AngularJS styleguide

Opinionated AngularJS styleguide
for teams
by @toddmotto

Table of Contents

AngularJS styleguide

Opinionated AngularJS styleguide for teams by @toddmotto

A standardised approach for developing AngularJS applications in teams. This styleguide touches on concepts, syntax, conventions and is based on my experience writing about, talking about, and building Angular applications.

Community

John Papa and I have discussed in-depth styling patterns for Angular and as such have both released separate styleguides. Thanks to those discussions, I've learned some great tips from John that have helped shape this guide. We've both created our own take on a styleguide. I urge you to check his out to compare thoughts.

> See the original article that sparked this off

Modules

- **Definitions**: Declare modules without a variable using the setter and getter syntax

```
// avoid
var app = angular.module('app', []);
app.controller();
app.factory();

// recommended
angular
  .module('app', [])
  .controller()
  .factory();
```

- Note: Using `angular.module('app', []);` sets a module, whereas `angular.module('app');` gets the module. Only set once and get for all other instances.

- **Methods**: Pass functions into module methods rather than assign as a callback

```
// avoid
angular
  .module('app', [])
  .controller('MainCtrl', function MainCtrl () {

  })
  .service('SomeService', function SomeService () {

  });

// recommended
function MainCtrl () {

}
function SomeService () {

}
angular
  .module('app', [])
  .controller('MainCtrl', MainCtrl)
  .service('SomeService', SomeService);
```

- This aids with readability and reduces the volume of code "wrapped" inside the Angular framework

- **IIFE scoping**: To avoid polluting the global scope with our function declarations that get passed into Angular, ensure build tasks wrap the concatenated files inside an IIFE

```
(function () {

  angular
    .module('app', []);

  // MainCtrl.js
  function MainCtrl () {
```

```
  }

  angular
    .module('app')
    .controller('MainCtrl', MainCtrl);

  // SomeService.js
  function SomeService () {

  }

  angular
    .module('app')
    .service('SomeService', SomeService);

  // ...

})();
```

Controllers

- **controllerAs syntax**: Controllers are classes, so use the `controllerAs` syntax at all times

```
<!-- avoid -->
<div ng-controller="MainCtrl">
  {{ someObject }}
</div>

<!-- recommended -->
<div ng-controller="MainCtrl as vm">
  {{ vm.someObject }}
</div>
```

- In the DOM we get a variable per controller, which aids nested controller methods, avoiding any `$parent` calls
- The `controllerAs` syntax uses `this` inside controllers, which gets bound to `$scope`

```
// avoid
function MainCtrl ($scope) {
  $scope.someObject = {};
  $scope.doSomething = function () {

  };
}

// recommended
function MainCtrl () {
  this.someObject = {};
  this.doSomething = function () {

  };
}
```

- Only use `$scope` in `controllerAs` when necessary; for example, publishing and subscribing events using `$emit`, `$broadcast`, `$on` or `$watch`. Try to limit the use of these, however, and treat `$scope` as a special use case
- **Inheritance**: Use prototypal inheritance when extending controller classes

```
function BaseCtrl () {
  this.doSomething = function () {

  };
}
BaseCtrl.prototype.someObject = {};
BaseCtrl.prototype.sharedSomething = function () {

};

AnotherCtrl.prototype = Object.create(BaseCtrl.prototype);
```

```
function AnotherCtrl () {
  this.anotherSomething = function () {

  };
}
```

- Use `Object.create` with a polyfill for browser support
- **controllerAs 'vm'**: Capture the `this` context of the Controller using `vm`, standing for `ViewModel`

```
// avoid
function MainCtrl () {
  this.doSomething = function () {

  };
}

// recommended
function MainCtrl (SomeService) {
  var vm = this;
  vm.doSomething = SomeService.doSomething;
}
```

Why? : Function context changes the `this` value, use it to avoid `.bind()` calls and scoping issues

- **Presentational logic only (MVVM)**: Presentational logic only inside a controller, avoid Business logic (delegate to Services)

```
// avoid
function MainCtrl () {

  var vm = this;

  $http
    .get('/users')
    .success(function (response) {
      vm.users = response;
    });

  vm.removeUser = function (user, index) {
    $http
      .delete('/user/' + user.id)
      .then(function (response) {
        vm.users.splice(index, 1);
      });
  };

}

// recommended
function MainCtrl (UserService) {

  var vm = this;

  UserService
    .getUsers()
    .then(function (response) {
```

```
        vm.users = response;
      });

    vm.removeUser = function (user, index) {
      UserService
        .removeUser(user)
        .then(function (response) {
          vm.users.splice(index, 1);
        });
    };

}
```

Why? : Controllers should fetch Model data from Services, avoiding any Business logic. Controllers should act as a ViewModel and control the data flowing between the Model and the View presentational layer. Business logic in Controllers makes testing Services impossible.

Services and Factory

- All Angular Services are singletons, using `.service()` or `.factory()` differs the way Objects are created.

  Services: act as a `constructor` function and are instantiated with the `new` keyword. Use `this` for public methods and variables

  ```
  function SomeService () {
    this.someMethod = function () {

    };
  }
  angular
    .module('app')
    .service('SomeService', SomeService);
  ```

  Factory: Business logic or provider modules, return an Object or closure

- Always return a host Object instead of the revealing Module pattern due to the way Object references are bound and updated

  ```
  function AnotherService () {
    var AnotherService = {};
    AnotherService.someValue = '';
    AnotherService.someMethod = function () {

    };
    return AnotherService;
  }
  angular
    .module('app')
    .factory('AnotherService', AnotherService);
  ```

  Why? : Primitive values cannot update alone using the revealing module pattern

Back to top

Directives

- **Declaration restrictions**: Only use `custom element` and `custom attribute` methods for declaring your Directives (`{ restrict: 'EA' }`) depending on the Directive's role

  ```
  <!-- avoid -->

  <!-- directive: my-directive -->
  <div class="my-directive"></div>

  <!-- recommended -->

  <my-directive></my-directive>
  ```

```
<div my-directive></div>
```

- Comment and class name declarations are confusing and should be avoided. Comments do not play nicely with older versions of IE. Using an attribute is the safest method for browser coverage.

- **Templating**: Use `Array.join('')` for clean templating

```
// avoid
function someDirective () {
  return {
    template: '<div class="some-directive">' +
      '<h1>My directive</h1>' +
    '</div>'
  };
}

// recommended
function someDirective () {
  return {
    template: [
      '<div class="some-directive">',
        '<h1>My directive</h1>',
      '</div>'
    ].join('')
  };
}
```

  Why? : Improves readability as code can be indented properly, it also avoids the `+` operator which is less clean and can lead to errors if used incorrectly to split lines

- **DOM manipulation**: Takes place only inside Directives, never a controller/service

```
// avoid
function UploadCtrl () {
  $('.dragzone').on('dragend', function () {
    // handle drop functionality
  });
}
angular
  .module('app')
  .controller('UploadCtrl', UploadCtrl);

// recommended
function dragUpload () {
  return {
    restrict: 'EA',
    link: function (scope, element, attrs) {
      element.on('dragend', function () {
        // handle drop functionality
      });
    }
  };
}
angular
  .module('app')
  .directive('dragUpload', dragUpload);
```

- **Naming conventions**: Never `ng-*` prefix custom directives, they might conflict future native directives

```
// avoid
// <div ng-upload></div>
function ngUpload () {
  return {};
}
angular
  .module('app')
  .directive('ngUpload', ngUpload);

// recommended
// <div drag-upload></div>
function dragUpload () {
  return {};
}
angular
  .module('app')
  .directive('dragUpload', dragUpload);
```

- Directives and Filters are the *only* providers that have the first letter as lowercase; this is due to strict naming conventions in Directives. Angular hyphenates `camelCase`, so `dragUpload` will become `<div drag-upload></div>` when used on an element.

- **controllerAs**: Use the `controllerAs` syntax inside Directives as well

```
// avoid
function dragUpload () {
  return {
    controller: function ($scope) {

    }
  };
}
angular
  .module('app')
  .directive('dragUpload', dragUpload);

// recommended
function dragUpload () {
  return {
    controllerAs: 'vm',
    controller: function () {

    }
  };
}
angular
  .module('app')
  .directive('dragUpload', dragUpload);
```

Directives

- **Declaration restrictions**: Only use `custom element` and `custom attribute` methods for declaring your Directives (`{ restrict: 'EA' }`) depending on the Directive's role

```
<!-- avoid -->

<!-- directive: my-directive -->
<div class="my-directive"></div>

<!-- recommended -->

<my-directive></my-directive>
<div my-directive></div>
```

- Comment and class name declarations are confusing and should be avoided. Comments do not play nicely with older versions of IE. Using an attribute is the safest method for browser coverage.

- **Templating**: Use `Array.join('')` for clean templating

```
// avoid
function someDirective () {
  return {
    template: '<div class="some-directive">' +
      '<h1>My directive</h1>' +
    '</div>'
  };
}

// recommended
function someDirective () {
  return {
    template: [
      '<div class="some-directive">',
        '<h1>My directive</h1>',
      '</div>'
    ].join('')
  };
}
```

Why? : Improves readability as code can be indented properly, it also avoids the `+` operator which is less clean and can lead to errors if used incorrectly to split lines

- **DOM manipulation**: Takes place only inside Directives, never a controller/service

```
// avoid
function UploadCtrl () {
  $('.dragzone').on('dragend', function () {
    // handle drop functionality
  });
}
angular
  .module('app')
```

```
  .controller('UploadCtrl', UploadCtrl);

// recommended
function dragUpload () {
  return {
    restrict: 'EA',
    link: function (scope, element, attrs) {
      element.on('dragend', function () {
        // handle drop functionality
      });
    }
  };
}
angular
  .module('app')
  .directive('dragUpload', dragUpload);
```

- **Naming conventions**: Never `ng-*` prefix custom directives, they might conflict future native directives

```
// avoid
// <div ng-upload></div>
function ngUpload () {
  return {};
}
angular
  .module('app')
  .directive('ngUpload', ngUpload);

// recommended
// <div drag-upload></div>
function dragUpload () {
  return {};
}
angular
  .module('app')
  .directive('dragUpload', dragUpload);
```

- Directives and Filters are the *only* providers that have the first letter as lowercase; this is due to strict naming conventions in Directives. Angular hyphenates `camelCase`, so `dragUpload` will become `<div drag-upload></div>` when used on an element.

- **controllerAs**: Use the `controllerAs` syntax inside Directives as well

```
// avoid
function dragUpload () {
  return {
    controller: function ($scope) {

    }
  };
}
angular
  .module('app')
  .directive('dragUpload', dragUpload);

// recommended
```

```
function dragUpload () {
  return {
    controllerAs: 'vm',
    controller: function () {

    }
  };
}
angular
  .module('app')
  .directive('dragUpload', dragUpload);
```

Filters

- **Global filters**: Create global filters using `angular.filter()` only. Never use local filters inside Controllers/Services

```
// avoid
function SomeCtrl () {
  this.startsWithLetterA = function (items) {
    return items.filter(function (item) {
      return /^a/i.test(item.name);
    });
  };
}
angular
  .module('app')
  .controller('SomeCtrl', SomeCtrl);

// recommended
function startsWithLetterA () {
  return function (items) {
    return items.filter(function (item) {
      return /^a/i.test(item.name);
    });
  };
}
angular
  .module('app')
  .filter('startsWithLetterA', startsWithLetterA);
```

- This enhances testing and reusability

Back to top

Routing resolves

- **Promises**: Resolve Controller dependencies in the `$routeProvider` (or `$stateProvider` for `ui-router`), not the Controller itself

```
// avoid
function MainCtrl (SomeService) {
  var _this = this;
  // unresolved
  _this.something;
  // resolved asynchronously
  SomeService.doSomething().then(function (response) {
    _this.something = response;
  });
}
angular
  .module('app')
  .controller('MainCtrl', MainCtrl);

// recommended
function config ($routeProvider) {
  $routeProvider
```

```
  .when('/', {
    templateUrl: 'views/main.html',
    resolve: {
      // resolve here
    }
  });
}
angular
  .module('app')
  .config(config);
```

- **Controller.resolve property**: Never bind logic to the router itself. Reference a `resolve` property for each Controller to couple the logic

```
// avoid
function MainCtrl (SomeService) {
  this.something = SomeService.something;
}

function config ($routeProvider) {
  $routeProvider
  .when('/', {
    templateUrl: 'views/main.html',
    controllerAs: 'vm',
    controller: 'MainCtrl'
    resolve: {
      doSomething: function () {
        return SomeService.doSomething();
      }
    }
  });
}

// recommended
function MainCtrl (SomeService) {
  this.something = SomeService.something;
}

MainCtrl.resolve = {
  doSomething: function (SomeService) {
    return SomeService.doSomething();
  }
};

function config ($routeProvider) {
  $routeProvider
  .when('/', {
    templateUrl: 'views/main.html',
    controllerAs: 'vm',
    controller: 'MainCtrl'
    resolve: MainCtrl.resolve
  });
}
```

- This keeps resolve dependencies inside the same file as the Controller and the router free from logic

Back to top

Publish and subscribe events

- **$scope**: Use the `$emit` and `$broadcast` methods to trigger events to direct relationship scopes only

```
// up the $scope
$scope.$emit('customEvent', data);

// down the $scope
$scope.$broadcast('customEvent', data);
```

- **$rootScope**: Use only `$emit` as an application-wide event bus and remember to unbind listeners

```
// all $rootScope.$on listeners
$rootScope.$emit('customEvent', data);
```

- Hint: Because the `$rootScope` is never destroyed, `$rootScope.$on` listeners aren't either, unlike `$scope.$on` listeners and will always persist, so they need destroying when the relevant `$scope` fires the `$destroy` event

```
// call the closure
var unbind = $rootScope.$on('customEvent'[, callback]);
$scope.$on('$destroy', unbind);
```

- For multiple `$rootScope` listeners, use an Object literal and loop each one on the `$destroy` event to unbind all automatically

```
var rootListeners = {
  'customEvent1': $rootScope.$on('customEvent1'[, callback]),
  'customEvent2': $rootScope.$on('customEvent2'[, callback]),
  'customEvent3': $rootScope.$on('customEvent3'[, callback])
};
for (var unbind in rootListeners) {
  $scope.$on('$destroy', rootListeners[unbind]);
}
```

Back to top

Performance

- **One-time binding syntax**: In newer versions of Angular (v1.3.0-beta.10+), use the one-time binding syntax `{{ ::value }}` where it makes sense

```
// avoid
<h1>{{ vm.title }}</h1>

// recommended
<h1>{{ ::vm.title }}</h1>
```

Why? : Binding once removes the watcher from the scope's `$$watchers` array after the `undefined` variable becomes resolved, thus improving performance in each dirty-check

- **Consider $scope.$digest**: Use `$scope.$digest` over `$scope.$apply` where it makes sense. Only child scopes will update

```
$scope.$digest();
```

Why? : `$scope.$apply` will call `$rootScope.$digest`, which causes the entire application `$$watchers` to dirty-check again. Using `$scope.$digest` will dirty check current and child scopes from the initiated `$scope`

Back to top

Angular wrapper references

- **$document and $window**: Use `$document` and `$window` at all times to aid testing and Angular references

```
// avoid
function dragUpload () {
  return {
    link: function ($scope, $element, $attrs) {
      document.addEventListener('click', function () {

      });
    }
  };
}

// recommended
function dragUpload () {
  return {
    link: function ($scope, $element, $attrs, $document) {
      $document.addEventListener('click', function () {

      });
    }
  };
}
```

- **$timeout and $interval**: Use `$timeout` and `$interval` over their native counterparts to keep Angular's two-way data binding up to date

```
// avoid
function dragUpload () {
  return {
    link: function ($scope, $element, $attrs) {
      setTimeout(function () {
        //
      }, 1000);
    }
  };
```

```
}

// recommended
function dragUpload ($timeout) {
  return {
    link: function ($scope, $element, $attrs) {
      $timeout(function () {
        //
      }, 1000);
    }
  };
}
```

Back to top

Comment standards

- **jsDoc**: Use jsDoc syntax to document function names, description, params and returns

```
/**
 * @name SomeService
 * @desc Main application Controller
 */
function SomeService (SomeService) {

  /**
   * @name doSomething
   * @desc Does something awesome
   * @param {Number} x - First number to do something with
   * @param {Number} y - Second number to do something with
   * @returns {Number}
   */
  this.doSomething = function (x, y) {
    return x * y;
  };

}
angular
  .module('app')
  .service('SomeService', SomeService);
```

Back to top

Minification and annotation

- **ng-annotate**: Use ng-annotate for Gulp as `ng-min` is deprecated, and comment functions that need automated dependency injection using `/** @ngInject */`

```
/**
 * @ngInject
 */
function MainCtrl (SomeService) {
  this.doSomething = SomeService.doSomething;
}
angular
```

```
  .module('app')
  .controller('MainCtrl', MainCtrl);
```

- Which produces the following output with the `$inject` annotation

```
/**
 * @ngInject
 */
function MainCtrl (SomeService) {
  this.doSomething = SomeService.doSomething;
}
MainCtrl.$inject = ['SomeService'];
angular
  .module('app')
  .controller('MainCtrl', MainCtrl);
```

Routing resolves

- **Promises**: Resolve Controller dependencies in the `$routeProvider` (or `$stateProvider` for `ui-router`), not the Controller itself

```
// avoid
function MainCtrl (SomeService) {
  var _this = this;
  // unresolved
  _this.something;
  // resolved asynchronously
  SomeService.doSomething().then(function (response) {
    _this.something = response;
  });
}
angular
  .module('app')
  .controller('MainCtrl', MainCtrl);

// recommended
function config ($routeProvider) {
  $routeProvider
  .when('/', {
    templateUrl: 'views/main.html',
    resolve: {
      // resolve here
    }
  });
}
angular
  .module('app')
  .config(config);
```

- **Controller.resolve property**: Never bind logic to the router itself. Reference a `resolve` property for each Controller to couple the logic

```
// avoid
function MainCtrl (SomeService) {
  this.something = SomeService.something;
}

function config ($routeProvider) {
  $routeProvider
  .when('/', {
    templateUrl: 'views/main.html',
    controllerAs: 'vm',
    controller: 'MainCtrl'
    resolve: {
      doSomething: function () {
        return SomeService.doSomething();
      }
    }
  });
}

// recommended
```

```
function MainCtrl (SomeService) {
  this.something = SomeService.something;
}

MainCtrl.resolve = {
  doSomething: function (SomeService) {
    return SomeService.doSomething();
  }
};

function config ($routeProvider) {
  $routeProvider
  .when('/', {
    templateUrl: 'views/main.html',
    controllerAs: 'vm',
    controller: 'MainCtrl'
    resolve: MainCtrl.resolve
  });
}
```

- This keeps resolve dependencies inside the same file as the Controller and the router free from logic

Publish and subscribe events

- **$scope**: Use the `$emit` and `$broadcast` methods to trigger events to direct relationship scopes only

```
// up the $scope
$scope.$emit('customEvent', data);

// down the $scope
$scope.$broadcast('customEvent', data);
```

- **$rootScope**: Use only `$emit` as an application-wide event bus and remember to unbind listeners

```
// all $rootScope.$on listeners
$rootScope.$emit('customEvent', data);
```

- Hint: Because the `$rootScope` is never destroyed, `$rootScope.$on` listeners aren't either, unlike `$scope.$on` listeners and will always persist, so they need destroying when the relevant `$scope` fires the `$destroy` event

```
// call the closure
var unbind = $rootScope.$on('customEvent'[, callback]);
$scope.$on('$destroy', unbind);
```

- For multiple `$rootScope` listeners, use an Object literal and loop each one on the `$destroy` event to unbind all automatically

```
var rootListeners = {
  'customEvent1': $rootScope.$on('customEvent1'[, callback]),
  'customEvent2': $rootScope.$on('customEvent2'[, callback]),
  'customEvent3': $rootScope.$on('customEvent3'[, callback])
};
for (var unbind in rootListeners) {
  $scope.$on('$destroy', rootListeners[unbind]);
}
```

Performance

- **One-time binding syntax**: In newer versions of Angular (v1.3.0-beta.10+), use the one-time binding syntax `{{ ::value }}` where it makes sense

  ```
  // avoid
  <h1>{{ vm.title }}</h1>

  // recommended
  <h1>{{ ::vm.title }}</h1>
  ```

  Why? : Binding once removes the watcher from the scope's `$$watchers` array after the `undefined` variable becomes resolved, thus improving performance in each dirty-check

- **Consider $scope.$digest**: Use `$scope.$digest` over `$scope.$apply` where it makes sense. Only child scopes will update

  ```
  $scope.$digest();
  ```

  Why? : `$scope.$apply` will call `$rootScope.$digest`, which causes the entire application `$$watchers` to dirty-check again. Using `$scope.$digest` will dirty check current and child scopes from the initiated `$scope`

Angular wrapper references

- **$document and $window**: Use `$document` and `$window` at all times to aid testing and Angular references

```
// avoid
function dragUpload () {
  return {
    link: function ($scope, $element, $attrs) {
      document.addEventListener('click', function () {

      });
    }
  };
}

// recommended
function dragUpload () {
  return {
    link: function ($scope, $element, $attrs, $document) {
      $document.addEventListener('click', function () {

      });
    }
  };
}
```

- **$timeout and $interval**: Use `$timeout` and `$interval` over their native counterparts to keep Angular's two-way data binding up to date

```
// avoid
function dragUpload () {
  return {
    link: function ($scope, $element, $attrs) {
      setTimeout(function () {
        //
      }, 1000);
    }
  };
}

// recommended
function dragUpload ($timeout) {
  return {
    link: function ($scope, $element, $attrs) {
      $timeout(function () {
        //
      }, 1000);
    }
  };
}
```

Comment standards

- **jsDoc**: Use jsDoc syntax to document function names, description, params and returns

```
/**
 * @name SomeService
 * @desc Main application Controller
 */
function SomeService (SomeService) {

  /**
   * @name doSomething
   * @desc Does something awesome
   * @param {Number} x - First number to do something with
   * @param {Number} y - Second number to do something with
   * @returns {Number}
   */
  this.doSomething = function (x, y) {
    return x * y;
  };

}
angular
  .module('app')
  .service('SomeService', SomeService);
```

Minification and annotation

- **ng-annotate**: Use ng-annotate for Gulp as `ng-min` is deprecated, and comment functions that need automated dependency injection using `/** @ngInject */`

```
/**
 * @ngInject
 */
function MainCtrl (SomeService) {
  this.doSomething = SomeService.doSomething;
}
angular
  .module('app')
  .controller('MainCtrl', MainCtrl);
```

- Which produces the following output with the `$inject` annotation

```
/**
 * @ngInject
 */
function MainCtrl (SomeService) {
  this.doSomething = SomeService.doSomething;
}
MainCtrl.$inject = ['SomeService'];
angular
  .module('app')
  .controller('MainCtrl', MainCtrl);
```

Angular docs

For anything else, including API reference, check the Angular documentation.

License

(The MIT License)