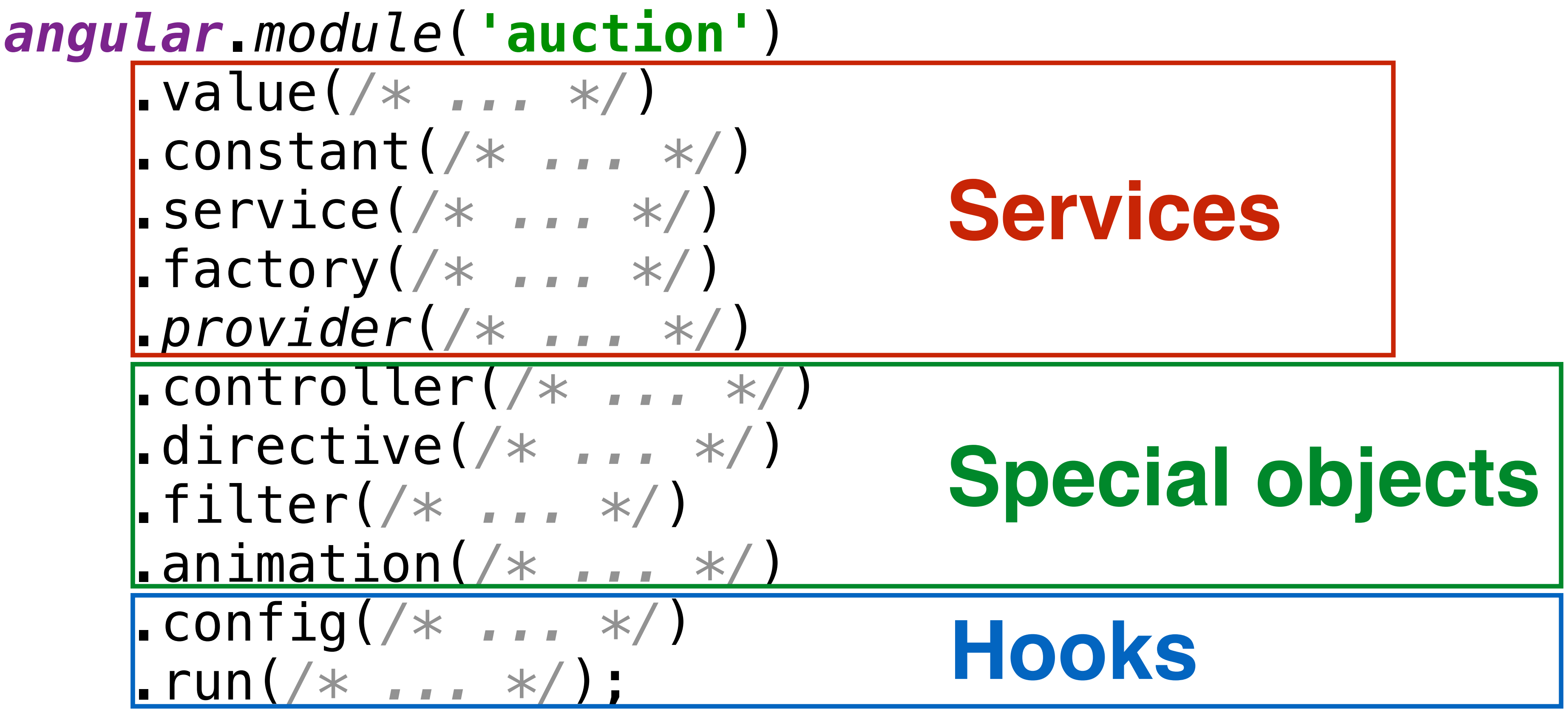
3. The same as 2, but using inline annotation form

```
angular.module('auction')
    .controller('SearchController', ['$scope', function ($scope) {
          //...
    }]);
```

The order matters

AngularJS DI: Registering Dependencies

- Most of dependencies differ by type of instantiating registered object
- Every registered object belongs to a single module
- All registered services are **singletons**
- Two phases: **configuration** (constant, provider, config) and **run** (all other)



```
angular.module('auction')
    .value(/* ... */)
    .constant(/* ... */)
    .service(/* ... */)
    .factory(/* ... */)
    .provider(/* ... */)
    .controller(/* ... */)
    .directive(/* ... */)
    .filter(/* ... */)
    .animation(/* ... */)
    .config(/* ... */)
    .run(/* ... */);
```

Services

Special objects

Hooks

DI: Hooks

AngularJS DI: config()

- Use to configure providers at configuration phase
- Can have only provider and constant dependencies

```
angular.module('auction', [])
    .config(['AuthenticationServiceProvider',
         function (authProvider) {
            authProvider.authType = 'forms';
         }]);
```

AngularJS DI: run()

- Runs after $injector is created, at the beginning of run phase

- Use for application initialization logic, e.g. global events binding, auto-login, geo-location.

```
angular.module('auction').run(['GeoService', function (geoService) {
    geoService.determineLocation();
}]);
```

Walkthrough 1

Update the page title when the route is changed
(follow the instructions in walkthrough_1_guide.html)

Revisiting Scopes: Events

- Scope can fire 2 types of events:
  - **$emit(name, args)** - bubbling event, goes through the hierarchy of parent scopes up to the rootScope
  - **$broadcast(name, args)** - (a.k.a. *capture* or *tunnelling*) propagates event down through the entire hierarchy of child scopes
- Scope can listen to events - **$on(name, listener)**
- **$broadcast()** and **$emit()** are syncronous

Walkthrough 1: Steps

1. In this walkthrough we will refactor the app in order to update page’s title (displayed in the browser’s tab) every time user navigates to a different page (i.e. routing event is successfully completed).

2. Import IntelliJ IDEA module `unit3` provided in the handouts.

3. Use detailed instruction provided in `walkthrough_1_guide.html` file.

DI: Services

AngularJS DI: value()

- Registers a static value.
- Available in the *run* phase

This name is used to register a value in Angular’s DI container. It can be used for injection in other places.

```
// JavaScript version
angular.module('auction')
    .value('securityToken', '0123456789')
    .controller('LoginController', ['securityToken', function (token) {
        //...
    }]);
```

Order matters, names can be different

AngularJS DI: constant()

- Similar to value(), but available in the *configuration* phase

```
angular.module('auction')
    .constant('locales', ['en-US', 'fr-CA'])
    .config([‘LocalizationServiceProvider', ‘locales',
     function (provider, locales) {
        provider.setSupportedLocales(locales);
      }]);
```

AngularJS DI: service()

- The object is instantiated with **new**

- Registered object must be a constructor function

- Example:

```
angular.module('auction')
    .service('AuthenticationService', function () {
        //...
    });
```

AngularJS DI: factory()

- A factory must be a function that will be **invoked** to get an instance of the service:

- Use factories to hide private computations:

```
angular.module('auction')
    .factory('CacheFactory', function () {
        var cache = {};
        return {
            add: function (key, value) {},
            getByKey: function (key) {}
        };
    });
```

Protected, can be accessed only by the factory

AngularJS DI: factory()

- Use to return a constructor function and repeatedly create new instances:

```
angular.module('auction')
    .factory('ProductModel', function () {
        return function (id, price) {
            this.id = id;
            this.price = price;
        }
     });
// Usage:
angular.module('auction')
    .controller('SearchController',
        function (ProductModel) {
            var product = new ProductModel();
        });
```

Use new to create instances

AngularJS DI: provider()

- Similar to a factory, but allows configuring provider on the application startup:

```
angular.module('auction')
    .provider('AuthenticationService', function () {
        this.authType;
        this.$get = function () {
            if (authType === 'basic') return new BasicAuthenticationService();
            if (authType === 'forms') return new FormsAuthenticationService();
            return new BasicAuthenticationService();
        };
    });

angular.module('auction', [])
    .config(['AuthenticationServiceProvider',
         function (authProvider) {
            authProvider.authType = 'forms';
         }]);

angular.module('auction')
    .controller('LoginController', ['AuthenticationService', function (authService) {
        authService.login();
    }]);
```

Configurable property

Notice name changes

Available on application startup

- E.g. $routeProvider allows configuring supported URLs

AngularJS DI: provider()

- Other factory methods are just syntactic sugar implemented on top of provider:

```
provider.service = function(name, Class) {
    provider.provide(name, function() {
        this.$get = function($injector) {
            return $injector.instantiate(Class);
        };
    });
}

provider.factory = function(name, factory) {
    provider.provide(name, function() {
        this.$get = function($injector) {
            return $injector.invoke(factory);
        };
    });
}

provider.value = function(name, value) {
    provider.factory(name, function() {
        return value;
    });
};
```

DI: Special Objects

AngularJS DI: controller()

- Registered objects available for **ngController** and **routing**.
- A controller **must** be a constructor function (i.e. instantiated using **new**)
- Unlike services, controllers are **not singletons**

```
// JavaScript version
angular.module('auction')
    .controller('SearchController', ['$scope', function ($scope) {
        //...
    }]);
```

AngularJS DI: directive()

- Uses factory() underneath
- Registers a special AngularJS object - directive
- Can have dependencies

Function name is not required, but is convenient for debugging - use names instead of anonymous functions in stack trace

```
angular.module('auction').directive('languageSwitcher',
    ['locales', function languageSwitcherDirectiveFactory(locales) {
        // directive definition object, mandatory AngularJS API
        return {
            restrict: 'E',
            link: function(scope, element) {
                element.text('Choose language: ' + locales.join(', '));
            }
        }
    }]);
```

```
<language-switcher></language-switcher>
```

AngularJS DI: filter()

- Uses factory() underneath
- Registers a special AngularJS object - filter
- Can have dependencies

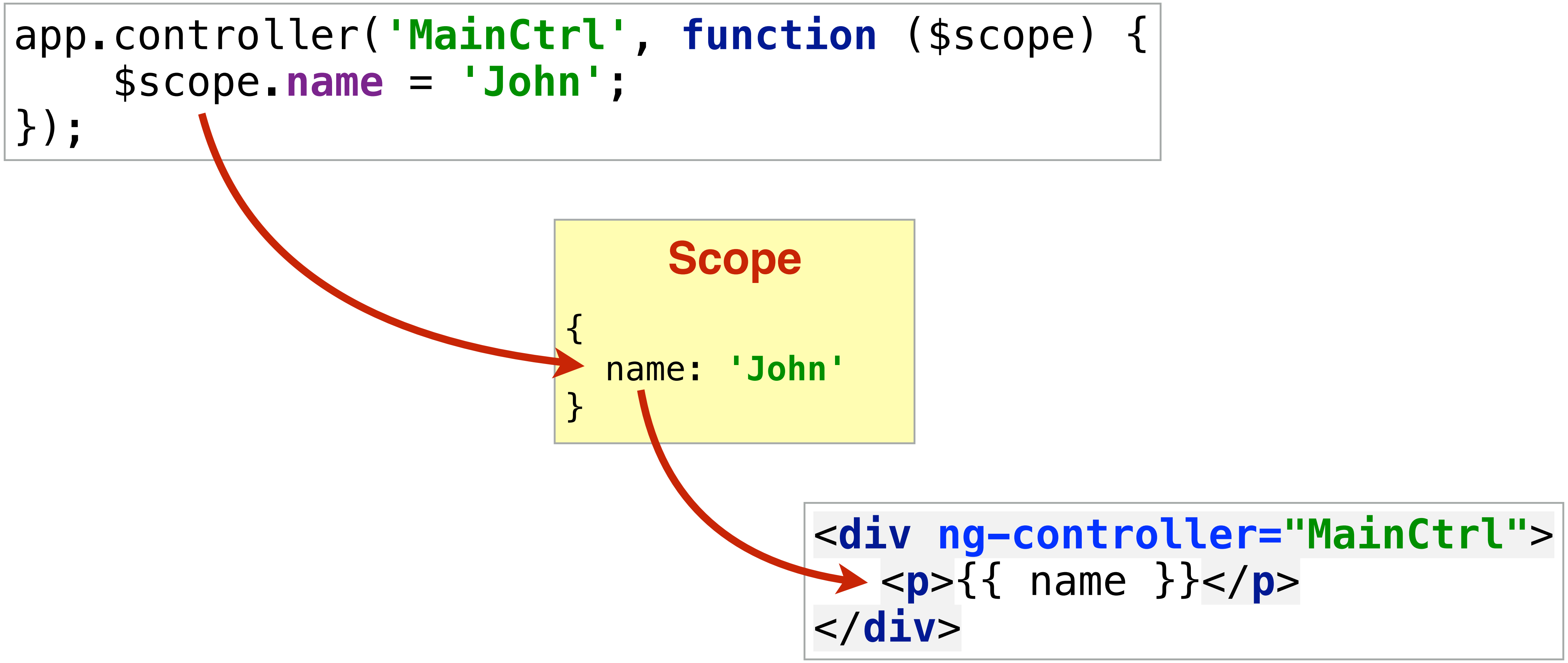
Returns a function that invoked each time the filter is applied

```
angular.module('auction').filter('join', function joinFilterFactory() {
    return function joinFilter(array, separator) {
        return array.join(separator);
    };
});

<p>{{ model.supportedLocales | join:', ' }}</p>
```

AngularJS Scopes

What is Scope?

Scope is a JavaScript object that keeps application models available as the data binding source on views.



```
app.controller('MainCtrl', function ($scope) {
    $scope.name = 'John';
});
```

Scope

```
{
  name: 'John'
}
```

```
<div ng-controller="MainCtrl">
  <p>{{ name }}</p>
</div>
```

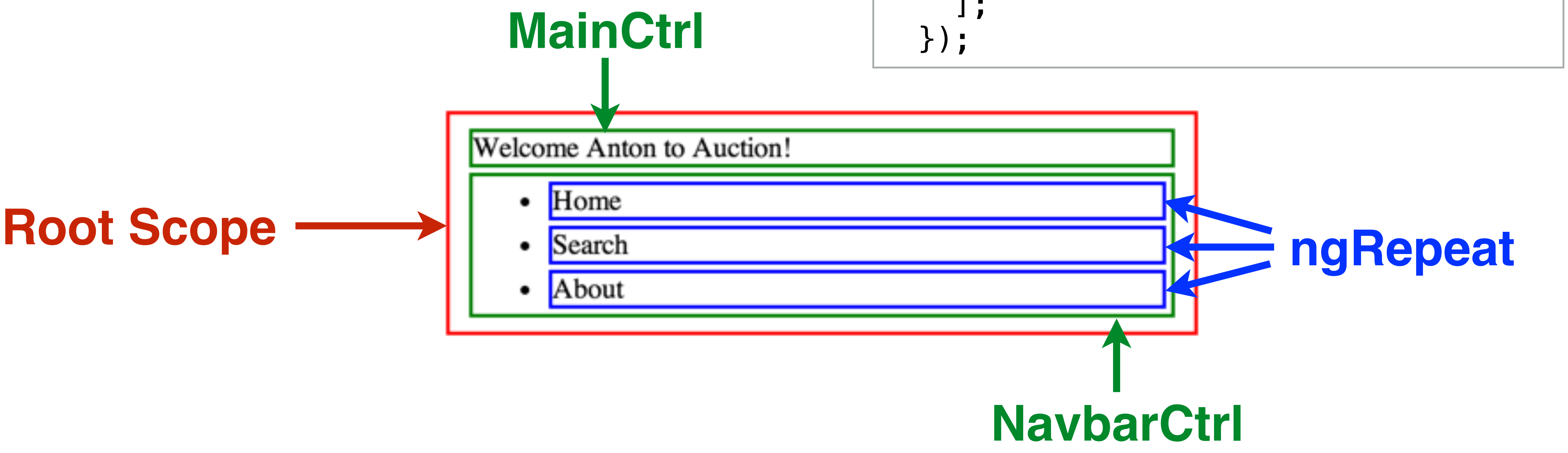
Scope Hierarchies

- Each application has only one **rootScope**
- Directives can create **child scopes** (e.g. ng-controller, ng-repeat)
- Child scopes **prototypically** inherit from their parents
- Directives can create **isolated scopes** (more on this later in this unit)

Scope Hierarchy Example

```
<div ng-controller="MainCtrl">
  Welcome {{ currentUser }} to {{ appName }}!
</div>
<ul ng-controller="NavbarCtrl">
  <li ng-repeat="item in menuItems">
    {{ item }}
  </li>
</ul>
```

```
app.controller('MainCtrl',
  function ($scope, $rootScope) {
    $scope.appName = 'Auction';
    $rootScope.currentUser = 'Anton';
  });

app.controller('NavbarCtrl',
  function ($scope) {
    $scope.menuItems = [
      'Home',
      'Search',
      'About'
    ];
  });
```



Welcome Anton to Auction!

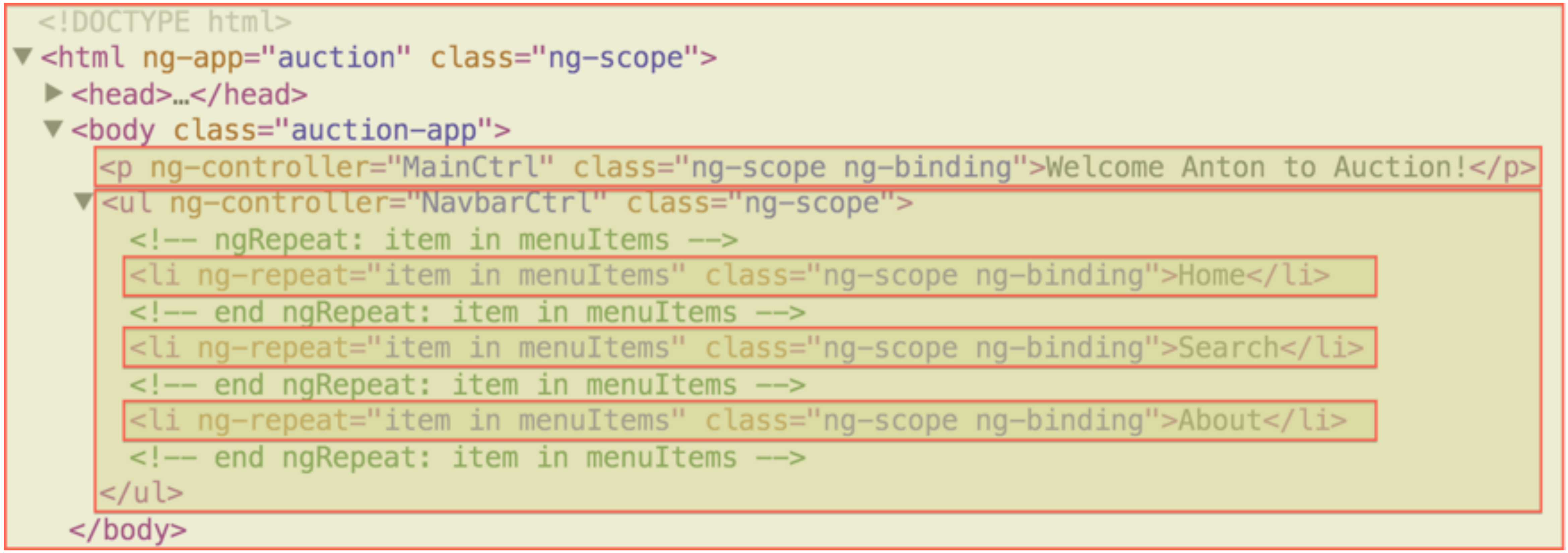
- Home
- Search
- About

Scope Hierarchy Example

- A Scopes hierarchy mimics the DOM structure

- To get the scope for any element (for **debugging only**):

```
angular.element(domEl).scope();
```

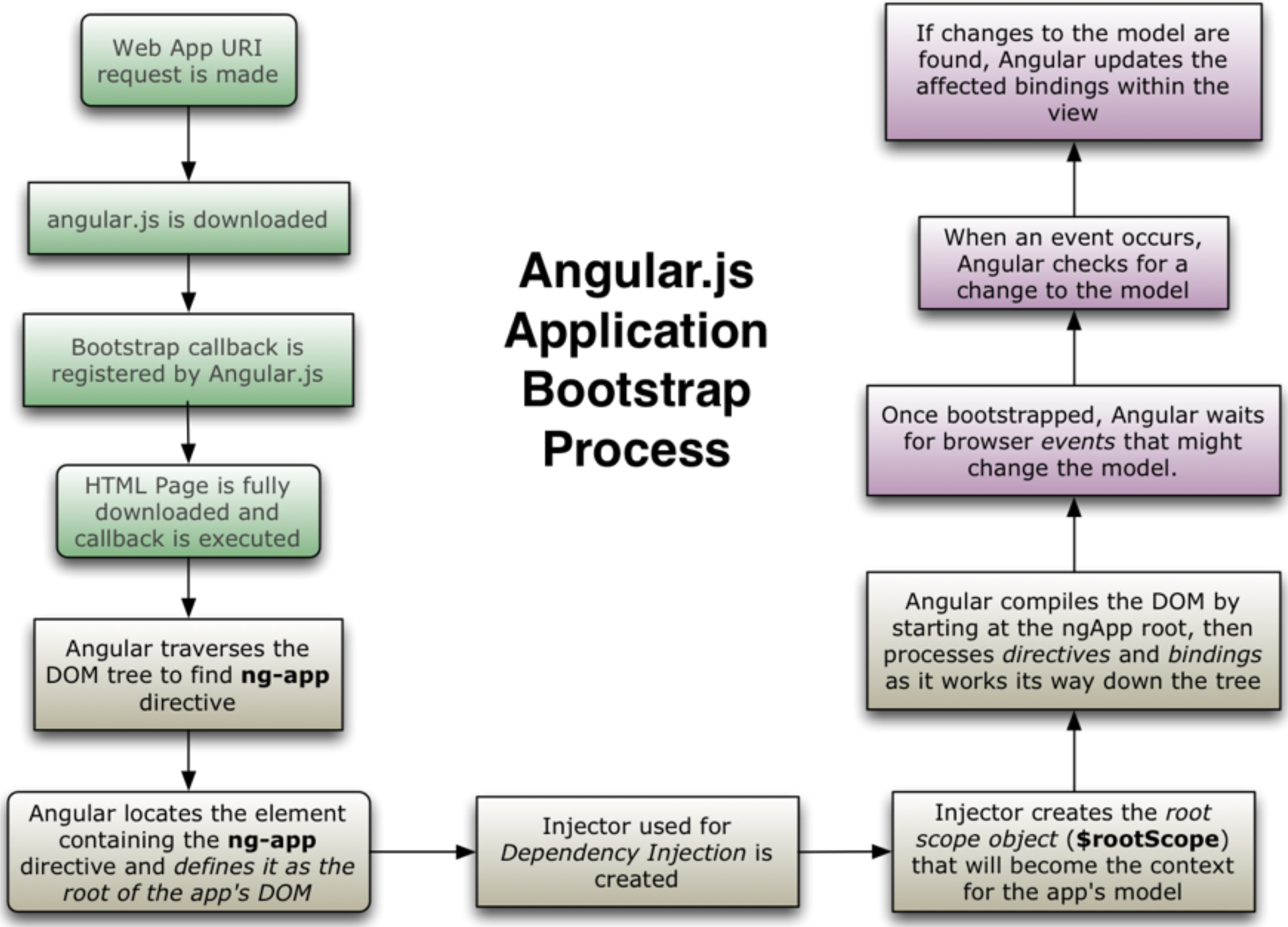


```
 <!DOCTYPE html>
▼<html ng-app="auction" class="ng-scope">
  ▶<head>…</head>
  ▼<body class="auction-app">
      <p ng-controller="MainCtrl" class="ng-scope ng-binding">Welcome Anton to Auction!</p>
    ▼<ul ng-controller="NavbarCtrl" class="ng-scope">
        <!-- ngRepeat: item in menuItems -->
        <li ng-repeat="item in menuItems" class="ng-scope ng-binding">Home</li>
        <!-- end ngRepeat: item in menuItems -->
        <li ng-repeat="item in menuItems" class="ng-scope ng-binding">Search</li>
        <!-- end ngRepeat: item in menuItems -->
        <li ng-repeat="item in menuItems" class="ng-scope ng-binding">About</li>
        <!-- end ngRepeat: item in menuItems -->
      </ul>
    </body>
```

How Scopes Work

- To make data-binding work AngularJS needs to:
  - observe the model changes → modify DOM
  - observe the DOM changes → modify models

Angular.js Application Bootstrap Process



Web App URI request is made

angular.js is downloaded

Bootstrap callback is registered by Angular.js

HTML Page is fully downloaded and callback is executed

Angular traverses the DOM tree to find **ng-app** directive

Angular locates the element containing the **ng-app** directive and *defines it as the root of the app's DOM*

Injector used for *Dependency Injection* is created

Injector creates the *root scope object* (**$rootScope**) that will become the context for the app's model

Angular compiles the DOM by starting at the ngApp root, then processes *directives* and *bindings* as it works its way down the tree

Once bootstrapped, Angular waits for browser *events* that might change the model.

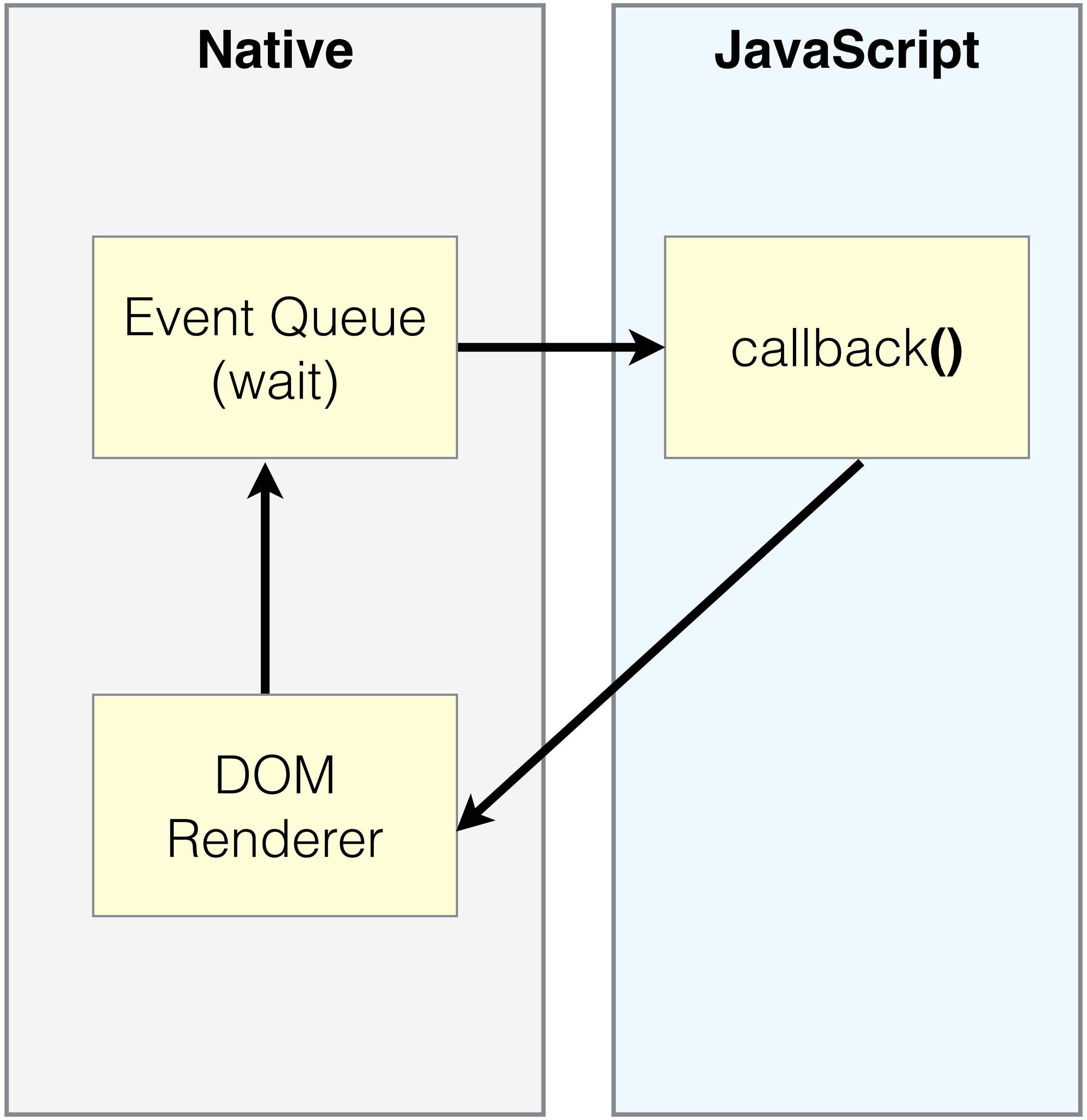
When an event occurs, Angular checks for a change to the model

If changes to the model are found, Angular updates the affected bindings within the view

http://www.codeproject.com/Articles/799241/AngularJS-The-next-BIG-thing

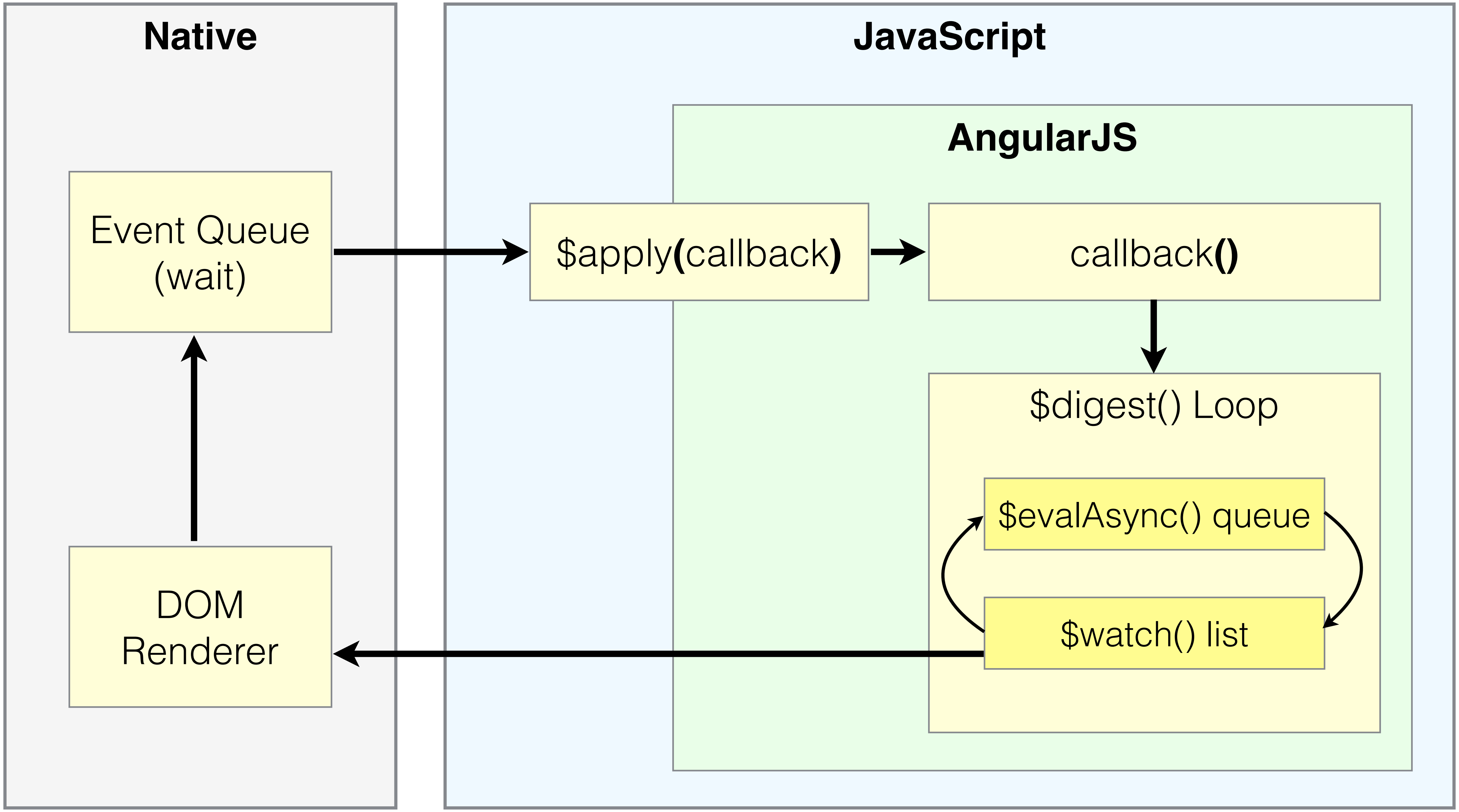
How Scopes Work

Native browser's event loop

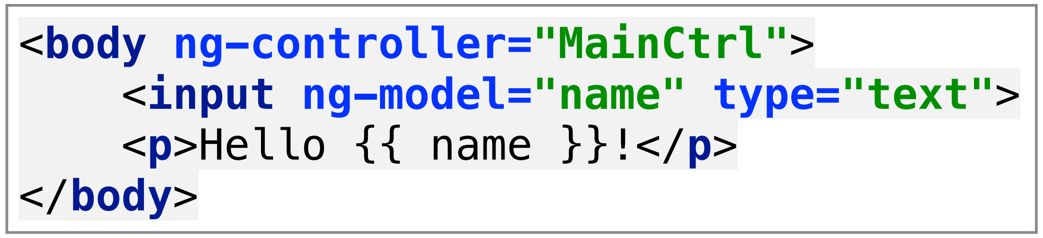


How Scopes Work

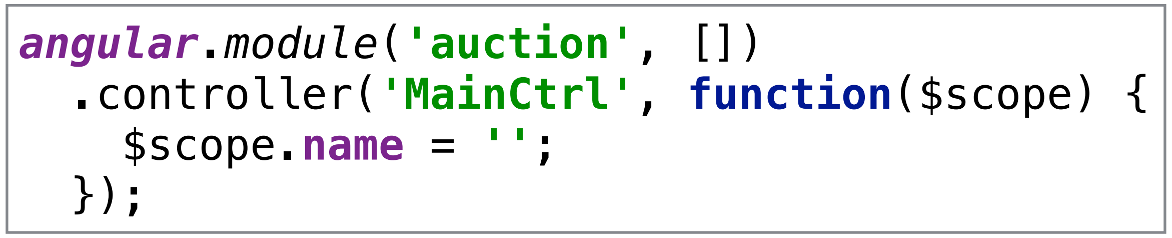
AngularJS modified event loop



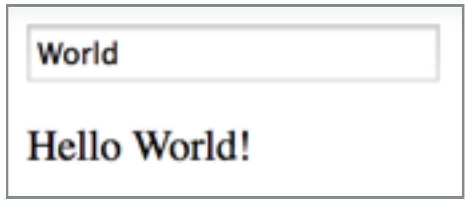
How Scopes Work: Example



```
<body ng-controller="MainCtrl">
    <input ng-model="name" type="text">
    <p>Hello {{ name }}!</p>
</body>
```



```
angular.module('auction', [])
  .controller('MainCtrl', function($scope) {
    $scope.name = '';
  });
```



World

Hello World!

1. App launches, passes *configuration* phase and enters *run* phase.
2. **ng-model** and **input** directives set up *keydown* listener on the **<input>** element
3. AngularJS compiles HTML and sets up **$watch** on **{{ name }}** changes
4. App enters browser's event loop
5. Pressing 'W' (of world) causes the browser to fire a *keydown* event on the **<input>**
6. **input** directive captures the change and calls **scope.$apply()**. Execution enters AngularJS modified event loop.
7. Inside **scope.$apply()** input directive calls **ngModelController.$setViewValue()** which updates internal state of view value and applies new value to **name** property. No **$digest** at this time.
8. **scope.$apply()** finishes the execution and the **$digest** loop begins.
9. **$watch** list detects a change on name property and notifies interpolation responsible for **{{ name }}** expression, which in turn updates DOM.
10. Execution exists AngularJS event loop, exits the keydown event and the JavaScript execution context.
11. The browser re-renders the view with updated text.

AngularJS Directives

Directives

- Attach behaviour to the DOM elements
- Can have visual and non-visual effect (*ng-controller* vs *ng-repeat*)
- Address two problems:
  - UI decomposition
  - Reusable components

How They Look Like

- Can be represented in several forms:
    - HTML element’s attribute: `ng-app`, `data-ng-app`
    - HTML element’s: `<auction-navbar>`
    - CSS classes `<div class=“auction-navbar”>`

Creating Custom Directives

```
angular.module('auction')
  .directive('auctionNavbar', function () {
    return {
      scope: true,
      restrict: 'E',
      templateUrl: 'views/partial/navbar.html'
    };
  });
```

Available as auction-navbar in HTML

New child scope is created for directive. Default - `false`.

Must be used as HTML element: `<auction-navbar>`

Path to the HTML template (partial view)

A Restrict Property

- Determines how to use a custom directive in HTML
- Can be one of the following:

```
'A' – <span auction–navbar></span>
'E' – <auction–navbar></auction–navbar>
'C' – <span class="auction–navbar"></span>
'M' – <!–– directive: auction–navbar ––>
```

Walkthrough 2

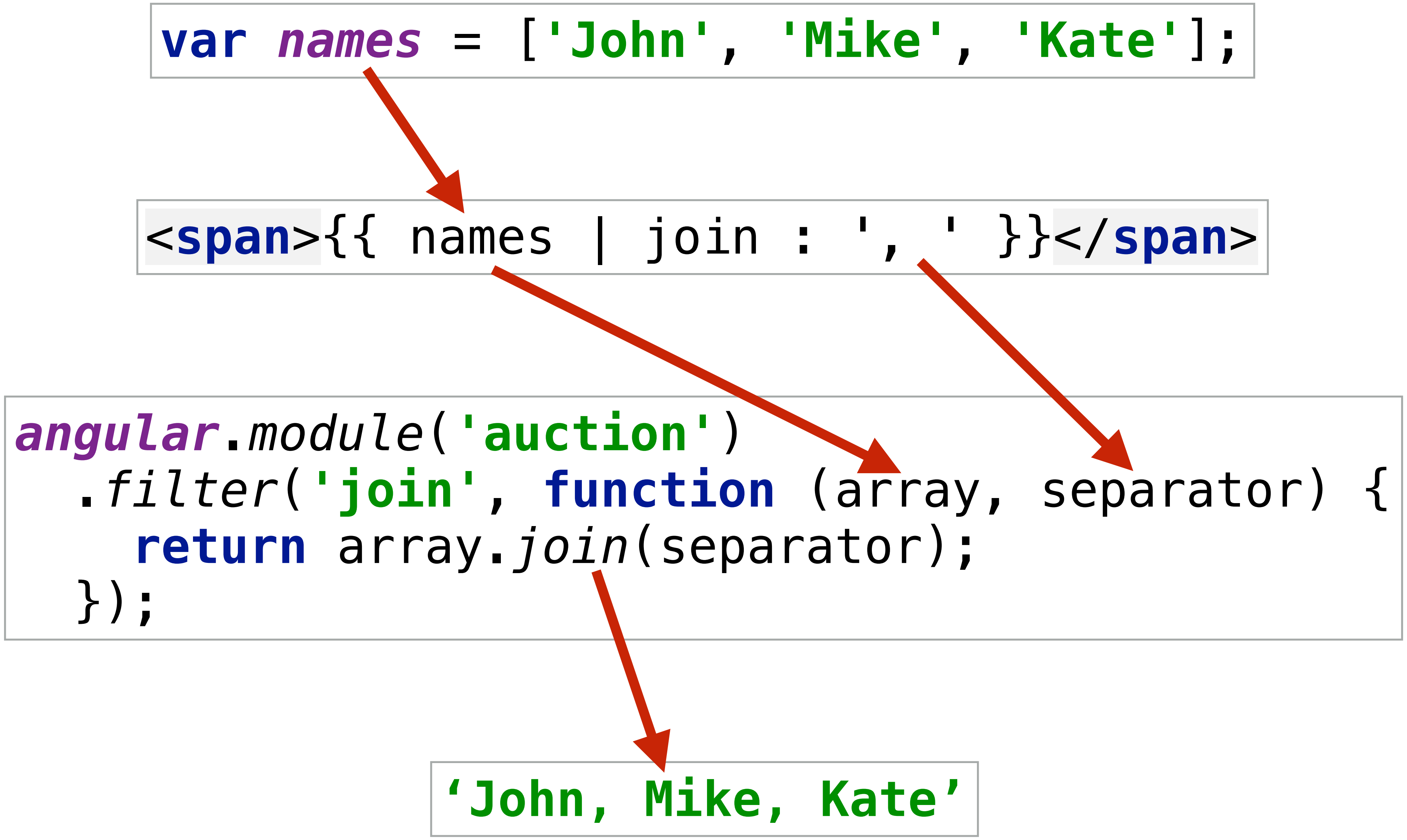
Decomposing the Auction app UI using directives
(follow the instructions in walkthrough_2_guide.html)

AngularJS Filters

Filter Features

- Transforms format of an expression value
- Can be used in HTML and directly invoked from code.
- Take at least one parameter - the value to transform.
- Can take arbitrary number of parameters.

Filter Example



```
var names = ['John', 'Mike', 'Kate'];
```

```
<span>{{ names | join : ', ' }}</span>
```

```
angular.module('auction')
  .filter('join', function (array, separator) {
    return array.join(separator);
  });
```

```
'John, Mike, Kate'
```

Revisiting Routing

Route parameters

1. Define a named placeholder

```
$routeProvider.when(‘/product/:id’, {
  templateUrl: 'views/search.html',
  controller: 'SearchCtrl'
});
```

Names should match

2. Substitute a placeholder in a template

```
<a href="#/product/{{ productId }}">Show Product</a>
<a ng-href="#/product/{{ productId }}">Show Product</a>
```

3. Access a parameter in a controller

```
angular.module('auction')
  .controller('ProductCtrl', function ($routeParams) {
    var productId = $routeParams.id;
  });
```

Promises

- A promise is an object that wraps a value that will be available *later* on as the result of an asynchronous operation.

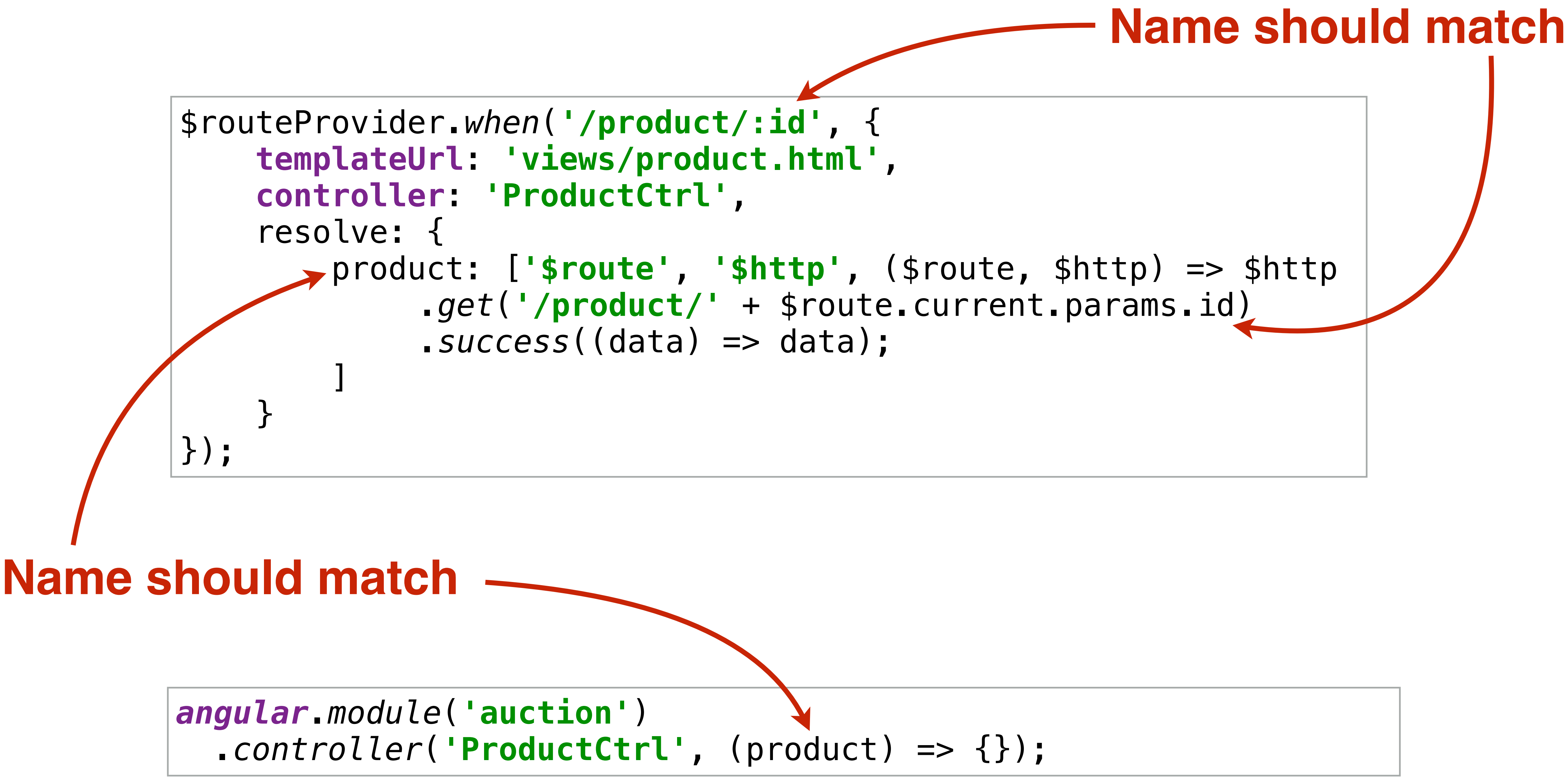
- Represented in AngularJS as **$q** service.

```
getFeatured() {
  var deferredProducts = this.$q.defer();

  this.$http.get('data/featured.json')
    .success((data) => deferredProducts.resolve(data.items))
    .error(() => deferredProducts.reject());

  return deferredProducts.promise;
}
```

Route’s Dependencies

- A route can define dependencies it must obtain before navigating to the view.
- Dependencies are defined using route’s **resolve** property.
- Dependencies will be injected into the target controller.
- If a dependency is promise, the $route service will wait until the promise is either resolved or rejected.

The *resolve* Example



Name should match

```
$routeProvider.when('/product/:id', {
    templateUrl: 'views/product.html',
    controller: 'ProductCtrl',
    resolve: {
        product: ['$route', '$http', ($route, $http) => $http
             .get('/product/' + $route.current.params.id)
             .success((data) => data);
        ]
    }
});
```

Name should match

```
angular.module('auction')
  .controller('ProductCtrl', (product) => {});
```

controllerAs

Use to automatically publish controller to scope:

```
// app.js
angular.module('auction', ['ngRoute'])
  .config(['$routeProvider', function ($routeProvider) {
    $routeProvider
      .when('/', {
        templateUrl: 'views/home.html',
        controller: 'HomeController',
        controllerAs: 'ctrl'
      });
    }]);
}());

// MainController.js
var HomeController = function (productService) {
    var _this = this;
    _this.products = [];
};
```

Additional Resources

- Understanding Scopes
- AngularJS Scopes
- AngularJS Directives

The Next Project Review

This project is about adding Product Details page. Use directory `homework3` from the handouts as the starting point.

1. Create `ProductDetailsController` that will handle user interactions on the Product Details page. Controller's constructor function should expect one parameter to be injected - `product` object. It will be resolved and provided by `resolve` function registered for the route.

2. Create `app/views/product.html` file and add HTML markup that will implement UI as shown in the `Single Item.png` mockup provided in the handouts. **Do not** implement a full-fledged image gallery with product's thumbnails, just add a static markup. The `Find More` button shouldn't display a Search Form, we will implement it in next homework.

3. Add `getProductById(productId)` method to the `ProductService`. The method should internally re-use one of the existing methods: `getFeaturedProducts()` or `find()`). Then a filtered collection of received products by product ID provided as an argument. If a product is found it should be returned to the call site, otherwise reject the promise.

4. In the `app.js` file add routing configuration to the new Product Details page. The route's path should contain the `:productId` parameter.

5. Add a `resolve` object for the route to pre-fetch the product from the server before the page is rendered. The same way we did on the "The *`resolve`* *Example*" slide, but do not call $http service - directly inject and re-use `ProductService.getProductById` method implemented in step 3.

6. Run the `grunt build` command. It will generate the `dist` subdirectory in the root directory of your app. The content of the `dist` can be deploy at GitHub Pages or any Web server.

7. Review a proposed solution at http://farata.github.io/modernwebdev-showcase/homework3/dist/#/