# [Developer Guide](https://docs.angularjs.org/guide)

## Section 1

# Conceptual Overview

This section briefly touches on all of the important parts of AngularJS using a simple example. For a more in-depth explanation, see the [tutorial](https://docs.angularjs.org/tutorial/).

| **Concept** | **Description** |
| --- | --- |
| [Template](https://docs.angularjs.org/guide/concepts#template) | HTML with additional markup |
| [Directives](https://docs.angularjs.org/guide/concepts#directive) | extend HTML with custom attributes and elements |
| [Model](https://docs.angularjs.org/guide/concepts#model) | the data shown to the user in the view and with which the user interacts |
| [Scope](https://docs.angularjs.org/guide/concepts#scope) | context where the model is stored so that controllers, directives and expressions can access it |
| [Expressions](https://docs.angularjs.org/guide/concepts#expression) | access variables and functions from the scope |
| [Compiler](https://docs.angularjs.org/guide/concepts#compiler) | parses the template and instantiates directives and expressions |
| [Filter](https://docs.angularjs.org/guide/concepts#filter) | formats the value of an expression for display to the user |
| [View](https://docs.angularjs.org/guide/concepts#view) | what the user sees (the DOM) |
| [Data Binding](https://docs.angularjs.org/guide/concepts#databinding) | sync data between the model and the view |
| [Controller](https://docs.angularjs.org/guide/concepts#controller) | the business logic behind views |
| [Dependency Injection](https://docs.angularjs.org/guide/concepts#di) | Creates and wires objects and functions |
| [Injector](https://docs.angularjs.org/guide/concepts#injector) | dependency injection container |
| [Module](https://docs.angularjs.org/guide/concepts#module) | a container for the different parts of an app including controllers, services, filters, directives which configures the Injector |
| [Service](https://docs.angularjs.org/guide/concepts#service) | reusable business logic independent of views |

## A first example: Data binding

In the following example we will build a form to calculate the costs of an invoice in different currencies.

Let's start with input fields for quantity and cost whose values are multiplied to produce the total of the invoice:

Code: test\_1.html

Try out the Live Preview above, and then let's walk through the example and describe what's going on.



This looks like normal HTML, with some new markup. In Angular, a file like this is called a "[template](https://docs.angularjs.org/guide/templates)". When Angular starts your application, it parses and processes this new markup from the template using the so-called"[compiler](https://docs.angularjs.org/guide/compiler)". The loaded, transformed and rendered DOM is then called the "view".

The first kind of new markup are the so-called "[directives](https://docs.angularjs.org/guide/directive)". They apply special behavior to attributes or elements in the HTML. In the example above we use the [ng-app](https://docs.angularjs.org/api/ng/directive/ngApp) attribute, which is linked to a directive that automatically initializes our application. Angular also defines a directive for the [input](https://docs.angularjs.org/api/ng/directive/input)element that adds extra behavior to the element. The [ng-model](https://docs.angularjs.org/api/ng/directive/ngModel) directive stores/updates the value of the input field into/from a variable.

**Custom directives to access the DOM**: In Angular, the only place where an application should access the DOM is within directives. This is important because artifacts that access the DOM are hard to test. If you need to access the DOM directly you should write a custom directive for this. The[directives guide](https://docs.angularjs.org/guide/directive) explains how to do this.

The second kind of new markup are the double curly braces {{ expression | filter }}: When the compiler encounters this markup, it will replace it with the evaluated value of the markup. An "[expression](https://docs.angularjs.org/guide/expression)" in a template is a JavaScript-like code snippet that allows to read and write variables. Note that those variables are not global variables. Just like variables in a JavaScript function live in a scope, Angular provides a "[scope](https://docs.angularjs.org/guide/scope)" for the variables accessible to expressions. The values that are stored in variables on the scope are referred to as the "model" in the rest of the documentation. Applied to the example above, the markup directs Angular to "take the data we got from the input widgets and multiply them together".

The example above also contains a "[filter](https://docs.angularjs.org/guide/filter)". A filter formats the value of an expression for display to the user. In the example above, the filter [currency](https://docs.angularjs.org/api/ng/filter/currency) formats a number into an output that looks like money.

The important thing in the example is that Angular provides *live* bindings: Whenever the input values change, the value of the expressions are automatically recalculated and the DOM is updated with their values. The concept behind this is "[two-way data binding](https://docs.angularjs.org/guide/databinding)".

## Adding UI logic: Controllers

Let's add some more logic to the example that allows us to enter and calculate the costs in different currencies and also pay the invoice.

Code: test\_2.html, test\_2.js

What changed?

First, there is a new JavaScript file that contains a so-called "[controller](https://docs.angularjs.org/guide/controller)". More exactly, the file contains a constructor function that creates the actual controller instance. The purpose of controllers is to expose variables and functionality to expressions and directives.

Besides the new file that contains the controller code we also added an [ng-controller](https://docs.angularjs.org/api/ng/directive/ngController) directive to the HTML. This directive tells Angular that the new InvoiceController is responsible for the element with the directive and all of the element's children. The syntax InvoiceController as invoice tells Angular to instantiate the controller and save it in the variable invoice in the current scope.

We also changed all expressions in the page to read and write variables within that controller instance by prefixing them with invoice. . The possible currencies are defined in the controller and added to the template using [ng-repeat](https://docs.angularjs.org/api/ng/directive/ngRepeat). As the controller contains a total function we are also able to bind the result of that function to the DOM using {{ invoice.total(...) }}.

Again, this binding is live, i.e. the DOM will be automatically updated whenever the result of the function changes. The button to pay the invoice uses the directive [ngClick](https://docs.angularjs.org/api/ng/directive/ngClick). This will evaluate the corresponding expression whenever the button is clicked.

In the new JavaScript file we are also creating a [module](https://docs.angularjs.org/guide/concepts#module) at which we register the controller. We will talk about modules in the next section.

The following graphic shows how everything works together after we introduced the controller:



## View independent business logic: Services

Right now, the InvoiceController contains all logic of our example. When the application grows it is a good practice to move view independent logic from the controller into a so called "[service](https://docs.angularjs.org/guide/services)", so it can be reused by other parts of the application as well. Later on, we could also change that service to load the exchange rates from the web, e.g. by calling the Yahoo Finance API, without changing the controller.

Let's refactor our example and move the currency conversion into a service in another file:

Code: test\_3.html, test\_3.js, test\_3\_finance2.js

What changed? We moved theconvertCurrency function and the definition of the existing currencies into the new file finance2.js. But how does the controller get a hold of the now separated function?

This is where "[Dependency Injection](https://docs.angularjs.org/guide/di)" comes into play. Dependency Injection (DI) is a software design pattern that deals with how objects and functions get created and how they get a hold of their dependencies. Everything within Angular (directives, filters, controllers, services, ...) is created and wired using dependency injection. Within Angular, the DI container is called the "[injector](https://docs.angularjs.org/guide/di)".

To use DI, there needs to be a place where all the things that should work together are registered. In Angular, this is the purpose of the so-called "[modules](https://docs.angularjs.org/guide/module)". When Angular starts, it will use the configuration of the module with the name defined by the ng-app directive, including the configuration of all modules that this module depends on.

In the example above: The template contains the directive ng-app="invoice2". This tells Angular to use the invoice2 module as the main module for the application. The code snippetangular.module('invoice2', ['finance2']) specifies that the invoice2 module depends on thefinance2 module. By this, Angular uses the InvoiceController as well as the currencyConverterservice.

Now that Angular knows of all the parts of the application, it needs to create them. In the previous section we saw that controllers are created using a factory function. For services there are multiple ways to define their factory (see the [service guide](https://docs.angularjs.org/guide/services)). In the example above, we are using a function that returns thecurrencyConverter function as the factory for the service.

Back to the initial question: How does the InvoiceController get a reference to the currencyConverterfunction? In Angular, this is done by simply defining arguments on the constructor function. With this, the injector is able to create the objects in the right order and pass the previously created objects into the factories of the objects that depend on them. In our example, the InvoiceController has an argument named currencyConverter. By this, Angular knows about the dependency between the controller and the service and calls the controller with the service instance as argument.

The last thing that changed in the example between the previous section and this section is that we now pass an array to the module.controller function, instead of a plain function. The array first contains the names of the service dependencies that the controller needs. The last entry in the array is the controller constructor function. Angular uses this array syntax to define the dependencies so that the DI also works after minifying the code, which will most probably rename the argument name of the controller constructor function to something shorter like a.

## Accessing the backend

Let's finish our example by fetching the exchange rates from the Yahoo Finance API. The following example shows how this is done with Angular:

Code: test\_4.html, test\_4.js, test\_4\_finance2.js

## Section 2

# Data Binding

Data-binding in Angular apps is the automatic synchronization of data between the model and view components. The way that Angular implements data-binding lets you treat the model as the single-source-of-truth in your application. The view is a projection of the model at all times. When the model changes, the view reflects the change, and vice versa.

## Data Binding in Classical Template Systems

  
Most templating systems bind data in only one direction: they merge template and model components together into a view. After the merge occurs, changes to the model or related sections of the view are NOT automatically reflected in the view. Worse, any changes that the user makes to the view are not reflected in the model. This means that the developer has to write code that constantly syncs the view with the model and the model with the view.

## Data Binding in Angular Templates

  
Angular templates work differently. First the template (which is the uncompiled HTML along with any additional markup or directives) is compiled on the browser. The compilation step produces a live view. Any changes to the view are immediately reflected in the model, and any changes in the model are propagated to the view. The model is the single-source-of-truth for the application state, greatly simplifying the programming model for the developer. You can think of the view as simply an instant projection of your model.

Because the view is just a projection of the model, the controller is completely separated from the view and unaware of it. This makes testing a snap because it is easy to test your controller in isolation without the view and the related DOM/browser dependency.

## Section 3

# Controllers

# Understanding Controllers

In Angular, a Controller is a JavaScript **constructor function** that is used to augment the [Angular Scope](https://docs.angularjs.org/guide/scope).

When a Controller is attached to the DOM via the [ng-controller](https://docs.angularjs.org/api/ng/directive/ngController) directive, Angular will instantiate a new Controller object, using the specified Controller's **constructor function**. A new **child scope** will be available as an injectable parameter to the Controller's constructor function as $scope.

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. Angular has [databinding](https://docs.angularjs.org/guide/databinding) for most cases and[directives](https://docs.angularjs.org/guide/directive) to encapsulate manual DOM manipulation.
* Format input — Use [angular form controls](https://docs.angularjs.org/guide/forms) instead.
* Filter output — Use [angular filters](https://docs.angularjs.org/guide/filter) instead.
* Share code or state across controllers — Use [angular services](https://docs.angularjs.org/guide/services) instead.
* Manage the life-cycle of other components (for example, to create service instances).

# Setting up the initial state of a *$scope* object

Typically, when you create an application you need to set up the initial state for the Angular $scope. You set up the initial state of a scope by attaching properties to the $scope object. The properties contain the**view model** (the model that will be presented by the view). All the $scope properties will be available to the template at the point in the DOM where the Controller is registered.

The following example demonstrates creating a GreetingController, which attaches a greetingproperty containing the string 'Hola!' to the $scope:

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

myApp.controller('GreetingController', ['$scope', function($scope) {

$scope.greeting = 'Hola!';

}]);

We create an [Angular Module](https://docs.angularjs.org/guide/module), myApp, for our application. Then we add the controller's constructor function to the module using the .controller() method. This keeps the controller's constructor function out of the global scope.

We have used an **inline injection annotation** to explicitly specify the dependency of the Controller on the $scope service provided by Angular. See the guide on [Dependency Injection](https://docs.angularjs.org/guide/di) for more information.

We attach our controller to the DOM using the ng-controller directive. The greeting property can now be data-bound to the template:

<div ng-controller="GreetingController">

{{ greeting }}

</div>

# Adding Behavior to a Scope Object

In order to react to events or execute computation in the view we must provide behavior to the scope. We add behavior to the scope by attaching methods to the $scope object. These methods are then available to be called from the template/view.

The following example uses a Controller to add a method to the scope, which doubles a number:

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

myApp.controller('DoubleController', ['$scope', function($scope) {

$scope.double = function(value) { return value \* 2; };

}]);

Once the Controller has been attached to the DOM, the double method can be invoked in an Angular expression in the template:

<div ng-controller="DoubleController">

Two times <input ng-model="num"> equals {{ double(num) }}

</div>

As discussed in the [Concepts](https://docs.angularjs.org/guide/concepts) section of this guide, any objects (or primitives) assigned to the scope become model properties. Any methods assigned to the scope are available in the template/view, and can be invoked via angular expressions and ng event handler directives (e.g. [ngClick](https://docs.angularjs.org/api/ng/directive/ngClick)).

# Using Controllers Correctly

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](https://docs.angularjs.org/guide/di) [Services](https://docs.angularjs.org/guide/services) sections of this guide.

# Associating Controllers with Angular Scope Objects

You can associate Controllers with scope objects implicitly via the [ngController directive](https://docs.angularjs.org/api/ng/directive/ngController) or [$route service](https://docs.angularjs.org/api/ngRoute/service/$route).

## Simple Spicy Controller Example

To illustrate further how Controller components work in Angular, let's create a little app with the following components:

* A [template](https://docs.angularjs.org/guide/templates) with two buttons and a simple message
* A model consisting of a string named spice
* A Controller with two functions that set the value of spice

The message in our template contains a binding to the spice model, which by default is set to the string "very". Depending on which button is clicked, the spice model is set to chili or jalapeño, and the message is automatically updated by data-binding.

Code: test\_5.html, test\_5.js

Things to notice in the example above:

* The ng-controller directive is used to (implicitly) create a scope for our template, and the scope is augmented (managed) by the SpicyController Controller.
* SpicyController is just a plain JavaScript function. As an (optional) naming convention the name starts with capital letter and ends with "Controller".
* Assigning a property to $scope creates or updates the model.
* Controller methods can be created through direct assignment to scope (see the chiliSpicymethod)
* The Controller methods and properties are available in the template (for the <div> element and its children).

## Spicy Arguments Example

Controller methods can also take arguments, as demonstrated in the following variation of the previous example.

Code: test\_6.html, test\_6.js

Notice that the SpicyController Controller now defines just one method called spicy, which takes one argument called spice. The template then refers to this Controller method and passes in a string constant 'chili' in the binding for the first button and a model property customSpice (bound to an input box) in the second button.

## Scope Inheritance Example

It is common to attach Controllers at different levels of the DOM hierarchy. Since the [ng-controller](https://docs.angularjs.org/api/ng/directive/ngController) directive creates a new child scope, we get a hierarchy of scopes that inherit from each other. The $scope that each Controller receives will have access to properties and methods defined by Controllers higher up the hierarchy. See [Understanding Scopes](https://github.com/angular/angular.js/wiki/Understanding-Scopes) for more information about scope inheritance.

Code: test\_7.html, test\_7.js

Notice how we nested three ng-controller directives in our template. This will result in four scopes being created for our view:

* The root scope
* The MainController scope, which contains timeOfDay and name properties
* The ChildController scope, which inherits the timeOfDay property but overrides (hides) thename property from the previous
* The GrandChildController scope, which overrides (hides) both the timeOfDay property defined inMainController and the name property defined in ChildController

Inheritance works with methods in the same way as it does with properties. So in our previous examples, all of the properties could be replaced with methods that return string values.

## Section 4

# Services

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

Angular services are:

* Lazily instantiated – Angular only instantiates a service when an application component depends on it.
* Singletons – Each component dependent on a service gets a reference to the single instance generated by the service factory.

Angular offers several useful services (like [$http](https://docs.angularjs.org/api/ng/service/$http)), but for most applications you'll also want to [create your own](https://docs.angularjs.org/guide/services#creating-services).

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

## Using a Service

To use an Angular service, you add it as a dependency for the component (controller, service, filter or directive) that depends on the service. Angular's [dependency injection](https://docs.angularjs.org/guide/di) subsystem takes care of the rest.

Code: test\_8.html, test\_8.js

## Creating Services

Application developers are free to define their own services by registering the service's name and **service factory function**, with an Angular module.

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.

### Registering Services

Services are registered to modules via the [Module API](https://docs.angularjs.org/api/ng/type/angular.Module). Typically you use the [Module#factory](https://docs.angularjs.org/api/ng/function/angular.module) API to register a service:

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

myModule.factory('serviceId', function() {

var shinyNewServiceInstance;

// factory function body that constructs shinyNewServiceInstance

return shinyNewServiceInstance;

});

Note that you are not registering a **service instance**, but rather a **factory function** that will create this instance when called.

### Dependencies

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.

For more on dependencies, see the [dependency injection](https://docs.angularjs.org/guide/di) docs.

The example module below has two services, each with various dependencies:

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

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

var messageQueue = [];

function log() {

if (messageQueue.length) {

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

messageQueue = [];

}

}

$interval(log, 50000);

return function(message) {

messageQueue.push(message);

}

}]);

batchModule.factory('routeTemplateMonitor', ['$route', 'batchLog', '$rootScope',

function($route, batchLog, $rootScope) {

$rootScope.$on('$routeChangeSuccess', function() {

batchLog($route.current ? $route.current.template : null);

});

}]);

In the example, note that:

* The batchLog service depends on the built-in [$interval](https://docs.angularjs.org/api/ng/service/$interval) and [$log](https://docs.angularjs.org/api/ng/service/$log) services.
* The routeTemplateMonitor service depends on the built-in [$route](https://docs.angularjs.org/api/ngRoute/service/$route) service and our custombatchLog service.
* Both services use the array notation to declare their dependencies.
* The order of identifiers in the array is the same as the order of argument names in the factory function.

### Registering a Service with *$provide*

You can also register services via the [$provide](https://docs.angularjs.org/api/auto/service/$provide) service inside of a module's config function:

angular.module('myModule', []).config(['$provide', function($provide) {

$provide.factory('serviceId', function() {

var shinyNewServiceInstance;

// factory function body that constructs shinyNewServiceInstance

return shinyNewServiceInstance;

});

}]);

This technique is often used in unit tests to mock out a service's dependencies.

## Section 5

# Scopes

# What are Scopes?

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

## Scope characteristics

* Scopes provide APIs ([$watch](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch)) to observe model mutations.
* Scopes provide APIs ([$apply](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$apply)) to propagate any model changes through the system into the view from outside of the "Angular realm" (controllers, services, Angular event handlers).
* Scopes can be nested to limit access to the properties of application components while providing access to shared model properties. Nested scopes are either "child scopes" or "isolate scopes". A "child scope" (prototypically) inherits properties from its parent scope. An "isolate scope" does not. See [isolated scopes](https://docs.angularjs.org/guide/directive#isolating-the-scope-of-a-directive)for more information.
* Scopes provide context against which [expressions](https://docs.angularjs.org/guide/expression) are evaluated. For example {{username}} expression is meaningless, unless it is evaluated against a specific scope which defines the username property.

## Scope as Data-Model

Scope is the glue between application controller and the view. During the template [linking](https://docs.angularjs.org/guide/compiler) phase the[directives](https://docs.angularjs.org/api/ng/provider/$compileProvider#directive) set up [$watch](https://docs.angularjs.org/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.

Both controllers and directives have reference to the scope, but not to each other. This arrangement isolates the controller from the directive as well as from the DOM. This is an important point since it makes the controllers view agnostic, which greatly improves the testing story of the applications.

Code: test\_9.html, test\_9.js

In the above example notice that the MyController assigns World to the username property of the scope. The scope then notifies the input of the assignment, which then renders the input with username pre-filled. This demonstrates how a controller can write data into the scope.

Similarly the controller can assign behavior to scope as seen by the sayHello method, which is invoked when the user clicks on the 'greet' button. The sayHello method can read the username property and create a greeting property. This demonstrates that the properties on scope update automatically when they are bound to HTML input widgets.

Logically the rendering of {{greeting}} involves:

* retrieval of the scope associated with DOM node where {{greeting}} is defined in template. In this example this is the same scope as the scope which was passed into MyController. (We will discuss scope hierarchies later.)
* Evaluate the greeting [expression](https://docs.angularjs.org/guide/expression) against the scope retrieved above, and assign the result to the text of the enclosing DOM element.

You can think of the scope and its properties as the data which is used to render the view. The scope is the single source-of-truth for all things view related.

From a testability point of view, the separation of the controller and the view is desirable, because it allows us to test the behavior without being distracted by the rendering details.

it('should say hello', function() {

var scopeMock = {};

var cntl = new MyController(scopeMock);

// Assert that username is pre-filled

expect(scopeMock.username).toEqual('World');

// Assert that we read new username and greet

scopeMock.username = 'angular';

scopeMock.sayHello();

expect(scopeMock.greeting).toEqual('Hello angular!');

});

## Scope Hierarchies

Each Angular application has exactly one [root scope](https://docs.angularjs.org/api/ng/service/$rootScope), but may have several child scopes.

The application can have multiple scopes, because some [directives](https://docs.angularjs.org/guide/directive) create new child scopes (refer to directive documentation to see which directives create new scopes). When new scopes are created, they are added as children of their parent scope. This creates a tree structure which parallels the DOM where they're attached.

When Angular evaluates {{name}}, it first looks at the scope associated with the given element for thename property. If no such property is found, it searches the parent scope and so on until the root scope is reached. In JavaScript this behavior is known as prototypical inheritance, and child scopes prototypically inherit from their parents.

This example illustrates scopes in application, and prototypical inheritance of properties. The example is followed by a diagram depicting the scope boundaries.

Code: test\_10.html, test\_10.js, test\_10\_style.css

Notice that Angular automatically places ng-scope class on elements where scopes are attached. The<style> definition in this example highlights in red the new scope locations. The child scopes are necessary because the repeater evaluates {{name}} expression, but depending on which scope the expression is evaluated it produces different result. Similarly the evaluation of {{department}}prototypically inherits from root scope, as it is the only place where the department property is defined.

## Retrieving Scopes from the DOM.

Scopes are attached to the DOM as $scope data property, and can be retrieved for debugging purposes. (It is unlikely that one would need to retrieve scopes in this way inside the application.) The location where the root scope is attached to the DOM is defined by the location of [ng-app](https://docs.angularjs.org/api/ng/directive/ngApp) directive. Typically ng-app is placed on the <html> element, but it can be placed on other elements as well, if, for example, only a portion of the view needs to be controlled by Angular.

To examine the scope in the debugger:

1. Right click on the element of interest in your browser and select 'inspect element'. You should see the browser debugger with the element you clicked on highlighted.
2. The debugger allows you to access the currently selected element in the console as $0 variable.
3. To retrieve the associated scope in console execute: angular.element($0).scope() or just type $scope

## Scope Events Propagation

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

Code: test\_11.html, test\_11.js

## Scope Life Cycle

The normal flow of a browser receiving an event is that it executes a corresponding JavaScript callback. Once the callback completes the browser re-renders the DOM and returns to waiting for more events.

When the browser calls into JavaScript the code executes outside the Angular execution context, which means that Angular is unaware of model modifications. To properly process model modifications the execution has to enter the Angular execution context using the [$apply](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$apply) method. Only model modifications which execute inside the $apply method will be properly accounted for by Angular. For example if a directive listens on DOM events, such as [ng-click](https://docs.angularjs.org/api/ng/directive/ngClick) it must evaluate the expression inside the $applymethod.

After evaluating the expression, the $apply method performs a [$digest](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest). In the $digest phase the scope examines all of the $watch expressions and compares them with the previous value. This dirty checking is done asynchronously. This means that assignment such as $scope.username="angular" will not immediately cause a $watch to be notified, instead the $watch notification is delayed until the $digestphase. This delay is desirable, since it coalesces multiple model updates into one $watch notification as well as it guarantees that during the $watch notification no other $watches are running. If a $watchchanges the value of the model, it will force additional $digest cycle.

1. **Creation**

The [root scope](https://docs.angularjs.org/api/ng/service/$rootScope) is created during the application bootstrap by the [$injector](https://docs.angularjs.org/api/auto/service/$injector). During template linking, some directives create new child scopes.

1. **Watcher registration**

During template linking directives register [watches](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) on the scope. These watches will be used to propagate model values to the DOM.

1. **Model mutation**

For mutations to be properly observed, you should make them only within the [scope.$apply()](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$apply). (Angular APIs do this implicitly, so no extra $apply call is needed when doing synchronous work in controllers, or asynchronous work with [$http](https://docs.angularjs.org/api/ng/service/$http), [$timeout](https://docs.angularjs.org/api/ng/service/$timeout) or [$interval](https://docs.angularjs.org/api/ng/service/$interval) services.

1. **Mutation observation**

At the end of $apply, Angular performs a [$digest](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest) cycle on the root scope, which then propagates throughout all child scopes. During the $digest cycle, all $watched expressions or functions are checked for model mutation and if a mutation is detected, the $watch listener is called.

1. **Scope destruction**

When child scopes are no longer needed, it is the responsibility of the child scope creator to destroy them via [scope.$destroy()](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$destroy) API. This will stop propagation of $digest calls into the child scope and allow for memory used by the child scope models to be reclaimed by the garbage collector.

### Scopes and Directives

During the compilation phase, the [compiler](https://docs.angularjs.org/guide/compiler) matches [directives](https://docs.angularjs.org/api/ng/provider/$compileProvider#directive) against the DOM template. The directives usually fall into one of two categories:

* Observing [directives](https://docs.angularjs.org/api/ng/provider/$compileProvider#directive), such as double-curly expressions {{expression}}, register listeners using the [$watch()](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) method. This type of directive needs to be notified whenever the expression changes so that it can update the view.
* Listener directives, such as [ng-click](https://docs.angularjs.org/api/ng/directive/ngClick), register a listener with the DOM. When the DOM listener fires, the directive executes the associated expression and updates the view using the [$apply()](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$apply) method.

When an external event (such as a user action, timer or XHR) is received, the associated [expression](https://docs.angularjs.org/guide/expression) must be applied to the scope through the [$apply()](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$apply) method so that all listeners are updated correctly.

### Directives that Create Scopes

In most cases, [directives](https://docs.angularjs.org/api/ng/provider/$compileProvider#directive) and scopes interact but do not create new instances of scope. However, some directives, such as [ng-controller](https://docs.angularjs.org/api/ng/directive/ngController) and [ng-repeat](https://docs.angularjs.org/api/ng/directive/ngRepeat), create new child scopes and attach the child scope to the corresponding DOM element. You can retrieve a scope for any DOM element by using anangular.element(aDomElement).scope() method call. See the [directives guide](https://docs.angularjs.org/guide/directive#isolating-the-scope-of-a-directive) for more information about isolate scopes.

### Controllers and Scopes

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

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

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

### Scope *$watch* Performance Considerations

Dirty checking the scope for property changes is a common operation in Angular and for this reason the dirty checking function must be efficient. Care should be taken that the dirty checking function does not do any DOM access, as DOM access is orders of magnitude slower than property access on JavaScript object.

### Scope *$watch* Depths



Dirty checking can be done with three strategies: By reference, by collection contents, and by value. The strategies differ in the kinds of changes they detect, and in their performance characteristics.

* Watching by reference ([scope.$watch](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) (watchExpression, listener)) detects a change when the whole value returned by the watch expression switches to a new value. If the value is an array or an object, changes inside it are not detected. This is the most efficient strategy.
* Watching collection contents ([scope.$watchCollection](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watchCollection) (watchExpression, listener)) detects changes that occur inside an array or an object: When items are added, removed, or reordered. The detection is shallow - it does not reach into nested collections. Watching collection contents is more expensive than watching by reference, because copies of the collection contents need to be maintained. However, the strategy attempts to minimize the amount of copying required.
* Watching by value ([scope.$watch](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) (watchExpression, listener, true)) detects any change in an arbitrarily nested data structure. It is the most powerful change detection strategy, but also the most expensive. A full traversal of the nested data structure is needed on each digest, and a full copy of it needs to be held in memory.

## Integration with the browser event loop



The diagram and the example below describe how Angular interacts with the browser's event loop.

1. The browser's event-loop waits for an event to arrive. An event is a user interaction, timer event, or network event (response from a server).
2. The event's callback gets executed. This enters the JavaScript context. The callback can modify the DOM structure.
3. Once the callback executes, the browser leaves the JavaScript context and re-renders the view based on DOM changes.

Angular modifies the normal JavaScript flow by providing its own event processing loop. This splits the JavaScript into classical and Angular execution context. Only operations which are applied in the Angular execution context will benefit from Angular data-binding, exception handling, property watching, etc... You can also use $apply() to enter the Angular execution context from JavaScript. Keep in mind that in most places (controllers, services) $apply has already been called for you by the directive which is handling the event. An explicit call to $apply is needed only when implementing custom event callbacks, or when working with third-party library callbacks.

1. Enter the Angular execution context by calling [scope](https://docs.angularjs.org/guide/scope).[$apply](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$apply)(stimulusFn), where stimulusFn is the work you wish to do in the Angular execution context.
2. Angular executes the stimulusFn(), which typically modifies application state.
3. Angular enters the [$digest](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest) loop. The loop is made up of two smaller loops which process[$evalAsync](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$evalAsync) queue and the [$watch](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) list. The [$digest](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest) loop keeps iterating until the model stabilizes, which means that the [$evalAsync](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$evalAsync) queue is empty and the [$watch](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) list does not detect any changes.
4. The [$evalAsync](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$evalAsync) queue is used to schedule work which needs to occur outside of current stack frame, but before the browser's view render. This is usually done with setTimeout(0), but thesetTimeout(0) approach suffers from slowness and may cause view flickering since the browser renders the view after each event.
5. The [$watch](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) list is a set of expressions which may have changed since last iteration. If a change is detected then the $watch function is called which typically updates the DOM with the new value.
6. Once the Angular [$digest](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest) loop finishes the execution leaves the Angular and JavaScript context. This is followed by the browser re-rendering the DOM to reflect any changes.

Here is the explanation of how the Hello world example achieves the data-binding effect when the user enters text into the text field.

1. During the compilation phase:
   1. the [ng-model](https://docs.angularjs.org/api/ng/directive/ngModel) and [input](https://docs.angularjs.org/api/ng/directive/input) [directive](https://docs.angularjs.org/guide/directive) set up a keydown listener on the <input> control.
   2. the [interpolation](https://docs.angularjs.org/api/ng/service/$interpolate) sets up a [$watch](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) to be notified of name changes.
2. During the runtime phase:
   1. Pressing an 'X' key causes the browser to emit a keydown event on the input control.
   2. The [input](https://docs.angularjs.org/api/ng/directive/input) directive captures the change to the input's value and calls[$apply](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$apply)("name = 'X';") to update the application model inside the Angular execution context.
   3. Angular applies the name = 'X'; to the model.
   4. The [$digest](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest) loop begins
   5. The [$watch](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) list detects a change on the name property and notifies the [interpolation](https://docs.angularjs.org/api/ng/service/$interpolate), which in turn updates the DOM.
   6. Angular exits the execution context, which in turn exits the keydown event and with it the JavaScript execution context.
   7. The browser re-renders the view with update text.

## Section 6

# Dependency Injection

Dependency Injection (DI) is a software design pattern that deals with how components get hold of their dependencies.

The AngularJS injector subsystem is in charge of creating components, resolving their dependencies, and providing them to other components as requested.

## Using Dependency Injection

DI is pervasive throughout AngularJS. You can use it when defining components or when providing run and config blocks for a module.

* Components such as services, directives, filters, and animations are defined by an injectable factory method or constructor function. These components can be injected with "service" and "value" components as dependencies.
* Controllers are defined by a constructor function, which can be injected with any of the "service" and "value" components as dependencies, but they can also be provided with special dependencies. See [Controllers](https://docs.angularjs.org/guide/di#controllers) below for a list of these special dependencies.
* The run method accepts a function, which can be injected with "service", "value" and "constant" components as dependencies. Note that you cannot inject "providers" into run blocks.
* The config method accepts a function, which can be injected with "provider" and "constant" components as dependencies. Note that you cannot inject "service" or "value" components into configuration.

See [Modules](https://docs.angularjs.org/guide/module#module-loading-dependencies) for more details about run and config blocks.

### 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) {

// ...

}])

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

// ...

}])

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

// ...

}]);

### Module Methods

We can specify functions to run at configuration and run time for a module by calling the config and run methods. These functions are injectable with dependencies just like the factory functions above.

angular.module('myModule', [])

.config(['depProvider', function(depProvider) {

// ...

}])

.run(['depService', function(depService) {

// ...

}]);

### Controllers

Controllers are "classes" or "constructor functions" that are responsible for providing the application behavior that supports the declarative markup in the template. The recommended way of declaring Controllers is using the array notation:

someModule.controller('MyController', ['$scope', 'dep1', 'dep2', function($scope, dep1, dep2) {

...

$scope.aMethod = function() {

...

}

...

}]);

Unlike services, there can be many instances of the same type of controller in an application.

Moreover, additional dependencies are made available to Controllers:

* [$scope](https://docs.angularjs.org/guide/scope): Controllers are associated with an element in the DOM and so are provided with access to the [scope](https://docs.angularjs.org/guide/scope). Other components (like services) only have access to the [$rootScope](https://docs.angularjs.org/api/ng/service/$rootScope) service.
* [resolves](https://docs.angularjs.org/api/ngRoute/provider/$routeProvider#when): If a controller is instantiated as part of a route, then any values that are resolved as part of the route are made available for injection into the controller.

## Dependency Annotation

AngularJS invokes certain functions (like service factories and controllers) via the injector. You need to annotate these functions so that the injector knows what services to inject into the function. There are three ways of annotating your code with service name information:

* Using the inline array annotation (preferred)
* Using the $inject property annotation
* Implicitly from the function parameter names (has caveats)

### 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);

In this scenario the ordering of the values in the $inject array must match the ordering of the parameters in MyController.

Just like with the array annotation, you'll need to take care to keep the $inject in sync with the parameters in the function declaration.

### Implicit Annotation

**Careful:** If you plan to [minify](http://en.wikipedia.org/wiki/Minification_(programming)) your code, your service names will get renamed and break your app.

The simplest way to get hold of the dependencies is to assume that the function parameter names are the names of the dependencies.

someModule.controller('MyController', function($scope, greeter) {

// ...

});

Given a function, the injector can infer the names of the services to inject by examining the function declaration and extracting the parameter names. In the above example, $scope and greeter are two services which need to be injected into the function.

One advantage of this approach is that there's no array of names to keep in sync with the function parameters. You can also freely reorder dependencies.

However this method will not work with JavaScript minifiers/obfuscators because of how they rename parameters.

Tools like [ng-annotate](https://github.com/olov/ng-annotate) let you use implicit dependency annotations in your app and automatically add inline array annotations prior to minifying. If you decide to take this approach, you probably want to use ng-strict-di.

Because of these caveats, we recommend avoiding this style of annotation.

## Using Strict Dependency Injection

You can add an ng-strict-di directive on the same element as ng-app to opt into strict DI mode:

<!doctype html>

<html ng-app="myApp" ng-strict-di>

<body>

I can add: {{ 1 + 2 }}.

<script src="angular.js"></script>

</body>

</html>

Strict mode throws an error whenever a service tries to use implicit annotations.

Consider this module, which includes a willBreak service that uses implicit DI:

angular.module('myApp', [])

.factory('willBreak', function($rootScope) {

// $rootScope is implicitly injected

})

.run(['willBreak', function(willBreak) {

// AngularJS will throw when this runs

}]);

When the willBreak service is instantiated, AngularJS will throw an error because of strict mode. This is useful when using a tool like [ng-annotate](https://github.com/olov/ng-annotate) to ensure that all of your application components have annotations.

If you're using manual bootstrapping, you can also use strict DI by providing strictDi: true in the optional config argument:

angular.bootstrap(document, ['myApp'], {

strictDi: true

});

## Why Dependency Injection?

This section motivates and explains AngularJS's use of DI. For how to use DI, see above.

For in-depth discussion about DI, see [Dependency Injection](http://en.wikipedia.org/wiki/Dependency_injection) at Wikipedia, [Inversion of Control](http://martinfowler.com/articles/injection.html) by Martin Fowler, or read about DI in your favorite software design pattern book.

There are only three ways a component (object or function) can get a hold of its dependencies:

1. The component can create the dependency, typically using the new operator.
2. The component can look up the dependency, by referring to a global variable.
3. The component can have the dependency passed to it where it is needed.

The first two options of creating or looking up dependencies are not optimal because they hard code the dependency to the component. This makes it difficult, if not impossible, to modify the dependencies. This is especially problematic in tests, where it is often desirable to provide mock dependencies for test isolation.

The third option is the most viable, since it removes the responsibility of locating the dependency from the component. The dependency is simply handed to the component.

function SomeClass(greeter) {

this.greeter = greeter;

}

SomeClass.prototype.doSomething = function(name) {

this.greeter.greet(name);

}

In the above example SomeClass is not concerned with creating or locating the greeter dependency, it is simply handed the greeterwhen it is instantiated.

This is desirable, but it puts the responsibility of getting hold of the dependency on the code that constructs SomeClass.



To manage the responsibility of dependency creation, each AngularJS application has an [injector](https://docs.angularjs.org/api/ng/function/angular.injector). The injector is a [service locator](http://en.wikipedia.org/wiki/Service_locator_pattern) that is responsible for construction and lookup of dependencies.

Here is an example of using the injector service:

// Provide the wiring information in a module

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

Teach the injector how to build a greeter service. Notice that greeter is dependent on the $window service. The greeter service is an object that contains a greet method.

myModule.factory('greeter', function($window) {

return {

greet: function(text) {

$window.alert(text);

}

};

});

Create a new injector that can provide components defined in our myModule module and request our greeter service from the injector. (This is usually done automatically by AngularJS bootstrap).

var injector = angular.injector(['ng', 'myModule']);

var greeter = injector.get('greeter');

Asking for dependencies solves the issue of hard coding, but it also means that the injector needs to be passed throughout the application. Passing the injector breaks the [Law of Demeter](http://en.wikipedia.org/wiki/Law_of_Demeter). To remedy this, we use a declarative notation in our HTML templates, to hand the responsibility of creating components over to the injector, as in this example:

<div ng-controller="MyController">

<button ng-click="sayHello()">Hello</button>

</div>

function MyController($scope, greeter) {

$scope.sayHello = function() {

greeter.greet('Hello World');

};

}

When AngularJS compiles the HTML, it processes the ng-controller directive, which in turn asks the injector to create an instance of the controller and its dependencies.

injector.instantiate(MyController);

This is all done behind the scenes. Notice that by having the ng-controller ask the injector to instantiate the class, it can satisfy all of the dependencies of MyController without the controller ever knowing about the injector.

This is the best outcome. The application code simply declares the dependencies it needs, without having to deal with the injector. This setup does not break the Law of Demeter.