

모던 자바 웹 프로그래밍

김대희 지음

내용

모던 자바 웹 프로그래밍	1
저자 소개	4
저자 서문	5
대상 독자 및 참고사항	6
1 AngularJS 의 기본	9
1.1 Hello World	9
1.2 데이터 바인딩	11
1.3 컨트롤러를 사용합니다	13
1.4 DI (Dependency Injection)	15
1.5 ng 서비스	18
1.6 jqLite	19
2 Every Store 개발	22
2.1 상품 목록	22
2.2 REST 서비스를 사용해봅시다	34
2.3 상품 상세화면	36
2.4 장바구니	44
2.5 사용자 코멘트 화면	53
3 클라이언트 모듈화하기	60
3.1 node.js 설치	60
3.2 Browserify 란	60
3.3 Gulp 로 거들기	61
3.4 번들 자동화하기	62
4 Every Store 다듬기	67

4.1 Gradle 빌드환경 구축하기	67
4.2 MongoDB 사용하기	69
5 Every Store 배포.....	82
5.1 Build	82
5.2 NginX 활용하기	82

저자 소개

저자 서문

필자는 바로 몇 년 전에만 서버에서 html 코드와 자바스크립트가 한데 뭉쳐진 스파게티 코드를 생산하는 개발을 했었다. 그러나 3년전 백본(bacbone.js)를 접하고 SPA(Single Page Application) 방식으로 개발을 진행해 왔다.

당시 jQuery와 모듈화 되지않고 글로벌영역을 오염시키는 모듈을 모두 걷어내고 백본으로 모두 교체하는 작업등의 성과를 얻었다. 그러나 문제는 상당히 많아지는 자바스크립트 파일 개수와 여러 개발자들에 의해 수정 되고 공통기능이 커짐에 따라 require.js를 사용하여 모듈화 작업을 진행했다.

지금도 백본은 꽤 쓸만하고 자잘은 버그도 없고 가볍고 단시간에 익힐 수 있는 좋은 프론트엔드 기술이지만 AngularJS를 접하고 나서 최근에는 거의 앵귤러(AngularJS)를 우선적으로 이용하게 되었다. 거기에서 node.js를 이용한 모듈화 및 번들툴로 사용하면서 프론트영역부터 백엔트 영역까지 전체적인 모든 업무를 담당할 수 있었다.

이 책은 풀스택(Full Stack) 즉 프론트와 백엔드를 모두 다루는 책이다. 전문 프론트 개발자가 아니더라도 요즘 처럼 특히 프론트 영역에서 프레임워크가 우후죽순처럼 쏟아져 나와고 있는 시점이라 프론트 영역의 소스를 수정하기 위해서 어느정도 프레임워크의 지식이 필요하게 되었다. 거기에서 싱글페이지(SPA) 로 구성하려면 앵귤러나 백본과 같은 기술뿐만 아니라 백엔드의 지원까지 조화가 필요하다.

자바에는 요즘 MEAN 스택 같은 유명한 Full Stack 프레임워크는 없지만 RESThub(<http://resthub.org>) 나 JHipster(<https://jhipster.gibhub.io>)를 이용할 수 있다. 이 책은 JHipster와 유사하게 백엔드로 스프링부트를 사용하고 프론트에서는 AngularJS 를 사용한다. JHipster를 사용해도 좋으나 처음 접하는 이에게는 어떤 코드를 생성하는 지 잘 알 수가 없다. 그래서 이 책에서는 풀스택의 쉬운 개발패턴을 소개한다. 자신만의 독자적인 방식이 아니라 범용적이고 잘 관리되고 있는 오픈 소스가 필요하다면 JHipster를 접해보기를 권한다.

대상 독자 및 참고사항

대상: 초중급

nodejs를 사용해본 경험이 있거나 자바스크립트의 디자인패턴 중 일부 체인닝패턴 IIFE 와 같은 것에 익숙해야 합니다.

몽고디비의 설치 및 기본적인 쿼리를 알고 있어야 합니다.

curl이나 크롬의 웹개발툴에 익숙해야 합니다.

스프링 프레임 워크에 기본적인 지식이 필요합니다.

이 책에서 사용하는 소스파일은 아래의 Github 에서 다운로드가 가능합니다.

https://github.com/clonekim/starting_modern_java_web

이 책은 미니 쇼핑카트, 에브리스토어(Every Store) 라는 싱글 페이지 웹앱을 구축하는 것이 최종 목표입니다.

에브리스토어는 흔히 볼수 있는 쇼핑몰 구조를 가지고 있습니다. 상품을 나열하고 선택하면 상세한 화면으로 이동하고 사용자가 코멘트를 작성할 수도 있고 바로 구입 또는 장바구니에 담을 수 있습니다.

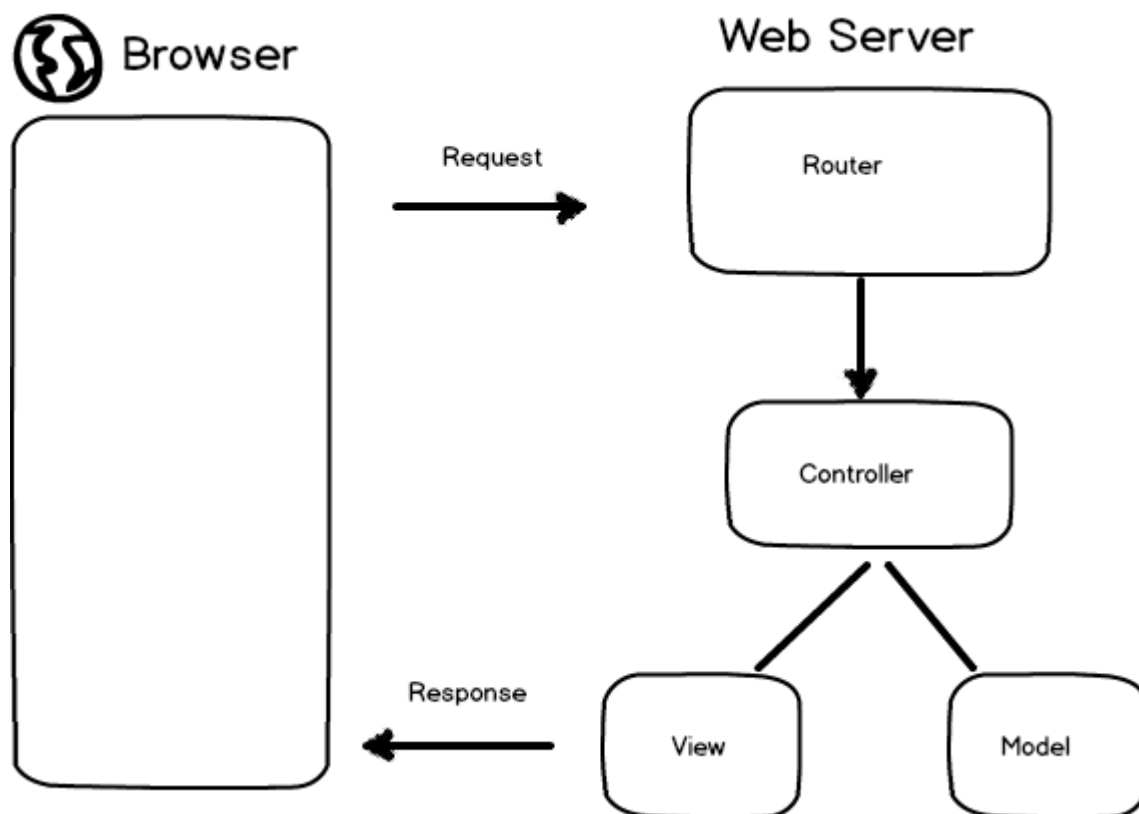
장바구니는 언제나 접근이 가능하고 장바구니에 담겨 있는 상품 개수는 항상 노출되도록 하기 위해서 화면의 상단에 항상 표시합니다.

장바구니 화면에서는 상품이 나열되고 전체 상품의 가격이 합산되어 출력되고 실제 결제는 이루어지지 않습니다. 물론 장바구니에서 상품을 클릭하면 다시 상품의 상세화면으로 이동합니다.

에브리스토어는 싱글페이지로 구성되기 때문에 먼저 싱글페이지웹과 종래의 웹이 어떻게 다른지 간략하게 설명하겠습니다.

그림 1 은 종래의 웹어플리케이션이 작동하는 방식입니다. 브라우저가 서버로부터 요청하고 결과를 받는 것은 변하지 않았으나 라우트 즉 각 웹사이트의 목적지가 서버에서 제공됩니다. 서버로부터 응답결과는 html 또는 json 등 어느것이든 될 수 있습니다. 그럼 SPA의 구조를 보겠습니다.

그림 1 일반적인 웹접근 방식

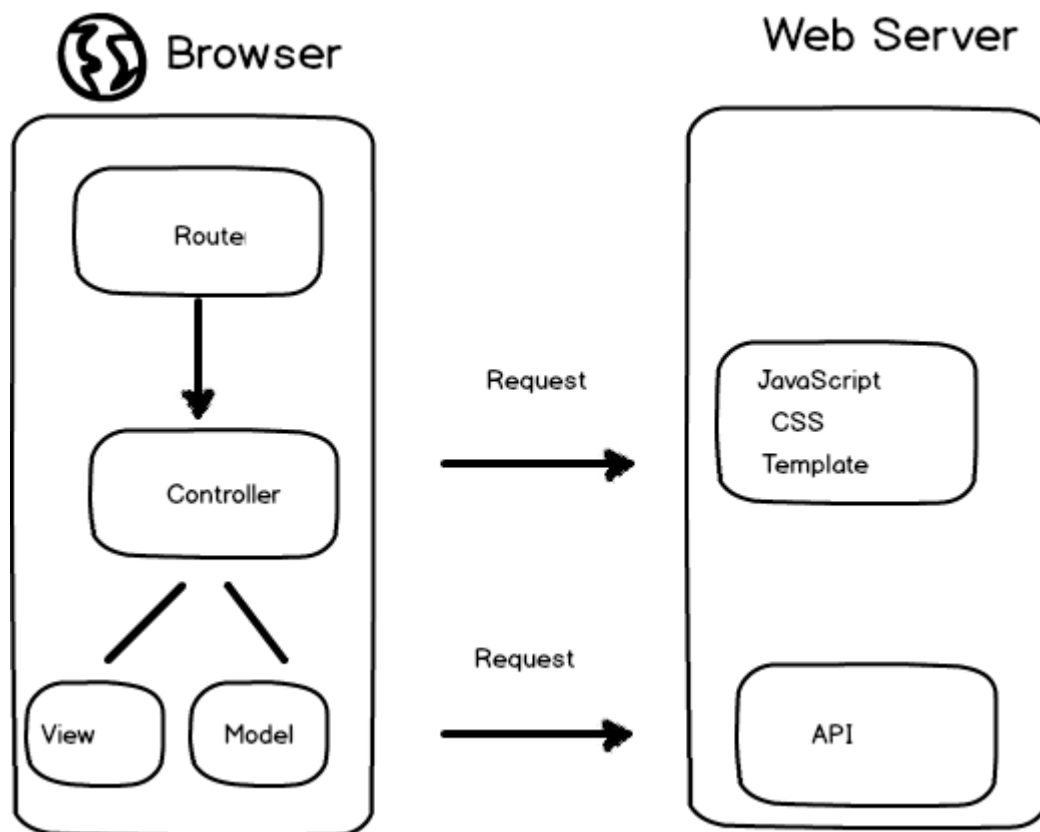


SPA에서는 모든 웹사이트의 목적지를 클라이언트가 가지고 있습니다.

처음 웹 브라우저가 서버로부터 html을 다운로드 후 브라우저에서 발생하는 라우트의 이동, 즉 다른 목적으로 이동하는데 네트워크 트래픽이 한번에 발생되지 않습니다. 대신 이동시에 해당 목적지에서 표현해야할 html 조각들을 그때그때 다운 받을 수 있습니다. 그리고 데이터부분을 취득하여 템플릿 조각에 삽입되고 파싱하여 최종적으로 렌더링 됩니다. 이 작업은 DOM을 사용하는 것입니다.

구조상으로는 클라이언트와 서버의 역할이 확실히 구분됩니다. 즉 서버에서는 더 이상 뷰페이지를 생산하지 않아도 되지만 클라언트가 상당히 무겁게 될 수 있기 때문에 기존과 전혀 다른 개발 패턴이 필요합니다.

그림 2 싱글페이지 구조



1 AngularJS의 기본

먼저 Angular(앵굴러)를 <https://www.angularjs.org> 에서 다운로드 받습니다.
이 책에서는 1.4.7 을 사용하였습니다.

1.1 Hello World

앵굴러를 사용한 가장 간단한 예제입니다

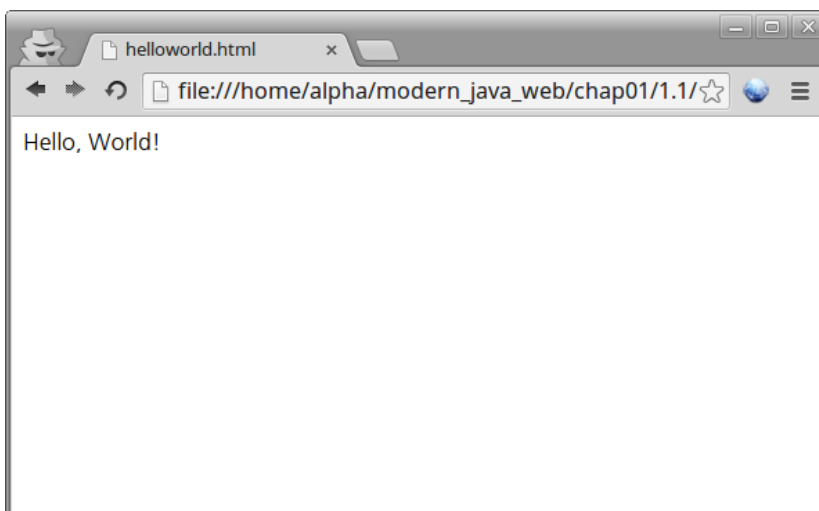
[helloworld.html]

```
<!DOCTYPE html>
<html ng-app>
  <head>
    <meta charset="utf-8">
    <script type="text/javascript" src="angular.min.js"></script>
  </head>
  <body>
    {{ 'Hello, World! ' }}
  </body>
</html>
```

무슨 일이 일어났나요?

웹브라우저에서 helloworld.html 를 열어보면 화면에 Hello, World! 를 출력합니다. (그림 3)

그림 3 hello world



출력된 문장은 앵귤러의 표현식을 통해서 출력되었습니다. Angular 표현식은 템플릿이라고 불립니다. 앵귤러의 템플릿은 html 기반에서 사용할 수 있어서 자유롭게 사용이 가능합니다

```
{{ expression }}
```

가령 아래와 같이 산술식도 사용할 수 있습니다

```
<div>
{{ 1 + 2 }}
</div>
```

다음 예제를 봅시다

[ng-repeat.html]

```
<div ng-init="num=42"> ---❶
  {{num}}
</div>
```

```
<div ng-if="enable"> ---❷
Hello, World
</div>
```

```
<div ng-init="items = ['사과', '감', '배', '복숭아']"> ---❸
  <ul>
    <li ng-repeat="item in items"> ---❹
      {{item}}
    </li>
  </ul>
</div>
```

❶ num라는 변수를 생성하고 초기화합니다. {{num}} 에 42가 출력됩니다

❷ enable이 참이면 Hello, World 를 출력합니다. enable이 존재하지 않기때문에 아무것도 출력되지 않습니다

❸ items라는 변수에 배열로 초기화합니다.

❹ 배열에 있는 값을 순환하면서 출력합니다

위의 예제는 Angular의 지시어(directive)를 사용한 예입니다.

앵귤러에는 상당히 많은 지시어가 있습니다. 그 밖에 지시어는

<https://code.angularjs.org/1.4.7/docs/api/ng/directive> 에서 확인이 가능합니다.

1.2 데이터 바인딩

Angular 가 가지고 있는 특징 중 하나인데 자바스크립트에서 설정한 변수를 html 에서 사용 시 양방향간에 바인딩이 되어 있어서 어느 한쪽에서 값을 변경하면 다른 한쪽도 적용됩니다.

이를 양방향 데이터바인딩이라고 합니다

[bind-test.html]

```
<!DOCTYPE html>
<html ng-app>
  <head>
    <meta charset="utf-8">
    <script type="text/javascript" src="angular.min.js"></script>
  </head>
  <body>
    <div ng-init="num=42">
      {{num}}
    </div>

    <div ng-if="enable">
      Hello, World
    </div>

    <div ng-init="items = ['사과', '감', '배', '복숭아']">
      <ul>
        <li ng-repeat="item in items">
          {{item}}
        </li>
      </ul>
    </div>

    <input type="text" ng-model="message"> ---❶
```

```
<div>
  {{message}} ---❷
</div>
</body>
</html>
```

❶ ng-model 을 통해서 바인딩이 될 변수를 설정합니다.

❷ div 에 그냥 표기만 하므로 단방향 바인딩입니다.

실행 결과를 보기 위해서 웹브라우저로 접근합니다. 텍스트 박스 아래에 출력된 문구가 없습니다.

이제 텍스트 입력 박스에 입력해 봅니다.

입력되는 문자가 바로 아래 div 영역에 출력됩니다. (그림 4)

그림 4



만약 단방향 데이터바인딩을 표현하려면 다음과 같이 합니다.

1.3 컨트롤러를 사용합니다

템플릿과 자바스크립트의 코드를 연결하기 위해서 컨트롤러를 사용합니다.

컨트롤러는 자신의 영역 즉 `$scope` 라는 컨텍스트를 가지고 있습니다. 자신의 스코프에 선언된 변수나 함수는 컨트롤러가 관리하는 템플릿에서 사용할 수 있습니다.

아래 예는 `$scope` 에 변수와 함수를 선언하는 예제 입니다.

```
var appModule = angular.module('app', []);
appModule.controller('myController', function($scope) { ---❶

    $scope.message = 'Hello World'; ---❷
    $scope.action = function() { ---❸
        alert("Good Bye");
    }
});
```

❶ myController를 선언합니다.

❷ `$scope`에 `message`라는 변수를 선언합니다.

❸ `$scope`에 `action`이라는 함수를 선언합니다.

이제 컨트롤러를 템플릿에 적용합니다. 전체 소스는 다음과 같습니다.

[bind-action.html]

```
<!DOCTYPE HTML>
<html lang="en" ng-app="app">
<head>
    <meta charset="utf-8">
    <script type="text/javascript" src="angular.min.js"></script>
    <script type="text/javascript">
        var appModule = angular.module('app', []);
        appModule.controller('myController', function($scope) {
            $scope.message = 'Hello World';
            $scope.action = function() {
                alert("Good Bye");
            }
        });
    </script>
</head>
<body>
    <div>Hello World</div>
    <div><button ng-click="action">Good Bye</button></div>
</body>
</html>
```

```
    });  
  </script>  
  
</head>  
<body>  
  
  <h1 ng-init="name='Hello, Angular'"> {{name}} </h1>  
  
  <div ng-init="variable=42" >  
    {{variable}}  
  </div>  
  
  <div ng-init="items = ['사과', '감', '배', '복숭아']">  
    <ul>  
      <li ng-repeat="item in items">  
        {{item}}  
      </li>  
    </ul>  
  </div>  
  
  <input type="text" ng-model="message">  
  <div>  
    {{message}}  
  </div>  
  
  <div ng-controller="myController"> ---❶  
    {{message}} ---❷  
    <button type="button" ng-click="action()">click</button> ---❸  
  </div>  
  
</body>  
</html>
```

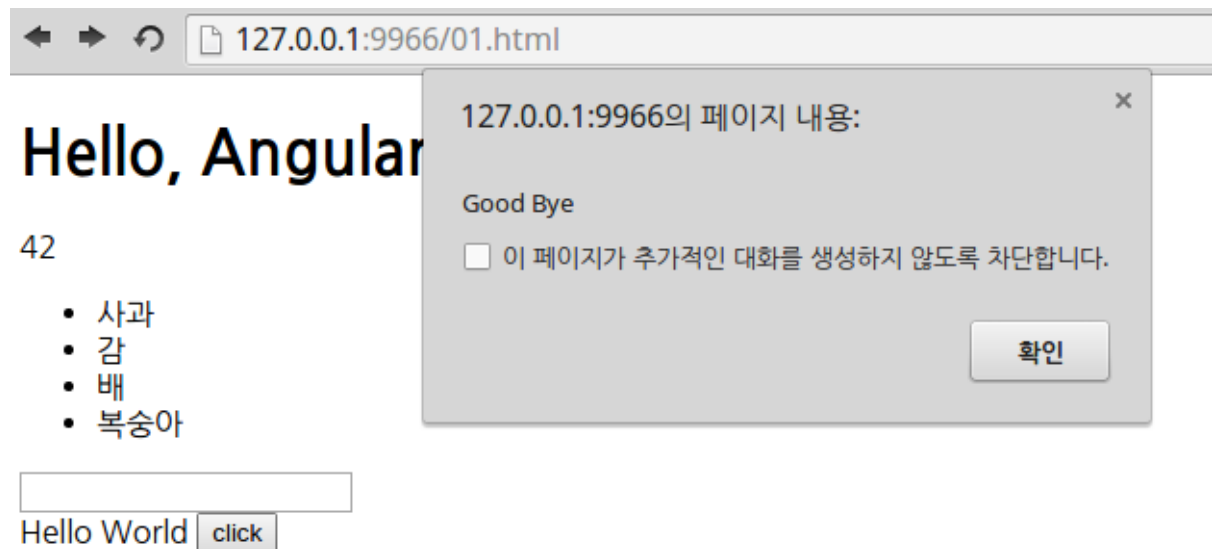
❶ 이 div 에 myController 를 사용하도록 합니다

❷ myController 가 관리하는 \$scope 의 message 변수의 값을 출력합니다

❸ \$scope 에 정의되어있는 action 함수를 사용하도록 합니다

여기서 ng-click 이라는 지시어를 통해서 컨트롤러가 가지고 있는 함수를 사용할 수 있습니다. 버튼을 클릭하면 다음과 같이 결과를 볼 수 있습니다. (그림 5)

그림 5



1.4 DI (Dependency Injection)

잠깐 여기서 사용한 \$scope 는 어디로부터 온것인가요.

이것은 Angular 의 DI(Dependency Injection)¹를 사용한 겁니다. DI 어디서 많이 들어봤는데 스프링 프레임워크를 통해서도 접해보았습니다.

그것과 같은 DI 입니다. 그러니까 여기서 사용하려는 \$scope 를 Angular 가 삽입 시켜 준것입니다.

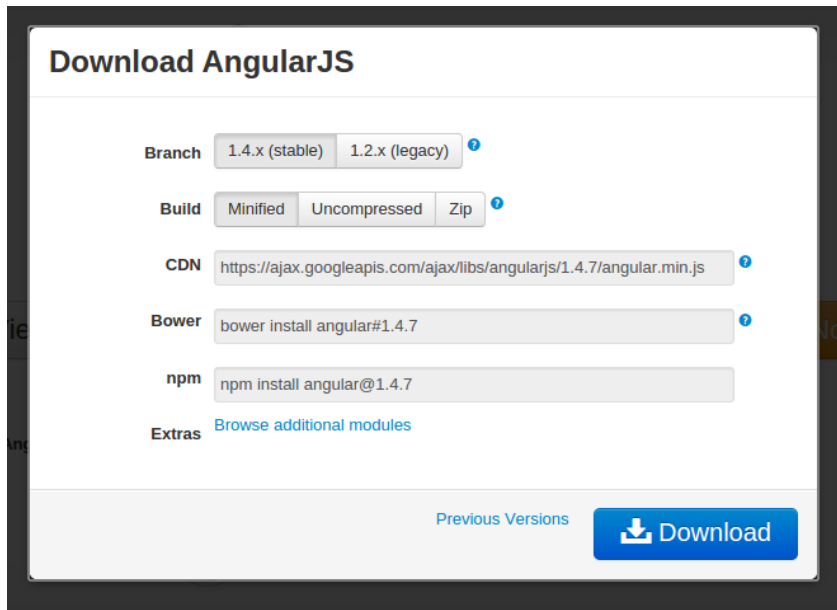
앵귤러의 모듈 중 에 angular-message 를 사용하는 예들 들어 보겠습니다.

우선 angular-message 가 필요하기 때문에 angular 공식 사이트로부터 다운로드 받습니다.

angular 의 기타 외부 모듈은 Extra (그림 6) 로부터 다운로드 받을 수 있습니다.

¹ <https://code.angularjs.org/1.4.7/docs/guide/di>

그림 6



angular-messages.js를 다운로드 받은 후 <https://code.angularjs.org/1.4.7/docs/api/ngMessages> 를 참조하자. ngMessages라는 모듈로 사용할 수 있다는 것을 알수 있다.

[ng-message.html]

```
<!DOCTYPE html>
<html ng-app="myapp">
  <head>
    <meta charset="utf-8">
    <script type="text/javascript" src="angular.min.js"></script>
    <script type="text/javascript" src="angular-messages.min.js"></script>
    <script type="text/javascript">
      angular.module('myapp', ['ngMessages']).controller('myController', function($scope) { ---❶

        $scope.weapon= {
          items: {
            "1": false,
            "2": false,
            "3": false,
            "4": false
          }
        }
      };
    </script>
  </head>
</html>
```

```

$scope.change = function() { ---❷

    angular.forEach($scope.weapon.items, function(val, key) {
        if($scope.status == key) {
            $scope.weapon.items[key] = true;
        }else {
            $scope.weapon.items[key] = false;
        }
    })
};

});

</script>
</head>
<body ng-controller="myController">
    <input type="text" ng-model="status" ng-change="change()"> ---❸

    <div ng-messages="weapon.items"> ---❹
        <p ng-message="1">칼</p>
        <p ng-message="2">권총</p>
        <p ng-message="3">레밍턴</p>
        <p ng-message="4">슈류탄</p>
    </div>

    <p ng-show="!status.length">1~4 번호를 입력하세요</p>

</body>
</html>

```

- ❶ 모듈에 ngMessages라는 의존성을 추가한다.
- ❷ 입력박스에 키가 입력시 변화를 감지하기 위한 change 이벤트를 작성한다. 텍스트 박스의 입력값에 따라 내부 weapon.items에 true 또는 false를 설정한다.
- ❸ 컨트롤러의 이벤트 리스너와 바인딩한다
- ❹ scope의 내부의 데이터를 설정한다.

1.5 ng 서비스

ng 모듈에는 어떤 서비스들이 있을 까요?

<https://code.angularjs.org/1.4.7/docs/api/ng/service> 를 참조하면 ng 모듈에서 기본적으로 제공되는 모듈들을 알수 있습니다. 이 모듈 중 일부를 DI 를 통해서 사용하는 예제를 봅시다.

제공되는 기본 서비스 중에서 \$log 와 \$interval 이라는 서비스를 사용합니다. (그림 7)

[ng-log.html]

```
<!DOCTYPE html>
<html ng-app="myapp">
  <head>
    <meta charset="utf-8">
    <script type="text/javascript" src="angular.min.js"></script>
    <script type="text/javascript">
      angular.module('myapp', []).controller('serviceController', ['$log', '$interval', function($log, $interval) {

        var num = 0;

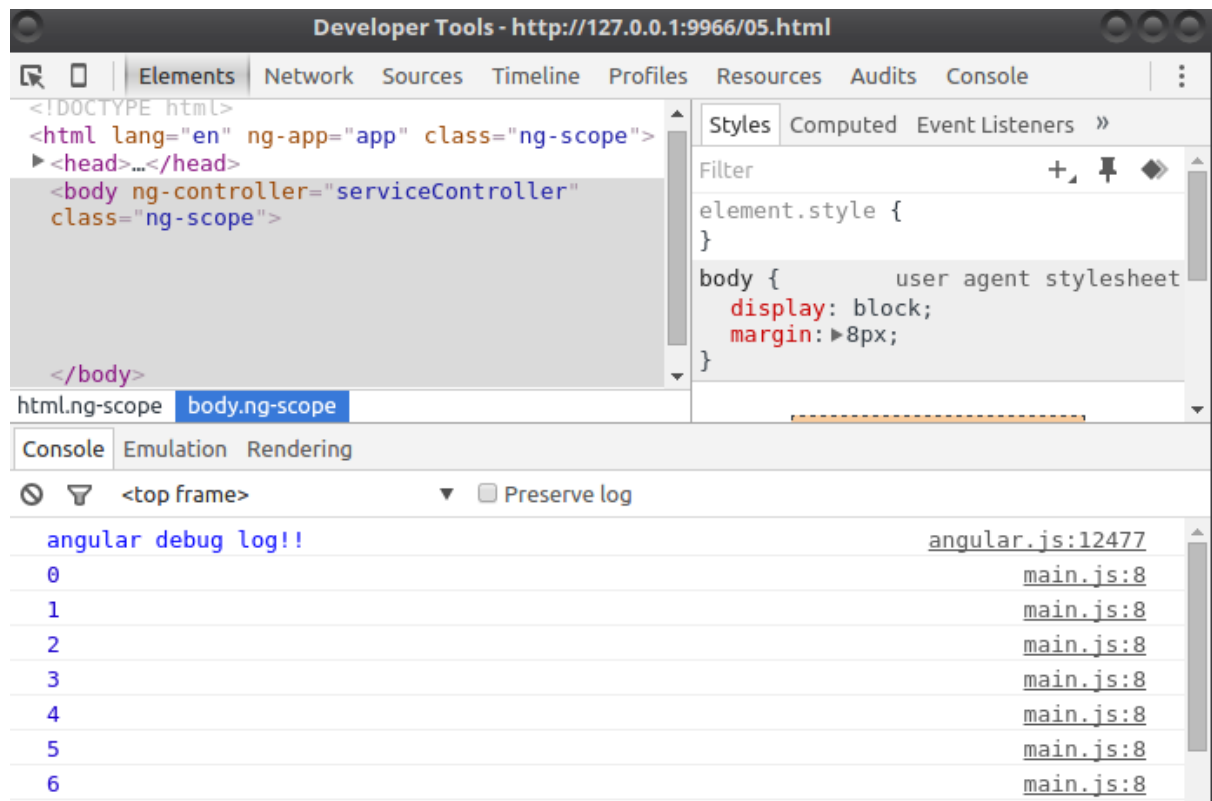
        $interval(function() {
          console.log(num++);
        }, 3000);

        $log.debug("angular debug log!!"); //debug가 초기값이다
      }]);

    </script>
  </head>
  <body ng-controller="serviceController">

  </body>
</html>
```

그림 7 서비스 사용 예



\$log 서비스를 통해서 콘솔에 메시지를 출력할 수 있습니다. 디폴트로 사용가능하게 되어있으나 아래와 같이 config 를 통해서 설정을 변경할 수 있습니다.

```
app.config(['$logProvider', function($logProvider) {  
    $logProvider.debugEnable(false);  
}]);
```

1.6 jqLite

하지만 DOM 를 직접 다뤄야 한다면 어떻게 할까요? jQuery 를 사용하고 싶은데 어떻게 해야하나 라고 의문을 가지는 독자분도 계실 겁니다.

DOM 를 직접 조작하려면 jQuery 를 사용하는게 가장 효과적일 지도 모릅니다.

Angular 에 jQuery 와 호환성을 가지는 jQuery 의 축소 버전인 jqLite 가 있습니다.

jqLite 가 jQuery 를 완벽하게 대체하는 것은 아니나 jQuery 와 호환가능하는 메서드들은 그림 8 을 참고하여 확인이 가능합니다

그림 8 jqLite의 메서드

Angular's jqLite

jqLite provides only the following jQuery methods:

- `addClass()`
- `after()`
- `append()`
- `attr()` - Does not support functions as parameters
- `bind()` - Does not support namespaces, selectors or eventData
- `children()` - Does not support selectors
- `clone()`
- `contents()`
- `css()` - Only retrieves inline-styles, does not call `getComputedStyle()`. As a setter, does not convert numbers to strings or append 'px'.
- `data()`
- `detach()`
- `empty()`
- `eq()`
- `find()` - Limited to lookups by tag name
- `hasClass()`
- `html()`
- `next()` - Does not support selectors
- `on()` - Does not support namespaces, selectors or eventData
- `off()` - Does not support namespaces, selectors or event object as parameter
- `one()` - Does not support namespaces or selectors
- `parent()` - Does not support selectors
- `prepend()`
- `prop()`
- `ready()`
- `remove()`
- `removeAttr()`
- `removeClass()`
- `removeData()`
- `replaceWith()`
- `text()`
- `toggleClass()`
- `triggerHandler()` - Passes a dummy event object to handlers.
- `unbind()` - Does not support namespaces or event object as parameter
- `val()`
- `wrap()`

jqLite 를 사용하려면 `angular.element` 로 해당 엘리먼트에 먼저 접근해야 합니다.

```
//angular.element가 돌려주는 것은 jqLite오브젝트이다
```

```
var el = angular.element(document.getElementById('message'));  
el.css('color', 'red');
```

```
var newEl = angular.element("<div></div>");
```

만약 jQuery 를 사용하게 되면 jqLite 를 대체하게 됩니다. angular 1.3 부터 jQuery 2.x 를 지원하기 시작하였고 jQuery 1.x 버전도 사용할 수 있지만 오작동할 소지가 있습니다.

2 Every Store 개발

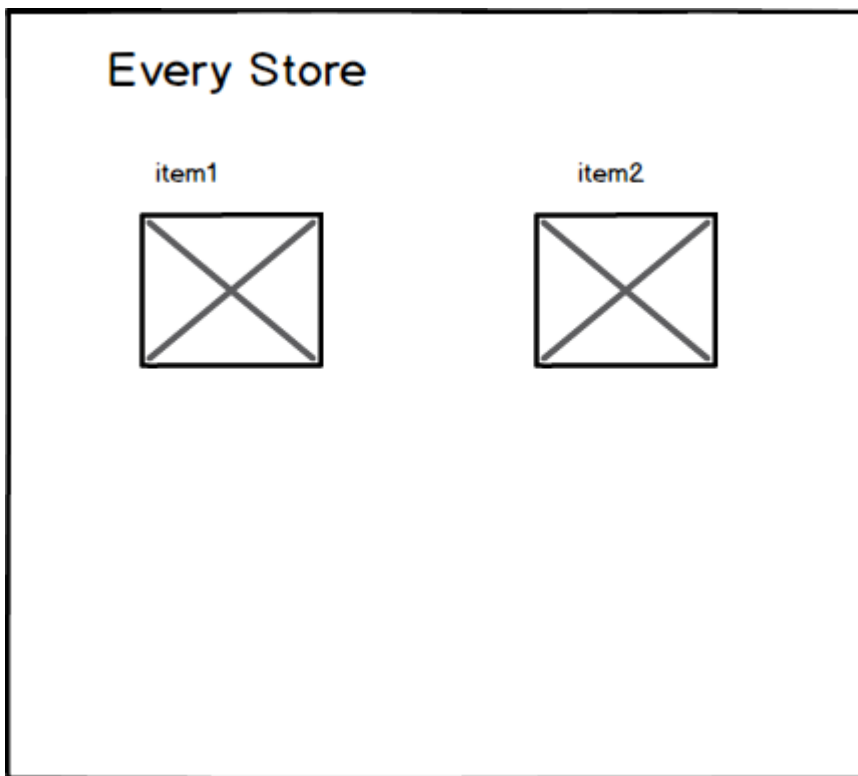
이번 장부터 Angular와 Spring Boot를 사용해서 미니 쇼핑카트를 만듭니다. 전체 쇼핑카트의 구성은 총 3개의 화면으로 구성되었습니다. 기능별로는 상품목록보기, 상품상세보기, 코멘트목록보기, 코멘트 작성하기, 장바구니 보기, 장바구니 카운트 등이 되겠습니다.

2.1 상품 목록

그림 9 는 에브리스토어에 접속하면 처음 보이는 화면입니다.

상품이 2 열로 나열되고 상품을 클릭하면 상세화면으로 이동하는 구조입니다.

그림 9 상품 목록 화면



상품목록에서 보여줄 것은 이름과 이미지만 보여 줍니다.

나머지는 상세화면에서 보여주도록 합니다.

그러면 서버로부터 받는 데이터는 대략 다음과 같습니다.

```
[  
  { id: 1, name: '상품1', img: 'image01.jpg'}, // 하나가 각 상품정보입니다  
  { id: 2, name: '상품2', img: 'image02.jpg' }  
]
```

일단 로컬에서 개발하기 때문에 순수 html 로 모두 구현할 수 있습니다만 어쨌든 서버는 추후에 필요하더라도 반드시 있어야 합니다.

서버프로그래밍을 하기 위해 서버가 필요합니다.

서버쪽이야 스프링 부트로 너무나 간단하고 쉽게 시작할 수 있습니다.

스프링 부트란?

Spring Boot (스프링 부트)는 스프링 프로젝트² 의 하나로

스프링 기반의 어플리케이션을 빠르게 구축할 수 방법을 제공합니다.

복잡한 의존성의 해결은 물론 스프링 부트의 소개 페이지에서 나오듯이 "Just Run" 할 수 있도록 대부분의 번거로운 설정을 자동화 합니다

서버 프로그래밍 준비

준비물은 스프링부트 CLI 와 텍스트 에디터가 필요합니다.

메모장이나 Sublime Text 나 각자 취향에 맞는 에디터만 있으면 됩니다.

CLI 는 왜 필요하나요?

스프링부트를 시작하려면 가능한 한 IDE(통합개발툴)도 필요하고 기본적인 빌드 환경도 필요합니다. 그런데 그 과정은 나중에 설명하기로 하고 빠른 프로토타입을 만들어야 할 경우에는 스프링부트 cli 가 적절한 선택입니다.

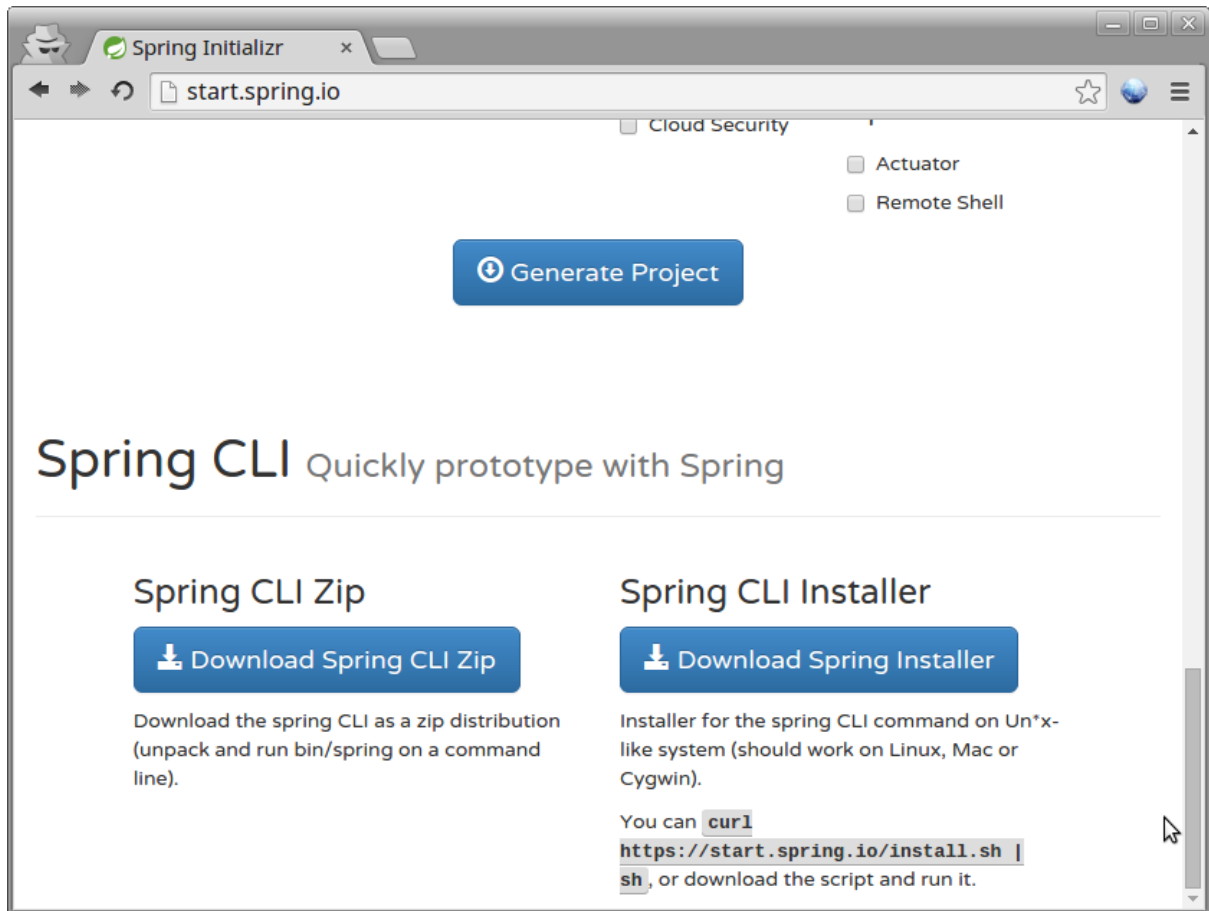
스프링 부트 CLI 는 어디서 얻나요?

<http://start.spring.io> (그림 10) 에서 다운로드 할 수 있습니다. 하지만 해당사이트의 내용이 자주 바뀌는 관계로 아래와 주소를 통해서 다운로드 가능합니다.

<http://repo.spring.io/release/org/springframework/boot/spring-boot-cli/1.2.5.RELEASE/spring-boot-cli-1.2.5.RELEASE-bin.zip>

² <http://spring.io>

그림 10 start.spring.io



만약 유닉스 호환 OS(맥, 리눅스) 를 사용하는 경우에는 curl 이나 gvm 를 통해서도 얻을 수 있습니다.

curl 를 통해서 설치하는게 가장 쉽습니다.

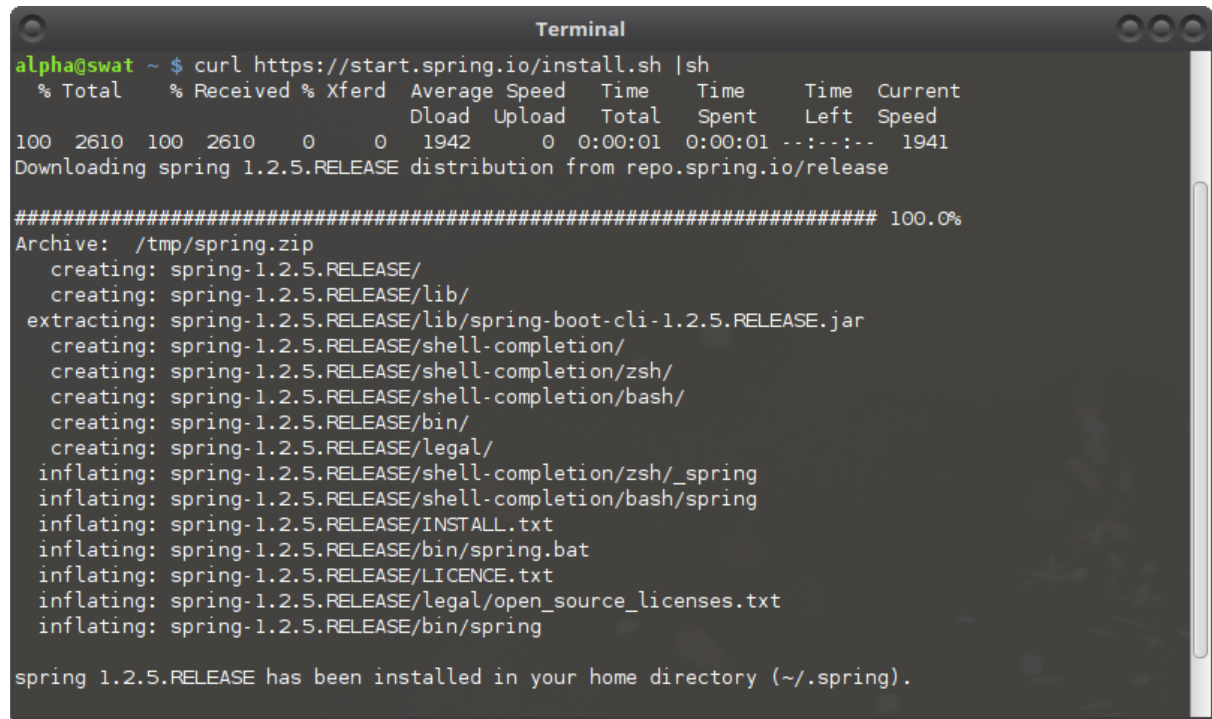
```
$ curl https://start.spring.io/install.sh | sh
```

curl를 통해 설치하면 사용자의 PATH에 바로 잡히기 때문에 바로 확인이 가능합니다.

명령어의 확인은 아래와 같습니다.

```
$ spring version
Spring CLI v1.2.5.RELEASE
```


그림 11 curl를 통한 설치

A terminal window titled "Terminal" showing the installation of Spring CLI. The user runs the command `curl https://start.spring.io/install.sh | sh`. The output shows a progress bar for downloading the distribution from `repo.spring.io/release`. After reaching 100.0%, it lists the files being created and inflated, including `spring-1.2.5.RELEASE/lib/spring-boot-cli-1.2.5.RELEASE.jar`, shell completion files for zsh and bash, and various scripts and licenses. The final message states: "spring 1.2.5.RELEASE has been installed in your home directory (~/.spring)."

```
alpha@swat ~ $ curl https://start.spring.io/install.sh | sh
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload  Total   Spent    Left   Speed
100 2610  100 2610    0     0  1942      0  0:00:01  0:00:01 --:--:-- 1941
Downloading spring 1.2.5.RELEASE distribution from repo.spring.io/release

##### 100.0%
Archive:  /tmp/spring.zip
  creating: spring-1.2.5.RELEASE/
  creating: spring-1.2.5.RELEASE/lib/
extracting: spring-1.2.5.RELEASE/lib/spring-boot-cli-1.2.5.RELEASE.jar
  creating: spring-1.2.5.RELEASE/shell-completion/
  creating: spring-1.2.5.RELEASE/shell-completion/zsh/
  creating: spring-1.2.5.RELEASE/shell-completion/bash/
  creating: spring-1.2.5.RELEASE/bin/
  creating: spring-1.2.5.RELEASE/legal/
  inflating: spring-1.2.5.RELEASE/shell-completion/zsh/_spring
  inflating: spring-1.2.5.RELEASE/shell-completion/bash/spring
  inflating: spring-1.2.5.RELEASE/INSTALL.txt
  inflating: spring-1.2.5.RELEASE/bin/spring.bat
  inflating: spring-1.2.5.RELEASE/LICENCE.txt
  inflating: spring-1.2.5.RELEASE/legal/open_source_licenses.txt
  inflating: spring-1.2.5.RELEASE/bin/spring

spring 1.2.5.RELEASE has been installed in your home directory (~/.spring).
```

GVM(Groovy enVironment Manager)를 통해서 설치하는 경우에는 gvm 를 통해서 프로그램을 버전 별로 설치 및 삭제 그리고 업데이트가 가능하다는 점입니다. 또한 추후에 소개할 Gradle 도 마찬가지로 쉽게 설치가 가능합니다.

다음은 gvm 을 통한 spring cli 설치 예입니다.

먼저 gvm 를 설치합니다 (그림 12)

그림 12 gvm설치

[illegible]

만약 설치 후 gvm 명령어를 사용할 수 없으면

설치 후 화면에 지시 사항대로 gvm 설 스크립트를 실행합니다

```
$ source .gvm/bin/gvm-init.sh
```

gvm 설치 후 spring cli 를 설치합니다. gvm 에서는 springboot 라는 이름으로 설치하면 됩니다.
버전을 명시하지 않으면 가장 최신버전을 설치합니다.

```
$ gvm install springboot
```

또는

```
$ gvm install springboot 1.2.5.RELEASE
```

```
$ spring -version
```

Spring CLI v1.2.5.RELEASE

준비물을 모두 갖추었으면 이제 아래와 같이 작성합니다.

아래의 내용을 app.groovy 로 저장합니다.

[app.groovy]

```
@RestController
class App {

    @RequestMapping("/")
    def index() {
        "Hello Spring"
    }
}
```

너무 짧은가요?

서버를 실행을 하려면 콘솔에서 아래와 같이 실행합니다.

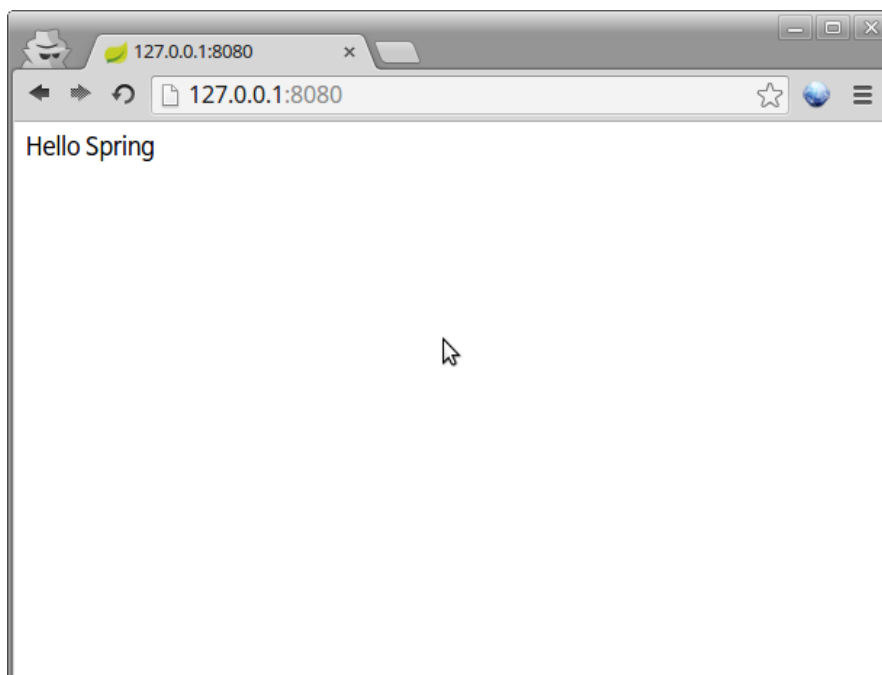
```
spring run app.groovy
```

최초 실행 시에는 CLI 툴이 스프링 부트 웹어플리케이션을 실행하기 위한 라이브러리를 다운로드 받기 때문에 시간이 조금 걸릴 수 있습니다.

일단 서버가 시작되면 웹브라우저로 <http://localhost:8080> 으로 접근해 봅니다.

"Hello Spring" 이라는 인사말을 볼 수 있습니다. (그림 13)

그림 13



서버를 종료하기 위해서는 Ctrl+C 를 누릅니다.

참고로 groovy 파일은 자바의 class 로 컴파일됩니다. 따라서 groovy 파일을 수정후 재확인을 하려면 서버를 다시 시작해야합니다.

뷰페이지 작성

index페이지를 보여주기위해서 뷰페이지를 작성합니다. 전통적으로 J2EE 환경에서 사용되는 뷰의 역할로 JSP를 주로 사용하였습니다. 여기서는 그루비템플릿(Groovy-template)를 사용 합니다. 그 외에 Thymeleaf, Freemarker, Velocity, Mustache 등의 템플릿을 사용할 수 있습니다.

[app.groovy]

```
@Grab("groovy-templates") ---❶
@Grab("org.webjars:bootstrap:3.3.4") ---❷
@Grab("org.webjars:angularjs:1.4.7") ---❸
@RestController
class App {

    @RequestMapping("/")
    def index() {
        new ModelAndView("index") ---❹
    }
}
```

- ❶ Grab를 사용함으로써 메이븐저장소에서 groovy-template이라는 라이브러리를 다운로드합니다
- ❷ bootstrap webjar 라이브러리를 다운로드합니다.
- ❸ angular webjar 라이브러리를 다운로드합니다.
- ❹ index.tpl이라는 그루비템플릿 파일을 반환합니다.

위 예제에서 @Grap 어노테이션은 Groovy Grape툴의 일부분으로서 메이븐저장소에서 외부 라이브러리를 다운로드 받는 역할을 합니다. @Grab를 사용하기 위한 문법은 Group ID: arifact ID:Verson 또는 artifact ID 만 사용합니다.

이제 템플릿 파일을 작성해야하는데 파일의 위치는 app.groovy가 있는 경로에 templates 라는 디렉토리 이하에 작성하면 됩니다. templates 디렉토리는 스프링부트에서 설정된 디렉토리입니다. 만약 디렉토리를 변경하려면 환경설정파일을 작성해야 하는데 이것은 나중에 다루도록 하겠습니다.

[index.tpl]

```

yieldUnescaped '<!doctype html>' ---❶
html('ng-app':"store") {
  head {
    title('Every Store')
    link(rel: 'stylesheet', href: 'webjars/bootstrap/3.3.4/css/bootstrap.min.css')
  }
  body{

    div(class: 'container') {
      div(class: 'navbar-header') {
        h1('Every Store')
      }
    }

    div(class: 'container' ) {
      div(class: 'panel panel-default') {
        div(class: 'panel-body' , 'ng-controller': 'GoodsController') { ---❷
          div(class: 'row') {

            div(class: "col-md-6", "ng-repeat": "item in goods" ) {
              div(class: "panel-body", "ng-cloak") {
                yieldUnescaped "" ---❸
                {{item.name}}
                
                "" ---❹
              }
            }
          }
        }
      }
    }
  }

  script(src: 'webjars/angularjs/1.4.7/angular.js') {}
  script(src: 'js/store.js'){ }
}

```

❶ yieldUnescaped 를 사용하면 그루비템플릿이 다음문장을 해석하지 않습니다

- ❷ ng-controller를 그대로 사용하면 컴파일 에러가 발생합니다. 그래서 쉽표로 표기합니다
- ❸ Angular 템플릿과 충돌을 피하기 위해서 yieldUnescaped를 사용합니다. 하지만 복수행 이기 때문에 "" 를 사용합니다. 다음 "" 가 나오기 전까지中间的 문자열을 무시합니다.
- ❹ yieldUnescaped의 종결 마침을 알립니다

index.tpl 은 그루비의 MarkupTemplateEngine 을 사용하여 서버에서 렌더링됩니다.
 index.tpl 에서 사용한 문법은 표 1 를 통해서 간략히 소개합니다. 상세한 문법은
<http://groovy-lang.org/templating.html> 에서 확인이 가능합니다.

표 1 그루비 마크업 예

그루비 마크업 소스	해석 후 결과
script { }	<script> </script>
yieldUnescaped ''' <script> alert('hello'); </script> '''	<script> alert('hello'); </script>
ul { li(class:'abc') { p("menu1") } }	 <li class='abc'> <p>menu1</p>
컴파일 에러발생 div("Click Here", ng-click: "hello();") 올바른 예 div("Click Here", "ng-click": "hello();")	ng-click 이라는 속성에 하이픈(-)이 포함되었기 때문에 따옴표를 사용합니다 <div ng-click='hello();'>Click Here</div>

[store.js]

```
angular.module('store', [])
```

```
  .controller('GoodsController', function($scope) {
```

```
    $scope.goods = [ ---❶
```

```
      { id: 1, name: "커피", price: 3000,  img: 'coffee.jpg'},
```

```
      { id: 2, name: "PS4", price: 400000, img: 'ps4.jpg' },
```

```
      { id: 3, name: "자전거", price: 200000, img: 'bicycle.jpg' }
```

```
    ]
```

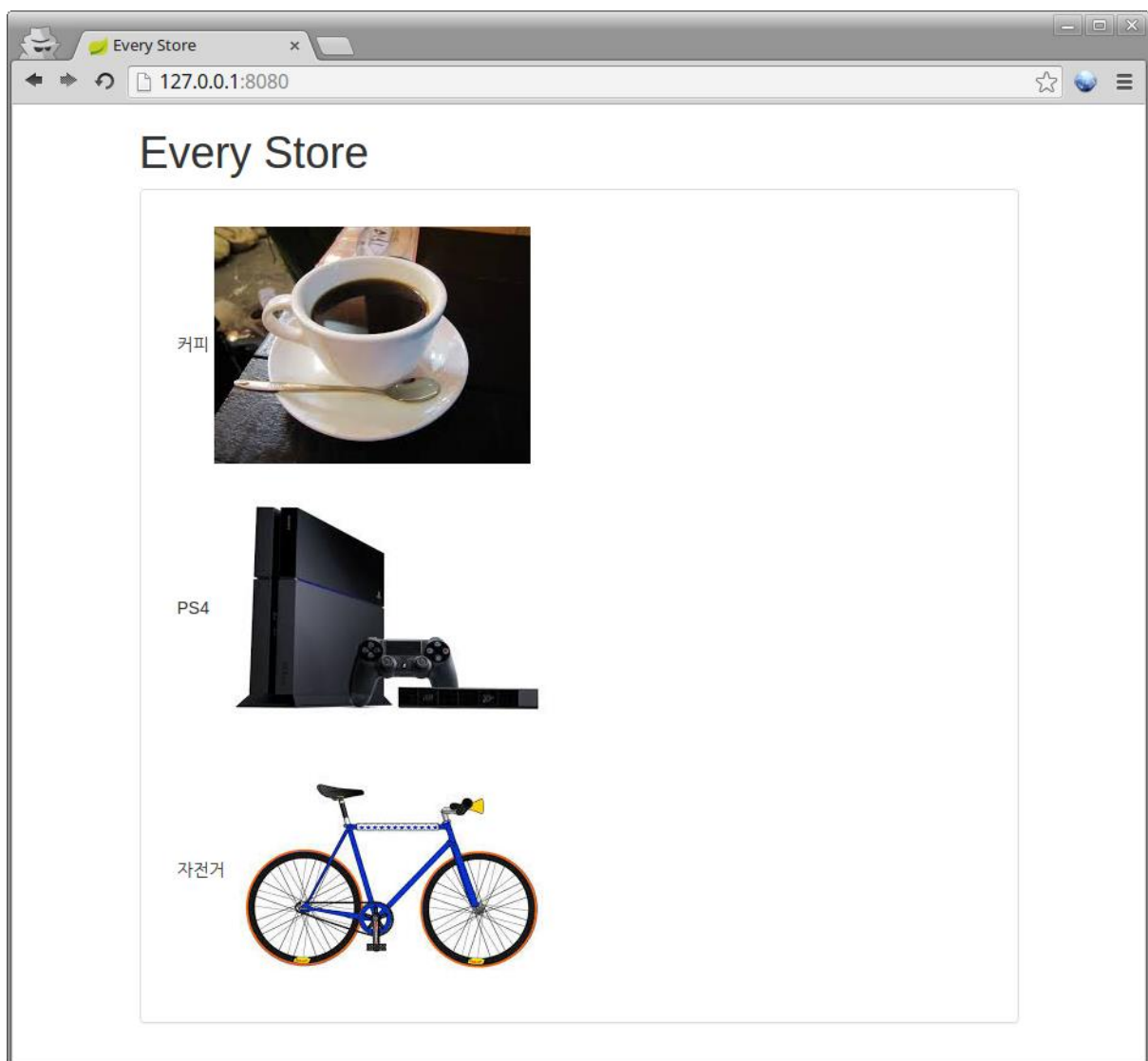
```
});
```

❶ scope 에 상품목록 데이터를 설정합니다. 일단은 임시로 클라이언트에서 모든 데이터를 가지고 있습니다.

이제 서버를 실행해서 <http://127.0.0.1:8080> 에 브라우저로 접근합니다.

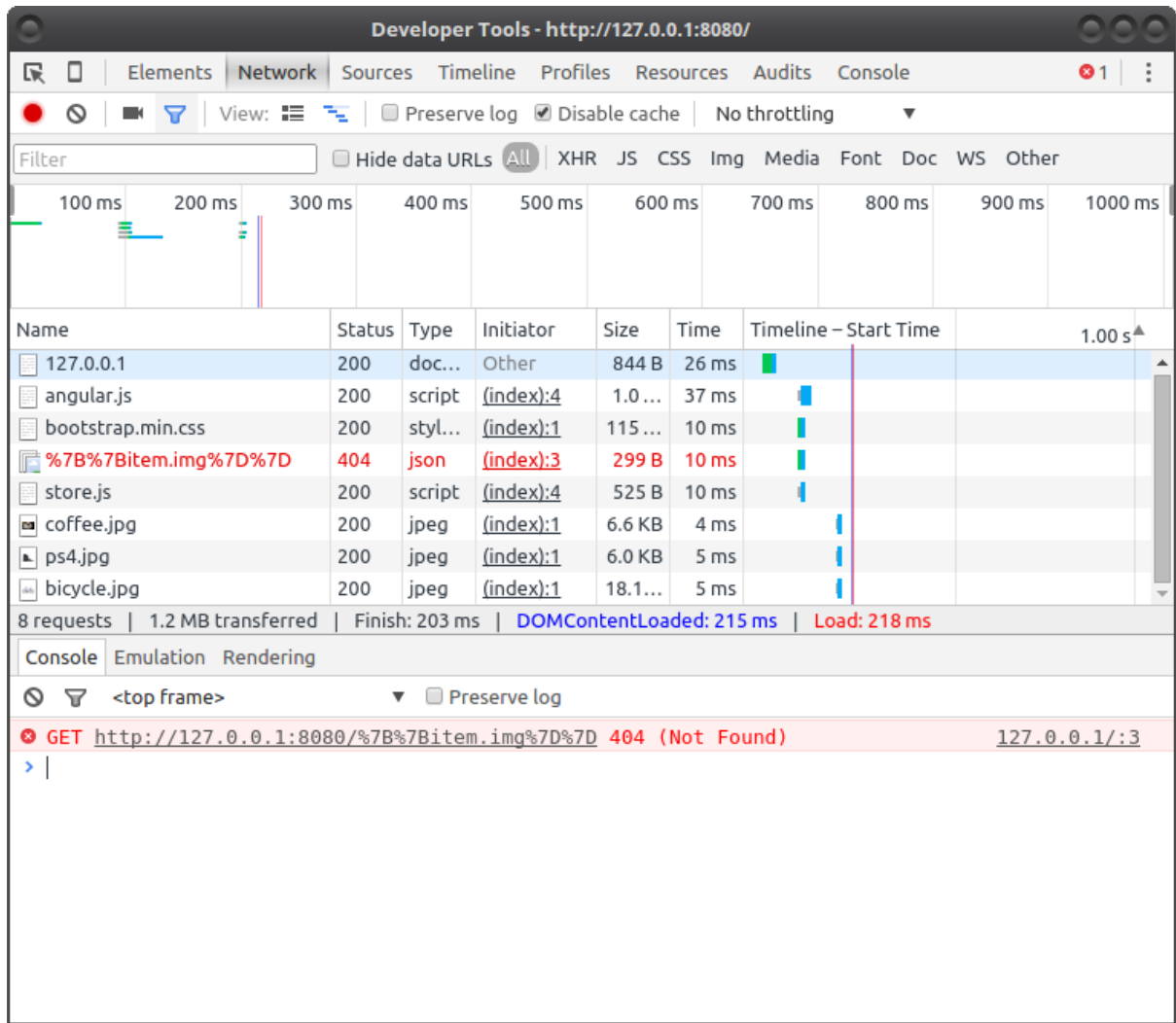
상품 목록 (그림 14)을 볼수가 있습니다.

그림 14



참고로 크롬의 경우 F12를 눌러서 개발모드로 볼 경우 특정 이미지를 못찾는 에러가 발생 (그림 15) 합니다.

그림 15



웹브라우저가 `` 해석하여 잘못된 이미지를 찾게 되서 발생하는 에러입니다. 이를 방지하기 위해서 소스의 `img` 부분을 아래와 같이 수정합니다

수정 전

```

```

수정 후

```

```

2.2 REST 서비스를 사용해봅시다

자바스크립트에서 제공하는 상품데이터 부분을 서버에서 제공하도록 수정합니다.

그러면 클라이언트에서 Ajax를 이용해서 데이터를 얻어야합니다. 다음은 REST api를 제공하는 서버의 소스입니다.

[app.groovy]

```
@Grab("groovy-templates")
@Grab("org.webjars:bootstrap:3.3.4")
@Grab("org.webjars:angularjs:1.4.7")
@RestController
class App {

    def goods = [ ---❶
        [ id: 1, name: "커피", price: 3000, img: 'coffee.jpg' ],
        [ id: 2, name: "PS4", price: 400000, img: 'ps4.jpg' ],
        [ id: 3, name: "자전거", price: 200000, img: 'bicycle.jpg' ]
    ]

    @RequestMapping("/")
    def index() {
        new ModelAndView("index")
    }

    @RequestMapping("/api/goods")
    def goods() {
        goods ---❷
    }
}
```

❶ java의 List문법입니다. List 안에 Map구조로 데이터를 넣습니다.

❷ 그루비 문법의 특징인 return 구문 없이 값을 돌려줍니다. 여기서는 상품리스트를 반환합니다.

\$http 서비스를 사용합니다

Angular의 \$http 서비스를 사용합니다. \$http는 DI를 통해서 삽입됩니다.

상품정보를 가져오는 서비스를 별도로 분리하여 REST 서비스를 사용하도록 합니다.

여기서 \$http를 사용한 후 돌려주는 값은 Promise³ 입니다. 즉 데이터를 취득 후 바로 처리 하지 않고 나중에 미룬다는 의미입니다. 그래서 비동기 처리에 사용하기가 용이 합니다.

참고로 앞으로 진행하는 store.js 의 소스 내용은 체이닝패턴⁴을 사용해서 기술합니다.

[store.js]

```
angular.module('store', [])

.service('GoodService', function($http) { ---❶

    this.getData = function() {

        return $http({

            method: 'GET',

            url: '/api/goods'

        });

    }

})

.controller('GoodsController', function($scope, GoodService) { ---❷

    $scope.goods = [];

    GoodService.getData().success(function(response, status, config, headers) { ---❸

        console.log('Response from server: ', response);

        $scope.goods = response;

    }).error(function(response, status, config, headers){

        alert("Error");

    });

});
```

❶ GoodService를 작성하고 상품정보를 가져오도록 위임합니다.

❷ GoodsController에서 GoodService를 사용하여 API를 호출합니다

❸ 상품 정보 호출 후 스코프에 전달하면 화면에서 상품 목록이 출력이 됩니다.

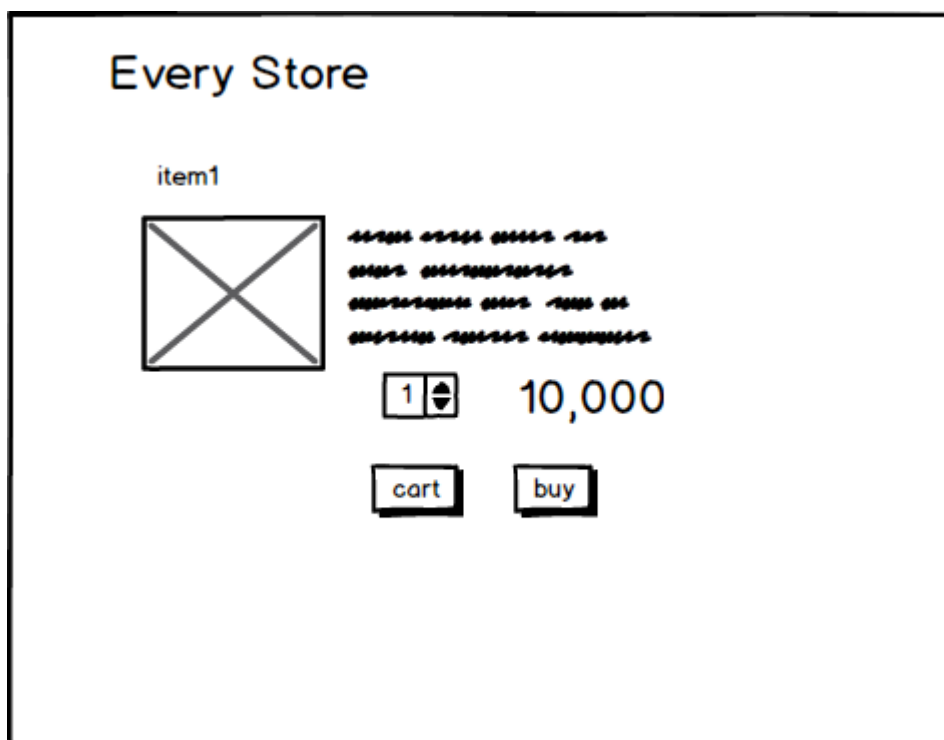
³ [https://code.angularjs.org/1.4.7/docs/api/ng/service/\\$q](https://code.angularjs.org/1.4.7/docs/api/ng/service/$q)

⁴ https://en.wikipedia.org/wiki/Method_chaining#jQuery

2.3 상품 상세화면

이번 절부터 더이상 그루비 템플릿을 사용하지 않습니다. 대신 Angular의 템플릿을 사용합니다. 처음 보이는 화면 즉 index.tpl은 그대로 유지합니다. 왜냐하면 처음 로딩 시 자바스크립트를 로딩해야하는 화면이 필요하기 때문입니다.

그림 16 상품상세화면



라우터를 사용하여 화면이동

라우터를 사용하면 화면의 주도권은 모두 클라이언트가 가지게 됩니다.

즉 화면이동이 해쉬태그를 사용하여 이동합니다.

해당 페이지로 이동하는 컨트롤러를 만들고 html조각을 렌더링하기 위해서 ngRoute를 사용합니다. 그리고 html 템플릿을 출력 하기 위해서 <ng-view>라는 지시어를 사용합니다.

상품목록 화면과 상품상세화면을 모두 ngRoute를 사용하기 위해서 뷰페이지를 분리해야 합니다. store.js에서 불러와야할 템플릿 페이지를 아래와 같이 정의 합니다.

[store.js]

```
angular.module('store', ['ngRoute']) ---❶
    .config(function($routeProvider) {

        $routeProvider ---❷
        .when('/:id', {
            templateUrl: 'views/detail.html',
            controller: 'DetailController'
        })
        .otherwise({
            templateUrl: 'views/index.html',
            controller: 'GoodsController'
        });

    })

    .service('GoodService', function($http) {
        this.getData = function() {
            return $http({
                method: 'GET',
                url: '/api/goods'
            });
        };

        this.getDetail = function(id) { ---❸
            return $http({
                method: 'GET',
                url: '/api/goods/' + id
            });
        }
    })

    .controller('GoodsController', function($scope, GoodService) {

        $scope.goods = [];

        GoodService.getData().success(function(response, status, config, headers) {
            console.log('Response from server: ', response);
```

```

        $scope.goods = response;
    }).error(function(response, status, config, headers){
        alert("Error");
    });
})
.controller('DetailController', function($scope, GoodService, $routeParams) { ---❶
    $scope.good = {};
    GoodService.getDetail($routeParams.id).success(function(response, status, config, headers) {
        console.log('Detail from server: ', response)
        $scope.good = response;
    }).error(function(response, status, config, headers){
        alert("Error");
    });
})
.filter("newLine", function() { ---❷
    return function(text){
        return text? text.replace(/\n/g, '<br/>'): "";
    }
});

```

❶ ngRoute 모듈을 설정합니다

❷ 라우터를 설정하기 위해서 routeProvider를 사용하고 각 라우터 경로를 담당할 컨트롤러와 템플릿을 지정합니다.

❸ 상품 상세 데이터를 가져오기 위해 http를 사용합니다

❹ 상세화면으로 진입하는 컨트롤러에서 서비스를 통해 결과를 바인딩합니다

❺ 상품의 상세설명에 개행이 있을 경우
 태그로 치환하기 위해 필터를 추가합니다.

이제 템플릿을 아래와 같이 라우터 별로 재작성합니다.

[views/index.html]

```

<div class="panel panel-default" ng-cloak>
    <div class="panel-body">

        <div class="col-md-6" ng-repeat="item in goods" ng-show="goods.length > 0">
            <div class="panel-body">
                <div>
                    <h3>{{item.name}}</h3>

```

```
        
        <div class="text-right">
            <a class="btn btn-primary" href="#{{item.id}}">선택</a>
        </div>
    </div>
</div>
</div>
</div>
```

[views/detail.html]

```
<div class="panel panel-default" ng-cloak>
    <div class="panel-body">
        <div class="text-left"><a href="/">처음</a></div>
        <div class="panel-body">
            <div class="row">
                <div class="col-xs-12 col-sm-6"><h3>{{good.name}}</h3></div>
            </div>
            <div class="row">
                <div class="col-xs-12 col-sm-6">
                    <div class="row">
                        <div> {{good.desc|newLine}}</div> ---❶
                    </div>
                    <div class="row">
                        <div class="col-sm-6" ng-init="qty=1">
                            <input type="number" size="2" style="width:50px;" ng-model="qty" >
                                {{ qty* good.price|currency:"":0}} 원 ---❷
                        </div>
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>
<p/>
<p/>
```

```
<div class="row text-center">
  <button class="btn btn-primary">Add to Cart</button>
  <button class="btn btn-default">Buy now</button>
</div>
</div>
</div>
</div>
```

- ❶ 상품상세설명 (good.desc)에 개행을 처리하기 위해 필터를 사용합니다
- ❷ 앵글러 템플릿표현식을 사용하여 가격 계산을 처리하고 currency라는 지시어를 적용합니다

index 뷰페이지에서는 상품목록을 열거하는 부분이 템플릿으로 교체됩니다.

[index.tpl]

```
yieldUnescaped '<!doctype html>'
html('ng-app':"store") {
  head {
    title('Every Store')
    link(rel: 'stylesheet', href: 'webjars/bootstrap/3.3.4/css/bootstrap.min.css')
  }
  body{
    div(class: 'container') {
      div(class: 'navbar-header') {
        h1('Every Store')
      }
    }

    div(class: 'container') {
      "ng-view" {} ---❶
    }

    script(src: 'webjars/angularjs/1.4.7/angular.js') {}
    script(src: 'webjars/angularjs/1.4.7/angular-route.js') {}
    script(src: 'js/store.js'){}
  }
}
```

❶ ng-view 지시어를 여기 사용합니다. 라우터에 해당하는 템플릿이 로딩될 위치입니다.

상품의 상세 정보 API 작성

상품의 상세설명을 추가위해 goods를 수정하고 상세정보를 돌려주는 메서드를 추가합니다.

[app.groovy]

```
@Grab("groovy-templates")
@Grab("org.webjars:bootstrap:3.3.4")
@Grab("org.webjars:angularjs:1.4.7")
@RestController
class App {

    def goods = [
        [ id: 1, name: "커피", price: 3000,  img: 'coffee.jpg' , desc: '맛 있는 커피'],
        [ id: 2, name: "PS4", price: 400000, img: 'ps4.jpg' , desc: '이번이 마지막 찬스 \n 최강의
콘솔게임머신'],
        [ id: 3, name: "자전거", price: 200000, img: 'bicycle.jpg' , desc: '그림에 나오는 자전거']
    ]

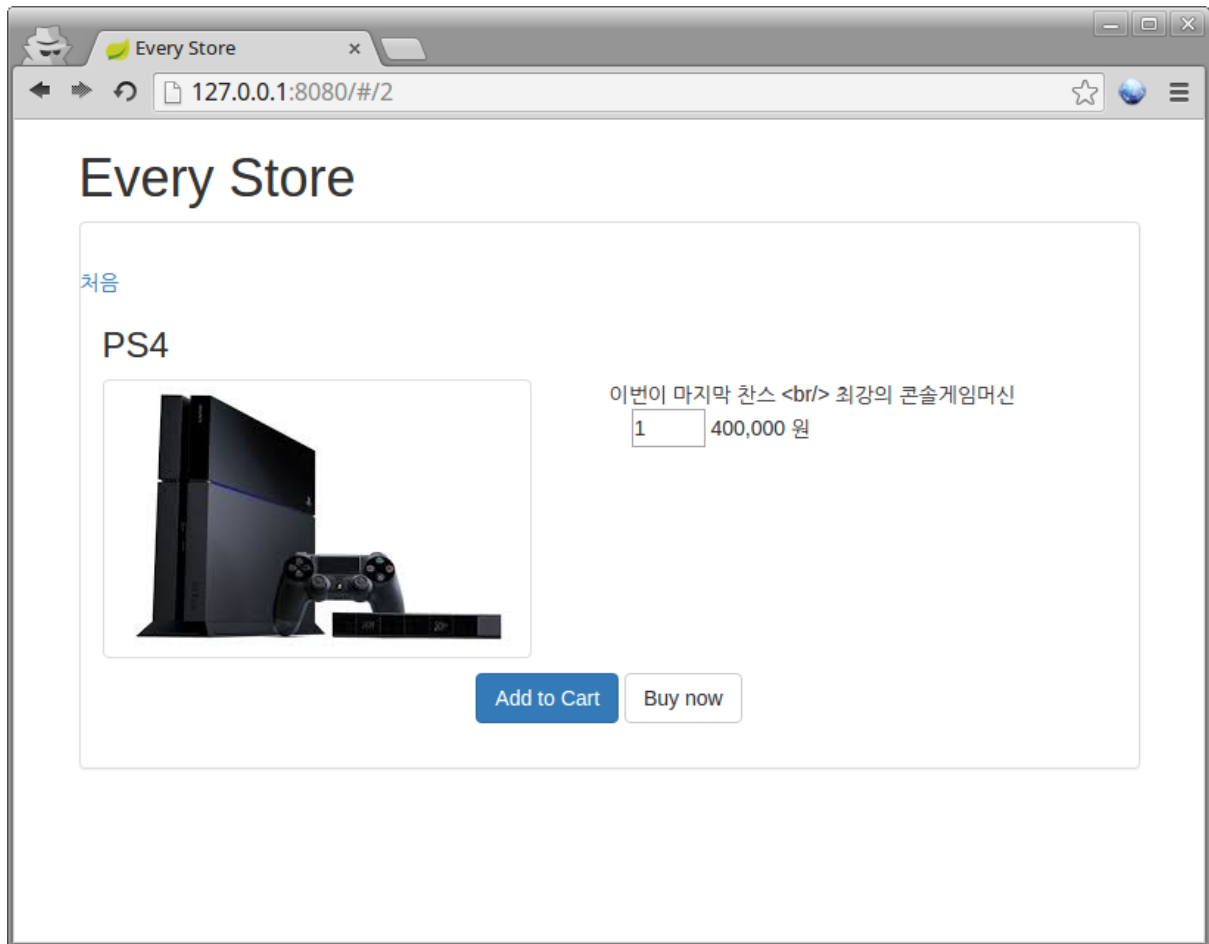
    ..생략..

    @RequestMapping("/api/goods/{id}") ---❶
    def good(@PathVariable Integer id) {
        println("=> " + id)
        goods.find { good -> good.id == id }
    }
}
```

❶ 아이디로 상품목록데이터에서 상품을 찾습니다.

실행된 결과를 보기 위해 상품목록에서 상품을 클릭하여 해당 상세정보를 이동합니다. 라우터가 작동되는 것을 알수 있습니다. 가령 아이디가 2인 상품의 상세정보를 보여준 화면은 주소에 #/2 라는 상품 아이디가 표기됩니다.

그림 17 상세화면 보기



상품 상세보기에서 상품 설명의 개행이 `
`로 치환되어 그대로 노출됩니다. 앵귤러에서는 XSS 공격을 막기 위해서 허가 되지 않은 html 태그는 사용하지 못합니다. 그래서 사용자가 제공한 데이터의 html 태그만 허용하기 위해서 `ng-bind-html`이라는 지시어를 사용합니다. `ng-bind-html`를 사용하기 위해서 `ngSanitize` 모듈이 필요합니다.

아래와 같이 해당 파일들을 수정 합니다.

Extra 모듈이기 때문에 아래와 같이 별도로 추가합니다.

[index.tpl]

..생략

```
script(src: 'webjars/angularjs/1.4.7/angular-sanitize.js') {}
```

의존성 주입을 위해 추가합니다.

[store.js]

```
angular.module('store', ['ngRoute', 'ngSanitize'])
```

```
...생략
```

ng-bind-html 지시어로 수정합니다.

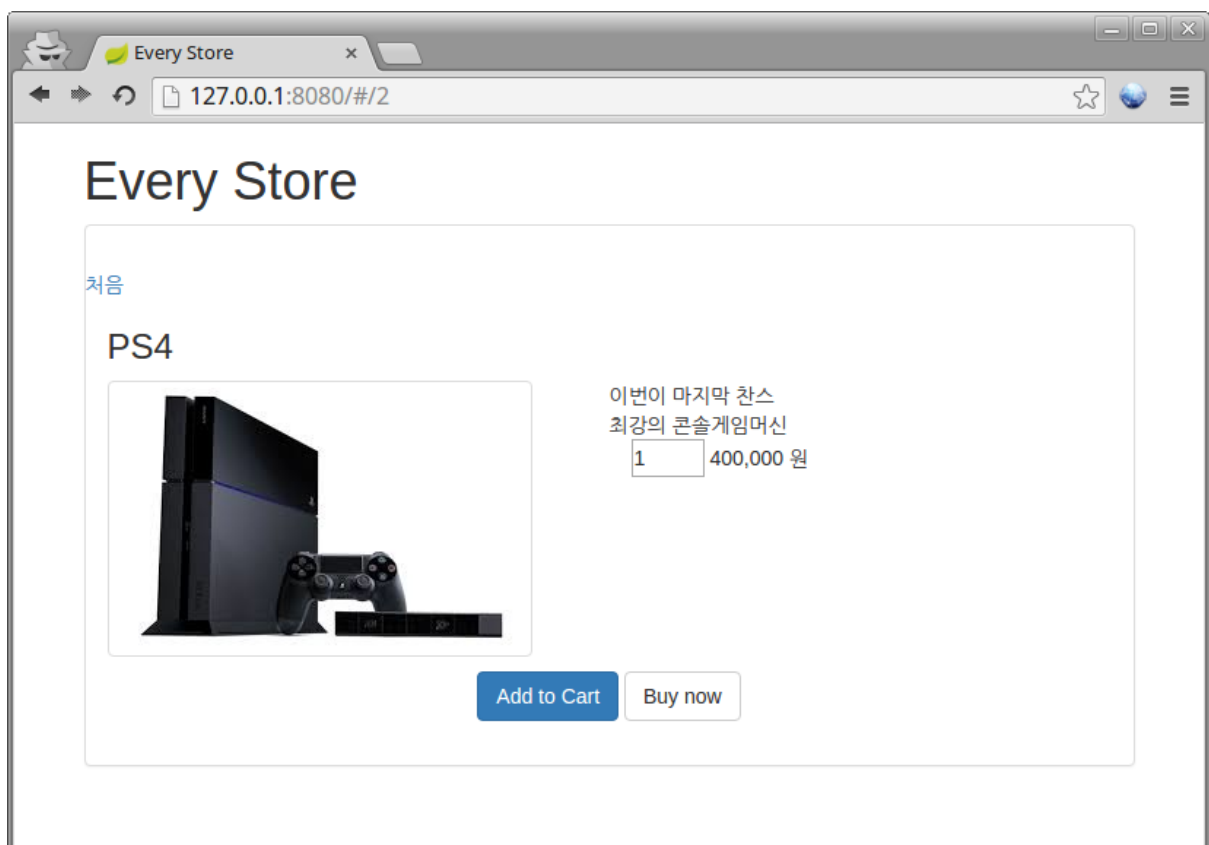
[views/detail.html]

```
...생략
```

```
<div ng-bind-html="good.desc|newLine"></div>
```

실행 후
이 제대로 작동합니다.

그림 18 ngSanitize 적용 후



2.4 장바구니

이제 상품의 상세화면에서 바로 장바구니에 담을 수 있도록 합니다.

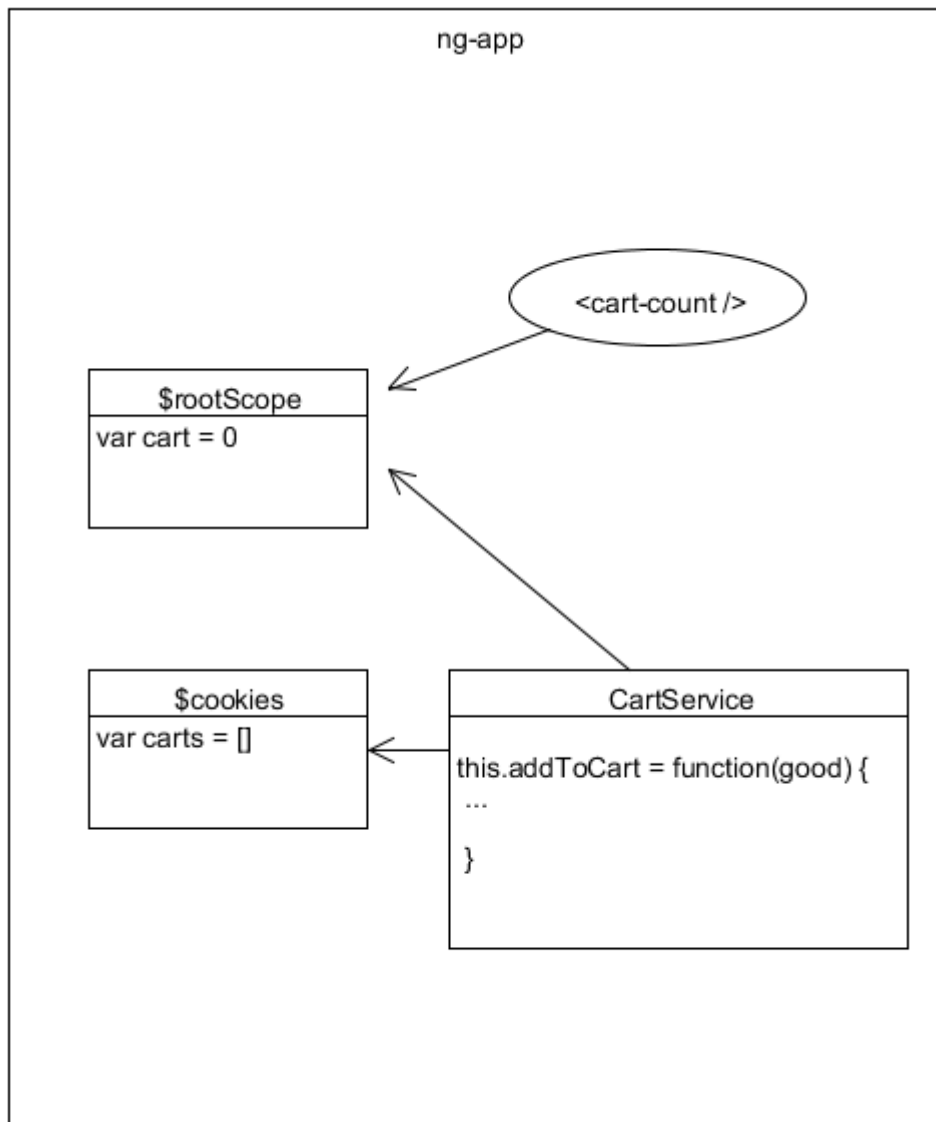
장바구니에 담은 즉시 쿠키에 저장하고 화면상의 피드백으로 장바구니에 담겨 있는 모든 상품의 총개수를 돌려주고 상단의 장바구니 카운터 마크에 표시하도록 처리합니다. 반대로 장바구니 목록 화면에서 삭제하면 쿠키에서 해당 상품을 삭제하고 장바구니 카운터에도 반영이 되도록 해야 합니다.

scope를 통해 데이터를 공유합니다

각 컨트롤러들은 자신만의 스코프를 가지고 있습니다. 컨트롤러끼리 데이터를 공유하기 위해 몇 가지 방법이 있습니다만, 여기서는 루트스코프(\$rootScope)를 이용합니다.

루트 스코프는 최상단에 있는 영역입니다. 따라서 장바구니 카운터에서는 루트 스코프에서 카운트를 가져오고 DetailController에서는 루트 스코프의 카운트를 서비스를 통해서 수정할 수 있습니다.

그림 19 \$rootScope



\$rootScope는 글로벌 영역이라서 오염될 수 있습니다.

<https://code.angularjs.org/1.4.7/docs/misc/faq> 를 참고하십시오.

장바구니 카운트 지시어 작성

지시어(directive)를 사용하여 장바구니 카운트를 작성합니다. 지시어를 사용하여 작성하면 재활용하기가 쉽습니다. 해당 지시어 태그의 이름만 작성하면 그만입니다.

[scope.js]

```
.directive('cartCount', function($rootScope, CartService) { ---❶
```

```
    return {
      restrict: "E", ---❷
      templateUrl: 'views/cartCount.html', ---❸
      link: function(scope, elem, attrs) { ---❹
        $rootScope.count = CartService.count();
      }
    }
  })
```

❶ \$rootScope와 CartService를 추가합니다.

❷ 해당 지시어가 Element 라는 것을 명시합니다

❸ 지시어에서 사용할 템플릿을 지정합니다

❹ 루트스코프의 변수에 장바구니의 총 상품개수를 할당합니다. 최초 화면이 로딩될 때 한번만 읽혀집니다. 그 후로는 컨트롤러에서 서비스를 통해 루트스코프를 변경하게 됩니다.

다음은 지시어에서 사용할 템플릿 파일입니다.

[views/cartCount.html]

```
<div class="navbar-right">
  <div class="navbar-text">
    <a href="#/cart">
      <span class="glyphicon glyphicon-shopping-cart"></span>
      Cart:
      {{count}} ---❶
    </a>
  </div>
</div>
```

❶ count 변수는 \$rootScope 에서 획득합니다.

장바구니 CRUD 서비스

장바구니 상품의 CRUD를 담당하는 CartService 를 작성하고 DetailController에서 사용하도록 합니다.

[store.js]

...생략

```
.service('CartService', function($rootScope, $cookies ){

    this.addToCart = function(good) { ---❶

        var find = false;

        var carts = this.getCart();

        carts.forEach(function(val) {

            if(val.id == good.id) {

                val.qty += good.qty

                find = true

            }

        });

        if(!find) {

            carts.push(good);

        }

        $cookies.putObject('carts', carts);

        $rootScope.count = this.count();

    };

    this.getCart = function() { ---❷

        return $cookies.getObject('carts') || [];

    };

    this.count = function() { ---❸

        var cnt = 0;

        var carts = this.getCart();

        carts.forEach(function(val) { cnt += val.qty ; });

        return cnt;

    };

};
```

```

this.remove = function(good) { ---❹
    var carts = this.getCart();
    if(carts.length == 1) {
        carts = [];
    }else {
        carts.forEach(function(val, index) {
            if(val.id == good.id) {
                carts.splice(index, 1);
            }
        });
    }

    $cookies.putObject('carts', carts);
    $rootScope.count = this.count();
    return carts;
}

```

- ❶ 상품을 쿠키에 담거나 같은 상품일 경우 수량을 수정하여 다시 쿠키에 저장합니다
- ❷ 쿠키에서 장바구니를 가져옵니다
- ❸ 장바구니 안의 상품의 총수량을 돌려줍니다
- ❹ 인자값으로 주어진 상품을 장바구니에서 삭제하고 다시 쿠키를 저장합니다

장바구니 담기

DetailController에 장바구니에 상품을 담기 위해서 서비스호출 부분을 아래와 같이 추가합니다.

[store.js]

...생략

```

.controller('DetailController', function($scope, GoodService, CartService, $routeParams) {

    $scope.addToCart = function(good, qty) {
        good.qty = qty;
        CartService.addToCart(good); ---❶
    };
})

```

...생략

템플릿에서 장바구니에 상품을 담기 위한 함수를 버튼에 연결만 하면 됩니다.

[views/detail.html]

수정 전

```
<button class="btn btn-primary">Add to Cart</button>
```

수정 후

```
<button class="btn btn-primary" ng-click="addToCart(good, qty)">Add to Cart</button>
```

이제 버튼을 클릭하면 수량과 상품정보를 받아서 CartService를 통해서 쿠키에 저장이 되고 장바구니 카운터를 갱신 할 것입니다.

장바구니 목록

다음은 장바구니 목록 화면의 소스 입니다. 장바구니에 상품이 있으면 상품목록을 보여주고 없을 경우에는 간단한 메시지를 출력합니다.

[views/cart.html]

```
<h3> 나의 장바구니 </h3>
```

```
<div ng-repeat="good in carts" ng-if="carts.length > 0">
```

```
  <div class="panel panel-default">
```

```
    <div class="panel-body">
```

```
      <div class="row">
```

```
        <div class="col-sm-8">
```

```
          <div class="col-sm-5"><h4><a href="#{{good.id}}">{{good.name}}</a> </h4></div>
```

```
          <div class="col-sm-3">
```

```
            <input type="number" ng-model="good.qty">
```

```
            {{good.qty * good.price |currency:"":0}} 원
```

```
          </div>
```

```

        </div>
        <div class="col-sm-4 text-right">
            <button class="btn btn-default btn-lg" ng-click="removeFromCart(good);"> ---❶
                <span class="glyphicon glyphicon-trash"></span></button>
            </div>
        </div>
    </div>
</div>
<div ng-if="carts.length ==0" class="panel panel-default"> ---❷
    <div class="panel-body">
        장바구니가 비어 있습니다
    </div>
</div>
<hr/>
<div class="text-right"> 총합계: {{ sum() | currency:"":0 }} 원 </div> ---❸

```

- ❶ 해당버튼을 클릭 시 해당 상품을 장바구니에서 삭제합니다
- ❷ 장바구니가 비어 있을 경우 메시지를 출력합니다
- ❸ 장바구니에 담겨 있는 모든 상품의 수량과 가격을 합산하여 출력합니다

장바구니화면에 대한 라우팅 설정과 함께 컨트롤러를 작성합니다.

[store.js]

```

angular.module('store', [ 'ngRoute', 'ngSanitize', 'ngCookies'])
    .config(function($routeProvider) {

        $routeProvider
            .when('/cart', { ---❶
                templateUrl: 'views/cart.html',
                controller: 'CartController'
            })
            .when('/:id', {
                templateUrl: 'views/detail.html',
                controller: 'DetailController'
            })
            .otherwise({

```

```
        templateUrl: 'views/index.html',
        controller: 'GoodsController'
    });
})
...생략
.controller('CartController', function($scope, CartService) {
    $scope.carts = CartService.getCart(); ---❷

    $scope.sum = function() { ---❸
        var sum = 0;
        $scope.carts.forEach(function(val){
            sum += val.price * val.qty
        });
        return sum;
    };

    $scope.removeFromCart = function(good) { ---❹
        $scope.carts = CartService.remove(good);
    }
})
```

❶ 장바구니 가기에 대한 라우트를 설정합니다

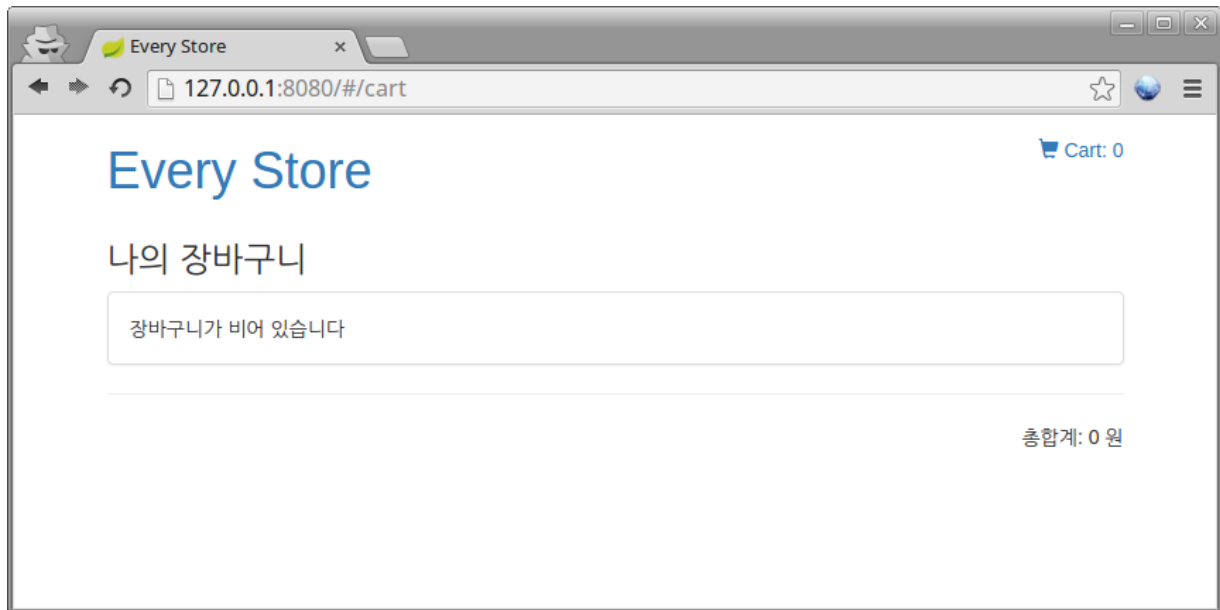
❷ 장바구니 화면으로 이동하면 최초로 장바구니 목록을 쿠키로부터 가져옵니다

❸ 장바구니에 담겨 있는 모든 상품의 수량과 가격을 계산하여 합산합니다

❹ 장바구니에서 해당 상품을 삭제합니다

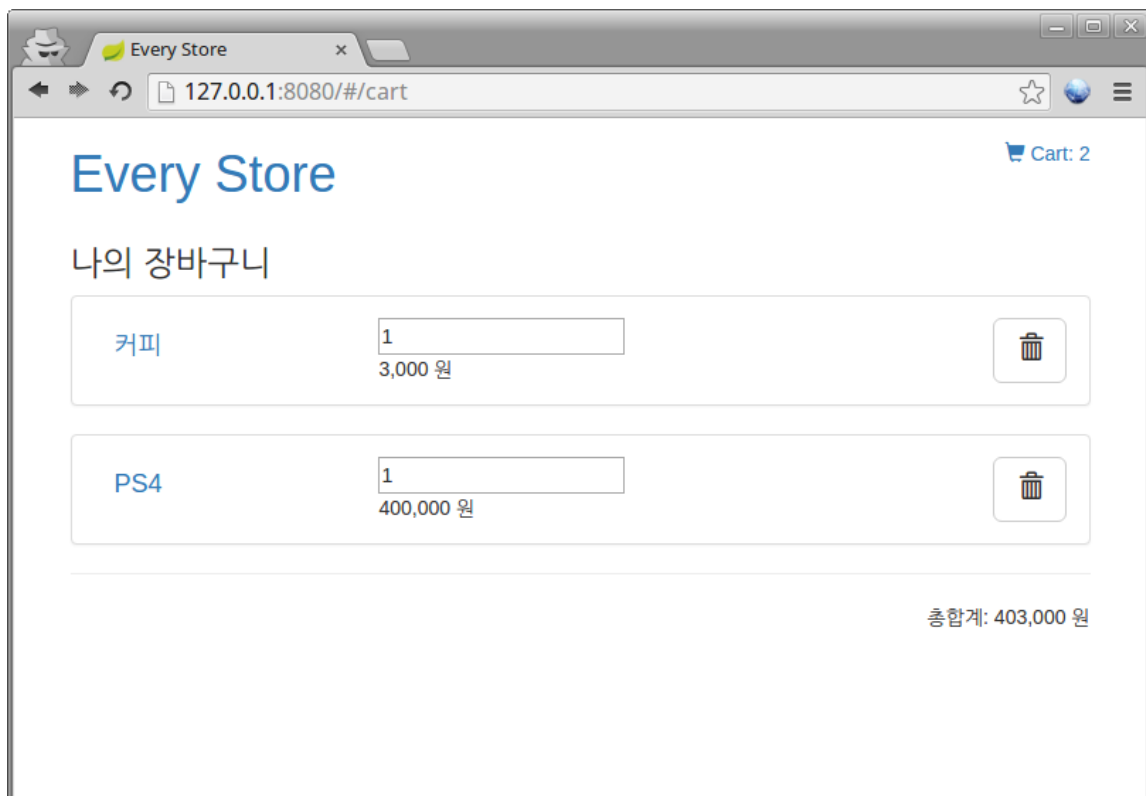
이제 웹브라우저에서 장바구니에 접근해 봅니다. 그림 20과 같이 빈 장바구니로 나타납니다.

그림 20 빈 장바구니



특정 상품을 장바구니에 담아 봅니다. 다시 장바구니로 이동하면 오른쪽 최상단에 전체 상품의 수량이 표기되고 장바구니 목록이 나타납니다. 여기에서 삭제 버튼을 클릭하면 해당 상품이 목록에서 삭제되고 바로 장바구니 수량도 바뀝니다.

그림 21 장바구니에 상품이 있는 경우




2.5 사용자 코멘트 화면

코멘트 화면은 상품에 대한 사용자의 코멘트 출력과 코멘트를 입력하는 화면으로 대략 그림 22와 같습니다.

그림 22

Every Store

item1



1

10,000

cart

buy

user1

user2

comment

코멘트 목록 가져오기

서버에서 코멘트 데이터와 목록을 가져올 수 있는 API를 작성합니다

[app.groovy]

```
@Grab("groovy-templates")
@Grab("org.webjars:bootstrap:3.3.4")
@Grab("org.webjars:angularjs:1.4.7")
@RestController
class App {
    def goods = [
        [ id: 1, name: "커피", price: 3000, img: 'coffee.jpg', desc: '맛 있는 커피'],
        [ id: 2, name: "PS4", price: 400000, img: 'ps4.jpg', desc: '이번이 마지막 찬스 \n 최강의
콘솔게임머신'],
        [ id: 3, name: "자전거", price: 200000, img: 'bicycle.jpg', desc: '그림에 나오는 자전거']
    ]

    def comments = [ ---❶
        [id: 1, writer: "김철수", desc: "별로예요", good_id : 1],
        [id: 2, writer: "홍길동", desc: "괜찮은 것 같아요", good_id: 1] ,
        [id: 3, writer: "김철수", desc: "갖고 싶네요!", good_id : 2],
    ]

    ..생략..

    @RequestMapping("/api/goods/{id}/comments") ---❷
    def comments(@PathVariable Integer id) {
        comments.findAll { comment -> comment.good_id == id }
    }
}
```

❶ 코멘트 데이터입니다

❷ 상품 아이디에 해당하는 코멘트 목록을 돌려줍니다

상품 상세 화면으로 이동하면 CommentController에서 CommentService 를 사용하여 서버로부터 코멘트 목록을 얻고 화면에 출력 합니다. 클라이언트에 서비스와 컨트롤러를 아래와 같이 추가합니다.

[store.js]

...생략

```
.service('CommentService', function($http) {  
    this.getComments = function(id) { ---❶  
        return $http({  
            method: 'GET',  
            url: '/api/goods/' + id + '/comments'  
        });  
    };  
  
    this.addComment = function(id, comment) { ---❷  
        return $http({  
            method: 'POST',  
            data: comment,  
            url: '/api/goods/' + id + '/comments'  
        });  
    }  
})
```

...생략

```
.controller('CommentController', function($scope, CommentService, $routeParams) {  
    $scope.comments = [];  
  
    CommentService.getComments($routeParams.id).success(function(response, status, config, headers){  
        $scope.comments = response; ---❸  
    });  
  
    $scope.addComment = function() { ---❹  
        //코멘트를 저장하는 부분을 여기에..  
    }  
})
```

❶ 서버로부터 코멘트 목록을 얻어옵니다

❷ 서버에 코멘트를 작성합니다

❸ 코멘트서비스를 이용하여 코멘트목록을 가져옵니다

❹ 코멘트를 저장하는 부분입니다. 앞으로 구현해야할 부분입니다

코멘트 추가

코멘트 목록과 함께 코멘트를 남길 수 있도록 입력폼을 구현합니다. 상품 상세 화면의 하단에 코멘트 목록과 코멘트 입력폼을 구현합니다.

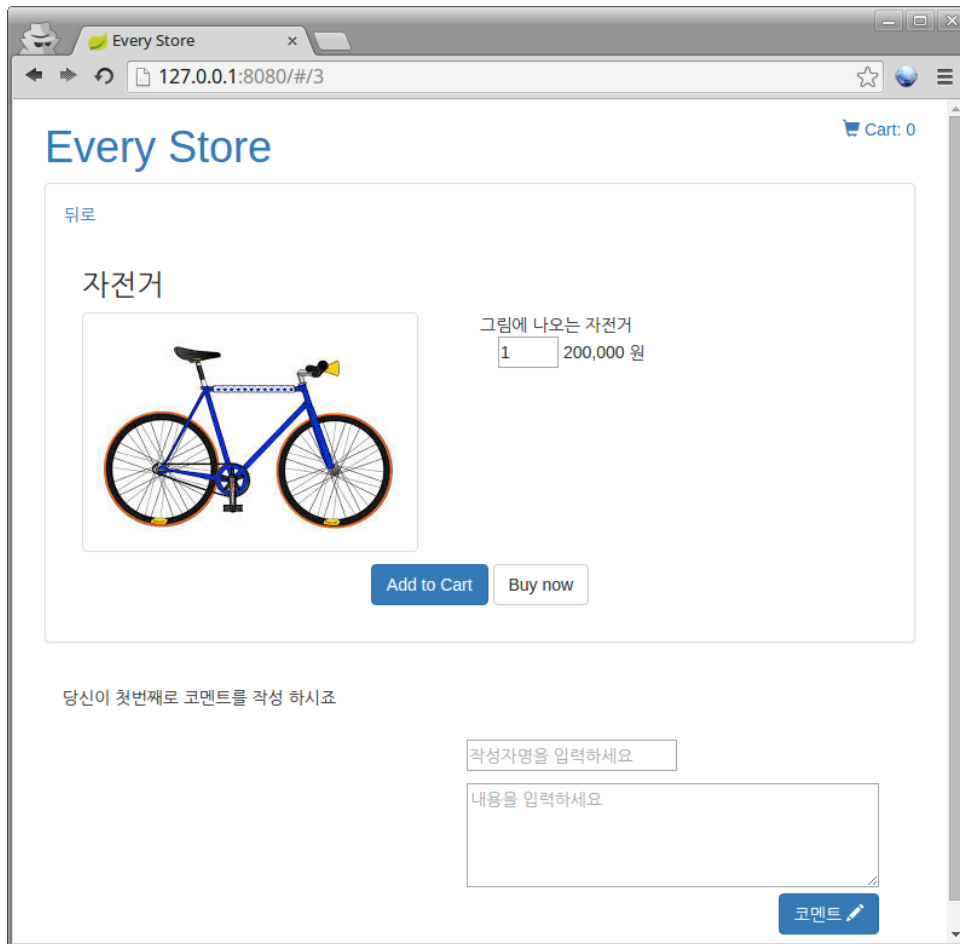
[views/detail.html]

```
<div class="panel-body" ng-controller="CommentController">
  <p ng-if="comments.length > 0">코멘트 해주시죠. 네? </p>
  <p ng-if="comments.length == 0">당신이 첫번째로 코멘트를 작성 하시죠</p>

  <div ng-repeat="comment in comments" ng-show="comments.length > 0">
    <div class="row">
      <div class="col-sm-2">
        <b>{{comment.writer}}</b>
      </div>
      <div class="col-sm-10" ng-bind-html="comment.desc | newLine"></div>
    </div>
  </div>

  <div class="panel-body">
    <div class="pull-right">
      <p><input type="text" ng-model="comment.writer" placeholder="작성자명을 입력하세요">
      </p>
      <textarea rows="4" cols="40" ng-model="comment.desc" placeholder="내용을
      입력하세요"></textarea>
      <div class="text-right">
        <button class="btn btn-primary" ng-click="addComment()">코멘트 <span class="glyphicon
        glyphicon-pencil"></span></button>
      </div>
    </div>
  </div>
</div>
```

그림 23 코멘트 입력 폼



코멘트 입력 시 Validation 처리

코멘트를 작성 시 작성자와 내용이 반드시 필요합니다. 그래서 에러를 검출하고 화면에 피드백을 주려면 클릭 시 발생하는 이벤트 처리에서 검증 로직을 추가합니다. CommentController에 아래와 같이 검증 로직을 추가합니다. 입력 폼에서 입력한 항목들을 모델로 바인딩하고 각 프로퍼티 유무만 확인하는 아주 간단한 로직입니다.

[store.js]

...생략

```
.controller('CommentController', function($scope, CommentService, $routeParams) {  
    $scope.comments = [];  
    $scope.comment = {}; ---❶  
  
    CommentService.getComments($routeParams.id).success(function(response, status, config, headers){  
        $scope.comments = response;  
    });  
});
```

```

});

$scope.addComment = function() {
    var comment = $scope.comment;
    comment.errors = { ---❷
        writer: comment.writer ? false: true,
        desc: comment.desc ? false: true
    }

    if(!comment.errors.writer && !comment.errors.desc) { ---❸
        delete comment.errors;
        CommentService.addComment($routeParams.id, comment).success(function(response, status,
config, headers){
            $scope.comments.push(response);
            $scope.comment= {}; ---❹
        }).error(function(response, status, config, headers){
            alert("Error");
        });
    }
}
}

```

❶ 코멘트 입력 폼에 바인딩된 모델입니다

❷ 저장버튼을 클릭 시 error 프로퍼티를 만들고 작성자와 내용이 있는지 없는지를 설정합니다

❸ error 프로퍼티가 가지고 있는 모든 프로퍼티가 에러를 가지고 있지 않으면 서버에 저장합니다. 저장 전에 errors 프로퍼티는 필요 없으므로 삭제합니다.

❹ 저장 후 코멘트입력 창의 데이터를 비우기 위해서 코멘트 모델을 초기화 시킵니다.

입력 검증 시 에러를 표기 하기 위해서 코멘트 입력폼을 아래와 같이 수정합니다

[views/detail.html]

...생략

```

<div class="panel-body">
    <div class="pull-right">
        <p>
            <input type="text" ng-model="comment.writer" placeholder="작성자명을 입력하세요">
            <div ng-if="comment.errors.writer" style="color:red"> ---❶
                *작성자명이 없네요

```

```

    </div>

    </p>

    <textarea rows="4" cols="40" ng-model="comment.desc" placeholder="내용을
    입력하세요"></textarea>

    <div ng-if="comment.errors.desc" style="color:red"> ---❷
      * 내용이 없네요
    </div>

    <div class="text-right">
      <button class="btn btn-primary" ng-click="addComment()">코멘트 <span class="glyphicon
      glyphicon-pencil"></span></button>
    </div>
  </div>
</div>
</div>

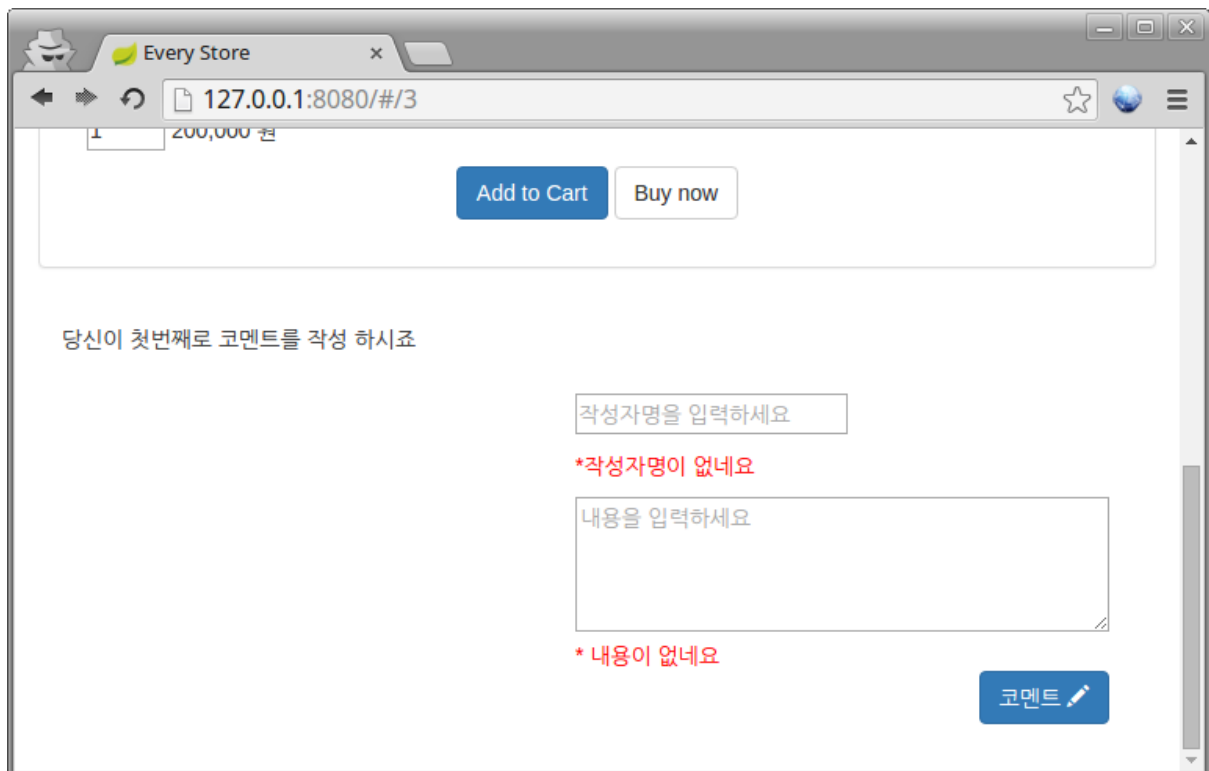
```

❶ 코멘트 모델의 errors의 속성 중 writer(작성자)가 true이면 메시지를 출력합니다

❷ 코멘트 모델의 errors의 속성 중 desc(내용)이 true이면 메시지를 출력합니다

입력창에 아무것도 입력하지 않고 코멘트 버튼을 클릭합니다 (그림 24). 코멘트 모델의 errors 프로퍼티의 조건에 맞게 에러메시지를 출력합니다.

그림 24 코멘트 입력 에러검증



3 클라이언트 모듈화하기

이번 장에서는 클라이언트의 소스를 모듈화하고 번들화 하는 방법을 소개합니다.

node.js의 모듈구조를 이용합니다. node.js는 자바스크립트의 모듈화를 위해서 CommonJS⁵의 스펙을 따릅니다.

3.1 node.js 설치

node.js는 <https://nodejs.org> 에서 다운로드 합니다.

시스템 PATH변수에 설정하는 것은 윈도우나 리눅스, 맥 모두 동일합니다.

설치 후 명령어로 확인이 가능합니다.

```
node -v  
v4.2.1
```

3.2 Browserify 란

browserify는 브라우저에서도 nodejs에서 처럼 모듈을 사용 할 수 있도록 환경을 제공해주는 것입니다. browserify를 통해서 nodejs의 모듈도 사용할 수 있습니다.

프로젝트를 만들어요

node 모듈을 패키지로 설치하고 관리하기 위한 프로젝트를 만들어야 합니다.

디렉토리를 만들고 해당 디렉토리로 이동 후 아래의 초기화 명령어를 실행합니다.

```
npm init
```

실행 후 package.json 파일이 생성됩니다.

browserify 설치

browserify는 글로벌 영역에 설치합니다. 이유는 어디서나 바로 명령어를 실행할 수 있도록 하기 위함입니다.

```
npm install -g browserify
```

⁵ <http://www.commonjs.org/>

browserify 을 이용하는 간단한 예를 보도록 하겠습니다.

[b.js]

```
module.exports= function( a, b) {  
  return a + b;  
}
```

[a.js]

```
var b = require('./b');  
console.log( b(2 ,4 ));
```

```
node a.js  
6
```

모듈 b 를 사용하여 결과를 콘솔에 출력합니다.

이제 browserify 를 사용하여 번들로 작성하겠습니다.

```
browserify a.js -o c.js ---❶  
node c.js ---❷  
6
```

❶ 옵션 -o 를 사용하여 의존성이 있는 모듈과 함께 번들화하여 c.js로 생성합니다

❷ 이제 c.js를 실행하면 결과는 같습니다.

browserify를 사용해서 작성한 모듈을 번들화하여 웹 브라우저에서 바로 사용할 수 있습니다.

3.3 Gulp 로 거들기

browserify를 사용해서 매번 소스를 수정 후에 번들작업을 반복 하는 것을 gulp로 자동화합니다.

```
npm install -g gulp
```

기본적인 gulp 사용법

[gulpfile.js]

```
var gulp = require('gulp');

gulp.task('default', function(){
  console.log("gulp task here!");
});
```

콘솔에서 그냥 gulp라고 입력하면 기본적으로 default라는 작업(task)을 실행합니다.

```
gulp
gulp task here!
```

3.4 번들 자동화하기

이제까지 작성한 에브리스토어의 store.js를 모듈구조로 바꾸고 결과물을 번들로 작성해 봅니다. angularjs와 관련 모듈을 설치합니다.

```
npm install --save-dev browserify
npm install --save-dev gulp
npm install --save-dev vinyl-source-stream
npm install --save angular
npm install --save angular-route
npm install --save angular-sanitize
npm install --save angular-cookies
```

체인닝패턴으로 구현된 store.js 의 소스를 이제 모듈화하기 위해서 성격별로 디렉토리를 만들고 소스를 재 작성합니다.

```
src
├─ controller
```

```
|   ├── CartController.js
|   ├── CommentController.js
|   ├── DetailController.js
|   └── GoodController.js
|── directive
|   └── CartCount.js
|── filter
|   └── newline.js
|── service
|   ├── CartService.js
|   ├── CommentService.js
|   └── GoodService.js
└── store.js
```

기존의 체인닝패턴을 유지하면서 컨트롤러, 서비스, 필터들을 모듈별로 분리하였습니다.
모듈화된 소스 중 일부만 보여드리면 아래와 같습니다.

[src/store.js]

```
var angular = require('angular');
var GoodService = require('./service/GoodService');
var CartService = require('./service/CartService');
var CommentService = require('./service/CommentService');
var GoodsController = require('./controller/GoodController');
var DetailController = require('./controller/DetailController');
var CartController = require('./controller/CartController');
var CommentController = require('./controller/CommentController');
var newLine = require('./filter/newline');
var CartCount = require('./directive/CartCount');

angular.module('store', [ require('angular-route'), require('angular-sanitize'), require('angular-cookies')])
    .config(function($routeProvider) {

        $routeProvider
            .when('/cart', {
                templateUrl: 'views/cart.html',
                controller: 'CartController'
            })
    })
```

```
.when('/:id', {
  templateUrl: 'views/detail.html',
  controller: 'DetailController'
})
.otherwise({
  templateUrl: 'views/index.html',
  controller: 'GoodsController'
});

})

.directive('cartCount', CartCount)
.service('GoodService', GoodService)
.service('CartService', CartService)
.service('CommentService', CommentService)
.controller('GoodsController', GoodsController)
.controller('DetailController', DetailController)
.controller('CartController', CartController)
.controller('CommentController', CommentController)
.filter("newLine", newLine);
```

[src/service/GoodService.js]

```
module.exports = function($http) {
  this.getData = function() {
    return $http({
      method: 'GET',
      url: '/api/goods'
    });
  };

  this.getDetail = function(id) {
    return $http({
      method: 'GET',
      url: '/api/goods/' + id
    });
  }
};
```

[src/controller/GoodController.js]

```
module.exports =function($scope, GoodService) {

    $scope.goods = [];

    GoodService.getData().success(function(response, status, config, headers) {
        $scope.goods = response;
    }).error(function(response, status, config, headers){
        alert("Error");
    });
}
```

이제 gulp로 자동화작업을 하기 위해서 아래와 같이 수정합니다.

[gulpfile.js]

```
var gulp = require('gulp'),
    browserify = require('browserify'),
    source = require('vinyl-source-stream');

gulp.task('build', function() {
    var b = browserify('./src/store.js', {debug:true}); ---❶
    return b.bundle() ---❷
        .pipe(source('all.js')) ---❸
        .pipe(gulp.dest('./target')); ---❹
});

gulp.task('watch', function() { ---❺
    gulp.watch('./src/**/*.js', ['build']); ---❻
});
```

❶ 메인소스파일을 기술합니다

❷ 번들화를 진행합니다. 관련된 모듈을 모두 찾아서 번들화합니다

❸ 번들화된 결과 파일을 all.js로 지정합니다

❹ 결과 파일은 현재 target이라는 경로에 위치합니다

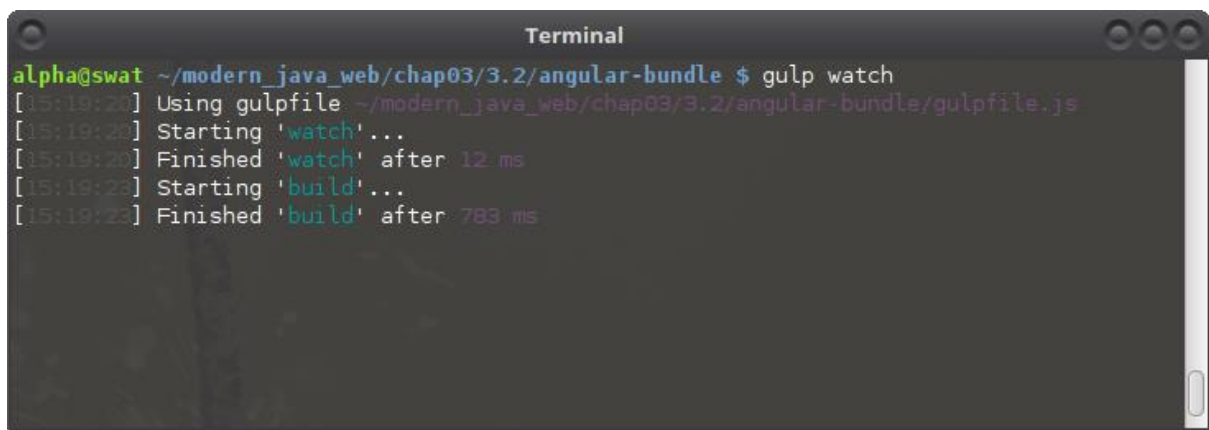
❺ watch 작업을 선언합니다.

⑥ 해당 src 디렉토리 이하에 있는 모든 자바스크립트 파일을 감시합니다. 만약 변경이 발생하면 상위에 선언된 build 작업을 실행합니다.

gulp를 실행하기 위해서 아래와 같이 입력합니다. 이제 소스 중 일부를 수정하고저장하면 build가 되고 target 디렉토리 이하에 all.js 라는 파일로 결과물이 생성됩니다.

```
gulp watch
```

그림 25 gulp task

A terminal window titled "Terminal" showing the execution of the 'gulp watch' command. The prompt is 'alpha@swat ~/modern_java_web/chap03/3.2/angular-bundle \$'. The output shows: '[15:19:20] Using gulpfile ~/modern_java_web/chap03/3.2/angular-bundle/gulpfile.js', '[15:19:20] Starting \'watch\'...', '[15:19:20] Finished \'watch\' after 12 ms', '[15:19:23] Starting \'build\'...', and '[15:19:23] Finished \'build\' after 783 ms'.

마지막으로 gulpfile.js에서 빌드 후 결과물이 에브리스토어의 public/js로 지정하도록 수정합니다. 이제 all.js 가 angularjs 와 store.js의 모든 소스를 가지고 있으므로 angularjs를 사용하는 부분을 아래와 같이 수정합니다.

[index.tpl]

수정 전

...생략

```
script(src: 'webjars/angularjs/1.4.7/angular.js') {}  
script(src: 'webjars/angularjs/1.4.7/angular-route.js') {}  
script(src: 'webjars/angularjs/1.4.7/angular-sanitize.js') {}  
script(src: 'webjars/angularjs/1.4.7/angular-cookies.js') {}  
script(src: 'js/store.js') {}
```

수정 후

```
script(src: 'js/all.js') {}
```

그리고 angularjs의 webjar로 필요없으므로 app.groovy 에서 아래 라인을 삭제합니다

```
@Grab("org.webjars:angularjs:1.4.7") //삭제 해야할 라인
```

4 Every Store 다듬기

4.1 Gradle 빌드환경 구축하기

이제까지 스프링부트 CLI를 통해서 빠른 프로토타입을 작성하였습니다. CLI는 그루비문법을 이해하고 자바의 import구문없이 @RestController와 같은 몇몇 어노테이션과 메인클래스를 작성합니다. 하지만 CLI가 지원하지 않는 라이브러리나 특정 환경을 지원하지 못하는 한계가 있기 때문에 빌드환경을 구축해야합니다.

Gradle 설치

윈도우의 경우

gradle을 <http://gradle.org> 에서 다운로드합니다. 다운로드 후 압축을 해제하고 설치경로를 GRADLE_HOME 로 설정하고 시스템 PATH변수에 추가합니다.

리눅스나 맥의 경우

기본적으로 윈도우에서 설치한 예와 같습니다. 그밖에 GVM을 사용하여 설치할 수 있습니다.

```
$ gvm install gradle
```

Gradle 의 기본적인 사용법

프로젝트 작성

```
gradle init -type groovy-library
```

작성 후 그루비 기반의 빌드환경이 작성됩니다. 디렉토리 구조는 아래 같습니다.

```
.
├─ build.gradle
├─ gradle
│  └─ wrapper
│     ├── gradle-wrapper.jar
│     └─ gradle-wrapper.properties
├─ gradlew
├─ gradlew.bat
├─ settings.gradle
└─ src
   ├── main
   │  └─ groovy
   │     └─ Library.groovy
   └─ test
      └─ groovy
         └─ LibraryTest.groovy
```

빌드 하기 위해서 gradle 명령어를 사용합니다.

빌드

```
|
| gradle build
|
```

빌드 후 결과물은 target 에 위치합니다. target 이라는 디렉토리는 없을 경우 빌드 시 생성됩니다.

4.2 MongoDB 사용하기

이제부터 스프링부트 CLI기반에서 작동하는 프로젝트를 Gradle 기반으로 재작성 하겠습니다.
또한 임시로 작성한 상품목록 또한 데이터베이스를 이용하도록 수정하고 몇가지 수정사항을 보완 할 것입니다. 처음부터 Gradle 을 이용해서 프로젝트 구조를 만들 수 있지만 여기서는 템플릿을 사용 할 것 입니다.

준비물

우선 몽고디비가 필요합니다. 이 책에서는 몽고디비 버전 3.0을 사용하였습니다.

윈도우의 경우

몽고디비 서버를 시작하기 위해 아래와 같이 입력합니다.

사전에 C:\data 디렉토리가 있어야 합니다

```
| mongod.exe --dbpath C:\data
```

리눅스와 맥OS의 경우

공식 웹사이트에서 설치정보를 확인합니다.

템플릿을 이용한 프로젝트 초기화

그루비템플릿과 몽고디비를 사용하는 웹어플리케이션을 구축을 쉽게 하기 위해서 Spring Initializr⁶에서 제공되는 템플릿을 사용합니다. 템플릿을 얻기 위해서 2가지 방법이 있습니다. 첫째로 Curl를 이용하는 방법과 두번째로 직접 사이트에서 다운로드하는 것입니다.

아래는 curl을 이용하는 방법입니다.

```
| $ curl https://start.spring.io/starter.tgz \  
| > -d name=everystore \  
| > -d dependencies=web,groovy-templates,data-mongodb \  
| > -d language=groovy \  
| > -d type=gradle-project \  
| > -d packageName=everystore \  
| > -d artifactId=everystore \  
|
```

⁶ <https://start.spring.io>

```
> -d baseDir=everystore | tar -xzvf -
```

curl 를 사용할 수 없는 경우에는 Spring Initializr 을 사용할 수 있습니다.

해당 사이트에서 원하는 빌드환경과 언어, 그리고 서비스타입을 선택하여 프로젝트의 빌드환경을 다운로드 받을 수 있습니다.

도메인 모델 작성

상품과 코멘트를 표현하기 위한 모델을 작성합니다.

[Good.groovy]

```
package everystore.model

import groovy.transform.ToString
import org.springframework.data.annotation.Id
import org.springframework.data.mongodb.core.mapping.Document
@Document ---❶
@ToString(includeNames = true) ---❷
class Good {

    @Id ---❸
    String id
    String name
    String desc
    double price
}
```

❶ 해당 모델이 도큐먼트로 취급됩니다

❷ 그루비에서 지원하는 ToString 어노테이션입니다. 일일이 ToString을 작성하지 않아도 됩니다

❸ 도큐먼트의 ID 즉 식별자임을 나타냅니다.

[Comment.groovy]

```
package everystore.model

import groovy.transform.ToString
```

```
import org.springframework.data.annotation.Id
import org.springframework.data.mongodb.core.mapping.Document
```

```
@Document
@ToString(includeNames = true)
class Comment {
    @Id
    String id
    String writer
    String desc
    String goodId
}
```

리포지토리 인터페이스를 작성

상품과 코멘트에 대한 CRUD 처리를 하기 위해 스프링부트의 data-mongodb에서 제공하는 Repository 인터페이스를 상속받습니다.

[GoodRepository.groovy]

```
package everystore.repository

import everystore.model.Good
import org.springframework.data.repository.CrudRepository

interface GoodRepository extends CrudRepository<Good, String> { ---❶
}
```

❶ Repository 중의 가장 기본이 되는 CrudRepository를 사용합니다. 만약 페이징 처리가 필요하다면 PagingAndSortingRepository 거나 CrudRepository와 PagingAndSortRepository 둘다 확장한 MongoRepository를 사용 할 수 있습니다. 참고로 CrudRepository 인터페이스를 잠깐 설명하겠습니다. T는 도메인 클래스이고 ID는 도메인의 식별자입니다.

```
public interface CrudRepository<T, ID extends Serializable> extends Repository<T, ID> {
... 생략
```

[CommentRepository.groovy]

```
package everystore.repository

import everystore.model.Comment
import org.springframework.data.repository.CrudRepository

interface CommentRepository extends CrudRepository<Comment, String> {
    List<Comment> findByGoodId(String id) ---❶
}
```

❶ 상품아이디로 해당 코멘트를 찾기 위해 선언합니다. 스프링 데이터가 제공하는 쿼리메커니즘을 이용해서 findBy[필드명]으로 선언 할 수 있습니다.

테스트를 통해서 검증합니다

사용자가 정의한 리포지토리 인터페이스를 가지고 CRUD에 대한 테스트를 하려고 합니다. 먼저 데이터베이스만 구동 가능한 환경이 필요합니다. 종래에는 XML을 이용해서 Spring 환경을 작성했으나 스프링 부트에서는 Java로 환경을 작성하는 것을 권장 합니다.

[MongoDbTestApplication.groovy]

```
package everystore

import com.mongodb.Mongo
import com.mongodb.MongoClient
import org.junit.runner.RunWith
import org.springframework.context.annotation.ComponentScan
import org.springframework.context.annotation.Configuration
import org.springframework.data.mongodb.config.AbstractMongoConfiguration
import org.springframework.data.mongodb.repository.config.EnableMongoRepositories
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner

@RunWith(SpringJUnit4ClassRunner) ---❶
@Configuration ---❷
@EnableMongoRepositories ---❸
@ComponentScan(basePackages = ["everystore.model", "everystore.repository"]) ---❹
class MongoDbTestApplication extends AbstractMongoConfiguration { ---❺

    @Override
    protected String getDatabaseName() {
        "everystore-test" ---❻
    }

    @Override
    Mongo mongo() throws Exception {
        new MongoClient("127.0.0.1", 27017) ---❼
    }

}
```

- ❶ 스프링에서 쉽게 JUnit를 사용하게끔 합니다. build.gradle에 dependencies 에 testCompile('org.springframework.boot:spring-boot-starter-test') 라고 기술되었기 때문에 가능합니다. 없다면 추가하도록 합니다.
- ❷ 스프링컨테이너에게 환경을 주입하기 위해 작성합니다. 즉 JavaConfig임을 알려줍니다.
- ❸ 몽고 리포지토리 인터페이스를 인식하기 위해 사용합니다
- ❹ 몽고 리포지토리 인터페이스를 사용하기 위한 도메인과 리포지토리를 찾기위해서 기술합니다.

- ⑤ 몽고디비 환경을 사용자가 확장하기 위한 클래스 입니다
- ⑥ 사용할 데이터 베이스 이름을 기술합니다
- ⑦ 몽고디비 접속 환경을 기술하여 MongoClient를 돌려줍니다.

[GoodRepositoryTest.groovy]

```
package everystore

import everystore.model.Comment
import everystore.model.Good
import everystore.repository.CommentRepository
import everystore.repository.GoodRepository
import org.junit.Assert
import org.junit.Test
import org.junit.runner.RunWith
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.test.context.ContextConfiguration
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner

@RunWith(SpringJUnit4ClassRunner)
@ContextConfiguration(classes = MongoddbTestApplication)
class GoodRepositoryTest {

    @Autowired
    GoodRepository goodRepository

    @Autowired
    CommentRepository commentRepository

    @Test
    void nullTest() {
        Assert.assertNotNull(goodRepository)
    }

    @Test
    void add() {
```

```
goodRepository.deleteAll()
commentRepository.deleteAll()

def coffee = new Good( name: "커피", price: 3000, img: "coffee.jpg", desc: "맛 있는 커피", qty: 10)
def ps4 = new Good( name: "PS4", price: 400000, img: "ps4.jpg", desc: "이번이 마지막 찬스 \n 최강의
콘솔게임머신", qty: 4)
def bicycle = new Good( name: "커피", price: 3000, img: "coffee.jpg", desc: "그림에 나오는 자전거",
qty: 3)

goodRepository.save(coffee)
goodRepository.save(ps4)
goodRepository.save(bicycle)

def comment1 = new Comment(writer: "김철수", desc: "별로예요", goodId: coffee.id)
def comment2 = new Comment(writer: "홍길동", desc: "괜찮은 것 같아요", goodId: coffee.id)
def comment3 = new Comment(writer: "김철수", desc: "갖고 싶어요!", goodId: ps4.id)

commentRepository.save(comment1)
commentRepository.save(comment2)
commentRepository.save(comment3)

}

@Test
void find() {
    def good = goodRepository.findOne("563c10fac9447a6373fa6c0a") ---❶
    Assert.assertNotNull(good)
    println good

    def comments = commentRepository.findByGoodId(good.id)
    Assert.assertEquals(2, comments.size())
    println comments
}
}
```

❶ 아이디에 해당하는 상품을 찾습니다. 여기에 기술된 아이디는 실제로 생성시마다 다르니 직접

실행 후 생성된 아이디를 기술하도록 합니다.

JUnit 테스트 코드를 실행합니다. 패키지 이름이 길어서 *(모든) 패키지의 GoodRepositoryTest 클래스의 nullTest라는 메서드를 테스트합니다. 메서드별로 테스트할 수 있습니다.

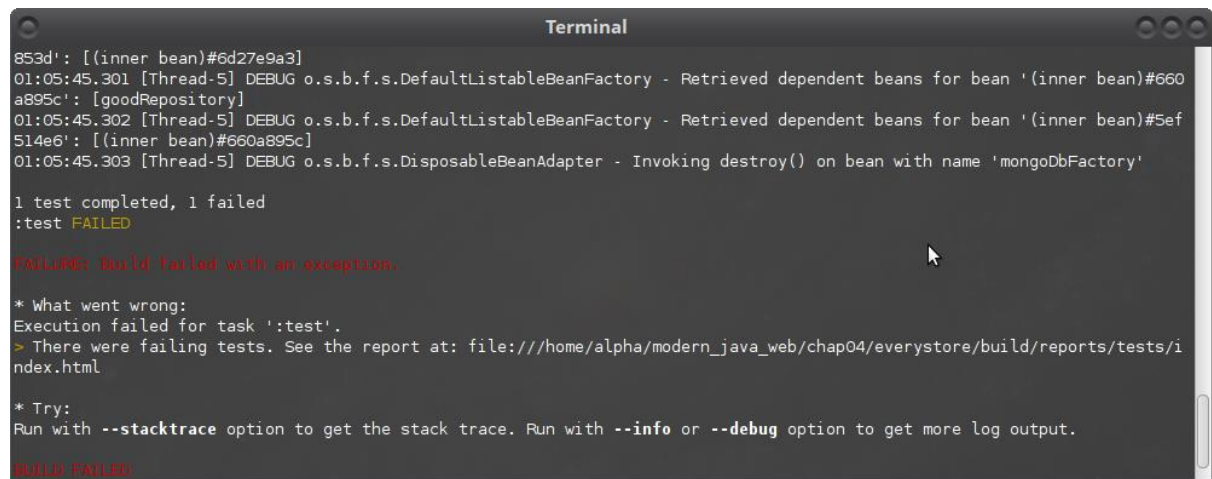
```
gradle test --tests *GoodRepositoryTest.nullTest
```

또는

```
gradle test --tests *Good*.find
```

실패 시 에러에 대한 상세한 결과가 html로 생성됩니다.

그림 26 에러 발생



```
Terminal
853d': [(inner bean)#6d27e9a3]
01:05:45.301 [Thread-5] DEBUG o.s.b.f.s.DefaultListableBeanFactory - Retrieved dependent beans for bean '(inner bean)#660a895c': [goodRepository]
01:05:45.302 [Thread-5] DEBUG o.s.b.f.s.DefaultListableBeanFactory - Retrieved dependent beans for bean '(inner bean)#5ef514e6': [(inner bean)#660a895c]
01:05:45.303 [Thread-5] DEBUG o.s.b.f.s.DisposableBeanAdapter - Invoking destroy() on bean with name 'mongoDbFactory'

1 test completed, 1 failed
:test FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':test'.
> There were failing tests. See the report at: file:///home/alpha/modern_java_web/chap04/everystore/build/reports/tests/index.html

* Try:
Run with --stacktrace option to get the stack trace. Run with --info or --debug option to get more log output.

BUILD FAILED
```

그림 27 에러에 대한 리포트

Test results - Class everystore.GoodRepositoryTest - Mozilla Firefox

Test results - Class everyst...

file:///home/alpha/modern_java_web/chap04/everystc

Class everystore.GoodRepositoryTest

all > everystore > GoodRepositoryTest

1	1	0	0.068s
tests	failures	ignored	duration

0% successful

Failed tests Tests Standard output

list

```
java.lang.AssertionError: expected:<1> but was:<[[everystore.model.Comment(id:563b7be4b280e7be7c96690e, writer:홍길...]]>
    at org.junit.Assert.fail(Assert.java:88)
    at org.junit.Assert.failNotEquals(Assert.java:834)
    at org.junit.Assert.assertEquals(Assert.java:118)
    at org.junit.Assert.assertEquals(Assert.java:144)
    at org.junit.Assert$assertEquals.call(Unknown Source)
    at org.codehaus.groovy.runtime.callsite.CallSiteArray.defaultCall(CallSiteArray.java:48)
    at org.codehaus.groovy.runtime.callsite.AbstractCallSite.call(AbstractCallSite.java:113)
    at org.codehaus.groovy.runtime.callsite.AbstractCallSite.call(AbstractCallSite.java:133)
    at everystore.GoodRepositoryTest.list(GoodRepositoryTest.groovy:57)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
```

컨트롤러를 작성합니다

컨트롤러에서는 임시로 작성한 데이터 부분을 리포지토리 인터페이스로 교체합니다.

[GoodController.groovy]

```
package everystore.controller

import everystore.model.Comment
import everystore.repository.CommentRepository
import everystore.repository.GoodRepository
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.web.bind.annotation.*
import org.springframework.web.servlet.ModelAndView

@RestController
class GoodController {

    @Autowired
    GoodRepository goodRepository ---❶

    @Autowired
    CommentRepository commentRepository ---❷

    @RequestMapping("/")
    def index() {
        new ModelAndView("index")
    }

    @RequestMapping("/api/goods")
    def goods() {
        goodRepository.findAll()
    }

    @RequestMapping("/api/goods/{id}")
    def good(@PathVariable String id) {
        goodRepository.findOne(id)
    }
}
```

```

@RequestMapping("/api/goods/{id}/comments")
def comments(@PathVariable String id) {
    commentRepository.findById(id)
}

@RequestMapping(value="/api/goods/{id}/comments", method= RequestMethod.POST)
def addComment(@PathVariable String id, @RequestBody Comment comment) { ---❸
    comment.goodId = id
    commentRepository.save(comment)
}
}

```

- ❶ 기존의 상품데이터를 GoodRepository 로 교체합니다.
- ❷ 기존의 코멘트 데이터를 CommentRepository로 교체합니다.
- ❸ 클라이언트에서 보내는 데이터를 이제 모델로 바인딩합니다

사용자 데이터 초기화하기

서버가 시작되면서 몽고디비에 데이터를 추가하려고 합니다. 따라서 메인클래스에서 CommandLineRunner 인터페이스를 구현합니다.

```

package everystore

import everystore.model.Comment
import everystore.model.Good
import everystore.repository.CommentRepository
import everystore.repository.GoodRepository
import groovy.util.logging.Slf4j
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.CommandLineRunner
import org.springframework.boot.SpringApplication
import org.springframework.boot.autoconfigure.SpringBootApplication

@SpringBootApplication ---❶
@Slf4j ---❷
class EverystoreApplication implements CommandLineRunner { ---❸

```

```

@Autowired GoodRepository goodRepository

@Autowired CommentRepository commentRepository

static void main(String[] args) {
    SpringApplication.run EverystoreApplication, args
}

@Override
void run(String... args) throws Exception { ---❶
    log.info("Creating user data initializing...")
    goodRepository.deleteAll()
    commentRepository.deleteAll()

    def coffee = new Good(name: "커피", price: 3000, img: "coffee.jpg", desc: "맛 있는 커피")
    def ps4 = new Good(name: "PS4", price: 400000, img: "ps4.jpg", desc: "이번이 마지막 찬스 \n 최강의
콘솔게임머신")
    def bicycle = new Good(name: "커피", price: 3000, img: "coffee.jpg", desc: "그림에 나오는 자전거")

    goodRepository.save(coffee)
    goodRepository.save(ps4)
    goodRepository.save(bicycle)

    def comment1 = new Comment(writer: "김철수", desc: "별로예요", goodId: coffee.id)
    def comment2 = new Comment(writer: "홍길동", desc: "괜찮은 것 같아요", goodId: coffee.id)
    def comment3 = new Comment(writer: "김철수", desc: "갖고 싶어요!", goodId: ps4.id)

    commentRepository.save(comment1)
    commentRepository.save(comment2)
    commentRepository.save(comment3)
}
}

```

- ❶ 프로그램의 메인클래스로 기술합니다. 즉 현재 클래스가 스프링 부트를 기동할 수 있는 메인 클래스가 됩니다.
- ❷ 그루비가Slf4j의 로그 인터페이스를 사용하도록 제공하는 어노테이션입니다
- ❸ CommandLineRunner를 구현합니다.

④ 구현할 메서드에서 초기화 작업을 처리할 수 있도록 기술합니다.

이제 스프링부트 서버를 초기값으로 test 라는 데이터베이스를 사용하게 됩니다. 시작하기 위해 앞서 몽고디비 환경을 스프링부트의 환경파일에 작성할 필요가 있습니다.

먼저 스프링 부트의 공식문서⁷ 를 참조하여 어떤 프로퍼티들을 사용할 수 있는지 알아 봅시다.

MONGODB (MongoProperties)

```
spring.data.mongodb.host= # the db host
spring.data.mongodb.port=27017 # the connection port (defaults to 27107)
spring.data.mongodb.uri=mongodb://localhost/test # connection URL
spring.data.mongodb.database=
spring.data.mongodb.authentication-database=
spring.data.mongodb.grid-fs-database=
spring.data.mongodb.username=
spring.data.mongodb.password=
spring.data.mongodb.repositories.enabled=true # if spring data repository support is enabled
```

스프링부트가 사용하는 환경파일은 application.properties 또는 application.yml 입니다. 템플릿을 사용하여 프로젝트를 구성하면 application.properties가 기본적으로 사용됩니다. 아래 파일에 사용할 데이터베이스만 기술할 것 입니다. 해당 파일은 resources 디렉토리 이하에 위치합니다.

[application.properties]

```
spring.data.mongodb.database=everystore
```

이제 서버를 시작하기 위해 아래와 같이 입력합니다.

```
gradle bootRun
```

⁷ <http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle>

5 Every Store 배포

5.1 Build

에브리스토어를 빌드하기 위해서 아래와 같이 입력합니다. -x 옵션은 빌드 시 테스트를 실행하지 않겠다는 의미입니다. 빌드 후 결과물은 build 디렉토리 아래에 위치 합니다.

```
gradle build -x test
```

빌드 후 결과물은 zip 또는 jar로 생성됩니다. 여기서 jar를 사용하겠습니다. jar는 build/libs에 위치합니다. 생성된 jar 는 통칭 FatJar로 불립니다. 하나의 파일에 모든 라이브러리를 포함하고 있기 때문입니다. 이 jar에는 정적 리소스까지 포함되어 있습니다. jar로 서버를 실행하기 위해서 아래와 같습니다.

```
java -jar everystore-0.0.1-SNAPSHOT.jar
```

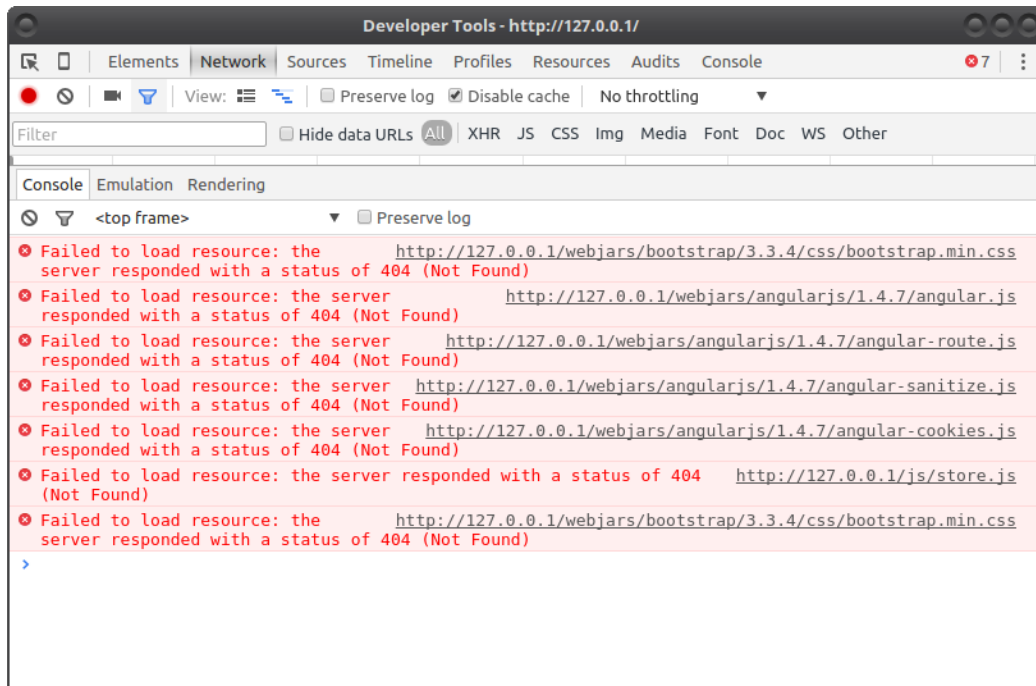
5.2 NginX 활용하기

빌드시 정적리소스 제거

일반적으로 정적리소스를 담당하는 서버와 API 서버로 나뉘서 서비스하는 경우가 많습니다. nginx와 함께 사용 시 정적리소스는 jar 내부에 있기 때문에 nginx를 통해서 서비스될 수가 없습니다. 만약 웹브라우저로 접근한다면 그림 28 과 같이 404 에러를 보게 됩니다.

이를 해결 하기 위해서 빌드시 정적리소스를 제외해야 합니다. gradle의 빌드 스크립트인 build.gradle를 다음과 같이 수정합니다.

그림 28 정적리소스 404에러



[build.gradle]

...생략

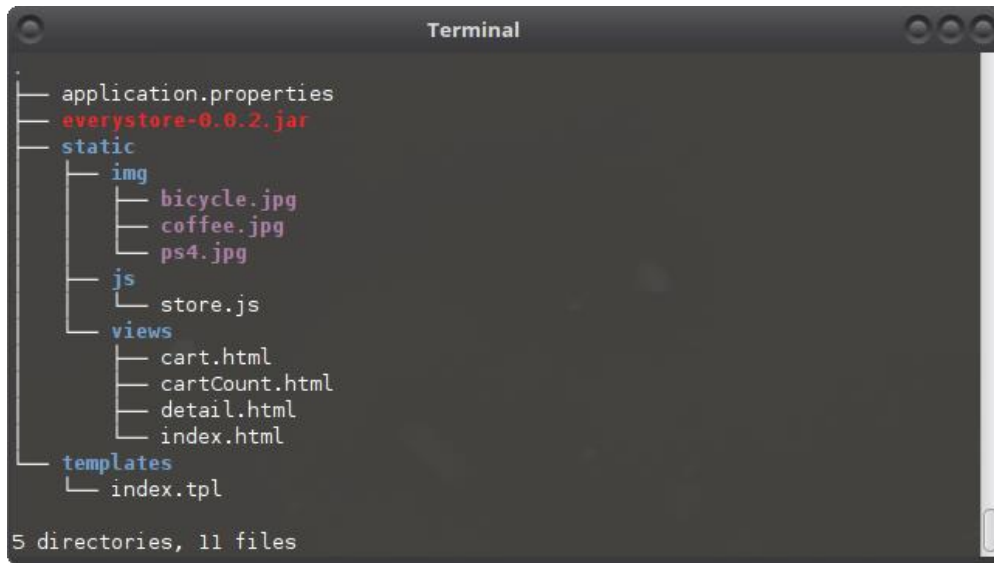
```
jar {  
    excludes= [ ❶  
        'static',  
        'templates'  
    ]  
  
    baseName = 'everystore'  
    version = '0.0.1-SNAPSHOT'  
}
```

...생략

❶ 빌드 시 제외할 파일이나 디렉토리를 기술하면 됩니다.

이제 다시 빌드를 실행합니다. 필자의 경우에는 application.properties 도 빌드에서 제외하고 아래와 같은 디렉토리 구조로 변경하였습니다.

그림 29



그리고 nginx에서 정적 리소스가 있는 경로를 설정하기 위해서 nginx의 config 파일을 수정합니다. 정적 리소스가 있는 위치를 root에 기술하였습니다.

[everystore.conf]

```
upstream backend{
    server 127.0.0.1:8080;
}

server {
    listen      80;
    server_name localhost;
    root /home/everystore/static;

    location / {
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $http_host;
        proxy_redirect off;
        proxy_pass http://backend;
    }
}
```

이제 nginx를 재시작하고 웹브라우저로 접속합니다. 정적리소스가 nginx로 부터 서비스 될 것 입니다. 하지만 webjar로 제공되는 것은 아직도 jar에 위치하고 있습니다.

angularjs의 webjar를 제거하려면 3장에서 소개한 번들로 작성하면 됩니다.