# AngularJS

https://docs.angularjs.org/guide/

# Introduction

# What Is Angular?

AngularJS is a structural framework for dynamic web apps. It lets you use HTML as your template language and lets you extend HTML's syntax to express your application's components clearly and succinctly. Angular's data binding and dependency injection eliminate much of the code you would otherwise have to write. And it all happens within the browser, making it an ideal partner with any server technology.

Angular is what HTML would have been, had it been designed for applications. HTML is a great declarative language for static documents. It does not contain much in the way of creating applications, and as a result building web applications is an exercise in *what do I have to do to trick the browser into doing what I want?*

The impedance mismatch between dynamic applications and static documents is often solved with:

- **a library** - a collection of functions which are useful when writing web apps. Your code is in charge and it calls into the library when it sees fit. E.g., `jQuery`.
- **frameworks** - a particular implementation of a web application, where your code fills in the details. The framework is in charge and it calls into your code when it needs something app specific. E.g., `durandal`, `ember`, etc.

Angular takes another approach. It attempts to minimize the impedance mismatch between document centric HTML and what an application needs by creating new HTML constructs. Angular teaches the browser new syntax through a construct we call *directives*. Examples include:

- Data binding, as in `{{}}`.
- DOM control structures for repeating, showing and hiding DOM fragments.
- Support for forms and form validation.
- Attaching new behavior to DOM elements, such as DOM event handling.
- Grouping of HTML into reusable components.

# A complete client-side solution

Angular is not a single piece in the overall puzzle of building the client-side of a web application. It handles all of the DOM and AJAX glue code you once wrote by hand and puts it in a well-defined structure. This makes Angular opinionated about how a CRUD (Create, Read, Update, Delete) application should be built. But while it is opinionated, it also tries to make sure that its opinion is just a starting point you can easily change. Angular comes with the following out-of-the-box:

- Everything you need to build a CRUD app in a cohesive set: Data-binding, basic templating directives, form validation, routing, deep-linking, reusable components and dependency injection.
- Testability story: Unit-testing, end-to-end testing, mocks and test harnesses.
- Seed application with directory layout and test scripts as a starting point.

# Angular's sweet spot

Angular simplifies application development by presenting a higher level of abstraction to the developer. Like any abstraction, it comes at a cost of flexibility. In other words, not every app is a good fit for Angular. Angular was built with the CRUD application in mind. Luckily CRUD applications represent the majority of web applications. To understand what Angular is good at, though, it helps to understand when an app is not a good fit for Angular.

Games and GUI editors are examples of applications with intensive and tricky DOM manipulation. These kinds of apps are different from CRUD apps, and as a result are probably not a good fit for Angular. In these cases it may be better to use a library with a lower level of abstraction, such as `jQuery`.

# The Zen of Angular

Angular is built around the belief that declarative code is better than imperative when it comes to building UIs and wiring software components together, while imperative code is excellent for expressing business logic.

- It is a very good idea to decouple DOM manipulation from app logic. This dramatically improves the testability of the code.
- It is a really, *really* good idea to regard app testing as equal in importance to app writing. Testing difficulty is dramatically affected by the way the code is structured.
- It is an excellent idea to decouple the client side of an app from the server side. This allows development work to progress in parallel, and allows for reuse of both sides.
- It is very helpful indeed if the framework guides developers through the entire journey of building an app: From designing the UI, through writing the business logic, to testing.
- It is always good to make common tasks trivial and difficult tasks possible.

Angular frees you from the following pains:

- **Registering callbacks:** Registering callbacks clutters your code, making it hard to see the forest for the trees. Removing common boilerplate code such as callbacks is a good thing. It vastly reduces the amount of JavaScript coding *you* have to do, and it makes it easier to see what your application does.
- **Manipulating HTML DOM programmatically:** Manipulating HTML DOM is a cornerstone of AJAX applications, but it's cumbersome and error-prone. By declaratively describing how the UI should change as your application state changes, you are freed from low-level DOM manipulation tasks. Most applications written with Angular never have to programmatically manipulate the DOM, although you can if you want to.
- **Marshaling data to and from the UI:** CRUD operations make up the majority of AJAX applications' tasks. The flow of marshaling data from the server to an internal object to an HTML form, allowing users to modify the form, validating the form, displaying validation errors, returning to an internal model, and then back to the server, creates a lot of boilerplate

code. Angular eliminates almost all of this boilerplate, leaving code that describes the overall flow of the application rather than all of the implementation details.

- **Writing tons of initialization code just to get started:** Typically you need to write a lot of plumbing just to get a basic "Hello World" AJAX app working. With Angular you can bootstrap your app easily using services, which are auto-injected into your application in a Guice-like dependency-injection style. This allows you to get started developing features quickly. As a bonus, you get full control over the initialization process in automated tests.

# 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.

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

# A first example: Data binding

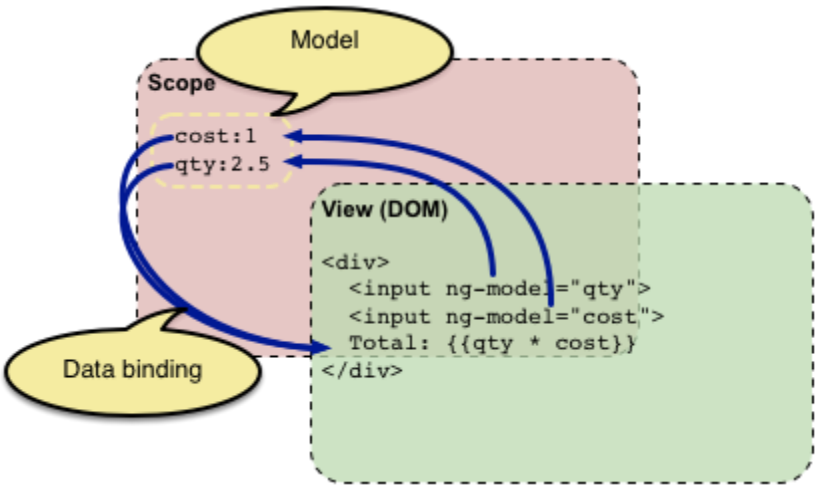
In the following example we will build a form to calculate the costs of an invoice in different currencies.

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

index.html

```
<div ng-app ng-init="qty=1;cost=2"> <b>Invoice:</b> <div> Quantity: <input type="number"
min="0" ng-model="qty"> </div> <div> Costs: <input type="number" min="0" ng-model="cost">
</div> <div> <b>Total:</b> {{qty * cost | currency}} </div> </div>
```

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. When Angular starts your application, it parses and processes this new markup from the template using the compiler. The loaded, transformed and rendered DOM is then called the view.

The first kind of new markup are the directives. They apply special behavior to attributes or elements in the HTML. In the example above we use the `ng-app` attribute, which is linked to a directive that automatically initializes our application. Angular also defines a directive for the `input` element that adds extra behavior to the element. The `ng-model` 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 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 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 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. A filter formats the value of an expression for display to the user. In the example above, the 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.

# 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.

invoice1.js index.html

```
angular.module('invoice1', []) .controller('InvoiceController', function() { this.qty = 1;
this.cost = 2; this.inCurr = 'EUR'; this.currencies = ['USD', 'EUR', 'CNY'];
this.usdToForeignRates = { USD: 1, EUR: 0.74, CNY: 6.09 }; this.total = function
total(outCurr) { return this.convertCurrency(this.qty * this.cost, this.inCurr, outCurr); };
this.convertCurrency = function convertCurrency(amount, inCurr, outCurr) { return amount *
this.usdToForeignRates[outCurr] / this.usdToForeignRates[inCurr]; }; this.pay = function pay()
{ window.alert("Thanks!"); }; });
```

```
<div ng-app="invoice1" ng-controller="InvoiceController as invoice"> <b>Invoice:</b> <div>
Quantity: <input type="number" min="0" ng-model="invoice.qty" required > </div> <div> Costs:
<input type="number" min="0" ng-model="invoice.cost" required > <select ng-
model="invoice.inCurr"> <option ng-repeat="c in invoice.currencies">{{c}}</option> </select>
</div> <div> <b>Total:</b> <span ng-repeat="c in invoice.currencies"> {{invoice.total(c) |
currency:c}} </span> <button class="btn" ng-click="invoice.pay()">Pay</button> </div> </div>
```

What changed?

First, there is a new JavaScript file that contains a 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.

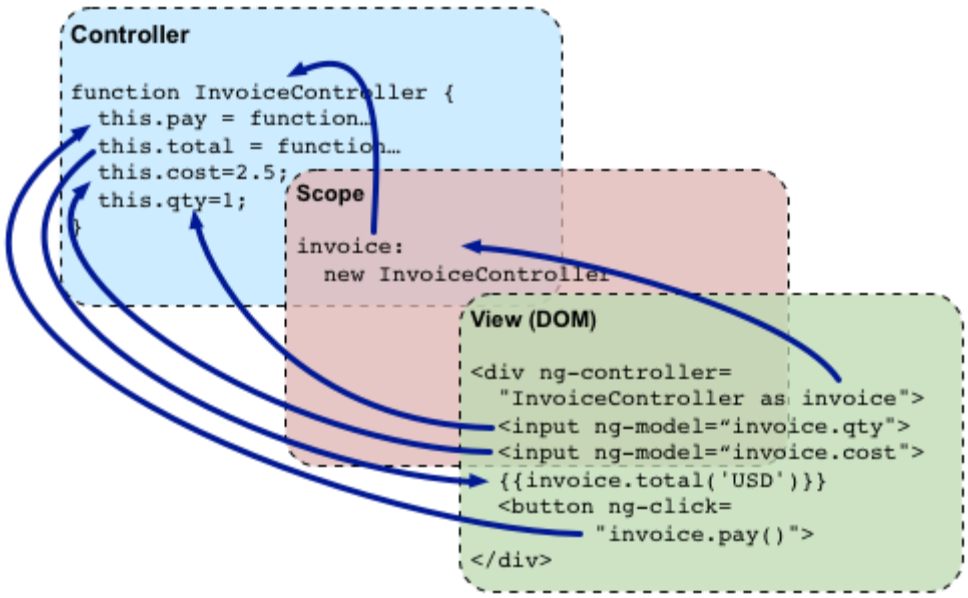
Besides the new file that contains the controller code we also added an `ng-controller` directive to the HTML. This directive tells Angular that the new `InvoiceController` is responsible for the element with the directive and all of the element's children. The syntax `InvoiceController as invoice` tells 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`. 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`. This will evaluate the corresponding expression whenever the button is clicked.

In the new JavaScript file we are also creating a 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:



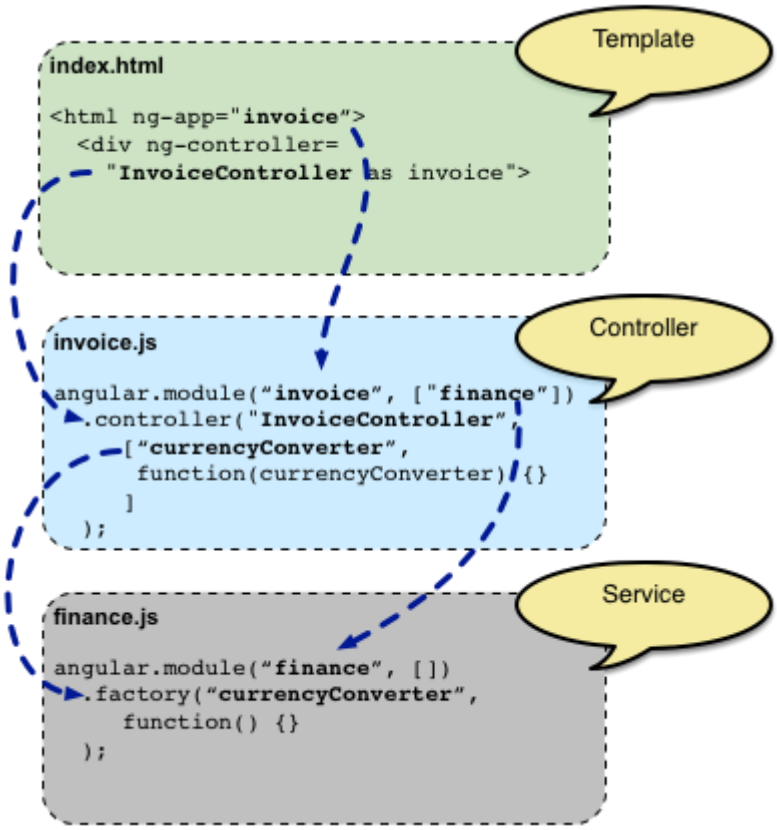

# View-independent business logic: Services

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 service, 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:

finance2.js invoice2.js index.html

```
angular.module('finance2', []) .factory('currencyConverter', function() { var currencies =
['USD', 'EUR', 'CNY']; var usdToForeignRates = { USD: 1, EUR: 0.74, CNY: 6.09 }; var convert =
function (amount, inCurr, outCurr) { return amount * usdToForeignRates[outCurr] /
usdToForeignRates[inCurr]; }; return { currencies: currencies, convert: convert }; });

angular.module('invoice2', ['finance2']) .controller('InvoiceController',
['currencyConverter', function(currencyConverter) { this.qty = 1; this.cost = 2; this.inCurr =
'EUR'; this.currencies = currencyConverter.currencies; this.total = function total(outCurr) {
return currencyConverter.convert(this.qty * this.cost, this.inCurr, outCurr); }; this.pay =
function pay() { window.alert("Thanks!"); }; }]);

<div ng-app="invoice2" ng-controller="InvoiceController as invoice"> <b>Invoice:</b> <div>
Quantity: <input type="number" min="0" ng-model="invoice.qty" required > </div> <div> Costs:
<input type="number" min="0" ng-model="invoice.cost" required > <select ng-
model="invoice.inCurr"> <option ng-repeat="c in invoice.currencies">{{c}}</option> </select>
</div> <div> <b>Total:</b> <span ng-repeat="c in invoice.currencies"> {{invoice.total(c) |
currency:c}} </span> <button class="btn" ng-click="invoice.pay()">Pay</button> </div> </div>
```




What changed? We moved the `convertCurrency` 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 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.

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 modules. 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 snippet `angular.module('invoice2', ['finance2'])` specifies that the `invoice2` module depends on the `finance2` module. By this, Angular uses the `InvoiceController` as well as the `currencyConverter` service.

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). In the example above, we are using a function that returns the `currencyConverter` function as the factory for the service.

Back to the initial question: How does the `InvoiceController` get a reference to the `currencyConverter` function? 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:

Edit in Plunker

invoice3.js finance3.js index.html

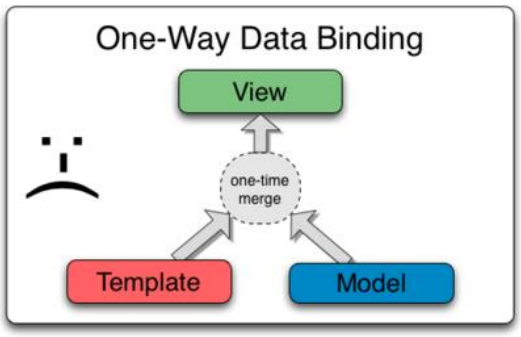
```
angular.module('invoice3', ['finance3']) .controller('InvoiceController',
['currencyConverter', function(currencyConverter) { this.qty = 1; this.cost = 2; this.inCurr =
'EUR'; this.currencies = currencyConverter.currencies; this.total = function total(outCurr) {
return currencyConverter.convert(this.qty * this.cost, this.inCurr, outCurr); }; this.pay =
function pay() { window.alert("Thanks!"); }; }]);

angular.module('finance3', []) .factory('currencyConverter', ['$http', function($http) { var
YAHOO_FINANCE_URL_PATTERN = '//query.yahooapis.com/v1/public/yql?q=select * from '+
'yahoo.finance.xchange where pair in ("PAIRS")&format=json&'+
'env=store://datatables.org/alltableswithkeys&callback=JSON_CALLBACK'; var currencies =
['USD', 'EUR', 'CNY']; var usdToForeignRates = {}; var convert = function (amount, inCurr,
outCurr) { return amount * usdToForeignRates[outCurr] / usdToForeignRates[inCurr]; }; var
refresh = function() { var url = YAHOO_FINANCE_URL_PATTERN. replace('PAIRS', 'USD' +
currencies.join('","USD')); return $http.jsonp(url).success(function(data) { var
newUsdToForeignRates = {}; angular.forEach(data.query.results.rate, function(rate) { var
currency = rate.id.substring(3,6); newUsdToForeignRates[currency] =
window.parseFloat(rate.Rate); }); usdToForeignRates = newUsdToForeignRates; }); }; refresh();
return { currencies: currencies, convert: convert, refresh: refresh }; }]);

<div ng-app="invoice3" ng-controller="InvoiceController as invoice"> <b>Invoice:</b> <div>
Quantity: <input type="number" min="0" ng-model="invoice.qty" required > </div> <div> Costs:
<input type="number" min="0" ng-model="invoice.cost" required > <select ng-
model="invoice.inCurr"> <option ng-repeat="c in invoice.currencies">{{c}}</option> </select>
</div> <div> <b>Total:</b> <span ng-repeat="c in invoice.currencies"> {{invoice.total(c) |
currency:c}} </span> <button class="btn" ng-click="invoice.pay()">Pay</button> </div> </div>
```

What changed? Our `currencyConverter` service of the `finance` module now uses the `$http`, a built-in service provided by Angular for accessing a server backend. `$http` is a wrapper around `XMLHttpRequest` and JSONP transports.

# Data Binding

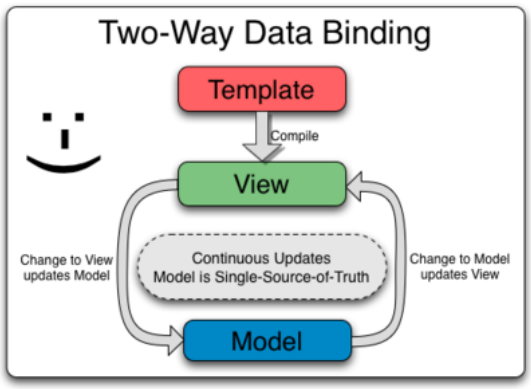
Data-binding in Angular apps is the automatic synchronization of data between the model and view components. The way that Angular implements data-binding lets you treat the model as the single-source-of-truth in your application. The view is a projection of the model at all times. When the model changes,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