**Controllers**

Use controllers to:

* Set up the initial state of the $scope object.
* Add behavior to the $scope object.

Do not use controllers to:

* Manipulate DOM — Controllers should contain only business logic. Putting any presentation logic into Controllers significantly affects its testability. AngularJS has databinding for most cases and directives to encapsulate manual DOM manipulation.
* Format input — Use AngularJS form controls instead.
* Filter output — Use AngularJS filters instead.
* Share code or state across controllers — Use AngularJS services instead.
* Manage the life-cycle of other components (for example, to create service instances).

These tags, areassociated with controllers, each one with one scope

The scope can be inheritated so the $scope of the controller of fogliodilavoro has inside the $scope of editor

NOT EXACTLY

These are directives associated with a component non a controller

<editor>

<fogliodilavoro></fogliodilavoro>

</editor>

In general, a Controller shouldn't try to do too much. It should contain only the business logic needed for a single view.

The most common way to keep Controllers slim is by encapsulating work that doesn't belong to controllers into services and then using these services in Controllers via dependency injection. This is discussed in the Dependency Injection and Services sections of this guide.

**Services**

AngularJS services are substitutable objects that are wired together using [dependency injection (DI)](https://code.angularjs.org/snapshot/docs/guide/di). You can use services to organize and share code across your app.

Each component dependent on a service gets a reference to the single instance generated by the service factory.

**Note:** Like other core AngularJS identifiers, built-in services always start with $ (e.g. $http, )

$window.alert(text); <- $window is a built in service

.

The **service factory function** generates the single object or function that represents the service to the rest of the application. The object or function returned by the service is injected into any component (controller, service, filter or directive) that specifies a dependency on the service.

Services can have their own dependencies. Just like declaring dependencies in a controller, you declare dependencies by specifying them in the service's factory function signature.

batchModule.factory('batchLog', ['$interval', '$log', function($interval, $log) {

var messageQueue = [];

function log() {

if (messageQueue.length) {

$log.log('batchLog messages: ', messageQueue);

messageQueue = [];

}

}

// start periodic checking

$interval(log, 50000);

return function(message) {

messageQueue.push(message);

}

}]);

**Factory Methods**

The way you define a directive, service, or filter is with a factory function. The factory methods are registered with modules. The recommended way of declaring factories is:

angular.module('myModule', [])

.factory('serviceId', ['depService', function(depService) {

// ... NUOVO SERVICE

}])

.directive('directiveName', ['depService', function(depService) {

// ... NUOVA DIRETTIVA (vedi dopo)

}])

.filter('filterName', ['depService', function(depService) {

// ... NUOVO FILTRO (vedi dopo)

}]);

**Scope**

[Scope](https://code.angularjs.org/snapshot/docs/api/ng/type/$rootScope.Scope) is an object that refers to the application model. It is an execution context for [expressions](https://code.angularjs.org/snapshot/docs/guide/expression). Scopes are arranged in hierarchical structure which mimic the DOM structure of the application. Scopes can watch [expressions](https://code.angularjs.org/snapshot/docs/guide/expression) and propagate events.

Scopes provide context against which [expressions](https://code.angularjs.org/snapshot/docs/guide/expression) are evaluated. For example {{username}} expression is meaningless, unless it is evaluated against a specific scope which defines the username property.

Scope is the glue between application controller and the view

During the template [linking](https://code.angularjs.org/snapshot/docs/guide/compiler) phase the [directives](https://code.angularjs.org/snapshot/docs/api/ng/provider/$compileProvider#directive) set up [$watch](https://code.angularjs.org/snapshot/docs/api/ng/type/$rootScope.Scope#$watch) expressions on the scope. The $watch allows the directives to be notified of property changes, which allows the directive to render the updated value to the DOM. (DONE AUTOMATICALLY)

You can think of the scope and its properties as the data which is used to render the view.

**Scopes Events**

Scopes can propagate events in similar fashion to DOM events. The event can be [broadcasted](https://code.angularjs.org/snapshot/docs/api/ng/type/$rootScope.Scope#$broadcast) to the scope children or [emitted](https://code.angularjs.org/snapshot/docs/api/ng/type/$rootScope.Scope#$emit) to scope parents.

$scope.$on('MyEvent', function() {

$scope.count++;

});

<button ng-click="$emit('MyEvent')">$emit('MyEvent')</button>

Scopes and controllers interact with each other in the following situations:

* Controllers use scopes to expose controller methods to templates (see [ng-controller](https://code.angularjs.org/snapshot/docs/api/ng/directive/ngController)).
* Controllers define methods (behavior) that can mutate the model (properties on the scope).
* Controllers may register [watches](https://code.angularjs.org/snapshot/docs/api/ng/type/$rootScope.Scope#$watch) on the model. These watches execute immediately after the controller behavior executes.

See the [ng-controller](https://code.angularjs.org/snapshot/docs/api/ng/directive/ngController) for more information.

**Model**

Model = scope of the controller

# Components

In AngularJS, a Component is a special kind of [directive](https://code.angularjs.org/snapshot/docs/guide/directive) (SEE DIRECTIVE DOWN HERE) that uses a simpler configuration which is suitable for a component-based application structure.

This makes it easier to write an app in a way that's similar to using Web Components or using the new Angular's style of application architecture.

Advantages of Components:

* simpler configuration than plain directives
* promote sane defaults and best practices
* optimized for component-based architecture
* writing component directives will make it easier to upgrade to Angular

When not to use Components:

* for directives that need to perform actions in compile and pre-link functions, because they aren't available
* when you need advanced directive definition options like priority, terminal, multi-element
* when you want a directive that is triggered by an attribute or CSS class, rather than an element

Components can be registered using the .component() method of an AngularJS module (returned by [angular.module()](https://code.angularjs.org/snapshot/docs/guide/module)). The method takes two arguments:

* The name of the Component (as string).
* The Component config object

function HeroDetailController() {

}

angular.module('heroApp').component('heroDetail', {

templateUrl: 'heroDetail.html',

controller: HeroDetailController,

bindings: {

hero: '='

}

});

HTML:

<hero-detail hero="ctrl.hero"></hero-detail>

## **Component-based application architecture**

As already mentioned, the component helper makes it easier to structure your application with a component-based architecture. But what makes a component beyond the options that the component helper has?

* **Components only control their own View and Data:** Components should never modify any data or DOM that is out of their own scope. Normally, in AngularJS it is possible to modify data anywhere in the application through scope inheritance and watches. This is practical, but can also lead to problems when it is not clear which part of the application is responsible for modifying the data. That is why component directives use an isolate scope, so a whole class of scope manipulation is not possible.
* **Components have a well-defined public API - Inputs and Outputs:** However, scope isolation only goes so far, because AngularJS uses two-way binding. So if you pass an object to a component like this - bindings: {item: '='}, and modify one of its properties, the change will be reflected in the parent component. For components however, only the component that owns the data should modify it, to make it easy to reason about what data is changed, and when. For that reason, components should follow a few simple conventions:
  + Inputs should be using < and @ bindings. The < symbol denotes [one-way bindings](https://code.angularjs.org/snapshot/docs/api/ng/service/$compile#-scope-) which are available since 1.5. The difference to = is that the bound properties in the component scope are not watched, which means if you assign a new value to the property in the component scope, it will not update the parent scope. Note however, that both parent and component scope reference the same object, so if you are changing object properties or array elements in the component, the parent will still reflect that change. The general rule should therefore be to never change an object or array property in the component scope. @ bindings can be used when the input is a string, especially when the value of the binding doesn't change.

bindings: {

hero: '<', //ONE WAY BINDING

comment: '@'

}

* + Outputs are realized with & bindings, which function as callbacks to component events.

bindings: {

onDelete: '&',

onUpdate: '&'

}

* + Instead of manipulating Input Data, the component calls the correct Output Event with the changed data. For a deletion, that means the component doesn't delete the hero itself, but sends it back to the owner component via the correct event.

<!-- note that we use kebab-case for bindings in the template as usual -->

<editable-field on-update="$ctrl.update('location', value)"></editable-field><br>

<button ng-click="$ctrl.onDelete({hero: $ctrl.hero})">Delete</button>

* + That way, the parent component can decide what to do with the event (e.g. delete an item or update the properties)

ctrl.deleteHero(hero) {

$http.delete(...).then(function() {

var idx = ctrl.list.indexOf(hero);

if (idx >= 0) {

ctrl.list.splice(idx, 1);

}

});

}

* **Components have a well-defined lifecycle** Each component can implement "lifecycle hooks". These are methods that will be called at certain points in the life of the component. The following hook methods can be implemented:
  + $onInit() - Called on each controller after all the controllers on an element have been constructed and had their bindings initialized (and before the pre & post linking functions for the directives on this element). This is a good place to put initialization code for your controller.
  + $onChanges(changesObj) - Called whenever one-way bindings are updated. The changesObj is a hash whose keys are the names of the bound properties that have changed, and the values are an object of the form{ currentValue, previousValue, isFirstChange() }. Use this hook to trigger updates within a component such as cloning the bound value to prevent accidental mutation of the outer value.
  + $doCheck() - Called on each turn of the digest cycle. Provides an opportunity to detect and act on changes. Any actions that you wish to take in response to the changes that you detect must be invoked from this hook; implementing this has no effect on when $onChanges is called. For example, this hook could be useful if you wish to perform a deep equality check, or to check a Date object, changes to which would not be detected by AngularJS's change detector and thus not trigger $onChanges. This hook is invoked with no arguments; if detecting changes, you must store the previous value(s) for comparison to the current values.
  + $onDestroy() - Called on a controller when its containing scope is destroyed. Use this hook for releasing external resources, watches and event handlers.
  + $postLink() - Called after this controller's element and its children have been linked. Similar to the post-link function this hook can be used to set up DOM event handlers and do direct DOM manipulation. Note that child elements that contain templateUrl directives will not have been compiled and linked since they are waiting for their template to load asynchronously and their own compilation and linking has been suspended until that occurs. This hook can be considered analogous to the ngAfterViewInit and ngAfterContentInit hooks in Angular. Since the compilation process is rather different in AngularJS there is no direct mapping and care should be taken when upgrading.

By implementing these methods, your component can hook into its lifecycle.

* **An application is a tree of components:** Ideally, the whole application should be a tree of components that implement clearly defined inputs and outputs, and minimize two-way data binding. That way, it's easier to predict when data changes and what the state of a component is.

## **Component-based Applications**

It is recommended to develop AngularJS applications as a hierarchy of Components. Each Component is an isolated part of the application, which is responsible for its own user interface and has a well defined programmatic interface to the Component that contains it. Take a look at the [component guide](https://code.angularjs.org/snapshot/docs/guide/component) for more information.

# COSE MINORI

# Templates

In AngularJS, templates are written with HTML that contains AngularJS-specific elements and attributes. AngularJS combines the template with information from the model and controller to render the dynamic view that a user sees in the browser.

These are the types of AngularJS elements and attributes you can use:

* [Directive](https://code.angularjs.org/snapshot/docs/guide/directive) — An attribute or element that augments an existing DOM element or represents a reusable DOM component.
* [Markup](https://code.angularjs.org/snapshot/docs/api/ng/service/$interpolate) — The double curly brace notation {{ }} to bind expressions to elements is built-in AngularJS markup.
* [Filter](https://code.angularjs.org/snapshot/docs/guide/filter) — Formats data for display.
* [Form controls](https://code.angularjs.org/snapshot/docs/guide/forms) — Validates user input.

# AngularJS Expressions

AngularJS expressions are JavaScript-like code snippets that are mainly placed in interpolation bindings such as <span title="{{ attrBinding }}">{{ textBinding }}</span>, but also used directly in directive attributes such as ng-click="functionExpression()".

expressions are evaluated against a [scope](https://code.angularjs.org/snapshot/docs/api/ng/type/$rootScope.Scope) object.

Attributes such as disabled are called boolean attributes. We cannot use normal attribute bindings with them, because if we put an AngularJS interpolation expression into such an attribute then the binding information would be lost, because the browser ignores the attribute value.

In the following example, the interpolation information would be ignored and the browser would simply interpret the attribute as present, meaning that the button would always be disabled.

Disabled: <input type="checkbox" ng-model="isDisabled" />

<button disabled="{{isDisabled}}">Disabled</button>

For this reason, AngularJS provides special ng-prefixed directives for the following boolean attributes: [disabled](https://code.angularjs.org/snapshot/docs/api/ng/directive/ngDisabled), [required](https://code.angularjs.org/snapshot/docs/api/ng/directive/ngRequired), [selected](https://code.angularjs.org/snapshot/docs/api/ng/directive/ngSelected),[checked](https://code.angularjs.org/snapshot/docs/api/ng/directive/ngChecked), [readOnly](https://code.angularjs.org/snapshot/docs/api/ng/directive/ngReadonly) , and [open](https://code.angularjs.org/snapshot/docs/api/ng/directive/ngOpen).

# Filters

Filters format the value of an expression for display to the user. They can be used in view templates, controllers or services

{{ expression | filter }}

E.g. the markup {{ 12 | currency }} formats the number 12 as a currency using the [currency](https://code.angularjs.org/snapshot/docs/api/ng/filter/currency) filter. The resulting value is $12.00.

Filters can be applied to the result of another filter.

Filters may have arguments. The syntax for this is

{{ expression | filter:argument1:argument2:... }}

Filters can also be used into Service and Controllers using a dependency injection of a “service” of filter type

Custom filters:

angular.module('myReverseFilterApp', [])

.filter('reverse', function() {

return function(input, uppercase) {. . .}

};

})

.controller('MyController', ['$scope', 'reverseFilter', function($scope, reverseFilter) {. . .);

}]);

# Forms

Controls (input, select, textarea) are ways for a user to enter data. A Form is a collection of controls for the purpose of grouping related controls together.

Form and controls provide validation services, so that the user can be notified of invalid input before submitting a form

<form novalidate class="css-form">

<label>Name: <input type="text" ng-model="user.name" required /></label><br />

<label>E-mail: <input type="email" ng-model="user.email" required /></label><br />

Gender: <label><input type="radio" ng-model="user.gender" value="male" />male</label>

<label><input type="radio" ng-model="user.gender" value="female" />female</label><br />

<input type="button" ng-click="reset()" value="Reset" />

<input type="submit" ng-click="update(user)" value="Save" />

</form>

<style type="text/css">

.css-form input.ng-invalid.ng-touched {

background-color: #FA787E;

}

.css-form input.ng-valid.ng-touched {

background-color: #78FA89;

}

</style>

## **Directives**

At a high level, directives are markers on a DOM element

A directive could be written in the same way as: ng-model, ngModel, ngmodel, …

Directives could be expanded as:

.directive('myCustomer', function() {

return {

template: 'Name: {{customer.name}} Address: {{customer.address}}'

};

Inside the HTML:

<div my-customer></div>

Module

You can think of a module as a container for the different parts of your app – controllers, services, filters, directives, etc.

Why?

Most applications have a main method that instantiates and wires together the different parts of the application.

AngularJS apps don't have a main method. Instead modules declaratively specify how an application should be bootstrapped. There are several advantages to this approach:

* The declarative process is easier to understand.
* You can package code as reusable modules.
* The modules can be loaded in any order (or even in parallel) because modules delay execution.
* Unit tests only have to load relevant modules, which keeps them fast.
* End-to-end tests can use modules to override configuration.

The empty array in angular.module('myApp', []). This array is the list of modules myApp depends on.

While the example above is simple, it will not scale to large applications. Instead we recommend that you break your application to multiple modules like this:

* A module for each feature
* A module for each reusable component (especially directives and filters)
* And an application level module which depends on the above modules and contains any initialization code.

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);

});

A module is a collection of configuration and run blocks which get applied to the application during the bootstrap process. In its simplest form the module consists of a collection of two kinds of blocks:

1. **Configuration blocks** - get executed during the provider registrations and configuration phase. Only providers and constants can be injected into configuration blocks. This is to prevent accidental instantiation of services before they have been fully configured.
2. **Run blocks** - get executed after the injector is created and are used to kickstart the application. Only instances and constants can be injected into run blocks. This is to prevent further system configuration during application run time.

angular.module('myModule', []).

config(function(injectables) { // provider-injector

// This is an example of config block.

// You can have as many of these as you want.

// You can only inject Providers (not instances)

// into config blocks.

}).

run(function(injectables) { // instance-injector

// This is an example of a run block.

// You can have as many of these as you want.

// You can only inject instances (not Providers)

// into run blocks

});

Run blocks are the closest thing in AngularJS to the main method

# Providers

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 AngularJS apps most of these objects are instantiated and wired together automatically by the [injector service](https://code.angularjs.org/snapshot/docs/api/auto/service/$injector).

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.

**Value Recipe**

Let's say that we want to have a very simple service called "clientId" that provides a string representing an authentication id used for some remote API. You would define it like this:

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

myApp.value('clientId', 'a12345654321x');

Notice how we created an AngularJS module called myApp, and specified that this module definition contains a "recipe" for constructing the clientId service, which is a simple string in this case.

And this is how you would display it via AngularJS's data-binding:

myApp.controller('DemoController', ['clientId', function DemoController(clientId) {

this.clientId = clientId;

}]);

## **Factory Recipe**

The Factory recipe adds the following abilities:

* ability to use other services (have dependencies)
* service initialization
* delayed/lazy initialization

The Factory recipe constructs a new service using a function with zero or more arguments (these are dependencies on other services). The return value of this function is the service instance created by this recipe.

myApp.factory('apiToken', ['clientId', function apiTokenFactory(clientId) {

var encrypt = function(data1, data2) {

// NSA-proof encryption algorithm:

return (data1 + ':' + data2).toUpperCase(); };

var secret = window.localStorage.getItem('myApp.secret');

var apiToken = encrypt(clientId, secret);

return apiToken;}]);

## **Service Recipe**

They don’t need the return:

The Service recipe produces a service just like the Value or Factory recipes, but it does so by invoking a constructor of the object that works as service

myApp.service('unicornLauncher', ["apiToken", UnicornLauncher]);

function UnicornLauncher(apiToken) {

this.launchedCount = 0;

this.launch = function() {

// Make a request to the remote API and include the apiToken

...

this.launchedCount++;

}

}

## **Provider Recipe**

As already mentioned in the intro, the Provider recipe is the core recipe type and all the other recipe types are just syntactic sugar on top of it. It is the most verbose recipe with the most abilities, but for most services it's overkill.

**Constant Recipe**

myApp.constant('planetName', 'Greasy Giant');

| **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 |

## **Decorators**

Decorators are a design pattern that is used to separate modification or decoration of a class without modifying the original source code. In AngularJS, decorators are functions that allow a service, directive or filter to be modified prior to its usage

There are two ways to register decorators

* $provide.decorator,
* module.decorator

### **$provide.decorator**

The [decorator function](https://code.angularjs.org/snapshot/docs/api/auto/service/$provide#decorator) allows access to a $delegate of the service once it has been instantiated. For example:

angular.module('myApp', [])

.config([ '$provide', function($provide) {

$provide.decorator('$log', [

'$delegate',

function $logDecorator($delegate) {

var originalWarn = $delegate.warn;

$delegate.warn = function decoratedWarn(msg) {

msg = 'Decorated Warn: ' + msg;

originalWarn.apply($delegate, arguments);

};

return $delegate;

}

]);

}]);

After the $log service has been instantiated the decorator is fired. The decorator function has a $delegate object injected to provide access to the service that matches the selector in the decorator. This $delegate will be the service you are decorating. **The return value of the function provided to the decorator will take place of the service, directive, or filter being decorated.**

### **module.decorator**

This [function](https://code.angularjs.org/snapshot/docs/api/ng/type/angular.Module#decorator) is the same as the $provide.decorator function except it is exposed through the module API. This allows you to separate your decorator patterns from your module config blocks.

angular

.module('theApp', [])

.factory('theFactory', theFactoryFn)

.decorator('theFactory', moduleDecoratorFn);