
AngularJS 도입 선택 가이드

네이버 AU개발랩

저작권

Copyright © NAVER Corp. All Rights Reserved.

이 문서는 네이버(주)의 지적 재산이므로 어떠한 경우에도 네이버(주)의 공식적인 허가 없이 이 문서의 일부 또는 전체를 복제, 전송, 배포하거나 변경하여 사용할 수 없습니다.

이 문서는 정보 제공의 목적으로만 제공됩니다. 네이버(주)는 이 문서에 수록된 정보의 완전성과 정확성을 검증하기 위해 노력하였으나, 발생할 수 있는 내용상의 오류나 누락에 대해서는 책임지지 않습니다. 따라서 이 문서의 사용이나 사용 결과에 따른 책임은 전적으로 사용자에게 있으며, 네이버(주)는 이에 대해 명시적 혹은 묵시적으로 어떠한 보증도 하지 않습니다.

관련 URL 정보를 포함하여 이 문서에서 언급한 특정 소프트웨어 상품이나 제품은 해당 소유자가 속한 현지 및 국내외 관련법을 따르며, 해당 법률을 준수하지 않음으로 인해 발생하는 모든 결과에 대한 책임은 전적으로 사용자 자신에게 있습니다.

네이버(주)는 이 문서의 내용을 예고 없이 변경할 수 있습니다.

문서 정보



본 문서는 [네이버 개발자 블로그 hello world](#)를 통해서 공개되었습니다.

문서 개요

이 문서는 AngularJS를 사용한 애플리케이션 개발을 계획하고 있거나 AngularJS 도입을 고려 중일 때 참고할 만한 내용을 기술한다. AngularJS를 사용하는 방법을 다루지는 않는다.

이 문서의 모든 내용은 AngularJS 버전 1.3.x를 기준으로 작성되었다.

독자

이 문서는 기존 프론트엔드 개발 방법(DOM을 선택하고 제어하는)과 라이브러리/프레임워크 사용에 익숙한 개발자, 또는 AngularJS의 실업무 도입 여부를 판단해야 하는 실무자를 대상으로 한다.

문의처

이 문서의 내용에 오류가 있거나 내용과 관련한 의문 사항이 있으면 아래의 연락처로 문의한다.

연락처: 박재성(jaesung.park@navercorp.com)

문서 버전 및 이력

버전	일자	이력사항	작성자
0.9	2014.11.28	초안 작성	Ajax 플랫폼
			김성철/박재성/손찬욱
1.0	2015.01.09	성능 비교 그래프 및 내용 수정	Ajax 플랫폼
		Angular 2.0 에 대한 참고 내용 보완	김성철/박재성/손찬욱

표기 규칙

참고 표기

참고

독자가 참고해야 할 내용을 기술한다.

소스 코드 표기

이 문서에서 소스 코드는 회색 바탕에 검정색 글씨로 표기한다.

```
COPYDATASTRUCT st;  
st.dwData = PURPLE_OUTBOUND_ENDING;  
st.cbData = sizeof(pp);  
st.lpData = &pp;  
::SendMessage(GetTargetHwnd(), WM_COPYDATA, (LPARAM)this->m_hWnd, (LPARAM)&st);
```

목차

AngularJS 소개	9
개요	9
AngularJS란?	9
관심도	9
브라우저 호환성	10
특징과 주요 기능	11
특징	11
주요 기능	11
초기화 과정	19
학습 비용	20
 AngularJS 사용 효과	 22
개발 방식 비교	22
DOM 제어 방식	22
AngularJS(MVC 방식)	23
정리	24
구현 코드 비교	25
마크업	25
자바스크립트	27
정리	30
 AngularJS 사용 시 고려 사항	 31
양방향 데이터 바인딩	31
대상 추출	31
확인 시점	31
확인 방법	32
고려 사항	34
양방향 데이터 바인딩 성능 비교	35
성능 비교 데이터	35
서버로드 방식 대 AngularJS 방식	36

단방향 대 양방향 데이터 바인딩	37
고려 사항	38
그 외 고려 사항	40
컴파일되지 않은 콘텐츠의 출력	40
뷰가 분리된 구성	41
이벤트 바인딩	42
 도입 여부 선택 가이드	 44
정리	44
AngularJS 도입 여부 판단 흐름도	45
 부록	 46
디버깅 도구	46
타 라이브러리와의 사용성	47
도입 사례	48
도입 이유 및 효과	48
개발 기간	49
정리	49

표 및 그림 목록

표 목록

표 1 경험에 따른 AngularJS 학습 시간 사례	21
표 2 Jindo 와 AngularJS의 개발 코드 크기 비교	30

그림 목록

그림 1 프레임워크 키워드 검색 빈도수	9
그림 2 도입 여부에 따른 MVC 프레임워크 선호도	10
그림 3 기존의 단방향 데이터 바인딩(one-way data binding)	12
그림 4 새로운 양방향 데이터 바인딩(two-way data binding)	13
그림 5 uppercase 필터가 적용된 결과 화면	17
그림 6 currency 필터가 적용된 결과 화면	17
그림 7 커스텀 필터 'nl2br'이 적용된 결과 화면	18
그림 8 반복 지시자가 수행된 결과	18
그림 9 AngularJS 애플리케이션이 자동으로 초기화되는 과정	19
그림 10 DOM 제어 방식의 개발 방식	22
그림 11 DOM 제어 방식의 데이터 입력 절차	23
그림 12 DOM 제어 방식의 데이터 출력 절차	23
그림 13 AngularJS의 개발 방식	24
그림 14 AngularJS의 데이터 입출력 절차	24
그림 15 Jindo의 뷰 영역	25
그림 16 AngularJS의 뷰와 템플릿	26
그림 17 Jindo와 AngularJS의 개발 코드 크기	30
그림 18 digest 루프	32
그림 19 Digest 루프 호출 횟수: 최소 2번, 최대 10번	33
그림 20 서버 로드 시간(단위: ms)	36
그림 21 AngularJS 로드 시간(단위 ms)	37
그림 22 단방향 바인딩 처리 시간(단위 ms)	37
그림 23 양방향 바인딩 처리 시간(단위 ms)	38
그림 24 두 가지 로드 시간 비교(단위 ms)	38
그림 25 두 가지 방식의 처리 시간(단위 ms)	39

그림 26 초기화 전 템플릿 구문이 잠시 출력되는 과정	40
그림 27 별도의 분리된 템플릿 파일을 활용한 예시	42
그림 28 템플릿 파일 로딩에 대한 XHR 로그	42
그림 29 <input> 요소에 입력된 문자를 출력하는 예시	42
그림 30 <input> 요소에 문자 입력 시 발생하는 이벤트 로그	43
그림 31 AngularJS 도입 여부 판단 흐름도	45
그림 32 Batarang 크롬 확장 플러그인	46
그림 33 AngularJS 스코프 확인	46
그림 34 Show bindings 활성화 상태	47
그림 35 CSS와 텍스트의 내용을 각각 jQLite와 Jindo를 사용해 변경하는 버튼	48

AngularJS 소개

이 장에서는 AngularJS를 간략히 소개하고 기본적인 기능과 특징을 설명한다. 이를 통해 왜 수많은 개발자들이 AngularJS에 관심을 갖게 되었는지 살펴본다.

개요

AngularJS 란?

AngularJS¹는 2009년 Miško Hevery와 Adam Abrons에 의해 개발된 MVC(또는 MVW, Model View Whatever)² 웹 프레임워크로, SPA(Single Page Application) 형태의 웹 애플리케이션을 빠르게 개발할 수 있도록 도와준다.

기존의 라이브러리나 프레임워크와는 다르게, AngularJS는 DOM 제어 방식과 관련된 기능에 중점을 두지 않으며, 데이터의 변화와 그에 따른 출력에 초점이 맞추어져 있다. 따라서, AngularJS를 사용한 개발은 기존 개발 방식과는 다른, 새로운 접근법이 필요하다.

관심도

현 시점에서 MVC 프레임워크 중 가장 많은 관심을 받고 있는 프레임워크로는 AngularJS와 BackboneJS, ExtJS, EmberJS 등이 있다. 그 중 AngularJS에 대한 관심도는 2013년을 기점으로 급상승한 것을 확인할 수 있다.



그림 1 프레임워크 키워드 검색 빈도수³

¹ 초기 BRAT Tech. LLC 라는 스타트업의 프로젝트로 진행되었으나, 이후 오픈소스로 전환되었으며, Miško Hevery 가 구글에 입사함에 따라 자연스럽게 구글의 지원을 받아 개발되고 있다.

² 많은 개발자들 사이에서 AngularJS 가 MVC, MVP 또는 MVVP 인지를 놓고 논란이 있어 왔으며, 이 논란을 종식시키고자 AngularJS 팀은 AngularJS 를 MVW 로 정의하였다. <https://plus.google.com/+AngularJS/posts/aZNVhj355G2>

또한 실제 업무나 서비스에서의 도입 여부를 묻는 개발자 대상 온라인 설문 조사에서도 AngularJS가 다른 프레임워크들에 비해 앞서 있다.



그림 2 도입 여부에 따른 MVC 프레임워크 선호도⁴

브라우저 호환성

AngularJS는 대다수의 모던 브라우저를 지원한다. 하지만, IE⁵의 경우 버전에 따라 지원이 제한된다. AngularJS 1.3부터는 IE9 이상을 지원하고 AngularJS 1.2에서 여전히 IE8을 지원하지만, AngularJS 개발팀에서 공식적으로 향후 적극적인 지원 계획이 없음을 표명하고 있어 AngularJS를 사용한 애플리케이션 개발은 최신 모던 브라우저만을 대상으로 고려해야 한다.

³ 구글 트렌드(2014/11 기준): <https://www.google.ie/trends/explore#q=ExtJs%2C%20AngularJs%2C%20EmberJs%2C%20Backbone.js>

⁴ InfoQ 온라인 설문(2014/11 기준): <http://www.infoq.com/research/top-javascript-mvc-frameworks>

⁵ AngularJS의 IE 호환성 여부는 다음 페이지를 참고한다. <https://docs.angularjs.org/guide/ie>

특징과 주요 기능

이 문서에서 AngularJS의 사용법을 다루지는 않는다. 하지만 전반적인 내용을 이해할 수 있게 기본적인 수준의 AngularJS 기능과 특징을 간략히 설명한다.

특징

데이터 중심적

기존의 DOM 제어 방식 중심의 Jindo나 jQuery 등과는 다른 개념의 접근이 필요하다. 기존 방식이 변경해야 할 대상 DOM을 먼저 선택한 후 필요한 작업을 수행하는 형태라면, AngularJS는 데이터 자체에 초점을 맞춰 작업을 수행하며, 변경된 값의 출력(DOM 영역의 업데이트)은 자동으로 수행된다.

테스트 주도적

의존성 주입(Dependency Injection)을 통해 목업 객체로 손쉽게 전환할 수 있으며, 뷰와 분리된 컨트롤러 내의 비즈니스 로직 코드는 테스트 친화적인 구조로 개발할 수 있다.

선언적 HTML

커스텀 태그 및 태그 내에서의 속성 등을 통해 선언적 형태로 사용할 수 있으며, 이를 통해 단순한 구조의 DOM을 만들 수 있다.

주요 기능

데이터 바인딩

간단한 모델과 뷰를 처리하는 기본적인 AngularJS 애플리케이션은 다음과 같이 구성할 수 있다.

```
<!doctype html>
<html ng-app> (1)
<head>
  <title>페이지</title>
  <!-- AngularJS 로딩 -->
  <script type="text/javascript" src="./angular.js"></script>
</head>
<body>
  <div>
    <input type='text' ng-model='name' /> (2)
    <h2>{{name}}</h2> (3)
  </div>
</body>
</html>
```

- **ng-app:** 이 지시자(directive) 내에 속한 영역은 AngularJS 애플리케이션으로 인식된다.(예제 코드에서는 <html> ... </html> 사이의 영역을 의미)
- **ng-model='name':** 데이터 바인딩을 위해 사용되는 지시자로, 값으로 지정한 'name'은 모델의 이름으로 사용된다.
- **{{name}}:** 데이터의 출력을 위한 템플릿 표현식으로, 이중 중괄호로 묶어 사용한다. 이곳에 사용된 모델명인 'name'은 ng-model='name'과 바인딩되어 처리된다.

위의 코드를 수행하면, <input> 영역에 사용자가 입력한 값이 {{name}} 영역에 바로 반영되어 출력된다. 기존 개발 방식과는 다르게 데이터의 값이 출력될 요소를 선택해 업데이트하는 과정과 사용자의 입력값 변경에 대한 이벤트 처리를 별도로 하지 않더라도, 이에 대한 과정은 AngularJS 내부에서 자동적으로 처리된다.

참고

AngularJS 앱 내에서 데이터 바인딩⁶은 모델과 뷰 사이에서의 데이터의 변화에 대한 자동 동기화를 의미하며, 이를 '양방향 데이터 바인딩(two-way data binding)'이라 부른다.

우리가 이미 알고 있는 기존의 단방향 데이터 바인딩(one-way data binding)을 활용한 뷰의 갱신은 다음 그림과 같이 '템플릿 + 데이터'를 조합하고 출력할 뷰를 구성해 화면에 갱신하는 형태로 이루어진다. 즉, 데이터가 변경되면 템플릿과 변경된 데이터를 재조합한 후 뷰를 재갱신하는 반복적인 과정을 거쳐야 한다.

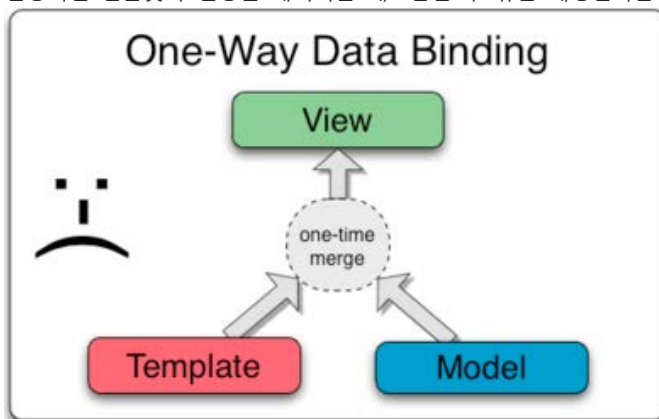
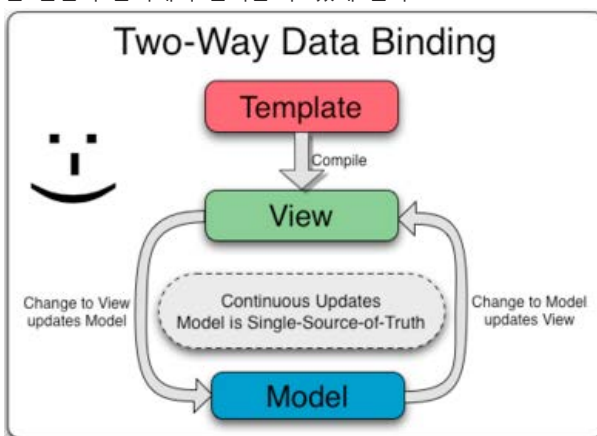


그림 3 기존의 단방향 데이터 바인딩(one-way data binding)

반면, AngularJS에서 채택하고 있는 양방향 데이터 바인딩은 템플릿을 컴파일⁷한 후, 모델-뷰 간의 변화를 지속적으로 확인해 변화가 생기면 자동으로 업데이트를 수행한다.

따라서 한 번 데이터 바인딩을 한 이후에는 별도로 뷰에 대한 업데이트를 신경 쓸 필요가 없어지며, 모델과 뷰를 완전히 분리해서 관리할 수 있게 된다.



⁶ <https://docs.angularjs.org/guide/databinding>

⁷ 컴파일에 대해서는, 이 문서의 "초기화 과정"을 참고한다.

그림 4 새로운 양방향 데이터 바인딩(two-way data binding)

모듈

모듈은 AngularJS 애플리케이션에서 객체들을 잘 조직화하기 위해 사용되며, 기본적으로 모든 애플리케이션은 모듈 단위로 구성된다. 기본적인 모듈의 선언은 다음과 같다.

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

이는 html 내에서 ng-app 지시자에 설정한 이름과 바인딩된다.

```
<html ng-app='myApp'>
```

angular.module() 메서드를 통해 모듈에 대한 선언(또는 정의)이 이루어지며, 두 번째 파라미터인 배열을 통해서 해당 모듈이 의존성을 갖는 모듈이 있는 경우 추가로 지정할 수 있다.

예를 들어 'myApp' 모듈이 'mySecondApp'이라는 모듈에 의존성을 갖는 경우 다음과 같이 한다.

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

컨트롤러

컨트롤러는 모듈 내에 정의되며, HTML 요소와 묶이도록 구성된다. 컨트롤러에는 비즈니스 로직이 구현되며, 데이터와 이벤트 핸들러 함수 등이 포함된다.

컨트롤러는 ng-controller 지시자를 통해 지정되며, 여기서 지정된 영역이 해당 컨트롤러의 스코프다.

```
<body ng-app='myApp'>
  <div ng-controller='myController'>
    <h2>{{name}}</h2>
  </div>
</body>
```

위의 예제에서는 ng-controller를 통해 지정된 <div> ... </div> 영역을 스코프로 갖는 myController 컨트롤러가 지정되었다. 해당 영역에 대한 스코프는 \$scope⁸라는 특별한 변수값을 통해 컨트롤러 함수에 전달된다.(이벤트 핸들러 함수에 이벤트 객체가 전달되는 경우를 떠올리면, 좀 더 이해하기 쉽다.)

이제 컨트롤러에서 {{name}}값을 업데이트하는 코드를 만들어 보자.

```
var app = angular.module('myApp', []); // 체이닝 형태로도 사용할 수 있다.
app.controller('myController', function($scope) {
  $scope.name = '안녕하세요';
});
```

\$scope 파라미터를 통해 컨트롤러 스코프 내의 name 값을 할당하는 것으로 값의 출력을 변경하게 된다.

컨트롤러에서는 사용자의 이벤트를 처리할 수도 있다. 예를 들어 ng-click 지시자는 클릭 이벤트를 처리할 수 있도록 한다.

```
<body ng-app='myApp'>
```

⁸ \$scope 객체에 대한 자세한 내용은 다음을 참고한다. <https://docs.angularjs.org/guide/scope>, <https://github.com/angular/angular.js/wiki/Understanding-Scopes>

```
<div ng-controller='myController'>
  <h2 ng-click='getName()'>{{name}}</h2>
</div>
</body>
```

```
angular.module('myApp', [])
.controller('myController', function($scope) {
  $scope.name = '안녕하세요';

  // ng-click='getName()'에 대한 이벤트 핸들러
  $scope.getName = function() {
    alert($scope.name);
  }
});
```

지시자

몇몇 지시자에 대해서는 이미 위에서 살펴봤다. AngularJS에서 제공하는 지시자를 사용할 수도 있지만, 개발자가 직접 지시자를 만들 수도 있다.

지시자는 `.directive()` 메서드를 이용해 만들 수 있다. 다음의 간단한 예제를 살펴보자.

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

// 사용할 지시자는 camel-case 형식으로 명명
app.directive('myInfo', function() {
  return {
    template: '이름: {{info.name}}, 주소: {{info.address}}'
  };
})
.controller("myController", function($scope) {
  $scope.info = {
    name: '홍길동',
    address: '경기도 성남시 분당구 불정로 6 그린 팩토리'
  };
});
```

이제 새롭게 만든 지시자를 아래와 같이 원하는 곳에 지정한다.

```
<body ng-app='myApp'>
  <div ng-controller='myController'>
    <span my-info></span>
  </div>
</body>
```

실제 수행되면, `` 영역은 다음과 같이 처리된다.

```
<span my-info="" class="ng-binding">이름: 홍길동, 주소: 경기도 성남시 분당구 불정로 6 그린
팩토리</span>
```

필요한 경우, 지시자를 반복해서 여러 곳에 사용할 수 있다.

옵션을 이용해 지시자가 사용되는 '형태'를 지정할 수도 있다. 위의 예제에서는 속성과 같이 사용되었지만 필요에 따라 태그, 클래스 또는 주석의 형태로도 사용할 수 있다.

다음의 예제는 간단한 태그 형태의 지시자를 만들어 사용하는 경우이다.

```
app.directive('myInfo2', function() {
  return {
    // restrict 옵션을 통해 지시자가 어떤 형태인지 지정할 수 있다.
    // E: Element, A: Attribute, C: Class, M: Comment
    restrict: 'E',

    // link 옵션의 함수는 DOM 제어가 필요한 경우 지정한다.
```

```
// 전달되는 파라미터는 다음과 같다.
// scope: 스코프 객체, element: jqLite로 래핑된 요소, attrs: normalize된 속성값
link: function(scope, element, attrs) {
    element.html("안녕하세요~@!");
}
};
})
```

일반 태그와 동일하게 만들어 지정하면 된다.

```
<body ng-app='myApp'>
  <div ng-controller='myController'>
    <my-info2></my-info2>
  </div>
</body>
```

실제 출력되는 결과는 다음과 같고, 지시자에서 지정한 것처럼 값이 반영된다.

```
<my-info2>안녕하세요~@!</my-info2>
```

서비스

서비스는 애플리케이션 내에서 비즈니스 로직과 데이터 등을 공유하기 위해 사용될 수 있는 객체이다.

```
app.factory('myService', function() {
    var oData = { name: '김길동', address: '서울시' };

    return {
        set: function(oVal) {
            oData = oVal;
        },
        get: function() {
            return oData;
        }
    };
});
```

.factory() 메서드는 전체 애플리케이션 내에서 싱글톤 객체로 생성하며, 반환되는 객체/함수는 애플리케이션 내의 다른 컴포넌트(컨트롤러, 서비스, 필터, 지시자) 등에 주입되어 사용될 수 있다.

위의 예제에서 등록된 서비스는 다음과 같이 컨트롤러에 파라미터로 전달해 주입할 수 있으며, 이를 통해 컨트롤러 간의 데이터 공유와 공용 함수 등을 등록해 사용할 수 있다.

```
app.controller("myController1", function($scope, myService) {
    // 서비스의 get() 메서드를 실행한다.
    console.log(myService.get());
    //--> { name: '김길동', address: '서울시' };

    // 서비스의 set() 메서드를 통해 값을 변경한다.
    myService.set({ name: '하하', address: '호호' });
});

app.controller("myController2", function($scope, myService) {
    // 서비스의 get() 메서드를 실행한다.
    console.log(myService.get());
    //--> ({ name: '하하', address: '호호' }
});
```

참고로, 컨트롤러에 전달되는 파라미터의 순서가 변경되더라도 동작에는 아무런 영향을 주지 않는다.

```
app.controller("myController1", function($scope, myService) { ... });
app.controller("myController2", function(myService, $scope) { ... });
```

그러나 전달되는 파라미터의 순서를 일관성 있게 만들고자 한다면(또는 코드의 minification 등으로 인해 디버깅이 어려운 점을 걱정하는 경우 등) 다음과 같이 배열 형태로 등록할 수도 있다.

```
app.controller("myController1", [
  '$scope',
  'myService',
  function($scope, myService) { ... }
]);

app.controller("myController2", [
  '$scope',
  'myService',
  function($scope, myService) { ... }
]);
```

라우팅

라우팅 기능은 URL을 기준으로 다른 컨트롤러를 수행하거나 템플릿을 사용하도록 설정할 수 있도록 해준다.

```
app.config(function($routeProvider) {
  $routeProvider
    .when('/person1', {
      controller: 'myController1',
      templateUrl: 'template1.html'
    })
    .when('/person2', {
      controller: 'myController2',
      templateUrl: 'template2.html'
    })
    .otherwise({ redirectTo: '/person1' });
});
```

.config() 메서드를 통해 URL을 기준으로 사용될 컨트롤러와 템플릿 파일을 각각 지정할 수 있다.

현재 브라우저에서 호출한 주소가 `http://test.com/list.html`이라고 했을 때, 실제 주소의 값은 다음과 같이 처리된다.

<http://test.com/list.html#/person1> 또는 <http://test.com/list.html#/person2>

그리고 list.html 파일 내에서 라우팅된 템플릿이 담길 영역은 다음과 같이 지정할 수 있다.

```
<body>
  <!-- 뷰 템플릿 파일의 내용이 담길 영역 -->
  <ng-view></ng-view>
</body>
```

기억할 점은 라우팅되는 주소의 값에 따라 다른 템플릿 뷰 파일을 사용해 관리할 수 있는데, 이 경우 각 템플릿 파일은 AJAX를 통해 로드된다는 사실이다.

필터

필터는 데이터를 특정 형태로 변형시키고자 할 때 사용한다. 필터의 사용은 템플릿 내에서 파이프 기호(|) 연산자를 통해 간단히 적용할 수 있다.

아래의 예제는 <input> 영역에 입력하는 값을 대문자로 치환하는 예이다.

```
<div>
  <input type="text" ng-model="val">
  <span>{{ val | uppercase }}</span>
</div>
```




그림 5 uppercase 필터가 적용된 결과 화면

여러 개의 필터를 중복해서 사용하는 것도 가능하다. 아래의 예제는 최종적으로 소문자를 출력한다.

```
<div>
  <input type="text" ng-model="val">
  <span>{{ val | uppercase | lowercase }}</span>
</div>
```

템플릿 영역이 아닌, 컨트롤러에서 필터를 사용하고자 할 때는 `$filter`를 사용한다.

```
<div ng-controller="myController3">
  <input type="text" ng-model="val">
  <span ng-click="showCurrency()">{{ val }}</span>
  <span>{{ val2 }}</span>
</div>
app.controller("myController3", function($scope, $filter) {
  $scope.showCurrency = function() {

    // 필터를 적용한다.
    $scope.val2 = $filter('currency')($scope.val);
  }
});
```

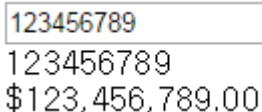


그림 6 currency 필터가 적용된 결과 화면

AngularJS에서 기본으로 제공하는 필터⁹들은 다음과 같다.

- currency, date, filter, json, limitTo, lowercase, uppercase, number, orderBy

기본으로 제공되는 필터 외에도 필요하면 직접 커스텀 필터를 만들 수 있다. 간단히 줄 바꿈 문자를 '
' 태그로 변환하는 필터의 예제를 살펴보자. 필터는 `.filter()` 메서드를 이용해 만들 수 있다.

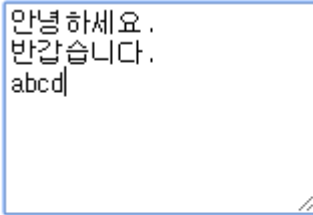
```
app.filter("nl2br", function($sce) {
  return function(text) {
    if(text) return $sce.trustAsHtml(text.replace(/\n/g, '<br>'));
  }
});
```

필터를 만들고, 다음과 같이 출력되는 내용에 'nl2br' 필터¹⁰를 적용해 보자.

```
<div ng-controller="myController4">
  <textarea ng-model="val" style="width:150px;height:100px"></textarea><br>
  <span ng-bind-html="val | nl2br"></span>
</div>
```

⁹ 필터 각각에 대한 자세한 설명은 다음 페이지를 참고한다. <https://docs.angularjs.org/api/ng/filter>

¹⁰ 특정 컨텍스트에 HTML 태그가 인식되는 값을 지정하기 위해선 `$sce`(Strict Contextual Escaping) 서비스와 `ng-bind-html` 지시자를 사용해야 한다. [https://docs.angularjs.org/api/ng/service/\\$sce](https://docs.angularjs.org/api/ng/service/$sce)



안녕하세요.
반갑습니다.
abcd

안녕하세요.
반갑습니다.
abcd

그림 7 커스텀 필터 'nl2br'이 적용된 결과 화면

참고

AngularJS는 HTML 자체를 템플릿으로 인식한다. 다른 템플릿 라이브러리의 처리와는 다르게 별도로 템플릿 영역을 분리해 구성할 필요 없이, 페이지 내에 템플릿 표현식을 사용해 필요한 데이터가 출력되도록 할 수 있다. 다음의 목록형 데이터를 출력하는 간단한 예제를 살펴보자.

```
app.controller("myControllerList", function($scope, $filter) {
    $scope.fruits = [
        { name: '사과', qty: 5 },
        { name: '감', qty: 2 },
        { name: '메론', qty: 9 },
        { name: '망고', qty: 3 },
        { name: '바나나', qty: 1 }
    ];
});
```

HTML 내에서 별도로 템플릿 영역을 분리해 선언할 필요 없이, 바로 아래와 같이 ng-repeat 지시자를 이용해 목록형 데이터를 반복해서 출력할 수 있다.

```
<div ng-controller="myControllerList">
  <ul>
    <li ng-repeat="fruit in fruits">
      <span>{{ fruit.name }}</span> / <span>{{ fruit.qty }}</span>
    </li>
  </ul>
</div>
```

위의 코드를 수행한 결과 화면은 아래와 같다.

- 사과 / 5
- 감 / 2
- 메론 / 9
- 망고 / 3
- 바나나 / 1

그림 8 반복 지시자가 수행된 결과

참고

현재 AngularJS 팀 내에선 차기 메이저 업데이트 버전인 2.0에 대해 논의가 진행되고 있다. 지금까지 공개된 내용에 따르면 1.x 버전과의 호환성을 지원하지 않을 것으로 보인다.(주요 문법의 변경과 폐지 등)¹¹

¹¹ ng-europe 2014: Angular 2.0 Core: <http://www.youtube.com/watch?v=gNmWybAyBHI>

또한 새로운 자바스크립트의 수퍼셋 언어인 AtScript¹²를 사용한 개발 계획을 발표한 상태이다.

초기화 과정

주요 기능에서 살펴본 것처럼 AngularJS로 개발된 애플리케이션을 실행하려면 HTML 내에 정의된 지시자와 템플릿 등에 대한 파싱 작업이 선행되어야 한다. 다음의 그림에서 초기화 과정을 간단하게 살펴본다.

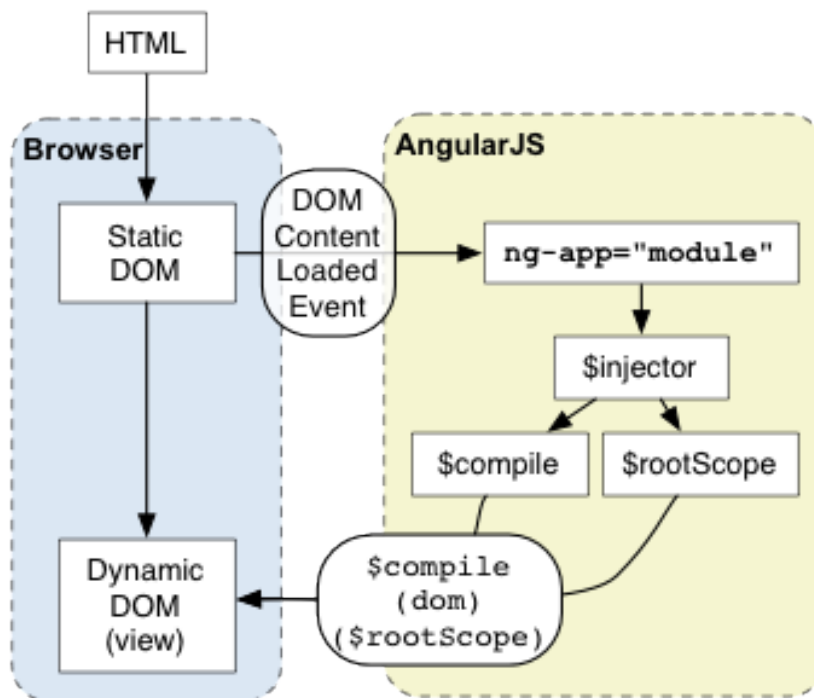


그림 9 AngularJS 애플리케이션이 자동으로 초기화되는 과정¹³

초기화는 시점에 따라 2가지 형태로 나눌 수 있다.

- Automatic Initialization: 페이지가 로딩되면서 자동 실행하도록 설정된 경우. DOMContentLoaded 이벤트가 발생하면 이벤트 핸들러를 통해 실행한다.
- Manual Initialization: 페이지가 로드된 후 수동으로 실행하도록 설정된 경우. document.readyState 속성의 값이 'complete'이면 실행한다.

AngularJS 애플리케이션의 초기화 작업은 다음의 과정을 거쳐 수행된다.

1. 문서 내에서 ng-app 지시자 확인

¹² AtScript 는 transpiler 를 통해 자바스크립트 또는 MS 의 TypeScript, Dart 등으로 컴파일될 수 있다고 한다. <http://sdtimes.com/atscript-google-new-superset-javascript-runtime/>

¹³ <https://docs.angularjs.org/guide/bootstrap>

2. 모듈 및 지시자와 연관된 모듈을 로딩
3. 애플리케이션 injector 생성
4. DOM의 컴파일¹⁴
5. 변경된 DOM의 화면 출력

이처럼 AngularJS는 표준 기술이 아니기 때문에 자체적인 구문을 파싱하는 전처리 과정을 반드시 거쳐야만 한다.

참고

초기화 시 실제 이벤트는 다음과 같이 바인딩된다.

- document 요소에 DOMContentLoaded 이벤트
 - window 객체에 load 이벤트 - DOMContentLoaded 이벤트를 지원하지 않는 브라우저 대상
-

학습 비용

AngularJS를 처음 접하고 사용할 때 학습에 대한 비용을 고려하지 않을 수 없다. 학습에 필요한 시간은 개개인에 따른 편차가 크기 때문에 정량적으로 표현하거나 산출하기 어렵다.

앞서 살펴본 것과 같이 간단한 수준에서의 데이터 출력과 관련된 작업은 몇 시간 내외의 학습만으로 충분할 수도 있지만, AngularJS의 기능을 충분히 활용하려면 수일에서 수개월까지의 시간이 필요할 수도 있다.

중요한 점은 학습에 소요되는 실질적인 시간의 '양'보다, DOM을 제어하는 형태의 개발 방법을 배경 지식으로 가진 개발자가 "AngularJS 방식으로 개발하기 위해 어떻게 해야 할까?"에 대한 것이다.

이에 대해 해외 개발자들의 지식 공유 사이트로 유명한, Stack Overflow(<http://stackoverflow.com/>)에서 많은 사람들의 관심을 끌었던 'jQuery 배경 지식을 가진 경우, 어떻게 AngularJS 방식으로 생각할 수 있을까요?'¹⁵라는 질문 글을 통해 살펴보는 것이 도움이 될 수도 있을 것이다.

해당 질문 글에서 가장 많은 수의 추천을 받은 답변은 다음과 같은 접근법이 필요하다고 언급하고 있다.

- 페이지를 디자인하고 DOM을 제어해 변경하는 방식으로 접근하지 말 것

¹⁴ 컴파일은 DOM 트리를 순회하면서, AngularJS의 지시를 찾아 매칭하고(컴파일 과정), 이후 지시자와 스코프를 결합(Link 과정)해 모델의 변경을 뷰에 출력하는 일련의 처리 작업을 의미한다. 좀더 자세한 내용은 [https://docs.angularjs.org/api/ng/service/\\$compile](https://docs.angularjs.org/api/ng/service/$compile) 과 <http://www.benlesh.com/2013/08/angular-compile-how-it-works-how-to-use.html> 을 참고한다.

¹⁵ "Thinking in AngularJS" if I have a jQuery background? <http://stackoverflow.com/questions/14994391/thinking-in-angularjs-if-i-have-a-jquery-background#15012542>

- DOM 제어 방식의 라이브러리를 사용한 애플리케이션을 구조적으로 만들기 위한 용도로 AngularJS를 사용하지 말 것
- 항상 아키텍처 관점에서 생각할 것
- 항상 테스트 주도 개발(TDD)을 수행할 것
- 지시자는 패키징된 라이브러리(또는 플러그인과 같은)가 아니다.

다음은 AngularJS를 사용해 실제 개발을 진행했던 사용자들이 경험한, 실제 학습에 소요된 기간에 대한 사례이다. 참여한 개발자들의 역량과 서비스의 난이도는 동일 선상에 있지 않음을 염두에 두고, 단지 참고 수준에서 확인하기 바란다.

표 1 경험에 따른 AngularJS 학습 시간 사례

구분	소요 시간/예측치	개발한 서비스 종류
사례 1	6 주	서비스의 관리자 페이지 개발
사례 2	7 일(2 일 학습 + 5 일 개발)	탭이 2 개인 단순 목록 형태 구현
사례 3	수치로 표현하기 어려우며, 다음과 같이 예측 <ul style="list-style-type: none">• 자바스크립트 초급: 비교적 쉽게 접근 가능• 자바스크립트 중/상급: 기존에 사용하던 프레임워크에 익숙해진 상태여서, 적응하는 데 약간의 시간이 필요할 수 있음.	모니터링 시스템 저작 도구 프로모션 페이지 등
사례 4	2 주	콘텐츠와 목록 등을 포함하는 엔드 페이지 구현

AngularJS 사용 효과

개발을 할 때, 생산성, 유연성, 유지보수성 등을 위해 프레임워크를 도입한다. 프론트엔드 영역만 해도 Jindo, jQuery, Backbone, Ember, Polymer, AngularJS 등 다양한 프레임워크가 존재한다. 프레임워크마다 고유의 특징점이 있으므로 각각의 프레임워크를 비교하는 것은 결코 쉬운 일이 아니다. 이를 조금이나마 손쉽게 해결하기 위한 방법으로 TodoMVC 프로젝트(<http://todomvc.com>)¹⁶가 있다. 이 프로젝트는 각각의 프레임워크를 이용하여 '할일 관리'라는 간단한 기능을 SPA(Single Page Application)¹⁷ 형태로 구현한다.

이 장에서는 DOM 제어 방식의 Jindo 프레임워크와 MVC 방식의 AngularJS를 이용하여 '할일 관리' 기능을 구현해 보고, 실제 구현 코드를 비교하여 개발 생산성과 유연성, 유지보수성 측면에서 어떤 차이가 있는지 살펴본다.

개발 방식 비교

DOM 제어 방식

Jindo나 jQuery는 화면에서 사용하는 요소(DOM)를 직접 제어하기 때문에, 다음과 같이 화면에 표현되는 요소를 각각 추출한 후, 그에 따른 이벤트 핸들러를 추가하는 방식으로 개발한다. 또한, 동적인 화면 구성을 위해, 변경되는 부분에 대해 기본 마크업 외에 별도의 템플릿이 필요하다.

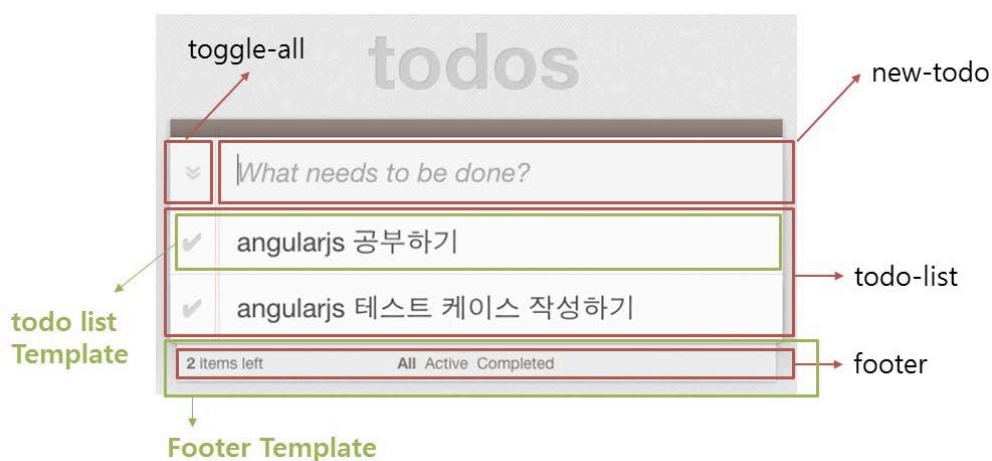


그림 10 DOM 제어 방식의 개발 방식

¹⁶ TodoMVC 프로젝트의 성능을 다양한 MVC 프레임워크를 사용해 비교한 결과를 다음의 링크에서 직접 확인해 볼 수 있다.

<http://dsuket.github.io/todomvc-perf-comparison/>

¹⁷ SPA(Single Page Application)는 페이지 전환 방식의 일반 웹페이지보다 더 쾌적한 사용자 경험을 제공한다. 이에 대한 자세한 내용은 다음 링크를 참조한다. http://en.wikipedia.org/wiki/Single-page_application

개발자는 화면의 요소에 대한 정보를 알아야 하며, 각 요소들의 이벤트 핸들러를 별도로 작성하여야 한다. 이벤트 핸들러에서는 사용자가 입력한 정보를 바탕으로 로컬 스토리지를 동기화하여야 하며, 동기화된 정보를 다시 화면에 동기화해야 한다. 또한, 화면 정보를 동기화하기 위해 별도의 템플릿을 작성하여야 하며, 이를 이용하여 변경된 DOM의 정보를 변경한다.

입력

사용자의 입력이 발생하면 그에 따라 이벤트가 발생하게 되고, 개발자는 그 이벤트가 발생했을 때 어떻게 동작하는지를 구현한다. 개발자는 앱의 todos 데이터를 갱신하고, 갱신된 정보를 로컬 스토리지에 반영하고, 템플릿을 이용하여 변경된 정보를 화면에 표현하는 기능을 구현한다.

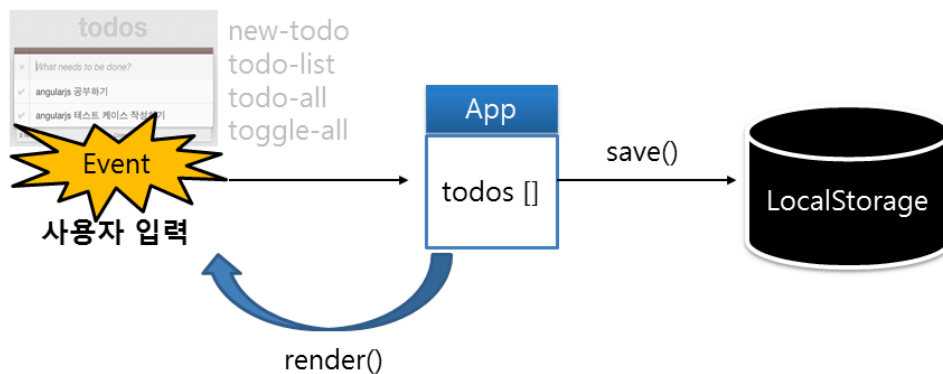


그림 11 DOM 제어 방식의 데이터 입력 절차

출력

개발자는 로컬 스토리지에서 todos 정보를 가지고 와서 앱의 todos 데이터를 갱신하고, 템플릿을 이용하여 갱신된 정보를 화면에 표현하도록 구현한다.

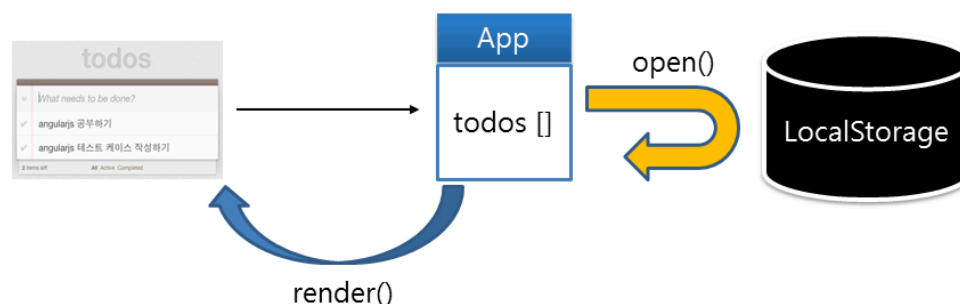


그림 12 DOM 제어 방식의 데이터 출력 절차

AngularJS(MVC 방식)

AngularJS는 화면을 표현하는 마크업을 템플릿으로 사용하며, 이 템플릿에 지시자(directive)로 기술하여 동적인 서비스 페이지를 구성한다.

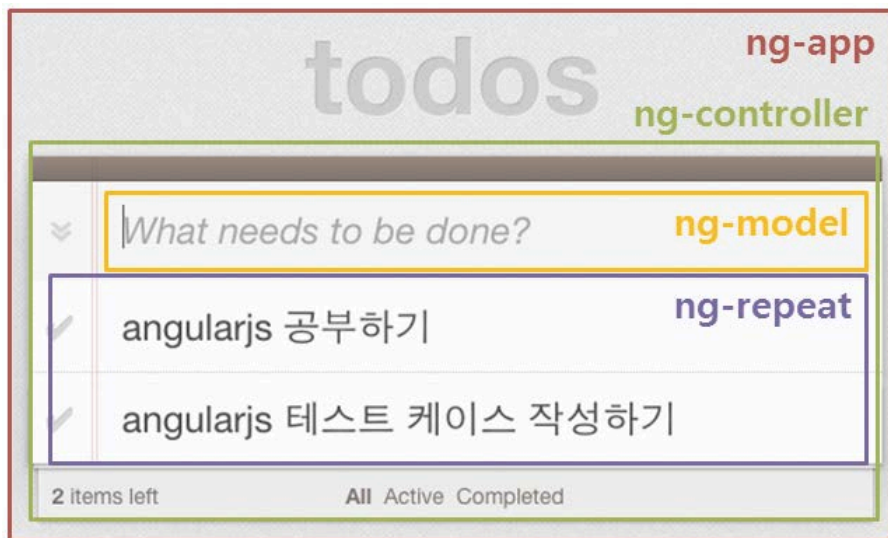


그림 13 AngularJS의 개발 방식

개발자는 템플릿을 작성하고, 템플릿의 요소에 적당한 지시자를 지정하고, 서비스 모듈(Module)을 구현한다. 서비스 모듈 구현 시 뷰(View)에서 발생한 이벤트를 감지할 컨트롤러(Controller)를 구현하고, 컨트롤러에서는 감지된 결과를 모델(Model)에 반영하는 기능을 구현한다. AngularJS의 양방향 데이터 바인딩에 의해 반영된 데이터는 자동으로 다시 뷰에 반영된다.

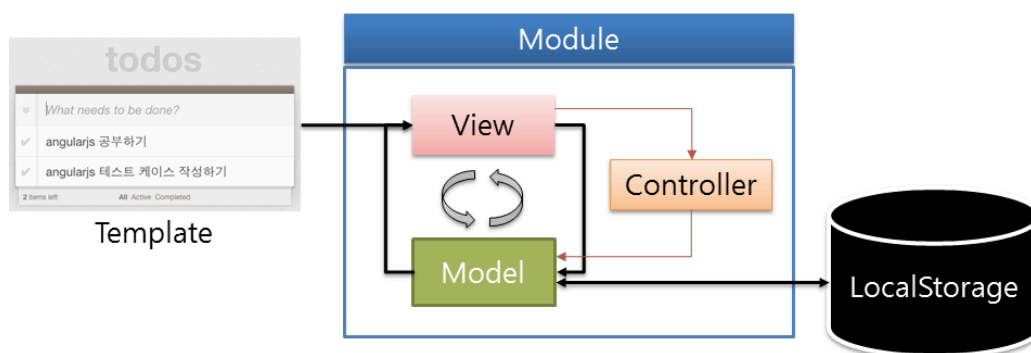


그림 14 AngularJS의 데이터 입출력 절차

정리

Jindo나 jQuery와 같은 'DOM 제어 방식의 개발'은 개발자가 마크업과 템플릿을 작성하여야 하며, 이를 바탕으로 제어할 각각의 요소를 알아야 한다. 또한 각각의 요소에 처리할 이벤트 핸들러를 작성하여야 한다. 각 핸들러에서는 데이터를 갱신하고 템플릿을 이용하여 갱신된 데이터를 화면 요소(DOM)에 반영하는 작업을 수행해야 한다.

반면, AngularJS는 개발자가 마크업을 작성하고, 마크업에 적당한 지시자를 선언한 후 모듈을 작성하기만 하면 서비스를 만들 수 있다. 개발의 난이도는 차이가 있을 수 있지만, 실제 개발자가 신경 써야 할 부분은 AngularJS가 정한 흐름 내로 한정되기 때문에, 서비스 구현에 더 충실할 수 있다.

구현 코드 비교

구체적으로 DOM 제어 기반의 Jindo와 MVC 방식의 AngularJS로 작성된 코드를 자세히 살펴해보도록 한다.

마크업

실제 화면에 표현되는 마크업이 Jindo 기반과 AngularJS 기반에서 어떻게 다른지 구체적으로 살펴본다.

Jindo

Jindo 기반의 마크업은 화면에 표현될 HTML과 동적으로 변경될 부분에 대한 별도의 템플릿으로 구성되어 있다. Jindo 기반의 '할일 관리'의 기본적인 마크업은 다음과 같다.

- 뷰(View)

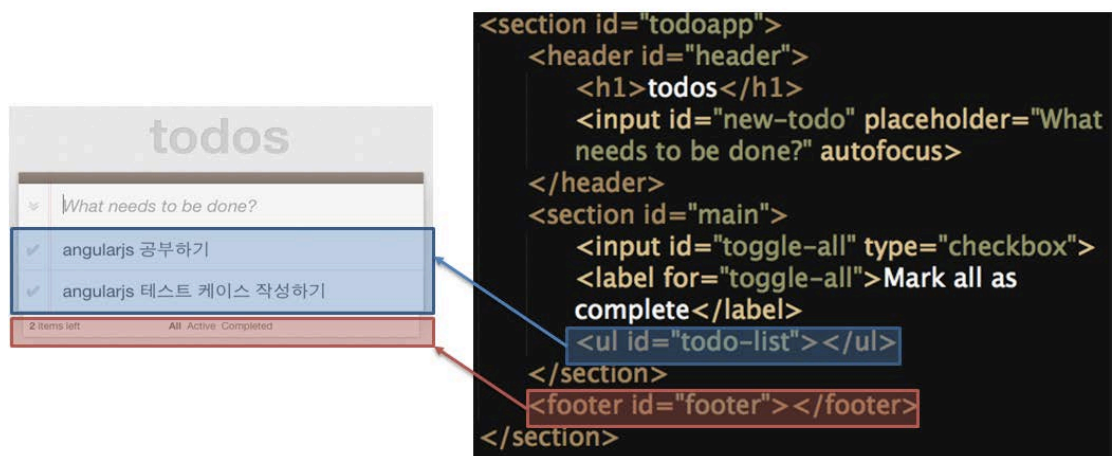


그림 15 Jindo의 뷰 영역

파란색 영역과 붉은색 영역은 데이터에 의해 동적으로 변경될 부분이다. 따라서, 이 부분에 대한 별도의 템플릿이 있어야 한다.

- 템플릿(Template)

파란색 배경의 영역에 해당되는 템플릿은 다음과 같다.

```

<script id="todo-template" type="text/template">
  {for todo in todos}
  <li {if todo.completed}class="completed"{/if} data-id="{=todo.id}">
    <div class="view">
      <input class="toggle" type="checkbox" {if todo.completed}checked{/if}>
      <label>{=todo.title}</label>
      <button class="destroy"></button>
    </div>
    <input class="edit" value="{=todo.title}">
  </li>
</script>

```

```
{/for}
</script>
```

붉은색 배경의 영역에 해당되는 템플릿은 다음과 같다.

```
<script id="footer-template" type="text/template">
<span id="todo-count"><strong>{=activeTodoCount}</strong> {=activeTodoWord}
left</span>
<ul id="filters">
  <li>
    <a {if filter == 'all'}class="selected"{/if} href="#/all">All</a>
  </li>
  <li>
    <a {if filter
== 'active'}class="selected"{/if} href="#/active">Active</a>
  </li>
  <li>
    <a {if filter
== 'completed'}class="selected"{/if} href="#/completed">Completed</a>
  </li>
</ul>
{if completedTodos}<button id="clear-completed">Clear completed
({=completedTodos})</button>{/if}
</script>
```

AngularJS

AngularJS 기반의 마크업은 화면에 표현될 HTML 자체가 뷰이자 템플릿이다.



그림 16 AngularJS의 뷰와 템플릿

AngularJS 기반의 '할일 관리'의 기본적인 마크업은 다음과 같다.

- 뷰(View) == 템플릿(Template)

```
<section id="todoapp" ng-app="todomvc" ng-controller="TodoCtrl">
<header id="header">
  <h1>todos</h1>
  <form id="todo-form" ng-submit="addTodo()">
    <input id="new-todo" placeholder="What needs to be done?" ng-
model="newTodo" autofocus>
  </form>
</header>
<section id="main" ng-show="todos.length" ng-cloak>
```

```

        <input id="toggle-all" type="checkbox" ng-model="allChecked" ng-
click="markAll(allChecked)">
        <label for="toggle-all">Mark all as complete</label>
        <ul id="todo-list">
            <li ng-repeat="todo in todos | filter:statusFilter track by $index" ng-
class="{completed: todo.completed, editing: todo == editedTodo}">
                <div class="view">
                    <input class="toggle" type="checkbox" ng-
model="todo.completed" ng-change="todoCompleted(todo)">
                    <label ng-
dblclick="editTodo(todo)">{{todo.title}}</label>
                    <button class="destroy" ng-
click="removeTodo(todo)"></button>
                </div>
                <form ng-submit="doneEditing(todo)">
                    <input class="edit" ng-trim="false" ng-
model="todo.title" ng-blur="doneEditing(todo)" todo-escape="revertEditing(todo)" todo-
focus="todo == editedTodo">
                </form>
            </li>
        </ul>
    </section>
    <footer id="footer" ng-show="todos.length" ng-cloak>
        <span id="todo-count"><strong>{{remainingCount}}</strong>
        <ng-pluralize count="remainingCount" when="{ one: 'item left', other:
'items left' }"></ng-pluralize>
        </span>
        <ul id="filters">
            <li>
                <a ng-class="{selected: location.path() == '/'}" "
href="#/">All</a>
            </li>
            <li>
                <a ng-class="{selected: location.path() == '/active'}"
href="#/active">Active</a>
            </li>
            <li>
                <a ng-class="{selected: location.path() == '/completed'}"
href="#/completed">Completed</a>
            </li>
        </ul>
        <button id="clear-completed" ng-click="clearCompletedTodos()" ng-
show="remainingCount < todos.length">Clear completed ({{todos.length -
remainingCount}})</button>
    </footer>
</section>
<footer id="info">
<p>Double-click to edit a todo</p>
<p>Credits:
    <a href="http://twitter.com/cburgdorf">Christoph Burgdorf</a>,
    <a href="http://ericbidelman.com">Eric Bidelman</a>,
    <a href="http://jacobmumm.com">Jacob Mumm</a> and
    <a href="http://igorminar.com">Igor Minar</a>
</p>
<p>Part of <a href="http://todomvc.com">TodoMVC</a></p>
</footer>

```

Jindo의 마크업과 유사하나, ng-XXX와 같은 지시자가 마크업에 표기되어 뷰와 템플릿 역할을 함께 하고 있다.

자바스크립트

비즈니스 로직이 존재하는 애플리케이션 영역은 Jindo 기반과 AngularJS 기반에서 어떻게 다른지 구체적으로 살펴본다.

Jindo

DOM 제어 기반의 Jindo 프레임워크는 마크업 요소(DOM)와 템플릿을 인지하고, 인지된 마크업 요소(DOM)에 이벤트 핸들러를 할당하는 작업이 필요하다.

- **마크업 요소의 인지**

아래 코드는 템플릿 요소를 확인하고, 마크업 요소(DOM)를 저장하는 코드이다.

```
cacheElements: function () {
    this.todoTemplate = jindo.$Template("todo-template");
    this.footerTemplate = jindo.$Template("footer-template");

    var $todoApp = jindo.$Element('todoapp');
    var $header = $todoApp.query('#header');

    this.$main = $todoApp.query('#main');
    this.$footer = $todoApp.query('#footer');
    this.$newTodo = $header.query('#new-todo');
    this.$toggleAll = this.$main.query('#toggle-all');
    this.$todoList = this.$main.query('#todo-list');
},
```

마크업에서 선언한 id(예. todoapp, #header, #main, #footer, #new-todo, #toggle-all, #todo-list)로 자바스크립트 코드를 구현하고 있기 때문에, 마크업이 변경되면 자바스크립트 코드도 함께 수정되어야 하는 문제가 있다.

- **이벤트 핸들러의 할당**

마크업 요소에 맞는 이벤트 핸들러를 할당하는 코드이다.

```
bindEvents: function () {
    var list = this.$todoList;
    this.$newTodo.attach('keyup', jindo.$Fn(this.create,this).bind());
    this.$toggleAll.attach('change', jindo.$Fn(this.toggleAll,this).bind());
    this.$footer.attach('click#clear-completed',
jindo.$Fn(this.destroyCompleted,this).bind());
    list.attach({
        'change@.toggle' : jindo.$Fn(this.toggle,this).bind(),
        'dblclick@label' : jindo.$Fn(this.edit,this).bind(),
        'keyup@.edit' : jindo.$Fn(this.editKeyup,this).bind(),
        'focusout@.edit' : jindo.$Fn(this.update,this).bind(),
        'click@.destroy' : jindo.$Fn(this.destroy,this).bind()
    });
},
```

각 이벤트 요소를 할당하는 경우에도 마크업에 명시된 class명(예: toggle, .edit, .destroy)이나 태그명(예: Label)이 주어지기 때문에, 마크업이 변경되면 자바스크립트 코드도 함께 수정되어야 하는 문제가 있다.

- **데이터의 변경, 템플릿을 이용하여 화면 반영**

사용자 입력에 의해 변경된 데이터를 변경하고, 템플릿을 이용하여 변경된 내용을 화면에 반영한다.

```
update: function (e) {
    // ...

    // todos 데이터 변경.
    if (val) {
        this.todos[i].title = val;
    } else {
        this.todos.splice(i, 1);
    }

    // 화면 갱신
    this.render();
},
```

```
render: function () {
    // 템플릿을 이용하여 화면 반영
    this.$todoList.html(this.todoTemplate.process({todos : todos}));
    // ...
}
```

사용자 이벤트에 의해 update 메서드가 호출되고, update에서는 todos 데이터를 변경하고, 화면 반영을 위해 render 메서드를 호출한다. Render 메서드에서는 todoTemplate를 이용하여 화면을 갱신한다.

AngularJS

MVC 기반의 AngularJS 프레임워크는 기본적으로 모델, 컨트롤러를 관리하는 모듈을 만든다. 템플릿에 의해 자동으로 만들어진 뷰에서 사용자의 액션이 발생할 경우, 처리할 컨트롤러를 만들고, \$scope에 모델을 만든다.

• 모듈 만들기

모듈은 컨트롤러, 서비스, 지시자를 포함하는 단위이다. 다음은 모듈을 이용하여 컨트롤러를 만드는 코드이다.

```
angular.module('todomvc', []).controller('TodoCtrl', function TodoCtrl($scope,
    $location, $filter) {
    //
}
```

모듈의 이름은 'todomvc'이고, 이에 포함된 컨트롤러는 'TodoCtrl'이다.

• \$scope에 모델 만들기

\$scope에 모델을 추가한다. 또한 추가한 모델의 데이터가 변경되었을 경우, 자동으로 뷰에 반영할 수 있도록 대상 모델을 scope.\$watch에 등록한다.

```
// todos, newTodo, editedTodo 모델
$scope.todos = [];
$scope.newTodo = '';
$scope.editedTodo = null;

$scope.$watch('todos', function () {
    // todo 모델이 변경되었을 경우, 로컬 스토리지에 반영한다.
    localStorage.setItem('todos-angularjs-perf', JSON.stringify(todos));
}, true);
```

todos 값이 변경되었을 경우, todos의 값에 해당하는 뷰가 자동으로 바뀌고, 로컬 스토리지 (LocalStorage)에 todos 데이터를 저장한다.

• 서비스 지시자 만들기

구현하고자 하는 기능 중, AngularJS에서 기본적으로 제공하는 지시자로 구현할 수 없을 경우에는 서비스에서 별도의 지시자를 만들어야 한다. AngularJS는 개발자에게 주어지는 자유도가 상대적으로 높지 않기 때문에, 기존 'DOM 제어 방식' 개발에 익숙한 개발자라면 이 부분이 상당한 아쉬움으로 남을 수 밖에 없다.

정리

개발 생산성

Jindo의 개발 코드와 AngularJS의 개발 코드 크기를 비교해 보면 아래와 같다.

표 2 Jindo 와 AngularJS의 개발 코드 크기 비교

구분	종류	개발 코드 크기(byte)	총 크기(byte)
Jindo	마크업	1,164(100%)	4,951(100.00%)
	자바스크립트	3,787(100%)	
AngularJS	마크업	2,059(176.89%)	3,161(63.84%)
	자바스크립트	1,102(29.09%)	

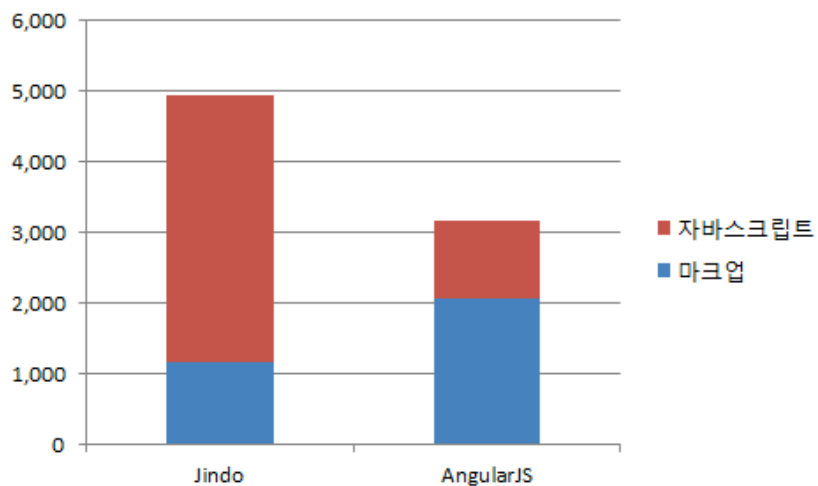


그림 17 Jindo와 AngularJS의 개발 코드 크기

마크업은 AngularJS 기반이 Jindo 기반에 비해 약 77% 정도 코드량이 증가하지만, 자바스크립트 코드는 AngularJS 기반일 때 Jindo 기반 코드에 비해 약 71% 정도 코드량이 감소했다. 총 코드 크기를 비교해 보면, 똑같은 기능을 구현할 때 AngularJS가 Jindo 기반보다 약 36% 정도 적은 양의 코드로 개발할 수 있다는 것을 알 수 있다.

유연성 및 유지보수성

DOM을 직접 제어하는 Jindo 프레임워크와는 다르게 AngularJS는 마크업과 자바스크립트 사이에 연관성이 거의 존재하지 않는다. 존재한다고 해도, 마크업과 자바스크립트를 연결하는 ng-controller 속성 정도이다. AngularJS는 DOM이 변경되더라도 직접적으로 자바스크립트에 영향을 주지 않기 때문에, 화면 구성의 변화가 자유로우며, 모듈별로 독립성을 가질 수 있다. 모듈이 독립성을 가질 수 있다는 것은 코드의 재사용률이 더 높아질 수 있고 테스트 코드를 쉽게 작성할 수 있다는 뜻이다.

AngularJS 사용 시 고려 사항

앞에서 살펴 봤듯이 AngularJS를 이용하여 개발하면 많은 이점이 있다. 적은 코드로 서비스를 손쉽게 만들 수 있기 때문에 개발 생산성이 좋다. 뿐만 아니라, 뷰 영역과 비즈니스 로직이 명확히 분리되어 있어서 코드의 재사용성이 좋고 테스트도 쉽다. 하지만 AngularJS를 사용할 때 개발자가 그 내부 원리를 잘 이해하지 못하고 무심코 사용하면 많은 문제점에 직면하게 된다. 이 장에서는 이러한 문제를 조금이나마 피하고자 AngularJS의 내부를 살펴보도록 한다.

양방향 데이터 바인딩

AngularJS의 최대 장점 중의 하나인 양방향 데이터 바인딩 기능은 개발자에게 많은 이점을 준다. 데이터가 변경되어 화면에 반영해야 하거나 사용자의 액션에 의해 데이터를 변경해야 할 때, 프레임워크가 자동으로 화면과 데이터를 동기화해주면 개발자는 비즈니스 로직에 더 집중할 수 있게 된다.

AngularJS에서는 이러한 기능을 구현하기 위해 Dirty Checking을 한다. 이 방식은 바인딩되는 모든 변수에 대해 특정 시점에 예전 값과 현재 값을 비교하여 값이 변경되었으면 DOM을 갱신한다.

Dirty Checking의 원리를 이해한 후, 잘못된 사용으로 발생할 수 있는 성능 문제를 확인해 보기로 한다.

대상 추출

Dirty Checking의 대상은 템플릿에서 추출한다. AngularJS는 템플릿에 명시된 지시자를 컴파일하고 링크 작업을 진행할 때, 해당 스코프(scope)의 `$scope.$watch`를 이용하여 Dirty Checking할 대상을 등록한다. 또는 개발자가 직접 `$scope.$watch`를 이용해 등록할 수 있다. 등록된 대상은 해당 스코프의 `$scope.$$watchers`를 통해 확인할 수 있다.

확인 시점

흔히 데이터의 변경 시점을 확인하기 위해서 주기적으로 변경 여부를 확인할 것으로 생각하겠지만, AngularJS는 좀 더 효과적인 전략을 사용한다. 데이터에 값이 설정되어 변경되는 시점에 Dirty Checking을 수행하는 것이다. 데이터를 변경하려면 먼저 자바스크립트가 수행되어야 한다. 구체적으로는 다음의 4 가지 상황 밖에 없다.

- HTML 요소의 이벤트 핸들러가 작동할 때
- Ajax 통신의 결과가 도착했을 때
- URL Hash값이 변경되었을 때
- timer에 의해 발생된 tick 이벤트의 핸들러가 작동할 때

이를 실제 AngularJS의 동작들로 살펴보면 다음과 같다.

- DOM 이벤트(`ng-click`, `ng-mousedown`, `ng-change`, `ng-checked` 등)가 발생한 후
- `$http`와 `$resource`에서 응답이 돌아왔을 때
- `$location`에서 URL을 변경한 후
- `$timeout` 이벤트가 발생한 후

AngularJS에서는 위의 동작이 발생할 때 Dirty Checking이 수행된다. 이외에도 `$scope.$apply()`나 `$scope.$digest()`가 호출될 때 Dirty Checking을 한다. 즉, AngularJS의 모듈을 사용하지 않고 데이터가 변경될 때는 Dirty Checking이 동작하지 않는다는 뜻이다. 따라서, 예기치 않은 동작을 방지하기 위해서라도 AngularJS로 개발할 때는 꼭 AngularJS 모듈을 사용해야 한다.

확인 방법

Dirty Checking 작업은 실제 `$scope.$digest()`에 의해 수행된다. 해당 스코프와 자식 스코프를 차례로 검색해서, `$scope.$watcher`에 등록된 모든 변수의 변경 여부를 검사한다. 변경 대상은 `$watch` 등록 시 전달한 표현식을 바탕으로 추출되어, 이전과 이후 값을 비교한다. 이 과정을 'digest 루프'라고 부른다.

이를 그림으로 표현하면 다음과 같다.

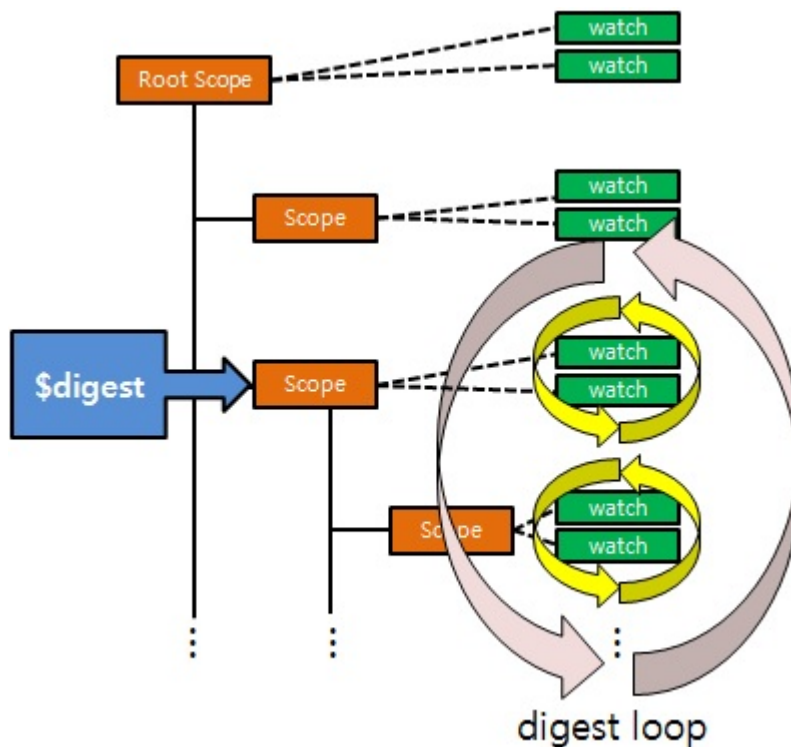


그림 18 digest 루프

여기서 우리가 알아야 할 것은, digest 루프는 `$scope.digest()`를 딱 한 번 호출하여도, 한 번만 돌지는 않는다는 점이다. `$scope.$digest()`가 한 번 호출되면 digest 루프가 최소 2번 호출된다. 이를 이해하기 위해 대상을 등록하는 `$scope.$watch` 함수를 자세히 살펴보자.

`$scope.$watch`의 함수 원형은 다음과 같다.

```
$watch(watchExpression, listener, [objectEquality]);
```

- 첫 번째 인자 `watchExpression`은 digest 루프 대상을 판단하는 watch 표현식으로, string이 될 수 있고 function 타입이 될 수 있다.
- 핸들러 함수의 첫 번째 인자에 의해 평가된 값이 이전과 다르면 두 번째 인자 `listener`의 핸들러 함수가 호출된다.

digest 루프가 돌 때 바인드된 대상별로 첫 번째 인자의 watch 표현식이 호출되고, 이 값에 의해 추출된 대상의 이전 값과 이후 값이 다르면 두 번째 인자의 핸들러 함수가 호출된다. 만약, digest 루프가 돌 때 watch의 핸들러 함수에 의해 `$scope`의 데이터가 변경되었다면, 다시 한 번 데이터의 변경 여부를 반영해야 하므로 최소 2번의 digest 루프가 호출된다.

AngularJS에서는 digest 루프가 watch 핸들러 함수의 반복된 `$scope` 데이터 변경으로 무한루프에 빠질 수 있기 때문에, 루프 호출 횟수에 10번의 제한을 두고 있다. 참고로, 이 값은 `$rootScopeProvider`의 `digestTtl(limit)` 함수를 통해 조정할 수 있다.

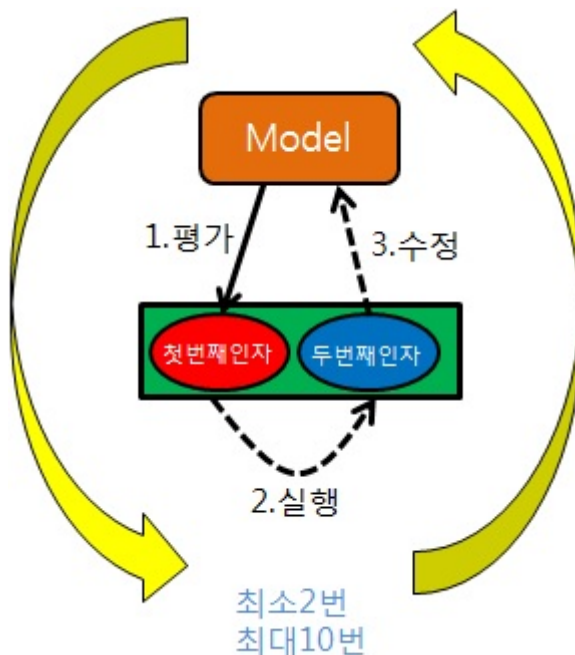


그림 19 Digest 루프 호출 횟수: 최소 2번, 최대 10번

`$scope.$digest`

실제 digest 루프를 실행하는 AngularJS의 `$scope.$digest` 코드는 다음과 같은 형태로 되어 있다.

```
do {
  dirty = false;
  current = target; // scope;
```

```
// scope를 확인
do {
  // watchers 대상이 존재할 경우 진행
  if ((watchers = current.$$watchers)) {
    length = watchers.length;

    // Scope별 검사
    while (length--) {
      watch = watchers[length];
      // watch 대상 값 변경 여부 검사
      if ((value = watch.get(current)) !== (last = watch.last) &&
          !(watch.eq ? equals(value, last) : (
            typeof value === 'number' &&
            typeof last === 'number' &&
            isNaN(value) && isNaN(last)
          ))) {
        dirty = true; // 변경 시 dirty 체크
        //...
      }
    }
  }
} while ((current = next)); // 다음 Scope로 이동
} while (dirty); // dirty가 체크되었을 경우, 한 번 더 digest 루프
```

\$scope.\$digest는 중첩 do~while으로 구성되어 있다. 첫 번째 do~while문에서는 스코프를 탐색하고, 두 번째 do~while문에서는 스코프 내의 바인딩 대상(watch)을 검사한다. 바인딩 대상이 변경되었으면 dirty를 체크하여 dirty가 없을 때까지 확인한다.

참고

실제 AngularJS 코드에서는 이보다 더 많은 조건들을 확인해서 스코프에 접근하고 바인딩 변수를 확인하지만, 여기서는 개념 이해를 돕기 위해 구조 형태로 기술하였다.

고려 사항

위에서 살펴본 바와 같이, digest 루프는 기본적으로 해당 스코프에 자식 스코프가 많이 있을수록, 또는 각 스코프의 watch 대상이 많을수록 느려진다. 뿐만 아니라, \$scope.\$watch 등록 시, watch 표현식은 현재 값의 변경 여부와 상관 없이 검사 시 매번 호출되기 때문에 가급적 가볍게 구성하고, 핸들러에서는 바인드된 데이터를 변경하지 않는다.

이를 간단한 수식으로 나타내면 다음과 같다.

- **스코프당 수행 시간 = 바인드 변수값 비교 수행 시간 * watch 표현식 수행 시간**
- 자신을 포함한 자식 스코프의 총 개수가 n개이면,
→ **한 번의 Digest 루프의 수행 시간 = 스코프당 수행 시간 * n**
- Digest 루프 호출 횟수는 최소 2에서 최대 10번이므로,
→ **총 수행 시간 = 한 번의 Digest 루프 수행 시간 * (2~10)**

실제 간단한 예를 들어 실제 루프의 수행 시간을 계산해 보자. 만약, 대상 스코프에 바인드된 변수가 100개이고 그 하위로 10개의 스코프가 있으며 각 스코프의 바인딩 변수가 50개라면, 바인드된 변수의 이전, 이후 값을 비교하는 횟수는 $(1 * 100) + (10 * 50) = 600$ 번이다. 이때, 이전, 이후 값 비교 시간이

1ms라면 총 600ms가 걸린다. 그런데 digest 루프는 기본적으로 2번 호출되니 $600\text{ms} \times 2 = 1,200\text{ms}$ 가 된다.

이뿐만이 아니다. 만약 watch 표현식을 확인하는 데 10ms가 걸린다면, 대략 수행되는 시간은 $1,200 \times 10\text{ms} = 12\text{초}$ 이다. 최악의 경우, watch의 핸들러에서 바인드된 값을 변경한다면, 최대 10번의 digest 루프가 돌아 총 $12\text{초} \times 10 = 120\text{초}$ 가 걸리게 된다. 물론, 이는 최악의 상황에 대한 예이지만, 이러한 상황을 피하기 위해서는 적어도, 다음 사항을 주의하여야 한다.

- \$scope의 바인딩 변수의 수는 가능한 한 줄인다.
- \$scope의 하위 \$scope 개수를 가능한 한 줄인다.
- \$scope.\$watch 등록 시, 가급적이면 watch 표현식을 가볍게 사용한다.
- \$scope.\$watch 등록 시, 가급적이면 핸들러에서 바인드된 데이터를 변경하지 않는다.

참고

AngularJS 2.0부터는 dirty checking이 Object.Observer를 이용하는 형태로 변경될 예정이다.

Object.Observer로 변경되면 특정 시점이 아닌 실제 데이터의 값이 변경될 때 이벤트를 통해 알 수 있기 때문에, 훨씬 효과적으로 데이터 변경 여부를 확인할 수 있다.

AngularJS 2.0에 대한 자세한 내용은 다음 링크를 참조하기 바란다.

<http://angularjs.blogspot.kr/2014/03/angular-20.html>

양방향 데이터 바인딩 성능 비교

HTML 페이지를 구성하는 방식은 크게 두 가지다. 하나는 서버에서 데이터를 가공하여 HTML을 구성하고 페이지를 로드하는 방식이고, 또 하나는 페이지를 로드한 후 자바스크립트를 통해 데이터를 구성하고 HTML을 가공하여 영역에 적용하는 방식이다. 기본적으로 서버로부터 HTML이 갖춰진 상태의 페이지를 로드하는 것과 로드 후 자바스크립트를 통해 HTML을 가공해 화면에 노출하는 방식 두 가지를 비교했을 때 어느 쪽이 페이지를 구성하는 속도가 더 빠를까?

또한 페이지 로드가 완료되었다고 가정하고 SPA(Single Page Application) 페이지에서 사용자의 동작에 의해 데이터를 구성하고 화면에 적용할 때 양방향 데이터 바인딩(two-way binding)과 단방향 데이터 바인딩(one-way binding)의 차이가 성능과 속도 측면에서 어떻게 다른지 확인해 보자.

성능 비교 데이터

우선 비교하려는 데이터는 비교 대상이 서로 같아야 하므로 네이버 뉴스에서 사용하는 기본적인 구조에 목록만 변경하여 처리되는 시간을 비교하고자 한다. 즉, 뉴스 서비스의 기사 목록을 제거하고 이곳에 목록을 변경하는 형태로 진행하고자 한다. HTML 마크업은 아래와 같다.

```
<ul>
  <li class="type_a">
    <a href="http://m.news.naver.com">
      목록 0
      <span class="by">디지털타임스 <span>오후 1:15</span>
    </span>
  </li>
</ul>
```

```

    </a>
  </li>

  ... 중략 ...
</ul>

```

ul > li 목록 형태로 구성되어 있으며, 목록 데이터를 100개에서 2,000개까지 100개 간격으로 구성하여 화면에 적용되는 시간을 확인해 보자.

서버로드 방식 대 AngularJS 방식

서버로드 방식은 말 그대로 서버로부터 HTML이 구성되고 이를 그대로 페이지에 로드하여 뷰 영역이 처리되는 방식이다. 반면, AngularJS 방식은 데이터가 자바스크립트 영역에 정의되어 있고 템플릿을 통해 스크립트로 HTML을 가공하여 화면에 적용하는 방식을 말한다.

테스트 시 초기 로드 아이템 개수에 차이를 두었는데, 100, 200, 300순으로 100개씩 증가하는 방식으로 2,000개까지 로드 속도를 비교해 봤다. 개수에 따라 총 10번을 시도했고 그에 따른 평균값 그래프는 다음과 같다.

서버로드 방식

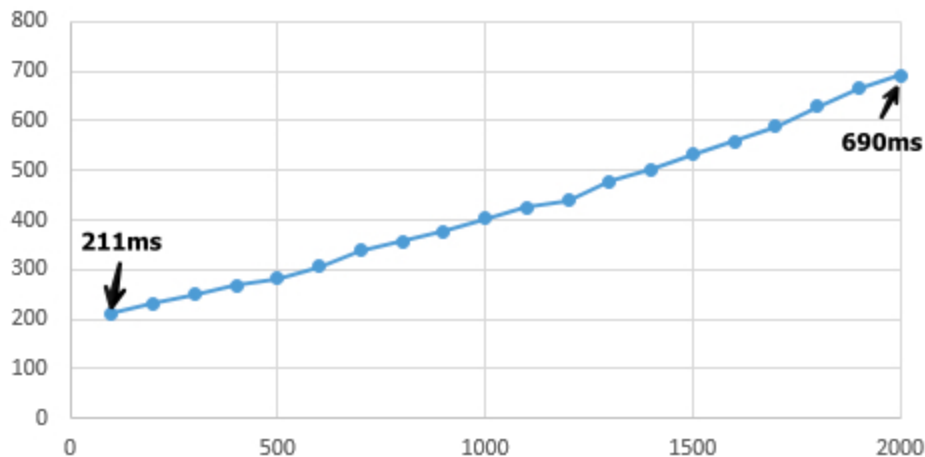


그림 20 서버 로드 시간(단위: ms)

100개의 기본 목록을 로드할 때 평균 약 211ms가 소요되었다. 목록의 개수가 늘어남에 따라 로딩 속도도 점차 늘어났으며, 2,000개의 목록을 로드할 때는 약 690ms가 소요되는 것을 확인하였다. 즉, 뷰 영역에서 처리하는 데이터 양에 따라 소요 시간도 비례하여 증가하는 것을 알 수 있다.

AngularJS 방식

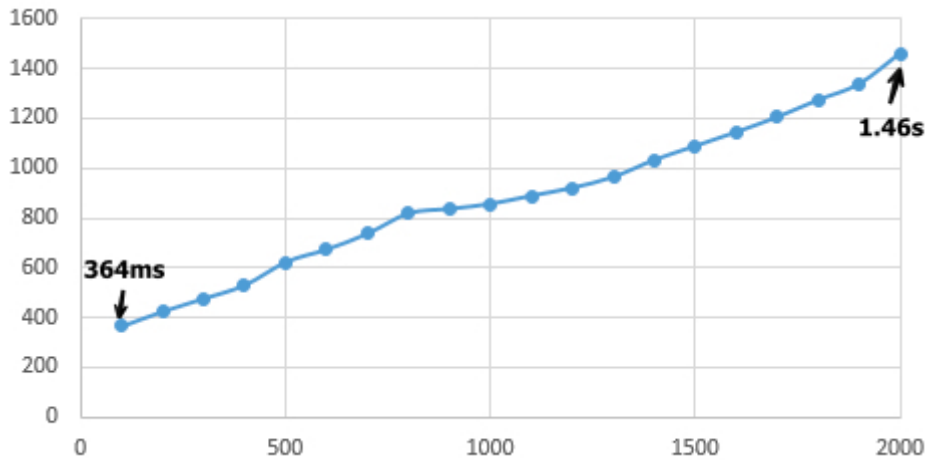


그림 21 AngularJS 로드 시간(단위 ms)

AngularJS 방식에서도 서버 로드 방식과 같이 로드 시간이 순차적으로 증가했다. 하지만 목록 100개의 로드 시간을 비교해보면 서버 로드 방식은 211ms인 반면 AngularJS 방식은 약 364ms 정도로, AngularJS 방식이 목록을 처리하는 데 150ms 정도 더 소요됐다. 또한 2,000개의 목록을 로드하는 시간은 약 1.46s로, 서버로드 방식보다 2배 이상의 시간이 소요되는 것을 확인할 수 있다.

단방향 대 양방향 데이터 바인딩

AngularJS에서는 양방향 바인딩 처리를 하고 있어 데이터가 변경되면 화면에 변경된 내용을 반영해주는 동작이 처리된다. 이는 기존 자바스크립트를 통해 데이터를 가공해 화면에 개발자가 직접 반영하는 동작 없이도 알아서 템플릿에 맞게 데이터가 적용되므로 편리하다. 그렇다면 양방향 바인딩과 단방향 바인딩 형태의 속도를 비교해 앞서 비교한 방식과 같이 100개에서 2,000개까지 100개의 간격으로 이루어진 데이터의 내용이 변경될 때 처리되는 시간을 확인해 보도록 하자.

마찬가지로 10번 시도를 통해 평균값을 냈다.

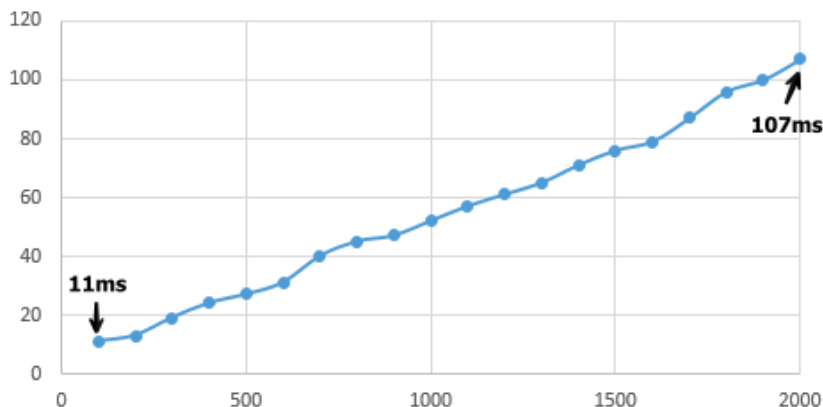


그림 22 단방향 바인딩 처리 시간(단위 ms)

위의 그림은 단방향 바인딩의 데이터를 처리하고 화면에 그려지는 시간을 그래프로 표현한 것이다. 데이터가 늘어남에 따라 처리하는 시간도 비례하여 상승하는 모습을 나타내고 있다. 데이터가 100개일 때 약 11ms 정도이고 2,000개일 때 약 107ms 정도 소요되는 것을 확인할 수 있다.

양방향 바인딩의 경우에 대해서도 알아보자.

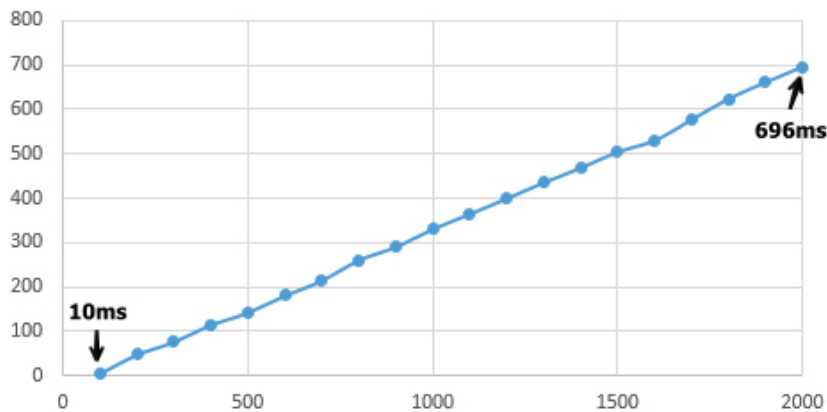


그림 23 양방향 바인딩 처리 시간(단위 ms)

양방향 바인딩의 경우에도 단방향과 마찬가지로 데이터의 양과 처리 시간은 비례한다. 앞의 결과와 마찬가지로 많은 양의 데이터를 처리하는 데 소요되는 시간은 오래 걸린다. 데이터가 100개일 때 10ms 정도 소요되고 2,000개일 때 약 696ms 정도 소요되는 것으로 나타났다.

고려 사항

서버로드 방식과 AngularJS 방식에 대한 비교와 단방향과 양방향 데이터 바인딩에 대한 두 가지 비교 결과를 확인해 보았다. 한눈에 알아보기 쉽도록 비교했던 결과를 하나의 차트를 통해 확인해 보도록 한다.

앞서 살펴봤던 서버로드 방식과 AngularJS 방식 두 개의 데이터를 하나의 차트로 표현하면 다음과 같이 정리할 수 있다.

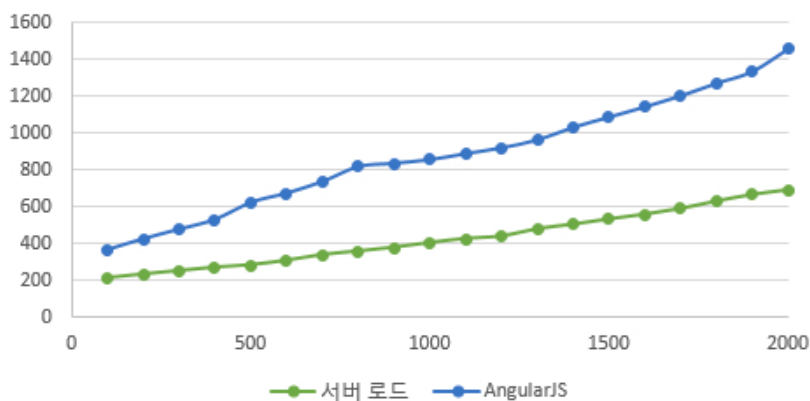


그림 24 두 가지 로드 시간 비교(단위 ms)

초기 페이지 로드 시 AngularJS를 통해 데이터를 처리하는 방식보다는 서버로부터 데이터를 내려받아 화면에 그려진 상태로 처리하는 것이 빠른 것으로 나타났다. 이와 같이 처음 페이지 로드 시 속도에 민감한 서비스라 판단된다면 AngularJS 방식보다는 서버로드 방식을 권장한다.

두 번째, 단방향 대 양방향 데이터 바인딩에 대한 데이터이다. 마찬가지로 100개에서 2,000개의 데이터 개수에 따라 단방향과 양방향 데이터 바인딩의 속도를 하나의 차트로 표현하면 다음과 같다.

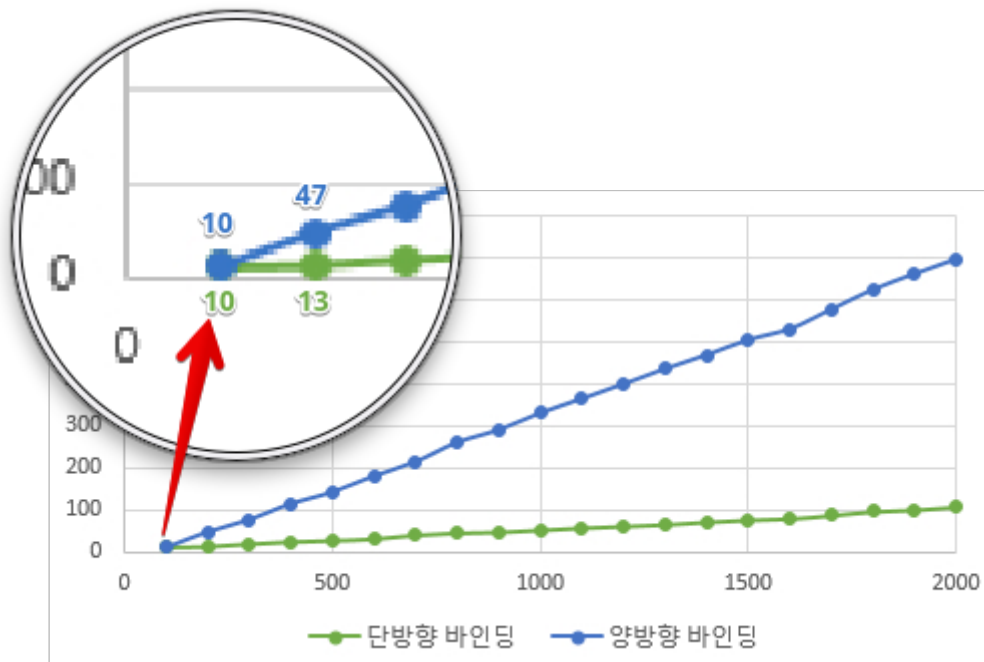


그림 25 두 가지 방식의 처리 시간(단위 ms)

단방향 바인딩의 처리 시간은 데이터의 양이 증가됨에 따라 완만하게 증가하지만, 양방향 바인딩의 경우 데이터의 양에 따라 처리 시간이 크게 증가되고 있는 것을 확인할 수 있었다. 따라서, 단방향과 양방향 바인딩을 통한 데이터 처리 시 100개 미만의 데이터에 대해서는 큰 차이를 보이지 않았으나 100개 이상의 데이터를 처리하는 경우에는 단방향 데이터 바인딩보다 양방향 데이터 바인딩의 처리 시간이 데이터의 개수에 비례해 증가하기 때문에 주의가 필요함을 알 수 있다.

- 100개 미만의 데이터: 단방향 및 양방향 데이터 바인딩 모두 10ms 내외로 비슷
- 200개 이상의 데이터: 200개의 데이터 처리 시 단방향 데이터 바인딩은 평균 13ms, 양방향 데이터 바인딩은 평균 47ms 소요되어 3배 이상의 차이를 보였으며, 데이터 개수 증가에 따라 처리 시간도 비례해 증가

그 외 고려 사항

컴파일되지 않은 콘텐츠의 출력

AngularJS는 표준 기술은 아니기 때문에, AngularJS 애플리케이션을 실행하기 위해선 반드시 전처리 과정이 필요함을 "초기화 과정"을 통해 살펴봤었다. 이 특성으로 인해 상황에 따라 컴파일(전처리)되지 않은 상태의 콘텐츠가 출력될 수도 있게 된다.

간단하게 목록을 출력하는 다음의 예제를 통해 살펴보자.

```
<body ng-app='myApp'>
  <div ng-controller='myControllerList'>
    <ul>
      <li ng-repeat='fruit in fruits'>
        <span>{{fruit.name}}</span> / <span>{{fruit.qty}}</span>
      </li>
    </ul>
  </div>
  <script type='text/javascript' src='angular.js'></script>
  <script type='text/javascript' src='myControllerList.js'></script>
</body>
```

```
app.controller("myControllerList", function($scope) {
  $scope.fruits = [
    { name: '사과', qty: 5 },
    { name: '감', qty: 2 },
    { name: '메론', qty: 9 },
    { name: '망고', qty: 3 },
    { name: '바나나', qty: 1 }
  ];
});
```

위와 같이 AngularJS를 로딩하는 스크립트 태그가 문서의 제일 아래에 위치한 경우, 아래와 같이 초기화가 이루어지기 전 잠시 템플릿 문법이 출력되고 이후 데이터가 바인딩된 뷰가 출력된다.

• {{fruit.name}} / {{fruit.qty}} →

- 사과 / 5
- 감 / 2
- 메론 / 9
- 망고 / 3
- 바나나 / 1

그림 26 초기화 전 템플릿 구문이 잠시 출력되는 과정

이처럼 AngularJS의 초기화가 완료되기 전 콘텐츠가 출력되는 상황이 만들어지는 경우에는 위와 같이 원하지 않는 형태의 출력이 발생될 수도 있다.

이러한 상황을 회피하려면 단순히 AngularJS 스크립트를 상단에 로드되도록 하거나, 컴파일이 필요한 영역을 먼저 숨김 처리한 후 컴파일된 이후 출력하게 하는 형태로 만들면 된다. 또한 AngularJS에서는 이러한 상황에 사용할 수 있는 `ng-cloak`¹⁸ 지시자를 제공하고 있다.

참고

AngularJS 애플리케이션에서 이러한 상황 설정은 억지스럽다고 느낄 수도 있다. 그러나 체감 성능 관점에서 스크립트가 렌더링과 DOM의 파싱을 막지 않도록 스크립트 로딩을 하단으로 이동하는 것은 많이 알려진 대표적인 성능 향상 방법 중 하나이다.

스크립트 로딩을 상단에서 실행하는 경우라도 페이지에서 포함하고 있는 콘텐츠의 양과 구조에 따라 위와 같이 일시적으로 초기화되지 않은 상태의 콘텐츠가 출력될 수 있는 상황은 충분히 예측해 볼 수 있다.

컴파일이 완료되기 전까지 화면을 보이지 않게 처리하는 방법도 비어 있는(blank) 화면을 일정 시간 노출되게 하는데, 이러한 상황은 체감 속도를 늦추므로 사용자에게 바람직하지 않을 것이다.

뷰가 분리된 구성

"특징과 주요 기능"의 "라우팅" 절에서 살펴봤던 것처럼 라우팅 시 별도의 외부 파일을 뷰로 갖도록 구성하는 경우, 각 뷰 파일의 로딩은 추가적인 AJAX 호출을 통해 이뤄진다.

```
<body ng-app="myApp">
  <div>
    <a href="#/view1">뷰1</a> / <a href="#/view2">뷰2</a>
    <ng-view></ng-view>
  </div>
</body>
```

```
angular.module('myApp', ['ngRoute'])
  .config(function($routeProvider) {
    $routeProvider
      .when('/view1', {
        controller: 'myController1',
        templateUrl: 'template1.html'
      })
      .when('/view2', {
        controller: 'myController2',
        templateUrl: 'template2.html'
      });
  })
  .controller("myController1", function($scope) {
    $scope.name = "가나다라";
  })
  .controller("myController2", function($scope) {
    $scope.name = "마바사";
  });
```

파일: template1.html

```
<div> 템플릿 1 : {{name}}</div>
```

파일: template2.html

```
<div>템플릿 2 : {{name}}</div>
```

¹⁸ `ng-cloak` 지시자는 `display` 속성을 제어하는 형태로 작동한다. <https://docs.angularjs.org/api/ng/directive/ngCloak>

위와 같이 구성된 경우, 각각 '뷰1'과 '뷰2' 링크를 클릭하면 라우팅에 의해 template1.html 파일과 template2.html 파일의 내용이 다음과 같이 출력된다.

뷰1 / 뷰2 뷰1 / 뷰2
 템플릿 1 : 가나다라 템플릿 2 : 마바사

그림 27 별도의 분리된 템플릿 파일을 활용한 예시

뷰는 아래의 로고를 통해 확인할 수 있는 것처럼 XHR을 통해 로딩된다. 한 번 로딩된 이후에는 내부적으로 캐시되므로 반복해서 호출되지는 않는다.

```
▶ XHR finished loading: GET "http://localhost/tt/angular/template1.html". angular.js:9689
▶ XHR finished loading: GET "http://localhost/tt/angular/template2.html". angular.js:9689
```

그림 28 템플릿 파일 로딩에 대한 XHR 로그

참고

템플릿 데이터의 캐시는 \$CacheFactoryProvider 객체 내의 클로저 변수로 저장된다. 캐시에 대한 데이터는 템플릿 URL을 키값으로 가지며, { "문서 URL": "템플릿 HTML", ... }과 같은 구조로 이뤄진다.

```
{
  template2.html: "<div> 템플릿 1 : {{name}}</div>",
  template2.html: "<div>템플릿 2 : {{name}}</div>"
}
```

이벤트 바인딩

양방향 데이터 바인딩을 위해서 AngularJS는 모델로 지정된 요소에 여러 개의 이벤트를 자동으로 바인딩한다. 예를 들어, <input> 요소를 모델로 지정하고 그 값을 뷰에 출력하는 간단한 애플리케이션의 경우, <input> 요소에는 실제로 다음의 이벤트들이 바인딩된다.

- input
- change
- blur
- compositionstart
- compositionend

확인을 위해 다음과 같이 간단하게 <input>의 값을 출력하도록 구성한 예제를 통해 살펴보자.

```
<div ng-controller="myController">
  <input type="text" ng-model="val"><br>
  <span>{{ val }}</span>
</div>
```

가나다
 가나다

그림 29 <input> 요소에 입력된 문자를 출력하는 예시

위의 그림과 같이 <input> 요소를 클릭하고 "가나다"라는 문자열을 입력한 후 페이지의 다른 영역을 클릭해 포커스를 잃는 순서대로 진행했을 때의 이벤트 핸들러의 수행 결과를 확인해 보자. 확인을 위해, AngularJS 내에서 이벤트 핸들러 함수인 `createEventHandler()`에서 이벤트 로그를 출력할 수 있도록 다음과 같이 추가했다.

```
function createEventHandler(element, events) {
  var eventHandler = function(event, type) {

    // 이벤트 발생에 대한 로그 출력
    console.log("이벤트 핸들러: ", event.type);

    // jQuery specific api
    event.preventDefault = function() {
      return event.defaultPrevented;
    };
    ...
  };
}
```

	이벤트 핸들러: compositionstart
3	이벤트 핸들러: input
	이벤트 핸들러: compositionend
	이벤트 핸들러: input
	이벤트 핸들러: compositionstart
2	이벤트 핸들러: input
	이벤트 핸들러: compositionend
	이벤트 핸들러: input
	이벤트 핸들러: compositionstart
	이벤트 핸들러: input
	이벤트 핸들러: compositionend
2	이벤트 핸들러: input
	이벤트 핸들러: change
	이벤트 핸들러: blur

그림 30 <input> 요소에 문자 입력 시 발생하는 이벤트 로그

그림에서와 같이 <input> 요소에 등록된 5개의 이벤트에 대한 핸들러가 수행됨을 확인할 수 있다. AngularJS는 직접 요소에 이벤트 바인딩을 하거나 직접 요소를 조작하지 않아도 되기 때문에 편리한 부분도 있지만, 이벤트 바인딩을 세부적으로 컨트롤해야 하는 상황에서는 불편할 수도 있다.

도입 여부 선택 가이드

웹 애플리케이션 개발 시, 어떤 상황에서 AngularJS 도입을 고려할지에 대한 고민이 필요할 수 있을 것이다. 이번 장에서는 AngularJS의 장단점을 다시 한 번 살펴보고, 도입 여부를 선택하는 데 도움이 될 간단한 선택 가이드를 제공한다.

정리

"AngularJS 소개"에서 우리는 AngularJS의 기본 개념과 철학, 특징을 확인했다.

"AngularJS 사용 효과"에서 간단한 TODO 애플리케이션의 실제 구현 프로세스와 코드를 이용해 DOM 제어 방식 대 AngularJS(MVC) 방식을 비교했다.

"AngularJS 사용 시 고려 사항"에서는 AngularJS의 대표적 기능 중 하나인 양방향 데이터 바인딩의 장단점을 알아보고, 단방향 바인딩과 양방향 바인딩 간의 성능 비교를 통해 고려해야 할 상황이 있음을 확인했다. 그 외 AngularJS의 내부 처리 방법들로 인해 특정 조건하에서는 추가적으로 고려해야 할 상황이 발생할 수 있음을 알 수 있었다.

AngularJS의 장점

- DOM 제어 기반 방식보다 적은 양의 코드로 개발 가능
- 구조화된 코드로 인해 향후 유지보수 비용 감소
- 코드(모듈) 간의 의존성이 적어 유연성과 재사용성이 증가
- 양방향 데이터 바인딩으로 인해 뷰의 갱신이 자동으로 이뤄지므로 뷰를 업데이트하는 등의 번거로운 작업이 감소하고 모델(데이터)에 더욱 집중 가능

AngularJS의 단점

- 최대 장점 중의 하나인 양방향 데이터 바인딩에서의 성능 저하(dirty-checking) 문제
- DOM 제어 방식이 아니므로, DOM의 세밀한 제어가 필요한 경우는 적합하지 않음
- 간단한 기능 구현이 아니라면, 어느 정도의 학습 비용(또는 기존 개발 방식에서의 인식 전환)은 필요
- 불필요할 수도 있는 이벤트들의 자동 바인딩

AngularJS 도입 여부 판단 흐름도

다음의 간단한 흐름도를 이용해 도입 여부를 직접 확인해 보기 바란다. 이 흐름도가 절대적인 결과를 도출하기 위한 것은 아니지만, 개발하고자 하는 기능과 상황에 따라 AngularJS의 도입 여부에 대한 판단이 달라질 수 있으므로 참고 수준에서 확인하기 바란다.

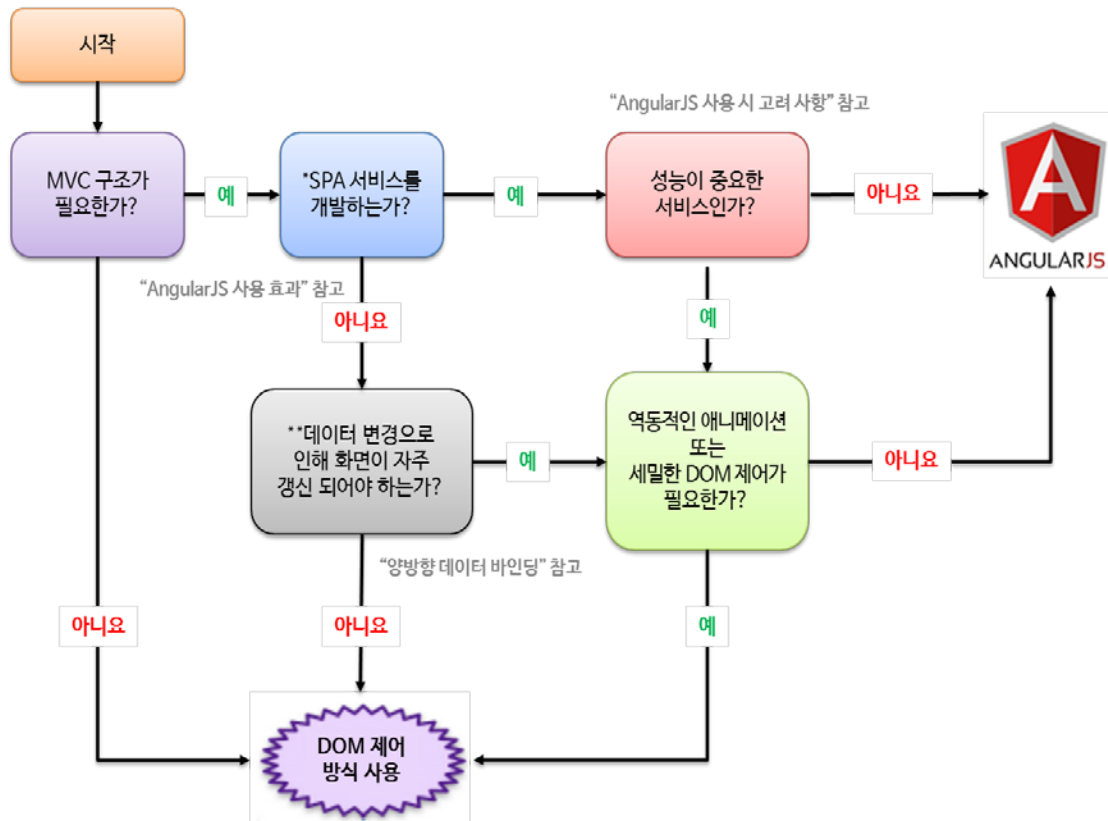


그림 31 AngularJS 도입 여부 판단 흐름도

참고

***SPA 서비스:** 구글의 Gmail, 네이버 캘린더 등의 서비스와 같이 최초 페이지 로딩 이후 페이지 이동 없이 사용하는 서비스와 같은 구조를 의미한다.

****데이터 변경:** 데이터(서버 또는 사용자의 입력) 값이 빈번하게 동기화되어 출력되어야 하는 경우를 의미한다.

부록

디버깅 도구

AngularJS에서 디버깅을 위해 Batarang이란 구글 크롬 확장 프로그램을 사용할 수 있다. 이 도구는 AngularJS의 디버깅이나 성능 메트릭스 등에 참고할 만한 정보를 보여주고 있어 개발 시에 참고할 수 있다.

우선 크롬 브라우저에서 확장 프로그램을 선택하여 "AngularJS" 키워드로 검색하면 다음과 같은 프로그램이 검색 결과로 나타난다. 해당 프로그램을 설치해보자.



그림 32 Batarang 크롬 확장 플러그인

확장 프로그램을 설치했으면 개발자 도구를 활성화한다. **Console** 탭 옆에 **AngularJS** 탭이 추가되어 있는 것을 확인할 수 있는데, 해당 탭 내에서 **Enable** 체크박스를 선택하면 작업하고 있는 페이지의 스코프(Scope)의 내용을 쉽게 확인할 수 있다.

다음은 **Models** 탭에 있는 스코프 내용을 확인하는 화면이다.

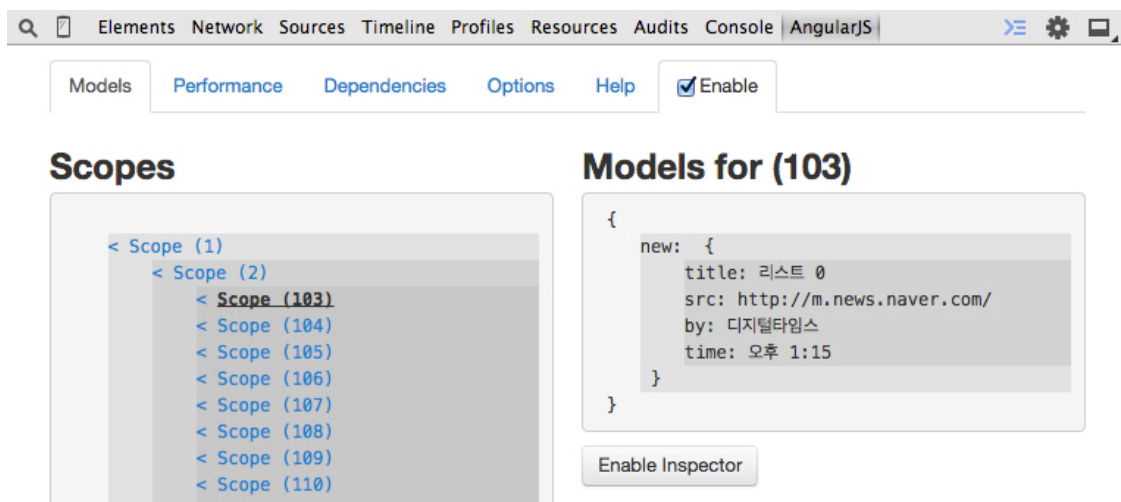


그림 33 AngularJS 스코프 확인

왼쪽에 정의된 스코프가 있으며, 스코프를 하나씩 눌러 어떠한 값이 정의되어 있는지 쉽게 확인할 수 있다. 그리고 **Enable Inspector**를 이용해 inspector가 지정한 위치의 스코프 내용을 확인할 수 있도록 제공한다.

추가로 **Options** 탭의 **Show applications**, **Show bindings**, **Show scopes**를 선택해 바인딩되어 있는 영역, 스코프 영역 등을 확인할 수 있다.

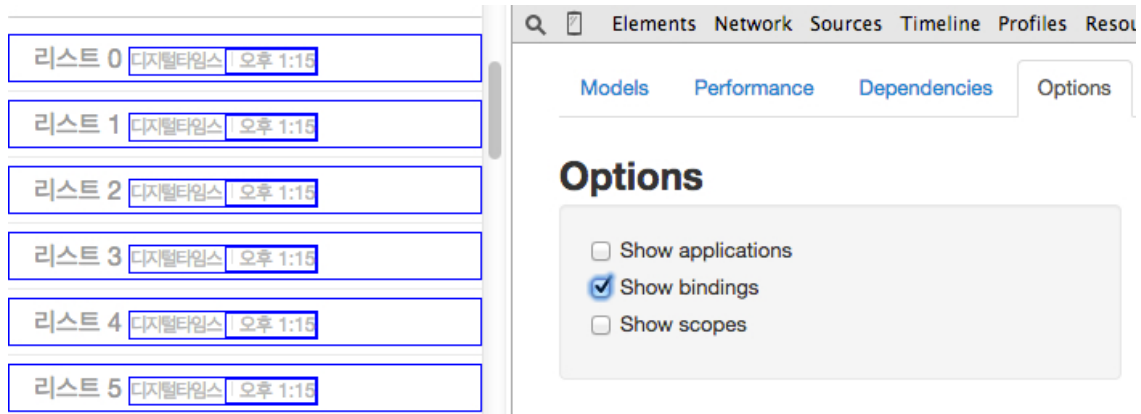


그림 34 Show bindings 활성화 상태

타 라이브러리와의 사용성

앞서 살펴본 것과 같이 AngularJS는 DOM 제어에 중점을 두지 않는다. 그러나 DOM을 제어하기 위해 기본적으로 jQuery¹⁹의 간소화 버전인 jQLite(jQuery Lite)²⁰를 라이브러리 내에 포함하고 있으며, 요소의 제어와 관련된 기능만을 제한적으로 제공한다. jQuery를 이미 사용하고 있다면(AngularJS 애플리케이션이 초기화되기 전에 로딩을 완료해야 함), 자동으로 jQuery를 사용하도록 설정된다²¹.

AngularJS 내부에서 사용되는 모든 요소에 대한 레퍼런스는 항상 jQLite(또는 jQuery)로 래핑되어 전달되며, 원 요소에 대한 래핑이 필요한 경우 다음과 같이 `angular.element()`를 사용할 수 있다.

```
// jQLite에서 제공하는 DOM 관련 메서드를 사용해야 하는 경우 다음과 같이 래핑한다.
angular.element(요소);
```

AngularJS 애플리케이션 내에서 외부 라이브러리(예: Jindo)를 같이 사용해야 할 때, 별도의 제약이 존재하지는 않는다.

다음은 클릭 이벤트 핸들러 내에서 event 객체를 이용해 이벤트가 발생한 요소를 얻은 후 해당 요소의 텍스트를 변경하는 간단한 예제다.

```
<div ng-controller="myController">
  <button ng-click="change($event)">테스트</button>
```

¹⁹ <https://docs.angularjs.org/misc/faq#does-angular-use-the-jquery-library->

²⁰ <https://docs.angularjs.org/api/ng/function/angular.element>

²¹ AngularJS 내부에서 `bindjQuery()` 함수를 이용해 jQLite가 jQuery의 기능들을 사용하도록 처리한다.

```
</div>
```

```
app.controller("myController", function($scope) {
    $scope.change = function(event) {
        var el = event.target;

        // jqLite을 사용해 요소의 css를 변경
        angular.element(el).css("fontSize", "20px");

        // jindo.$Element를 사용해 요소의 텍스트 값을 변경
        jindo.$Element(el).text("가나다라");
    };
});
```

버튼을 클릭하면 아래의 그림과 같이 '테스트'에서 '가나다라'라는 텍스트로 변경되며, 글꼴의 크기 또한 20px로 변경되는 것을 확인할 수 있다.



그림 35 CSS와 텍스트의 내용을 각각 jqLite와 Jindo를 사용해 변경하는 버튼

이처럼, 다른 라이브러리를 같이 사용하는 데는 큰 문제가 없으나, 병행해서(jQuery 포함) 사용하는 것은 AngularJS를 통한 애플리케이션 개발에서의 장점들을 퇴색시킬 수 있어 추천하지는 않는다.

도입 사례

다음은 실제 서비스에서 AngularJS를 사용해 본 개발자들의 의견을 정리한 내용이다. 일반 사용자를 대상으로 하는 서비스 영역과 어드민 도구 등 다양한 형태의 서비스 개발에 AngularJS를 도입해 사용한 개발자들의 실사용 경험을 인터뷰로 수집했다. 다만, 구체적으로 어떤 서비스 개발에 적용되었는지 언급하는 데는 어려움이 있어 사례별로 도입 이유 및 효과만 간략히 공유하고자 한다.

도입 이유 및 효과

사례 1

- 사용자의 동작에 따라 변경되는 요소를 쉽게 적용 가능
- 보다 쉬운 유지보수
- 인수인계에 대한 부담감 해소
- 모델을 이용한 간편한 데이터 바인딩 처리
- 프론트엔드 기술에 익숙하지 않더라도 부담 없이 개발 가능
- 코드 패턴이 동일하여, 개인 간의 역량 차이에 따른 코드 결과물의 차이 감소

사례 2

- SPA(Single Page Application) 형태의 개발 필요
- 특정 영역의 간편한 목록 업데이트
- 협업이 가능한 형태로 명확히 분리 가능

- 양방향 데이터 바인딩을 잘 이용하면 코드량 간소화
- 개발 속도 향상 및 유지보수 비용 절감

사례 3

- SPA에 최적화된 프레임워크
- 다른 프레임워크에 비해 쉽고 명확함
- 문서와 질의응답이 잘 되어 있는 버전 관리
- Grunt²², Yeoman²³, Bower²⁴ 등의 도구와의 사용 편리성

개발 기간

개발 기간은 스펙과 기능에 따라 달라질 수도 있지만, 한 개발자에 의하면 모니터링 시스템은 약 1년 반 정도의 시간을 할애하여 개발되었고, 저작 도구는 2개월, 프로모션 페이지는 3개월만에 AngularJS를 가지고 서비스를 만든 것으로 답했다.

정리

서비스의 특성과 개발자 간 개인차에 따라 달라질 수 있지만, 인터뷰에 참여한 개발자 대다수는 양방향 데이터 바인딩으로 인해 뷰 처리가 쉽고, 개발 속도가 빠르며, 협업에 대한 부담감을 줄일 수 있다는 부분을 AngularJS의 장점으로 꼽았다.

²² Grunt는 다양한 플러그인이 제공되는 빌드 도구이다. <http://gruntjs.com/>

²³ Yeoman은 다양한 프레임워크/라이브러리를 사용하는 프로젝트를 구성할 수 있도록 도와주는 스캐폴딩 도구이다. <http://yeoman.io/>

²⁴ Bower는 웹 패키징 매니저로, 웹 개발에 필요한 다수의 도구를 저장하는 저장소이다. <http://bower.io/>
