

AngularJS

Everything you need to understand to start with AngularJS

2012 was the year of the rise of the Javascript MVC frameworks and libraries with countless of those frameworks that were released or that went under the spotlight. Among others, you must have heard about [Backbone.js](#), [Ember.js](#) but my personal favorite is [AngularJS](#). AngularJS is a Javascript MVC framework created by Google to build properly architected and maintainable web applications.

If you haven't tried Angular yet, you're missing out on why people say JavaScript is the most flexible language in the world.

Angular is the only framework that doesn't make MVC seem like putting lipstick on a pig.

Most frameworks nowadays are simply a bundling of existing tools. They are an integrated tool set, but not very elegant. Angular is the next generation framework where each tool was designed to work with every other tool in an interconnected way.

MVC done right

Most frameworks implement MVC by asking you to split your app into MVC components, then require you to write code to string them up together again. That's a lot of work. Angular implements MVC by asking you to split your app into MVC components, then just let Angular do the rest. Angular manages your components for you and also serves as the pipeline that connects them.

Because Angular acts as the mediator, developers also won't feel tempted to write shortcuts between components that break abstractions just to make them fit easier.

Meet AngularJS

AngularJS is a client-side MVC framework written in JavaScript. It runs in a web browser and greatly helps us (developers) to write modern, single-page, AJAX-style web applications. It is a general purpose framework, but it shines when used to write CRUD (Create Read Update Delete) type web applications.

Getting familiar with the framework

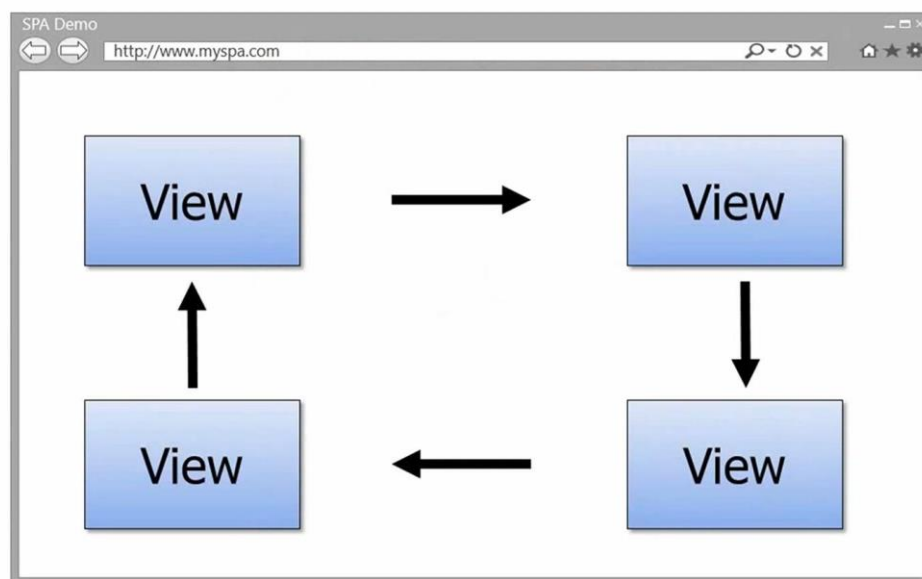
AngularJS is a recent addition to the client-side MVC frameworks list, yet it has managed to attract a lot of attention, mostly due to its innovative templating system, ease of development, and very solid engineering practices. Indeed, its templating system is unique in many respects:

- It uses HTML as the templating language
- It doesn't require an explicit DOM refresh, as AngularJS is capable of tracking user actions, browser events, and model changes to figure out when and which templates to refresh
- It has a very interesting and extensible components subsystem, and it is possible to teach a browser how to interpret new HTML tags and attributes .

The templating subsystem might be the most visible part of AngularJS, but don't be mistaken that AngularJS is a complete framework packed with several utilities and services typically needed in single-page web applications. AngularJS also has some hidden treasures, **dependency injection (DI)** and **strong focus on testability**. The built-in support for DI makes it easy to assemble a web application from smaller, thoroughly tested services. The design of the framework and the tooling around **it promote testing practices** at each stage of the development process.

Single page Application

Single Page Application (SPA)



First off, a Single Page Application is one in which we have a shell page and we can load multiple views into that.

So a traditional app, as you know you typically blink and load everything again. It's not very efficient on the bandwidth, especially in the mobile world.

In a SPA we can load the initial content upfront and then the different views or the little kind of mini- web pages can be loaded on the fly and embedded into the shell.

AngularJS, as we're going to see, is a very good SPA framework, but it's not just for that. You don't have to load these dynamic views with it. In fact if you wanted you could just use it for some of the cool separation of code that I'll show you and data binding, but I will focus on SPAs here.

The Challenge with SPAs

DOM Manipulation

History

Module Loading

Routing

Caching

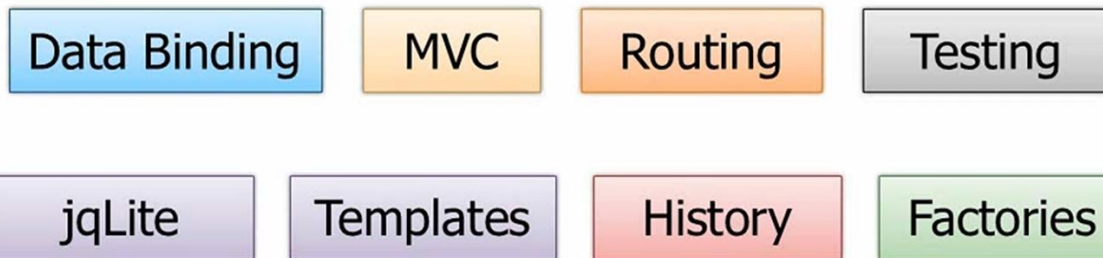
Object Modeling


Data Binding

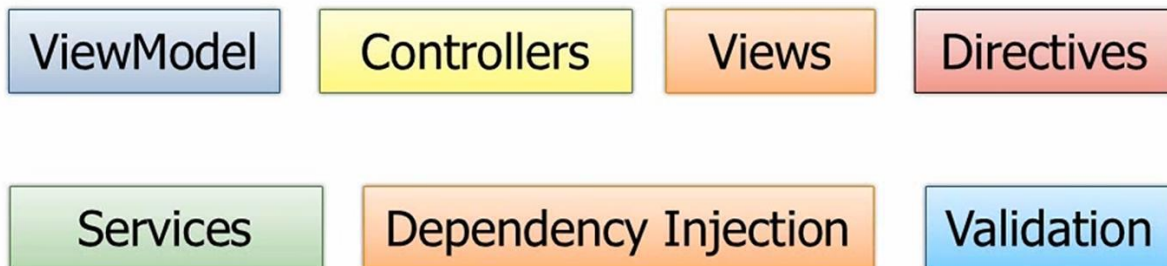
Ajax/Promises

View Loading





 AngularJS is a full-featured
SPA framework



Conceptual Overview

Conceptual Overview

This section briefly touches on all of the important parts of AngularJS using a simple example.

Concept	Description
Template	HTML with additional markup
Directives	extend HTML with custom attributes and elements

Concept	Description
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
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 Injector
Service	reusable business logic independent of views

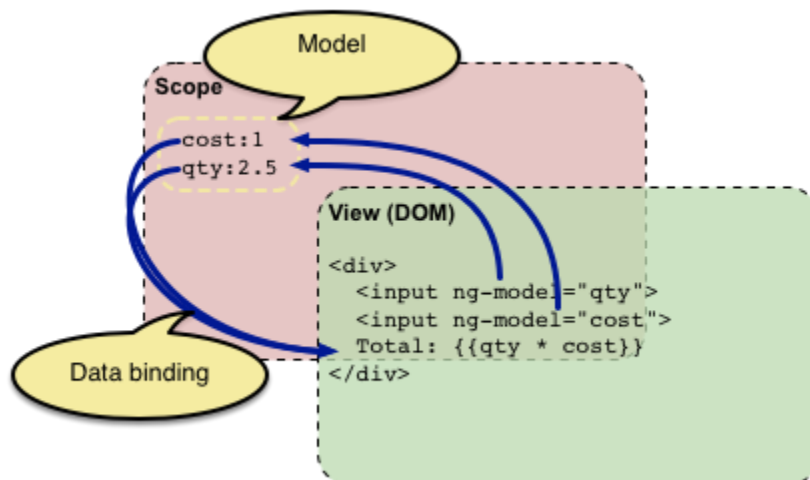
A first example: Data binding

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:

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

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 so called "[compiler](#)". The loaded, transformed and rendered DOM is then called the "[view](#)".



The first kind of new markup are the so called "[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. E.g. it is able to automatically validate that the entered text is non empty by evaluating the `required` attribute. The `ng-model` directive stores/updates the value of the input field into/from a variable and shows the validation state of the input field by adding css classes. In the example we use these css classes to mark an empty input field with a red border.

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.

Invoice.js

```
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!");
};

});

```

INDEX.HTML

```

<div ng-app="invoice1" ng-controller="InvoiceController as invoice">
    <b>Invoice:</b>
    <div>
        Quantity: <input type="number" ng-model="invoice.qty" required >
    </div>
    <div>
        Costs: <input type="number" 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 so called ["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.

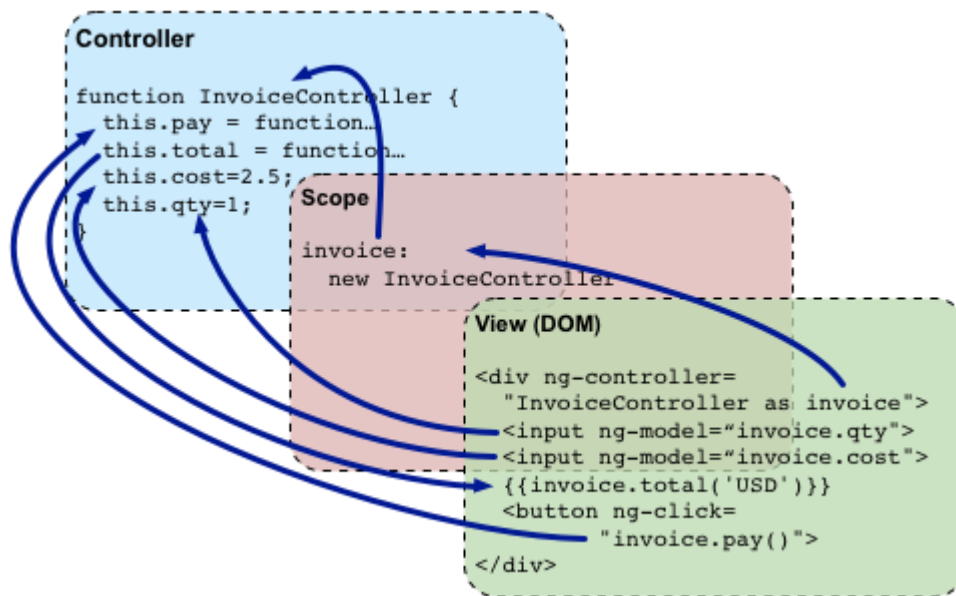
Besides the new file that contains the controller code we also added a `ng-controller` directive to the HTML. This directive tells angular that the new `InvoiceController` is responsible for the element with the directive and all of the element's children. The syntax `InvoiceController as invoice` tells 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 so called "[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:

```
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!");
        };
    }]);

```

```
        {{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 so called "[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 `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.

