Hanbit eBook
Realtime 36

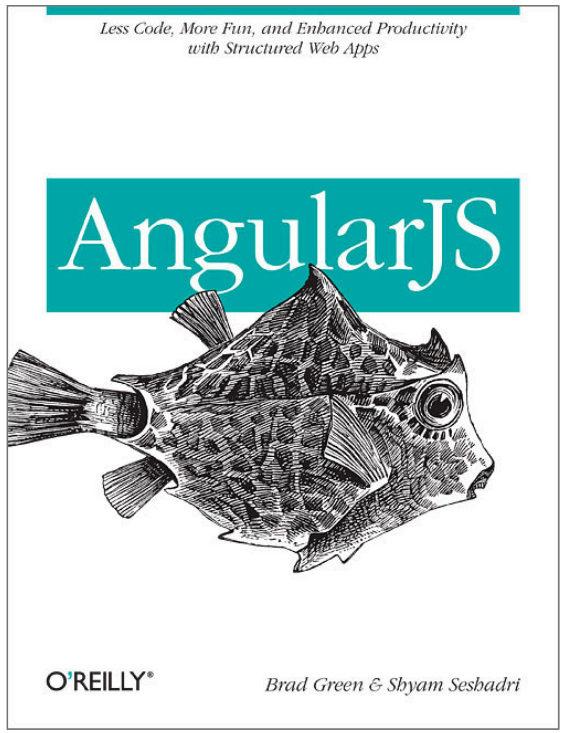
MVC 패턴을 구현하는 자바스크립트 프레임워크

AngularJS 기초편

AngularJS

브래드 그린, 샤이엄 세샤드리 지음 / 김지원 옮김

O'REILLY® 한빛미디어 Hanbit Media, Inc.



이 도서는 O'REILLY의
AngularJS의
번역서입니다.

MVC 패턴을 구현하는 자바스크립트 프레임워크

AngularJS 기초편

MVC 패턴을 구현하는 자바스크립트 프레임워크 AngularJS 기초편

초판발행 2013년 7월 31일

지은이 브래드 그린, 샤이엄 세샤드리 / **옮긴이** 김지원 / **펴낸이** 김태헌
펴낸곳 한빛미디어(주) / **주소** 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부
전화 02-325-5544 / **팩스** 02-336-7124
등록 1999년 6월 24일 제10-1779호
ISBN 978-89-6848-641-8 15000 / **정가** 9,900원

책임편집 배용석 / **기획** 김병희 / **편집** 이세진
디자인 표지 여동일, 내지 스튜디오 [밈], 조판 김현미
마케팅 박상용, 박주훈

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오.
한빛미디어 홈페이지 www.hanb.co.kr / **이메일** ask@hanb.co.kr

지금 하지 않으면 할 수 없는 일이 있습니다.
책으로 펴내고 싶은 아이디어나 원고를 메일(ebookwriter@hanb.co.kr)로 보내주세요.
한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

저자 소개

지은이_ 브래드 그린Brad Green

구글 AngularJS 프로젝트 팀에서 엔지니어 관리자를 맡고 있으며, 접근성과 지원 공학을 총괄 감독한다. 구글에 입사하기 전에는 인터넷 기업을 만들어 팔던 AvantGo 사에서 초창기 모바일 웹 개발자로 근무하다가, 출장요식업에 뛰어들어 고단한 몇 년을 보냈다. 대학을 졸업하고 NeXT Computer 사에서 스티브 잡스 밑에서 데모 소프트웨어를 만들고 잡스의 슬라이드 프레젠테이션을 디자인했던 것이 브래드의 첫 직장 경험이다. 브래드는 아내와 두 자녀를 데리고 캘리포니아 주 마운틴 뷰에 살고 있다.

지은이_ 샤이엄 세샤드리Shyam Seshadri

Fundoo Solutions 사의 사주이자 CEO다. AngularJS에 관해 컨설팅하고 워크숍을 개최한다. 인도 시장을 겨냥한 혁신적 제품 개발에 주력하며, AngularJS를 주제로 한 워크숍을 운영하고 컨설팅한다. Fundoo Solutions 사를 창립하기 전에는 하이데라바드에 있는 명문 Indian School of Business에서 MBA 과정을 마쳤다. 샤이엄은 대학 졸업 후 첫 직업으로 구글에서 다수의 프로젝트를 진행했다. 그 중에는 AngularJS가 처음으로 사용된 구글 피드백Google Feedback 프로젝트도 있다. 그리고 다양한 내부 도구도 제작했다. 현재는 인도 나비뭄바이에서 회사를 운영하고 있다.

역자 소개

옮긴이_ **김지원**

웹 기술뿐 아니라 온갖 분야에 발을 뻗고 싶어하는 바람기를 지녔지만 역부족이다. 배워야 할 것이 갈수록 늘어나 시간의 결핍을 느낀다. 워드프레스, 프라이드치킨, 꿀, 분재, 컴퓨터 음악을 좋아한다. 기술 문서, 매뉴얼, 유비쿼터스 관련 논문을 번역한 바 있고 해외 논문 DB 구축 관련 작업에도 참여했다. 번역서로는 『한 권으로 끝내는 정규표현식』(한빛미디어, 2010), 『웹 표준 가이드: HTML5+CSS3』(한빛미디어, 2010), 『프로젝트로 배우는 HTML5+자바스크립트』(한빛미디어, 2012), 『리팩토링』(한빛미디어, 2012), 『엘리멘틀 디자인 패턴』(한빛미디어, 2013), 『HTML5+CSS3 디자인 패턴』(한빛미디어, 2013), 『HTML5 웹소켓 프로그래밍』(한빛미디어, 2013) 등이 있다.

저자 서문

Angular 프레임워크의 기원은 2009년의 구글 피드백Google Feedback이라는 프로젝트로 거슬러 올라간다. 필자는 테스트 가능한 코드를 작성하면서 수개월간 개발 속도와 기능의 문제로 난관을 겪었다. 6개월여간 작성한 프론트엔드front-end 코드가 대략 17,000줄이었다. 그런데 당시 프로젝트 팀원이던 미스코 헤브리Misko Hevery는 자신이 취미로 작성한 오픈소스 라이브러리를 사용하면 2주도 안 걸려서 우리가 작성한 코드 전체를 새로 작성할 수 있을 거라고 큰소리쳤다.

필자는 프로젝트가 2주쯤 지연돼도 별문제는 없을 것이고 설령 미스코가 장담한 대로 기한 내에 다시 작성하지 못 하더라도 기한에 쫓겨 허둥대는 미스코의 표정을 보는 재미라도 있겠다는 생각에 그렇게 해보라고 했다. 예상대로 미스코는 기한을 넘겨 3주만에 완료했다. 그래도 6개월이 걸렸던 개발을 3주라는 단시간에 재현했다는 점에 우리 팀 전원은 경악했는데, 더 놀랍게도 미스코가 새로 작성한 애플리케이션의 코드 분량이 원래의 17,000줄의 1/10도 안 되는 1,500줄에 불과했다. 미스코가 이뤄낸 성과는 추진해볼 가치가 있어 보였다.

미스코가 간단한 선언 문서로 창안했던 각종 개념을 중심으로 해서, 미스코와 나는 웹 개발자의 경험을 간소화할 팀을 만들기로 했다. 이 책의 공동 저자인 샤이엄 세샤드리Shyam Seshadri는 구글 피드백 팀에서 Angular의 첫 출시 애플리케이션 개발을 지휘했다.

그때부터 우린 구글의 여러 팀과 수백 명의 오픈소스 기부자의 도움을 받아 Angular를 개발했다. 많은 개발자가 일상적인 작업에 Angular 프레임워크를 이용하며 엄청난 Angular 지원망에 기여한다.

우리는 여러분이 나눠주는 지식을 얻게 되리란 생각에 감개무량하다.

감사의 글

Angular 프레임워크를 탄생시킨 미스코 헤브리에게 각별히 고마움을 전한다. 미스코 덕택에 웹 애플리케이션 작성법을 기존과 전혀 다른 방식으로 생각하고 실천할 수 있었다. 이고르 미나Igor Minar는 Angular 프로젝트의 안정화와 체계화에 기여했고 활성화된 지금의 오픈소스 커뮤니티의 모체를 만들었다. 보이타 지나Vojta Jina는 Angular의 많은 부분을 작성했으며 덕분에 우리는 테스트를 유례없이 신속하게 할 수 있었다. 나오미 블랙Naomi Black, 존 린퀴스트John Lindquist, 매사이어스 마셔스 니멜라Mathias Matias Niemela는 숙련된 솜씨로 편집을 도와주었다. 앞서 나열한 모든 분들과 더불어, 다방면에서 도움을 주고 실시간 애플리케이션 제작 과정에서 피드백을 통해 우리에게 Angular의 가치 있는 사용법을 알려준 Angular 커뮤니티 분들께 감사의 인사를 남긴다.

브래드 그린, 샤이엄 세샤드리

역자 서문

왜 AngularJS인가? 개발자가 구글의 AngularJS 플랫폼을 선택할 수밖에 없는 이유는 다음과 같다.

- 양방향 데이터 바인딩이 가능하다 - AngularJS로 개발한 애플리케이션은 클라이언트에서 서버로뿐만 아니라 서버에서 클라이언트로도 실시간 변경 감지가 이뤄진다. 감시, 리스너, 캡처 기능을 통해 개발한 코드가 실행되고 모델을 조작한 후 발생하는 변경사항을 감시한다.
- 모델, 뷰, 컨트롤러, 서비스 등 여러 구성요소로 분리된다 - 지시어, 필터, 모듈 등의 추상 객체를 이용해 균형을 맞출 수 있다. 이로써 복잡도의 감소와 관심사의 분리라는 두 마리 토끼를 얻을 수 있다.
- 편리하고 친숙한 패턴이 많다 - MVC나 종속물 주입 같은 유명한 패턴 외에도 종속물 관리 같은 다수의 패턴이 들어 있어서 체계적인 구성으로 개발할 수 있다.
- 테스트용 코드를 쉽게 작성할 수 있다 - AngularJS 공식 온라인 강좌 페이지에도 Jasmine 문법을 사용한 단위 테스트와 클라이언트-서버 테스트를 코드로 작성하는 방법이 예시돼 있다.

모든 프레임워크가 그렇듯 비록 AngularJS 역시 완벽할 순 없지만, 사소한 단점에 비해 얻을 수 있는 것이 많다. AngularJS에 관한 전반적인 내용이 이 책의 본문에 자세히 설명돼 있으니 자세한 얘기는 본문을 숙지하기 바란다.

AngularJS를 개발에 사용하면 길고 복잡한 코딩의 분량을 획기적으로 줄일 수 있다. 아주 간단하고 코딩 길이가 짧은 프로젝트라면 물론 큰 이득을 얻지 못 할 수도 있지만, 데이터 수정 시에 빠른 반응성이 요구되는 UI를 구상하고 있다면

AngularJS는 개발에 있어서 반드시 고려할 만하다. 플랫폼에 이미 내장돼 있는 지시어 말고도 개발자가 직접 정의한 지시어를 템플릿에 사용함으로써 모델, 컨트롤러와 바로 연결이 가능한 점은 강력한 기능이다. 종속물 주입 또한 상위의 기능에 필요한 종속물(종속 객체, 종속 함수, 종속 모듈 등)을 마치 혈관으로 연결된 링거에 주사기로 항생제를 주입하듯이 손쉽게 끼워 넣음으로써 매우 직관적이며 메서드 체인 형태로 호출이 가능하다.

이 모든 잡다한 서론을 뒤로 하고 지금 당장 본문의 첫 페이지로 가서 어떤 놀라운 장점이 있고 어떤 부분이 자신의 개발에 필요한지 살펴보자.

번역을 마무리하며,

김지원

도움을 주신 분들

베타테스터_ 원종필

탁월함의 갈망에 빠져있는 게임 클라이언트 프로그래머다. Unity를 사용해서 일을 하고 있으며, 업무의 질을 높이기 위해 고민 중이다. 최근에는 웹 개발에 관심을 두고 있다.

베타테스터_ 이기동

중3 때 ASP를 시작으로 VB, C#, FLEX, MFC를 거쳐, 현재는 JAVA와 JQM을 이용하여 전자지갑 모바일 웹을 개발하고 있는 책 읽기 좋아하는 프로그래머다. 튜닝이나 해킹에 관심이 많으며, 요즘은 먼가 만들 생각에 아두이노에 빠져있다. 제일 사랑하는 건 아내고, 좋아하는 건 아내가 해준 밥이며, 설거지를 무척이나 싫어한다.

베타테스터_ 이수한

자신에게 떨어지는 일거리를 코딩으로 승화하고 싶어하는 평범한 월급쟁이 직장인이다. 게임공학이 전공이만 운 좋게 게임 외에 다양한 프로젝트를 수행하였다. 평소 서버, 플랫폼 쪽 구현 기술에 관심이 많으며 언젠가는 나만의 플랫폼을 설계 및 구현해보겠다는 꿈을 꾸고 있는 중이다.

베타테스터_ 이아름

공대 아름이와 이름만 같은 코스모스 졸업을 앞둔 취업준비생이다. 정보통신공학 전공으로 현재 프로그래머가 되기 위해 밤낮없이 코딩에 매진하고 있다.

대상 독자 및 예제 파일

초급 **초중급** 중급 중고급 고급

이 도서는 AngularJS의 핵심을 빠르게 확인하고 싶은 독자를 대상으로 한다. 도서의 내용을 보다 잘 이해하려면, HTML과 자바스크립트를 어느 정도 알고 있어야 한다. 다음과 같은 독자들에 많은 도움이 될 것이다.

- 규모 있는 웹 애플리케이션 프로젝트의 실무 개발자
- 프레임워크 기반으로 자바스크립트에 익숙해지려는 웹 퍼블리셔
- jQuery 입문 이상으로 나아가려는 자바스크립트 개발자

이 도서의 예제 소스 코드는 다음 웹 사이트에서 다운받을 수 있다.

- https://github.com/shyamseshadri/angularjs-book (영문 버전)
- http://www.hanb.co.kr/exam/2641 (한글 버전)

영문 버전은 AngularJS 1.0.4 버전을 사용하였으며, 한글 버전은 AngularJS 1.0.7을 사용하였다. 이 도서에 있는 예제는 AngularJS 1.0.7을 사용하여 소스 코드를 테스트하였다.

소스 코드에는 구글 AngularJS 프레임워크 파일의 URL로 인클루딩되어 있다. 하지만 소스 코드의 간결성을 위해 일부 코드는 AngularJS 사용하였다. 이 도서에서 사용한 AngularJS 프레임워크 파일은 다음 웹 사이트에서 다운받을 수 있다.

- http://angularjs.org/

한빛 eBook 리얼타임

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾는 것도 쉽지 않습니다. 또한 잘 정리되어 책으로 나오기까지는 오랜 시간이 걸립니다. 어떻게 하면 조금이라도 더 유용한 정보를 빠르게 얻을 수 있을까요? 어떻게 하면 남보다 조금 더 빨리 경험하고 습득한 지식을 공유하고 발전시켜 나갈 수 있을까요? 세상에는 수많은 종이책이 있습니다. 그리고 그 종이책을 그대로 옮긴 전자책도 많습니다. 전자책에는 전자책에 적합한 콘텐츠와 전자책의 특성을 살린 형식이 있다고 생각합니다.

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

1. **eBook Only - 빠르게 변화하는 IT 기술에 대해 핵심적인 정보를 신속하게 제공합니다.**

   500페이지 가까운 분량의 잘 정리된 도서(종이책)가 아니라, 핵심적인 내용을 빠르게 전달하기 위해 조금은 거칠지만 100페이지 내외의 전자책 전용으로 개발한 서비스입니다. 독자에게는 새로운 정보를 빨리 얻을 수 있는 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내는 선배, 전문가, 고수 분에게는 보다 쉽게 집필할 수 있는 기회가 될 수 있으리라 생각합니다. 또한 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 하고 있습니다.

2. **무료로 업데이트되는, 전자책 전용 서비스입니다.**

   종이책으로는 기술의 변화 속도를 따라잡기가 쉽지 않습니다. 책이 일정 분량 이상으로 집필되고 정리되어 나오는 동안 기술은 이미 변해 있습니다. 전자책으로 출간된 이후에도 버전 업을 통해 중요한 기술적 변화가 있거나 저자(역자)와 독자가 소통하면서 보완하여 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

3. 독자의 편의를 위하여 DRM-Free로 제공합니다.

구매한 전자책을 다양한 IT 기기에서 자유롭게 활용할 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해 독자 여러분이 언제 어디서 어떤 기기를 사용하더라도 편리하게 전자책을 볼 수 있도록 하기 위함입니다.

4. 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다.

종이책을 그대로 옮겨 놓아 가독성이 떨어지고 읽기 힘든 전자책이 아니라, 전자책의 환경에 가능한 한 최적화하여 쾌적한 경험을 드리고자 합니다. 링크 등의 기능을 적극적으로 이용할 수 있음은 물론이고 글자 크기나 행간, 여백 등을 전자책에 가장 최적화된 형태로 새롭게 디자인하였습니다.

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

차례

1 | AngularJS 이해하기

뛰어난 웹 기반 애플리케이션을 제작하는 우리의 능력이 대단해진 만큼, 개발 과정도 많이 복잡해졌다. 필자는 Angular 팀에서 Ajax 애플리케이션을 개발할 때 부딪히는 난관을 줄이고자 했으며, 구글에 있을 때는 지메일, 구글맵, 캘린더 등 대형 웹 애플리케이션의 힘든 제작 과정을 처음부터 끝까지 해냈다. 그리고 이 경험을 활용해서 많은 사람에게 도움을 줄 수 있을 것이라 생각했다.

필자는 두세 줄의 코드를 작성했던 맨 처음과 비슷하게 웹 애플리케이션을 작성하고 싶었고, 그렇게 만든 결과물은 너무 형편없어 아연실색했다. 필자는 코딩 과정이 웹 브라우저의 특이한 내부 동작 원리에 맞춰나가는 게 아니라 창조한다는 느낌이길 바랬다.

그와 동시에, 처음에 어떤 설계가 애플리케이션을 쉽게 제작하고 이해할 수 있을지 판단할 수 있으면서 애플리케이션이 발전해가는 중간중간에 용이한 테스트/확장/유지보수를 위해 합리적 선택을 계속 해나갈 수 있는 환경을 원했다.

이를 위해서 필자는 Angular 프레임워크를 사용했는데, 얻어낸 성과를 생각하면 아주 기쁘다. 특히, Angular 관련 오픈소스 커뮤니티는 필자에게 많은 지원과 정보를 제공해주었다. 여러분도 커뮤니티에 가입해서 Angular를 훨씬 더 개선할 수 있는 방법을 공유했으면 좋겠다.

이 책의 예제 소스 코드는 GitHub[01]에 올려두었다. GitHub 페이지에서 소스 코드를 직접 볼 수도 있고 다운받아 실행해볼 수 있다.

---

01 https://github.com/shyamseshadri/angularjs-book

1.1 AngularJS를 이해하는 데 필요한 각종 개념

Angular 애플리케이션을 제작할 때 활용하는 핵심 개념이 몇 가지 있다. 필자가 그 개념들을 만든 것이 아니라 단지 다양한 개발 환경에서 효율성이 높은 문법을 비중 있게 차용해서 HTML, 웹 브라우저, 그 밖의 익숙한 각종 웹 표준 기술을 포괄하는 방식으로 구현했을 뿐이다.

1.1.1 클라이언트 측 템플릿

멀티 페이지multi-page 웹 애플리케이션은 서버에 있는 데이터를 HTML과 조립하고 연결한 후, 완성된 페이지를 웹 브라우저에 보내는 방법으로 페이지를 생성한다. Ajax 애플리케이션(싱글 페이지single-page 애플리케이션) 대부분도 이와 같은 방식으로 HTML을 생성한다. Angular 애플리케이션은 템플릿과 데이터가 웹 브라우저로 보내진 후, 웹 브라우저에서 조립된다는 점이 다르다. 결국 서버의 역할은 템플릿에 사용될 정적 리소스를 올려두고 템플릿에 필요한 데이터를 필요할 때마다 제공하는 것뿐이다.

데이터와 템플릿을 웹 브라우저에서 조립하는 작업을 Angular로 작성하면 소스 코드 1-1과 같다. 이번 예제는 프로그래밍의 입문으로 여겨지는 Hello World로, "Hello, World"를 하나의 문자열로 작성하는 것이 아니라 "Hello"라는 인사말을 나중에 수정할 수 있게끔 데이터로 구조화할 것이다.

hello.html 파일을 열고 소스 코드 1-1과 같이 템플릿을 작성하자.

소스 코드 1-1 Hello World 애플리케이션의 템플릿(hello.html)

```
<html ng-app>
<head>
    <script src="angular.js"></script>
    <script src="controllers.js"></script>
</head>
<body>
    <div ng-controller='HelloController'>
        <p>{{greeting.text}}, World</p>
    </div>
</body>
</html>
```

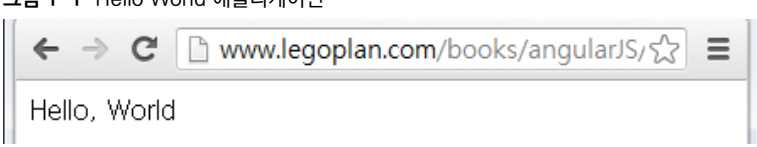
controllers.js 파일의 로직은 소스 코드 1-2와 같다.

소스 코드 1-2 Hello World 애플리케이션의 컨트롤러(controllers.js)

```
function HelloController($scope) {
    $scope.greeting = {
        text : 'Hello'
    };
}
```

선호하는 웹 브라우저에서 소스 코드 1-1의 hello.html을 로딩하면 그림 1-1과 같이 나타난다.

그림 1-1 Hello World 애플리케이션



위에서 사용한 HelloController 메서드는 흔히 사용하는 대부분의 메서드와 달리 다음과 같은 특징이 있다.

- HTML 파일 안에는 이벤트 리스너를 붙일 위치를 식별할 수 있는 클래스나 ID가 전혀 없다.
- HelloController 메서드가 greeting.text에 Hello를 할당한다면 이벤트 리스너를 등록하거나 콜백callback을 작성하지 않아도 된다.
- HelloController 메서드는 일반 자바스크립트 클래스며 Angular에 속한 어느 클래스도 상속받지 않는다.
- HelloController 메서드는 필요한 $scope 객체를 생성하지 않고 매개변수로 받는다.
- HelloController의 생성자를 호출하는 코드나 생성 시점을 알아내는 코드를 작성하지 않아도 된다.

나중에 더 많은 차이점을 알아보겠지만, 이 정도의 차이점만 보더라도 Angular 애플리케이션의 구조가 기존 애플리케이션과 다르다는 것만큼은 확실히 알 수 있다.

필자가 왜 Angular를 이용한 설계 방식을 택했고, Angular는 어떠한 원리로 돌아가는지 궁금할 것이다. 이제 Angular가 다른 곳에서 차용해온 유익한 몇 가지 개념을 살펴보자.

1.1.2 모델 뷰 컨트롤러(MVC)

MVCModel View Controller 애플리케이션 구조는 1970년대에 스몰토크Smalltalk의 구성 요소로서 도입됐다. 스몰토크에 속해 있던 초창기때부터 MVC는 사용자 인터페이스가 필수인 거의 모든 데스크톱 개발 환경에 사용됐다. C++, 자바, Objective-C 중 어느 것을 사용하더라도 MVC 구조를 일부 활용할 수 있다. 그러나 최근까지 MVC는 웹 개발에 거의 사용되지 않았다.

MVC는 개발자가 코드에서 데이터(모델model) 관리, 애플리케이션 로직(컨트롤러controller), 데이터 표현(뷰view)를 명확하게 서로 분리하는 것이 주된 목적이다.

뷰는 모델에서 데이터를 가져와 사용자에게 표시한다. 사용자가 클릭이나 타이핑 등으로 애플리케이션을 조작하면, 컨트롤러는 모델에 있는 데이터를 변경하는 식으로 응답한다. 모델은 변경이 발생했음을 뷰에 알려서, 뷰가 변경사항을 현재 표시 중인 내용에 적용하게 한다.

Angular 애플리케이션에서 뷰는 DOMDocument Object Model, 문서 객체 모델이고, 컨트롤러는 자바스크립트 클래스며, 모델 데이터는 객체 속성에 저장된다.

필자가 MVC를 깔끔하다고 생각하는 이유는 다음과 같다.

첫째, MVC에는 무엇을 어디에 넣어야 할지에 대한 멘탈 모델mental model이 있어서, 개발자가 매번 멘틀 모델을 생각하지 않아도 된다. 둘째, 자신이 작성한 코드가 MVC 구조에 따라 구성되어 있음을 한눈에 알 수 있어서 프로젝트 팀원들도 무슨 내용의 코드인지를 쉽게 이해할 수 있다. 셋째, MVC는 애플리케이션을 보다 쉽게 확장하고 유지하며 테스트할 수 있는 굉장한 장점들이 있다.

1.1.3 데이터 바인딩

Ajax 싱글 페이지 애플리케이션이 보편화되기 전에는, 전송된 데이터를 사용자에게 보여주기 전에 HTML의 문자열을 데이터와 합치는 방식으로 사용자 인터페이스를 손쉽게 작성할 수 있는 레일스Rails, PHP, JSP 같은 플랫폼을 이용했다.

jQuery 같은 라이브러리는 MVC 모델을 클라이언트로 확장해서 개발자가 비슷한 방식으로 인터페이스를 작성할 수 있으며, 페이지 전체가 아니라 DOM의 일부분만을 따로 업데이트할 수 있다. 이 책에서는 템플릿 HTML 문자열을 데이터와 합친 후, 플레이스홀더placeholder 요소에 있는 innerHtml 속성을 설정하여 합친 데이

터를 DOM 내부의 원하는 위치에 삽입하는 방식을 사용한다.

이 방식은 다른 건 다 좋은데, UI에 업데이트된 데이터를 삽입하거나 사용자 입력 내용에 따라 데이터를 변경할 때, UI와 자바스크립트 속성에 데이터를 할당하기 위해 추가해야 할 작업이 상당히 많다.

그런데 개발자가 코딩하지 않고도 이 모든 작업이 저절로 처리된다면 얼마나 좋을까? 개발자는 단지 UI의 어느 부분에 어떤 자바스크립트 속성을 할당할지 선언만 하고 동기화는 자동으로 이뤄지게 할 수 있다면 얼마나 좋을까? 이런 식의 프로그래밍을 '데이터 바인딩'이라고 한다. 필자는 Angular에 데이터 바인딩 기능을 포함시켰다. 데이터 바인딩을 MVC와 연동하면, 코딩 작업 없이 뷰와 모델을 작성할 수 있기 때문이다. 데이터를 한 곳에서 다른 곳으로 옮길 때 필요한 작업 대부분은 자동으로 이뤄진다.

이러한 기능을 눈으로 확인하기 위해 소스 코드 1-1의 템플릿을 동적으로 수정해 보자. 앞에서와 같이 HelloController 메서드는 greeting.text 값을 한 번 지정하고 이후에는 절대로 변경하지 않는다. 동적으로 만들기 위해 사용자의 입력 내용에 따라 greeting.text의 값이 변하는 텍스트 인풋을 추가하자. 수정한 템플릿은 소스 코드 1-3과 같다.

소스 코드 1-3 Hello World 애플리케이션의 템플릿 1차 수정(HelloDynamic.html)

```
<html ng-app>
<head>
   <script src="angular.js"></script>
   <script src="controllers.js"></script>
</head>
<body>
   <div ng-controller='HelloController'>
      <input ng-model='greeting.text'>
```

```
        <p>{{greeting.text}}, World</p>
    </div>
</body>
</html>
```

HelloController 컨트롤러는 그대로 두면 된다. 이 파일을 웹 브라우저에서 로딩하면 그림 1-2와 같은 화면이 나타난다.

그림 1-2 Hello World 애플리케이션의 기본 상태

인풋 필드input field 안에 있는 Hello를 '안녕'으로 수정하면 화면이 그림 1-3과 같이 변한다.

그림 1-3 Hello World 애플리케이션에서 입력 내용을 수정한 결과

인풋 필드에 'change 이벤트 리스너'를 붙이지 않고도 동적으로 업데이트되는 UI를 작성했다. 서버에서 변경된 내용이 UI에 동적으로 업데이트될 뿐만 아니라, UI에서 변경된 내용이 서버에 동적으로 저장되기도 한다. 앞서 작성한 컨트롤러에서 서버로 요청하고, 응답을 받으며, scope.greeting.text 속성에 반환되는 값을 지정할 수도 있다. Angular는 인풋의 내용과 중괄호 안의 텍스트를 반환되는 값으로 자동 업데이트한다.

1.1.4 종속물 주입

앞에서도 언급했지만 HelloController를 통해 이뤄지는 작업 대부분을 개발자가 작성할 필요가 없음은 수백 번 강조해도 지나치지 않다. 예컨대 데이터 바인딩을 수행하는 $scope 객체는 자동으로 넘어오기 때문에, 함수를 호출해서 객체를 생성하지 않아도 된다. 단지 HelloController 생성자 안에 $scope를 넣기만 하면 요청이 이뤄진다.

뒤에 나올 장들에서 살펴보겠지만, 요청할 수 있는 것은 $scope 객체만이 아니다. 예를 들어 사용자 웹 브라우저에 표시된 현재 URL에 데이터를 바인딩하려면, 소스 코드 1-4와 같이 HelloController 생성자 메서드에 $location 매개변수도 추가로 전달해서 이를 관리하는 객체를 요청할 수 있다.

소스 코드 1-4 Hello World 애플리케이션의 컨트롤러에 위치를 추가(controllers.js)

```
function HelloController($scope, $location) {
    $scope.greeting = {
        text : 'Hello'
    };
    // 여기에 $location을 사용해 유용한 기능을 구현
}
```

이 놀라운 결과를 가능케 한 것은 Angular의 '종속물 주입 시스템Dependency Injection System'이다. 종속물 주입을 이용하면 개발자는 종속물을 작성하지 않아도 되며 작성한 클래스가 직접 필요한 것을 요청하는 개발 방식을 따르면 된다.

이런 개발 방식은 '디미터의 법칙Law of Demeter'[02]이라는 디자인 패턴을 따른다. 디미

02 역자 주_ 디미터의 법칙 패턴이란 객체 간의 약결합을 요구하는 객체지향 프로그램을 개발하기 위한 소프트웨어 설계 패턴으로 다음과 같은 원칙을 지닌다.
- 각 객체는 자신에게 밀접히 관련된 객체들에 대한 정보만을 지니되, 한정된 정보만 지녀야 한다.
- 각 객체는 밀접히 관련된 객체끼리만 소통해야 하며, 관련 없는 객체와 소통해선 안 된다.
- 직접적으로 관련된 객체끼리만 소통해야 한다.

터의 법칙을 다른 말로 '최소 지식의 원칙the principle of least knowledge'이라고 한다. HelloController의 역할은 greeting 모델의 초기 상태를 설정하는 것으로, 디미터의 법칙 패턴에 따라 HelloController는 $scope 객체가 생성되는 원리나 $scope 객체가 있는 위치 같은 불필요한 정보에는 관여하지 말아야 한다.

이 특징은 Angular 프레임워크로 생성된 객체에만 적용되는 것이 아니다. 소스 코드 1-4의 미완성된 부분을 직접 작성해보자.

1.1.5 지시어

Angular의 가장 큰 장점 중 하나는 템플릿을 HTML 파일로 작성할 수 있다는 것이다. 이것이 가능한 이유는 프레임워크 코어에 강력한 DOM 변환 엔진을 넣어두었기 때문이다. DOM 변환 엔진을 통해 개발자는 HTML 문법을 확장할 수 있다.

앞에서 템플릿 예제를 통해 HTML 명세에는 나와 있지 않은 새로운 속성 몇 가지를 이미 살펴보았다. 예제는 데이터 바인딩을 위한 이중 중괄호 표기, 컨트롤러가 뷰의 어느 부분을 감독할지를 지정하는 ng-controller, 인풋을 해당 모델의 구성물에 바인딩하는 ng-model을 사용한다. 이러한 HTML 확장 문법을 '지시어directive'라고 한다.

Angular에는 애플리케이션의 뷰를 쉽게 정의할 수 있는 여러 지시어가 있다(나중에 더 많은 지시어를 살펴볼 것이다). 이런 지시어는 템플릿을 정의할 수 있다. 지시어를 선언해서 애플리케이션의 구동 원리를 지정할 수도 있고, 재사용 가능한 구성요소를 생성할 수도 있다.

그리고 Angular에 내장된 지시어를 사용할 수도 있지만, 자신만의 지시어를 작성해서 HTML 템플릿 기능을 자유롭게 확장할 수도 있다.

1.2 장바구니 예제

조금 복잡한 예제를 통해 Angular의 더 많은 기능을 알아보자. 쇼핑몰 애플리케이션을 제작한다고 생각해보자. 쇼핑몰 애플리케이션의 어딘가에는 사용자의 장바구니를 표시하고 사용자가 수정할 수 있게 해야 한다. 다음 장바구니 부분의 코드를 살펴보자.

소스 코드 1-5 장바구니 애플리케이션(order-form.html)

```
<!doctype html>
<html lang='ko' ng-app>
<head>
    <title>장바구니</title>
</head>
<body ng-controller="CartController">
    <h1>내 장바구니</h1>
    <div ng-repeat="item in items">
        <span>{{item.title}}</span>
        <input ng-model="item.quantity">
        <span>{{item.price | currency}}</span>
        <span>{{item.price * item.quantity | currency}}</span>
        <button ng-click="remove($index)">삭제</button>
    </div>
    <script src="angular.js"></script>
    <script>
        function CartController($scope) {
            $scope.items = [ {
                title : '페인트 그릇',
                quantity : 8,
                price : 3.95
            }, {
                title : '땡땡이 리본',
```

```
                quantity : 17,
                price : 12.95
            }, {
                title : '공깃돌',
                quantity : 5,
                price : 6.95
            } ];
            $scope.remove = function(index) {
                $scope.items.splice(index, 1);
            }
        }
    </script>
</body>
</html>
```

웹 브라우저에서 소스 코드 1-5를 로딩하면 그림 1-4와 같은 UI를 볼 수 있다.

그림 1-4 장바구니 UI

여기서는 소스 코드 1-5를 간단히 분석해보자. 더 자세한 내용은 이 책의 나머지 장을 참고하기 바란다.

맨 위의 코드부터 차근차근 살펴보자.

```
<html lang='ko' ng-app>
```

어떤 요소에 ng-app 지시어를 지정하면, 페이지에서 그 요소에 해당하는 부분을 Angular가 관리하게 된다. 앞의 코드에서 ng-app 지시어를 지정한 요소가 〈html〉이므로, Angular는 페이지 전체를 관리하게 된다. 이렇게 Angular가 페이지 전체를 관리하게 지정하는 것이 목적에 대부분 맞지만, 다른 메서드를 사용해 페이지를 관리하는 기존 애플리케이션에 Angular를 통합할 때는 애플리케이션의 〈div〉 요소에 지정하기도 한다.

```
〈body ng-controller="CartController"〉
```

Angular에서 개발자는 '컨트롤러'라고 하는 자바스크립트 클래스로 구성된 페이지 영역을 관리한다. CartController가 〈body〉와 〈/body〉 사이의 모든 것을 관리하도록 선언하려면, 앞의 코드처럼 body 태그에 컨트롤러를 삽입하면 된다.

```
〈div ng-repeat="item in items"〉
```

앞의 코드에서 ng-repeat 지시어는 지정된 〈div〉 요소 내부의 DOM을 items 배열의 모든 원소에 한 번씩 복사하라고 명령한다. 그리고 〈div〉의 모든 사본에서 현재 요소[03]에 item이라는 속성을 지정해서 템플릿 안에 사용할 수 있게 한다. 보다시피 이것은 제품명, 수량, 단가, 총액, 항목 삭제 버튼이 들어 있는 〈div〉를 3개 생성한다.

```
〈span〉{{item.title}}〈/span〉
```

03 역자 주_ ng-repeat는 프로그램 루프 같은 개념이다. 이 지시어가 지정된 요소 안에 포함된 요소는 계속 반복된다. c 언어 같으면 i=0; i〈5; i++ 라는 for문 조건절 하위에서 i가 가리키는 것이 현재가 된다.

Hello World 예제에서 보았다시피 이중 중괄호 {{ }}를 사용해 데이터 바인딩하면, 변수의 값을 페이지 안에 삽입하고 동기화를 유지할 수 있다. 표현식 {{item.title}}은 반복문 안에 현재 item을 가져와서, item의 title 속성에 할당돼 있는 내용을 DOM에 삽입한다.

```
<input ng-model="item.quantity">
```

앞의 코드처럼 ng-model 지시어를 지정하면 인풋 필드와 item.quantity의 값 사이에 데이터 바인딩이 이뤄진다.

〈span〉 요소로 감싸인 {{ }} 표기법은 "값을 여기에 삽입해라"는 일방적인 관계를 설정한다. 사용자 입장에서는 이 기능으로 충분하지만, 장바구니 애플리케이션에서는 사용자가 인풋의 수량을 변경하는 시점을 알아야 실시간으로 표시되는 총액을 변경할 수 있다.

ng-model 지시어를 지정했으므로 변경사항이 계속 모델에 반영된다. ng-model 지시어의 선언으로 인해 item.quantity의 값이 텍스트 필드에 삽입될 뿐만 아니라, 사용자가 새 값을 입력할 때마다 item.quantity를 자동으로 업데이트한다.

```
<span>{{item.price | currency}}</span>
<span>{{item.price * item.quantity | currency}}</span>
```

단가와 총액을 달러 단위로 표시하고자 한다. Angular에 있는 '필터' 기능을 이용하면 텍스트를 변환할 수 있다. 값을 화폐 단위로 형식화하는 currency라는 내장 필터가 있는데, 이 필터를 이용하면 값을 달러 단위의 형식으로 변환할 수 있다. 필터에 대한 상세한 내용은 다음 장을 참고하자.

```
<button ng-click="remove($index)">삭제</button>
```

앞의 코드로 인해, 제품 옆에는 장바구니 항목 삭제를 위해 사용자가 클릭할 수 있는 [삭제] 버튼이 생긴다. [삭제] 버튼을 클릭하면 remove() 함수가 호출된다. 삭제할 항목을 알려주기 위해 remove() 함수의 인자로 ng-repeat 반복 횟수가 저장되어 있는 $index를 전달했다.

```
function CartController($scope) {
```

CartController 메서드는 장바구니 로직을 관리한다. 컨트롤러에 $scope가 필요하다는 것을 Angular에 알리기 위해 $scope를 매개변수로 전달했다. $scope를 통해 UI를 구성하는 요소들의 데이터 바인딩이 이뤄진다.

```
$scope.items = [
   { title : '페인트 그릇', quantity : 8, price : 3.95 },
   { title : '땡땡이 리본', quantity : 17, price : 12.95 },
   { title : '공깃돌', quantity : 5, price : 6.95 }
];
```

$scope.items를 정의해서 사용자 장바구니에 든 항목들의 집합을 나타내는 더미 데이터 해시dummy data hash를 생성했다. UI와 데이터 바인딩이 가능해야 하므로, 사용자 장바구니에 든 항목을 $scope에 추가한다.

당연한 이야기지만, 장바구니가 메모리에서만 구동될 수는 없으므로 서버와 통신해서 데이터를 데이터베이스 같은 영구 저장소에 적절히 저장하게 해야 한다. 데이터의 영구 저장에 대해서는 뒤에 나오는 장들에서 다루겠다.

```
$scope.remove = function(index) {
    $scope.items.splice(index, 1);
}
```

UI에 바인딩하려면 remove() 함수가 필요하므로 $scope에도 remove() 함수를 추가했다. 메모리에서만 구동되는 장바구니 애플리케이션은 remove() 함수가 배열에 있는 항목을 삭제하는 것만 가능하다. ng-repeat 지시어로 생성된 〈div〉 목록은 데이터 바인딩에 의해 실시간 반영되므로, 항목을 삭제하면 해당 항목의 행이 사라지면서 자동으로 아래 항목이 그 행으로 올라온다. 이 remove() 함수는 사용자가 항목의 [삭제] 버튼을 클릭할 때마다 UI에서 호출됨을 명심하자.

1.3 나머지 장에서 다룰 내용

이 장에서는 Angular의 가장 기본적인 문법과 아주 간단한 예제만을 살펴보았다. 나머지 장에서는 Angular 프레임워크의 필수 기능에 대해서 알아볼 것이다.

2 | AngularJS 애플리케이션 해부

필요한 함수만 골라 사용하는 일반 라이브러리와 달리, Angular 프레임워크의 모든 구성요소는 협업 솔루션으로 사용하게끔 설계되어 있다. 이 장에서는 Angular의 기본 구성요소를 간결하게 설명하겠다. 이를 통해 각 요소의 연동 방법을 이해할 수 있을 것이다. 구성요소 대부분에 대한 상세한 설명은 나머지 장에서 다룬다.

2.1 Angular 호출

Angular를 시작하기 위해 모든 애플리케이션에 반드시 다음 두 가지 작업을 해야 한다.

1. angular.js 라이브러리 파일을 로딩해야 한다.
2. ng-app 지시어를 지정해서 DOM의 어느 부분을 관리할지 Angular에 전달해야 한다.

2.1.1 스크립트 로딩

angular.js 라이브러리 로딩은 간단하며 일반 자바스크립트 라이브러리를 로딩하는 것과 방법이 같다. 구글의 CDN(Content Delivery Network, 콘텐츠 전송 네트워크)에 올려져 있는 스크립트 파일을 로딩하는 방법은 다음과 같다.

```
<script
  src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.7/angular.min.js">
</script>
```

구글의 CDN을 이용하는 것은 여러모로 좋다. 구글 서버는 빠른데다가 어떤 애플

리케이션이든 캐시에 스크립트를 저장하는 기능이 있기 때문이다. 따라서 사용자는 Angular 플랫폼 기반의 애플리케이션을 여러 개 사용할 때 Angular 스크립트 파일을 한 번만 받으면 된다. 게다가 사용자가 구글 CDN의 Angular 링크를 이용한 웹 사이트를 방문한 적이 있으면, 다른 웹 사이트를 방문할 때 Angular 스크립트 파일을 다시 받지 않아도 된다.

Angular 스크립트 파일을 로컬이나 다른 위치에 두고 사용하고 싶다면, 구글 Angular 스크립트 파일을 받아서 자신이 원하는 위치에 두고 src 속성에 정확한 파일 경로를 지정하면 된다.

2.1.2 ng-app 지시어로 Angular의 경계 선언

ng-app 지시어를 이용하면 페이지의 어느 부분을 관리할지 Angular에 명령할 수 있다. 순수 Angular 애플리케이션을 제작할 때는 ng-app 지시어를 다음과 같이 〈html〉 태그에 넣는다.

```
<html ng-app>
    ...
</html>
```

앞의 코드는 페이지 안의 모든 DOM 요소를 관리하라고 Angular에 명령한다.

기존 애플리케이션이 있는데, 그 애플리케이션이 자바나 레일스 같은 타 기술을 이용해 DOM을 관리하게 돼 있다면, 다음과 같이 Angular가 관리하게 할 부분을 〈div〉 같은 요소로 묶은 후 ng-app 지시어를 그 요소에 넣으면 된다.

자바나 레일스 같은 기술을 이용해 DOM을 관리하는 애플리케이션이 있다면, 다음과 같이 Angular가 관리하게 할 부분을 〈div〉 같은 요소로 묶은 후 ng-app 지시어를 그 요소에 넣으면 된다.

```
<html>
    ...
    <div ng-app>
        ...
    </div>
    ...
</html>
```

2.2 모델 뷰 컨트롤러

Angular가 모델 뷰 컨트롤러 애플리케이션 설계 방식을 지원한다고 1장에서 설명했다. Angular 애플리케이션은 설계가 아주 유연하지만, 다음과 같은 기본적으로 지켜야 할 규칙이 있다.

- 모델에는 애플리케이션의 현 상태를 나타내는 데이터가 들어간다.
- 뷰는 현 상태를 나타내는 데이터를 표시한다.
- 컨트롤러는 모델과 뷰의 관계를 관리한다.

모델은 객체 속성을 사용하거나 데이터가 담긴 기본 타입만을 사용해 작성한다. 모델 변수는 특별할 것이 없다. 사용자에게 어떤 텍스트를 표시하려면 문자열을 다음과 같이 지정하면 된다.

```
var someText = '지금 여러분의 여정은 시작됐습니다.';
```

뷰는 HTML 페이지로 템플릿을 작성하고 거기에 모델에서 전달받은 데이터를 합쳐서 생성한다. 앞에서 보았다시피 DOM 안에 플레이스홀더placeholder를 삽입하고, 플레이스홀더의 텍스트를 다음과 같이 지정하면 된다.

```
<p>{{someText}}</p>
```

이중 중괄호 문법은 기존 템플릿에 새로운 내용을 삽입한다는 뜻에서 인터폴레이션interpolation이라고 한다.

컨트롤러는 클래스 또는 타입이며, 다음과 같이 모델을 구성할 객체나 기본 타입을 전달 받은 $scope 객체에 할당해서 Angular에 알리는 것이 목적이다.

```
function TextController($scope) {
    $scope.someText = someText;
}
```

이제까지 설명한 코드를 조합하면 다음과 같다.

소스 코드 2-1 1.dataBinding.html

```
<html ng-app>
<body ng-controller="TextController">
    <p>{{someText}}</p>

    <script
        src="https://ajax.googleapis.com/ajax/libs/angularjs/1.0.7/
        angular.min.js">
    </script>

    <script>
        function TextController($scope) {
            $scope.someText = '지금 여러분의 여정은 시작됐습니다.';
        }
    </script>
```

```
</body>
</html>
```

앞의 코드를 웹 브라우저에서 로딩하면 다음과 같이 출력되는 것을 볼 수 있다.

지금 여러분의 여정은 시작됐습니다.

이 기본 스타일 모델은 단순한 경우에나 사용할 수 있다. 애플리케이션 대부분은 데이터를 보관할 모델 객체를 작성해야 한다. 메시지 모델 객체를 작성하고 그 객체를 통해 someText를 저장해보자.

```
var someText = '지금 여러분의 여정은 시작됐습니다.';
```

그러기 위해서는 앞의 코드를 다음과 같이 수정한다.

```
var messages = {};
messages.someText = '지금 여러분의 여정은 시작됐습니다.';
function TextController($scope) {
    $scope.messages = messages;
}
```

그리고 이것을 템플릿 파일에서 다음과 같이 적용하자.

```
<p>{{messages.someText}}</p>
```

나중에 $scope 객체를 설명할 때 보겠지만, 이런 식으로 모델 객체를 작성하면 프로토타입 상속prototypal inheritance으로 인해 $scope 객체에서 발생할 수 있는 예기치 못한 동작을 막을 수 있다.

장기적으로 바람직한 습관을 이야기하면서도, 앞의 예제는 TextController를 전역 스코프에 작성했다. 예제니까 그래도 되지만, 원래 컨트롤러는 애플리케이션의 관련 부분들을 나타내는 네임스페이스를 사용하는 모듈 안에 정의하는 것이 올바른 방법이다.[01] 수정한 코드는 다음과 같다.

소스 코드 2-2 2.dataBinding.html

```
<html ng-app='myApp'>
<body ng-controller="TextController">
    <p>{{someText.message}}</p>

    <script
        rc="https://ajax.googleapis.com/ajax/libs/angularjs/1.0.7/
        angular.min.js">
    </script>

    <script>
        var myAppModule = angular.module('myApp', []);

        myAppModule.controller('TextController',
            function TextController($scope) {
            var someText = {};
            someText.message = '지금 여러분의 여정은 시작됐습니다.';
            $scope.someText = someText;
        });
    </script>
</body>
</html>
```

01 역자 주_ 'myApp'은 애플리케이션 관련 부분을 나타내는 네임스페이스고 myAppModule은 네임스페이스 myApp을 사용하는 모듈이다.

수정한 코드를 살펴보자. 먼저 ng-app 지시어에 모듈명 myApp을 지정했다. 이후에 스크립트에서 Angular 객체를 호출해서 myApp이라는 모듈을 생성한 후, 생성한 모듈의 컨트롤러 함수를 호출하여 작성한 컨트롤러 함수를 매개변수로 전달했다.

뒤에서 모듈에 대해 보다 자세히 살펴볼 것이다. 지금은 전역 네임스페이스를 제쳐두고 생각하는 것이 좋은데, 그러기 위한 방법으로 모듈을 사용한다.

2.3 템플릿과 데이터 바인딩

Angular 애플리케이션에서 템플릿은 다른 정적 리소스처럼 서버에서 로딩되거나, <script> 태그 안에 정의하는 HTML 문서에 불과하다. 템플릿 안에 UI를 정의할 때는 표준 HTML과 Angular 지시어를 사용한다. Angular 지시어를 사용하려면 UI 컴포넌트가 필요하다.

Angular는 우선 웹 브라우저에서 템플릿을 데이터와 합쳐 완전한 애플리케이션을 만든다. 이렇게 템플릿과 데이터를 합치는 코드는 1장에서 장바구니에 든 품목들을 표시하는 예제를 통해 살펴보았다.

```
<div ng-repeat="item in items">
   <span>{{item.title}}</span>
   ...
</div>
```

앞의 코드는 바깥의 를 한 번 표시한 후, items 배열의 원소마다 안의 내용을 반복해서 표시한다.

그럼 이 데이터는 어디에서 왔을까? 장바구니 예제에서는 데이터를 코드의 배열 안에 정의했다. 이런 방식은 UI를 제작하면서 어떤 식으로 돌아가는지 테스트할

때 적합하다. 그러나 애플리케이션 대부분은 서버에 저장되어 있는 데이터를 사용한다. 웹 브라우저에서 띄운 애플리케이션이 서버에 접속해서 사용자가 로딩한 페이지에 필요한 데이터를 요청하면, Angular는 그 데이터를 템플릿에 결합시킨다.

기본적인 시작 단계는 다음과 같다.

1. 사용자가 애플리케이션의 첫 페이지를 요청한다.
2. 사용자의 웹 브라우저가 서버로 HTTP 접속을 하고 템플릿이 든 index.html 페이지를 로딩한다.
3. Angular가 index.html 페이지 안에 로딩되고, 페이지 로딩이 완료될 때까지 기다렸다가 ng-app 지시어를 찾아서 템플릿 경계를 인식한다.
4. Angular는 템플릿을 살피면서 지시어와 바인딩을 찾아 리스너 등록과 DOM 조작을 수행하고 서버에서 초기 데이터를 가져온다. 마침내 애플리케이션의 부트스트랩이 완료되고 템플릿은 DOM을 통해 뷰로 변환된다.
5. 애플리케이션은 서버에 접속해서 필요한 경우 사용자에게 보여줄 추가 데이터를 로딩한다.

1단계부터 3단계까지는 모든 Angular 애플리케이션에 공통적으로 적용된다. 개발자는 4단계와 5단계를 선택할 수 있다. 두 단계는 동시에 이뤄질 수도 있고 순차적으로 이뤄질 수도 있다. 맨 처음에 뜨는 뷰에는 요청 횟수를 줄여 성능을 높이기 위해 표시할 데이터를 HTML 템플릿과 함께 수신하는 방법도 있다.

Angular를 이용해 애플리케이션을 구조화하면, 애플리케이션의 템플릿과 그 안에 채워질 데이터를 따로 유지할 수 있다. 그래서 템플릿을 캐시에 저장할 수 있다. 최초 로딩 후에는 새 데이터만 웹 브라우저로 수신되면 된다. 자바스크립트, 이미지, CSS를 비롯한 리소스와 마찬가지로, 이렇게 독립된 템플릿을 캐시에 저장하면 애플리케이션의 성능이 훨씬 더 좋아진다.

2.3.1 텍스트 표시

ng-bind 지시어를 사용하면 UI 내부의 어느 위치에서든 텍스트를 표시하고 업데이트할 수 있다. ng-bind 지시어의 문법은 두 가지다. 다음과 같은 이중 중괄호 문법은 앞서 살펴보았다.

```
<p>{{greeting}}</p>
```

또 다른 문법은 다음과 같이 ng-bind 지시어를 속성으로 지정하는 것이다.

```
<p ng-bind="greeting"></p>
```

앞의 두 방법 중 어느 것을 사용하든 결과는 같다. 만약 greeting 모델 변수에 "Hi there" 값이 지정돼 있다면 Angular는 다음과 같은 HTML을 생성한다.

```
<p>Hi there</p>
```

이 HTML은 웹 브라우저에 "Hi there"이라는 문자열을 표시한다.

그런데 왜 한 문법이 다른 형태의 문법보다 더 많이 사용될까? 필자가 이중 중괄호를 사용하는 인터폴레이션 문법을 만든 이유는 보다 읽기 편하고 코딩을 위한 타이핑 분량을 줄이기 위해서다. 두 문법 모두 결과는 같지만, 이중 중괄호 문법을 사용하면 애플리케이션의 최초 페이지인 index.html을 로딩할 때 Angular가 중괄호 문법을 데이터로 치환하기 전에 사용자가 렌더링 중인 템플릿을 보게 될 가능성이 있다. 단, 로딩된 이후에는 뷰에 이런 문제가 생기지 않는다.

이렇게 되는 이유는 웹 브라우저가 HTML 페이지를 로딩한 후, 그 페이지를 렌더

링하고 나서야 Angular가 개발자의 의도대로 해석하기 때문이다.

다행히 대부분의 템플릿엔 {{ }} 문법을 사용할 수 있다. 그러나 index.html 페이지에서 데이터 바인딩을 할 경우엔, 앞에서 설명한 문제점이 있으니 이중 중괄호 문법 대신 ng-bind 지시어를 사용해야 사용자가 데이터 로딩 중에 이상한 화면을 보지 않게 된다.

2.3.2 폼 인풋

Angular를 이용하면 폼 요소 사용이 간단하다. 앞의 예제에서 보았듯이 ng-model 속성을 사용하면 요소를 모델 속성에 바인딩할 수 있다. 이것은 텍스트 인풋, 라디오 버튼, 체크박스 같은 일반 폼 요소 모두에 적용된다. 체크박스를 다음과 같은 속성에 바인딩할 수도 있다.

```
<form ng-controller="SomeController">
   <input type="checkbox" ng-model="youCheckedIt">
</form>
```

앞의 코드를 해석하면 다음과 같다.

1. 사용자가 박스를 체크하면 SomeController의 $scope에 있는 youCheckedIt 이라는 속성이 'true'가 되고, 이 박스의 체크를 해제하면 youCheckedIt 속성이 'false'가 된다.
2. SomeController에 있는 $scope.youCheckedIt을 'true'로 지정하면 박스는 UI에서 체크된 상태로 표시되고, 'false'로 지정하면 박스가 체크 해제된 상태가 된다.

이제 사용자가 어떠한 조작을 하면 그에 대한 조치를 취해야 하는 경우를 생각해보자. 인풋 요소에 ng-change 속성을 사용하면, 사용자가 인풋 값을 수정하면 호출

되는 컨트롤러 메서드를 지정할 수 있다. 간단한 계산 메서드를 호출해서 초기 자본금이 얼마나 필요한지를 계산한 후 창업주에게 표시해 주는 폼을 작성하면 다음과 같다.

소스 코드 2-3 startup.html

```
<form ng-controller="StartUpController">
    창업 예산: <input ng-change="computeNeeded()"
                      ng-model="funding.startingEstimate">
    권장 자본금: {{funding.needed}}
</form>
```

예제가 간단해야 하므로, 계산 결과는 사용자가 입력한 예산의 10배가 되게 하자. 다음과 같이 초기 기본 값을 0으로 지정한다.

```
function StartUpController($scope) {
    $scope.funding = {
        startingEstimate : 0
    };

    $scope.computeNeeded = function() {
        $scope.needed = $scope.funding.startingEstimate * 10;
    };
}
```

그러나 앞의 코드는 전략상 잠재적인 문제가 있다. 사용자가 인풋 필드에 입력하면 권장 액수를 재계산하는 기능만 구현되어 있다. 사용자가 정확히 입력해서 인풋 필드가 업데이트되면 상관 없지만, 다른 인풋이 이 속성에 바인딩되면 어떻게 될까? 서버에서 보낸 데이터를 수신할 때에 필드가 업데이트된다면 어떻게 될까?

업데이트되는 방식과 상관없이 필드를 업데이트하려면 $watch()라는 $scope 함수를 사용해야 한다(watch 함수에 대한 상세 내용은 이 장의 뒷부분에서 자세히 설명하겠다). 간단히 설명하면 $watch() 함수를 호출할 때는 감시할 표현식과 그 표현식이 변경될 때 호출될 콜백을 매개변수로 전달하면 된다.

이 예제에서는 감시할 대상이 funding.startingEstimate이므로 이것이 변할 때마다 computeNeeded() 함수를 호출하면 된다. 이를 위해 $watch() 함수를 수정하면 StartUpController 메서드는 다음과 같이 된다.

```
function StartUpController($scope) {
    $scope.funding = {
        startingEstimate : 0
    };

    computeNeeded = function() {
        $scope.funding.needed = $scope.funding.startingEstimate * 10;
    };

    $scope.$watch('funding.startingEstimate', computeNeeded);
}
```

$watch() 함수에 매개변수로 전달하는 감시 대상 표현식은 문자열이기 때문에 작은 따옴표로 묶었다. 문자열은 Angular의 표현식과 다를 바 없다. 표현식은 간단한 작업을 수행할 수 있고 $scope 객체 안의 각종 속성에 접근할 수 있다. 표현식에 대해서는 이 장의 뒷부분에서 다루겠다.

함수의 반환 값을 감시할 수도 있지만, funding.startingEstimate 속성을 감시해봤자 결과는 초깃값인 0이며 0에 10을 곱해도 결과는 0이므로 값이 변할 일이 없

기 때문에 감시할 필요가 없다.

게다가 funding.needed는 funding.startingEstimate가 변경될 때마다 자동으로 업데이트되므로 템플릿을 다음과 같이 더 간단히 작성할 수 있다.

```
<form ng-controller="StartUpController">
    창업 예산: <input ng-model="funding.startingEstimate">
    권장 자본금: {{funding.needed}}
</form>
```

변경사항에 대해 일일이 조치를 해서는 안 되는 경우도 있다. 즉, 사용자가 준비됐음을 알려올 때까지 기다렸다가 조치를 해야 하는 경우다. 그러한 예로는 구매 완료하기나 채팅 메시지 전송하기 등이 있다.

폼이 여러 개의 인풋으로 구성돼 있을 때, 폼이 전송되는 순간에 호출할 함수를 지정하려면 폼 자체에 ng-submit 지시어를 사용하면 된다. 다음은 앞의 예제를 확장해서 사용자가 버튼을 클릭해 창업에 필요한 자본금을 요청할 수 있게 수정한 코드다.

```
<form ng-submit="requestFunding()" ng-controller="StartUpController">
    창업 예산: <input ng-change="computeNeeded()" ng-model="startingEstimate">
    권장 자본금: {{needed}}
    <button>저의 창업에 투자해주세요!</button>
</form>

function StartUpController($scope) {
    $scope.computeNeeded = function() {
        $scope.needed = $scope.startingEstimate * 10;
    };
```

```
    $scope.requestFunding = function() {
        window.alert("안됐지만 고객부터 더 모으세요.");
    };
}
```

ng-submit 지시어는 웹 브라우저가 폼을 전송할 때 기본 방식인 POST 동작을 수행하지 않도록 자동으로 방지해 주는 역할도 한다.

폼을 전송하지 않는 상호작용 기능을 제공해야 할 때처럼 다양한 이벤트를 처리할 수 있도록, Angular에는 웹 브라우저의 기본 이벤트 속성과 비슷한 이벤트 처리 지시어가 있다. 웹 브라우저의 onclick 이벤트 속성은 Angular의 ng-click 지시어와 같고 ondblclick 이벤트 속성은 Angular의 ng-dblclick 지시어와 같다.

이것을 적용하기 위해, 인풋 값을 0으로 초기화하는 초기화 버튼을 넣어 창업 자본금 계산 메서드를 마지막으로 확장하면 다음과 같다.

```
<form ng-submit="requestFunding()" ng-controller="StartUpController">
    창업 예산: <input ng-change="computeNeeded()" ng-model="startingEstimate">
    권장 자본금: {{needed}}
    <button>저의 창업에 투자해주세요!</button>
    <button ng-click="reset()">초기화</button>
</form>

function StartUpController($scope) {
    $scope.computeNeeded = function() {
        $scope.needed = $scope.startingEstimate * 10;
    };

    $scope.requestFunding = function() {
        window.alert("안됐지만 고객부터 더 모으세요.");
```

```
    };

    $scope.reset = function() {
        $scope.startingEstimate = 0;
    };
}
```

2.3.3 절제된 자바스크립트에 대해

자바스크립트를 이용하여 개발하다 보면, 주위 사람에게서 '절제된 자바스크립트unobtrusive JavaScript'를 작성해야 한다거나 HTML에 있는 click과 mousedown을 비롯한 각종 인라인 이벤트 핸들러의 남용은 바람직하지 않다는 조언을 듣게 되는 경우가 있다. 그 말은 정답이다.

절제된 자바스크립트라는 개념은 지금껏 여러 가지 의미로 해석돼 왔지만, 절제된 자바스크립트 방식의 코딩을 하는 이유를 살펴보면 다음과 같다.

1. 자바스크립트를 지원하지 않는 웹 브라우저를 사용하는 사람도 있을 수 있다. 누구든지 개발자의 모든 콘텐츠를 볼 수 있어야 하고 웹 브라우저에서 코드 실행 없이 애플리케이션을 이용할 수 있어야 한다.
2. 작동 방식이 다른 웹 브라우저를 사용하는 사람도 있을 수 있다. 스크린 리더를 사용하는 시각 장애인이나 일부 모바일 폰 사용자는 자바스크립트가 사용된 웹 사이트를 이용할 수 없다.
3. 자바스크립트는 플랫폼마다 다르게 동작한다. IEInternet Explorer, 인터넷 익스플로러가 그렇게 만든 주범이다. 웹 브라우저마다 서로 다른 이벤트 처리 코드를 넣어야 한다.
4. 인라인 이벤트 핸들러는 전역 네임스페이스global namespace에 속하는 함수를 참조한다. 이름이 같은 여러 함수로 구성된 라이브러리를 연동하려면 고생이기 때문이다.

5. 인라인 이벤트 핸들러는 구조와 동작을 뒤섞어버린다. 그래서 코드를 이해하고 확장하며 유지보수하기가 한층 어려워진다.

자바스크립트를 절제된 방식으로 작성하면 개발이 훨씬 순조롭다. 그러나 코드가 복잡해지고 가독성이 떨어진다는 것이 단점이다. 이벤트 핸들러가 취할 조치를 해당 요소를 사용해 선언하는 게 아니라, 요소에 ID를 지정하고 ID로 요소를 참조한 후 콜백 함수를 전달하면서 이벤트 핸들러를 지정해야 하기 때문이다. 잘 알려진 위치에 이런 관계를 이어주는 구조를 만드는 방법도 있지만, 대부분의 애플리케이션은 이런 핸들러 설정이 여기저기 흩어진 채로 완성된다.

필자는 Angular를 개발하면서 이 문제점을 재검토했다.

앞에 설명한 개념들이 등장한 시점부터 세월은 흘러 세상이 변했다. 앞의 이유 중 1번은 이제 옛말이 됐다. 자바스크립트 기능이 없는 웹 브라우저는 1990년대에 제작된 웹 사이트밖에 이용하지 못할 정도로 많은 웹 사이트가 자바스크립트를 사용한다. 요즘의 스크린리더는 자바스크립트도 지원하므로 2번의 이유도 지금은 해당 사항이 없다. 'ARIA[Accessible Rich Internet Applications'](02) 시맨틱 태그를 적절히 사용하면 기능이 아주 복잡한 UI도 접근 가능해진다. 현재는 모바일 폰에서도 데스크톱 컴퓨터와 똑같이 자바스크립트가 실행된다.

그럼 인라인 방식의 가독성과 간결성은 유지하면서 3번과 4번의 항목들을 해결할 수 있는지가 관건이다.

앞에서 말했지만 인라인 이벤트 핸들러 대부분은 ng-이벤트핸들러명="표현식" 같은 형태로 Angular에서 제공한다. 여기서 '이벤트핸들러명' 대신에 click, mousedown, change 등을 기입하면 된다. 사용자가 어떤 요소를 클릭한 시점을

02 역자 주_ ARIA(접근가능 리치 인터넷 애플리케이션)는 월드 와이드 웹 컨소시엄 산하의 웹 접근성 전문 기구인 WAI(웹 접근성 협회)가 발행한 기술 명세로, 웹 페이지 (특히 Ajax, HTML, 자바스크립트를 비롯한 관련 기술로 개발된 동적 콘텐츠 및 UI 컴포넌트)의 접근성을 증대시키는 방법이 설명되어 있다.

통지 받으려면 다음과 같이 ng-click 지시어를 사용하면 된다.

```
<div ng-click="doSomething()">...</div>
```

이러면 의도하지 않은 결과가 발생하는 것은 아닌지 걱정되겠지만 안심해도 된다. Angular 지시어는 기존의 이벤트 핸들러와 다음과 같은 점이 다르기 때문이다.

- Angular 지시어는 모든 웹 브라우저에서 똑같이 동작한다. Angular가 개발자 대신 알아서 처리한다.
- Angular 지시어는 전역 네임스페이스엔 적용되지 않는다. 지정한 표현식 doSomething()은 대상 div 요소의 컨트롤러 스코프 내에 있는 함수와 데이터에만 접근할 수 있다.

앞의 두 번째 항목은 조금 난해할 수도 있으니, 다음 예제를 보면서 이해하자. 일반적인 애플리케이션에는 내비게이션 바와 콘텐츠 영역이 들어가기 마련이다. 보통, 콘텐츠 영역은 내비게이션 바에서 어떤 메뉴 옵션을 선택하는지에 따라 바뀐다. 이러한 애플리케이션의 뼈대를 작성하면 다음과 같다.

```
<div class="navbar" ng-controller="NavController">
    …
    <li class="menu-item" ng-click="doSomething()">Something</li> …
</div>

<div class="contentArea" ng-controller="ContentAreaController">
    …
    <div ng-click="doSomething()">...</div>
    …
</div>
```

navbar 클래스 요소 안의 〈li〉와 contentArea 클래스 요소 안의 〈div〉는 둘 다 사용자가 클릭하면 doSomething()이라는 함수를 호출한다. 개발자는 doSomething() 호출로 인해 참조될 함수를 다음과 같이 컨트롤러 함수 안에 지정한다. 참조될 함수는 같은 함수일 수도 있고 서로 다른 함수일 수도 있다.

```
function NavController($scope) {
    $scope.doSomething = doA;
}

function ContentAreaController($scope) {
    $scope.doSomething = doB;
}
```

여기서 doA() 함수와 doB() 함수는 개발자가 정의하기에 따라 같은 함수가 될 수도 있고 다른 함수가 될 수도 있다. 마지막으로, 절제된 자바스크립트 방식의 코딩을 하는 이유는 구조와 동작을 뒤섞어버린다는 5번의 주장이다. 이 주장의 논거는 애매모호하다. 5번의 이유로 인해 구체적으로 어떤 부정적인 결과가 발생하는지를 개발자는 알 수 없기 때문이다. 그러나 이와 매우 비슷한 문제, 즉 표현과 애플리케이션 로직이 뒤섞여버릴지도 모른다는 실무적인 문제를 예상할 수 있다. 앞에 설명한 5번의 이유로 인해 개발자끼리 구조/동작 카테고리에 속하는 문제를 논의할 때마다 두고두고 언급될 정도의 부작용이 생길 것이 뻔하다.

자신의 시스템에 이런 뒤섞임 문제가 있는지를 알아낼 수 있는 간단한 검증 방법이 있다. 애플리케이션 로직을 대상으로 DOM이 없어도 실행할 수 있는 단위 테스트를 작성하는 게 가능할까? Angular에서는 가능하다. 비즈니스 로직이 들어 있고 DOM 참조는 들어 있지 않은 컨트롤러를 작성하면 된다. 문제는 이벤트 핸들러에 있었던 게 아니라, 기존의 자바스크립트 작성 방식에 있다. 이 책에서 지금껏 작성

했던 모든 컨트롤러 안에는 DOM 참조나 DOM 이벤트가 전혀 들어 있지 않다. 이렇게 DOM을 배제한 채 컨트롤러를 작성하는 일은 간단하다. 요소를 배치하고 이벤트를 처리하는 모든 작업은 Angular 프레임워크 내에서 자동으로 이뤄지기 때문이다.

다만 Angular의 이런 특성 때문에 단위 테스트를 작성할 땐 문제가 된다. DOM이 필요한 경우엔 테스트 안에 DOM 구조를 작성해야 해서 테스트 코드가 복잡해진다. 페이지가 바뀌면 테스트할 DOM도 변경해야 하므로 유지보수할 코드도 늘어난다. 게다가 DOM 접근이 느린데, 이 때문에 테스트가 늦어지면 피드백도 늦어지고 결국 애플리케이션 출시도 늦춰지게 된다. 단위 테스트와는 달리, Angular 컨트롤러 테스트에는 이런 문제점이 전혀 없다.

여기까지 읽느라 수고했다. 이제는 선언식 이벤트 핸들러를 마음 놓고 사용해도 간결성과 가독성은 유지되므로 모범 사례를 위반한다는 불편한 마음은 갖지 않아도 된다.

2.3.4 리스트, 테이블, 각종 반복 요소

Angular 지시어 중에서 가장 많이 사용되는 ng-repeat 지시어는 컬렉션 안의 모든 항목당 한 번씩 요소의 사본을 생성한다. 리스트를 생성해야 할 때는 ng-repeat 지시어를 사용해야 한다.

학생 당번 명부를 표시하는 교사용 애플리케이션을 작성한다고 하자. 서버에서 학생 데이터를 가져와 작성하는 것이 일반적이지만, 여기서는 간단히 학생 데이터를 자바스크립트 모델로 정의하자.

```
var students = [{name:'Mary Contrary', id:'1'},
                {name:'Jack Sprat', id:'2'},
                {name:'Jill Hill', id:'3'}];
```

```
function StudentListController($scope) {
    $scope.students = students;
}
```

이 학생 리스트를 표시하는 코드는 다음과 같이 작성하면 된다.

```
<ul ng-controller='StudentListController'>
    <li ng-repeat='student in students'>
        <a href='/student/view/{{student.id}}'>{{student.name}}</a>
    </li>
</ul>
```

ng-repeat 지시어는 해당 태그와 그 안의 모든 HTML 태그를 복제한다. 웹 브라우저에 표시되는 결과는 그림 2-1과 같다.

그림 2-1 링크가 걸린 당번 학생 리스트(studentsList.html)

- Mary Contrary
- Jack Sprat
- Jill Hill

표시되는 리스트 항목에는 링크가 걸려 있다. 'Mary Contrary'를 클릭하면 루트를 기준으로 /student/view/1로, Jack Sprat을 클릭하면 /student/view/2로, Jill Hill을 클릭하면 /student/view/3으로 이동한다.

앞에서 보았지만 학생 배열만 수정하면 렌더링되는 리스트도 자동으로 바뀐다. 새로운 학생을 리스트에 끼워 넣는 작업을 하려면 자바스크립트를 다음과 같이 작성한다.

```
var students = [{name:'Mary Contrary', id:'1'},
                {name:'Jack Sprat', id:'2'},
                {name:'Jill Hill', id:'3'}];

function StudentListController($scope) {
    $scope.students = students;

    $scope.insertTom = function () {
        $scope.students.splice(1, 0, {name:'Tom Thumb', id:'4'});
    };
}
```

그리고 앞의 컨트롤러를 호출하는 버튼을 템플릿에 추가하는 코드는 다음과 같다.

```
<ul ng-controller='StudentListController'>
    <li ng-repeat='student in students'>
        <a href='/student/view/{{student.id}}'>{{student.name}}</a>
    </li>
    <button ng-click="insertTom()">삽입하기</button>
</ul>
```

웹 브라우저에 표시된 이 코드의 결과 화면에서 [삽입하기] 버튼을 한 번 클릭하면 그림 2-2 같이 Tom Thumb 학생이 리스트에 추가로 표시된다.

그림 2-2 버튼 클릭으로 리스트에 삽입된 당번 학생 항목(studentsList.html)

- Mary Contrary
- Tom Thumb
- Jack Sprat
- Jill Hill

삽입하기

ng-repeat 지시어는 $index와 불boolean값을 통해 현재 요소에 인덱스 참조를 부여하는 기능도 한다. 이때, 불boolean값은 $first를 통해 컬렉션의 첫 번째 요소가 현재 요소인지 아닌지를 나타내고, $middle을 통해 중간 요소 중 하나가 현재 요소인지 아닌지를 나타내며, $last를 통해 마지막 요소가 현재 요소인지 아닌지를 나타낸다.

$index를 테이블 행에 라벨을 표시하는 용도로 사용할 수도 있다. 그럴 경우 템플릿은 다음과 같다.

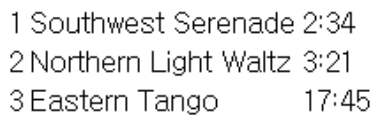
```
<table ng-controller='AlbumController'>
   <tr ng-repeat='track in album'>
      <td>{{$index + 1}}</td>
      <td>{{track.name}}</td>
      <td>{{track.duration}}</td>
   </tr>
</table>
```

그리고 이 템플릿에 필요한 컨트롤러는 다음과 같다.

```
var album = [{name:'Southwest Serenade', duration: '2:34'},
             {name:'Northern Light Waltz', duration: '3:21'},
             {name:'Eastern Tango', duration: '17:45'}];

function AlbumController($scope) {
   $scope.album = album;
}
```

웹 브라우저에 표시되는 결과는 다음과 같다.

그림 2-3 테이블 구조로 표시된 음반의 트랙 라벨(tableWithIndexLabels.html)



2.3.5 감추기와 보이기

메뉴, 상황 인식context-sensitive 도구 등 다양한 부분에서 요소를 보이거나 감추는 기능은 중요한 역할을 한다. Angular의 다른 모든 기능이 다 그렇듯이, UI 변화는 모델 변경을 통해 유도해야 하고 그 변화를 UI에 반영하려면 지시어를 사용해야 한다.

Angular에서 UI에 요소를 보이거나 감추는 지시어로는 ng-show와 ng-hide가 있다. 두 지시어는 전달받은 표현식에 따라 요소를 보이거나 감추는 기능은 같지만 작동은 서로 반대다. 요컨대, ng-show 지시어는 표현식이 true이면 요소를 보여주고 false이면 요소를 감추는 반면, ng-hide 지시어는 표현식이 true이면 요소를 감추고 false이면 요소를 보여준다. 둘 중에서 자신의 의도를 더 분명히 나타낼 수 있는 지시어를 사용하면 된다.

두 지시어는 드러내는 display:block 스타일과 감추는 display:none 스타일을 요소에 적절히 지정해야 한다. SF에 등장하는 살인 광선용 제어 패널을 제작한다는 가정 하에 다음 예제를 살펴보자.

소스 코드 2-4 deathRayMenu.html

```
<div ng-controller='DeathrayMenuController'>
    <button ng-click='toggleMenu()'>메뉴 On / Off</button>
    <ul ng-show='showMenu'>
        <li ng-click='stun()'>기절시키기</li>
        <li ng-click='disintegrate()'>분해하기</li>
        <li ng-click='erase()'>역사에서 지우기</li>
```

```
    </ul>
</div>
function DeathrayMenuController($scope) {
    $scope.showMenu = false;
    $scope.toggleMenu = function() {
        $scope.showMenu = !$scope.showMenu;
    };

    // 여기에 살인 광선의 기능을 연습 삼아 추가해보자.
}
```

2.3.6 CSS 클래스와 스타일

이제까지 살펴본 바와 같이, 애플리케이션에 클래스와 스타일을 동적으로 지정하려면 {{ }} 인터폴레이션 문법을 사용해서 데이터 바인딩을 하면 된다. 템플릿 안에 부분적으로 일치하는 클래스명을 작성할 수도 있다. 예컨대 어떤 메뉴들을 조건에 따라 비활성화하려면 코드를 다음과 같이 작성해서 사용자에게 시각적으로 알릴 수도 있다.

먼저 CSS 코드를 다음과 같이 작성하자.

```
.menu-disabled-true {
    color: gray;
}
```

그러면 살인 광선의 '기절시키기' 기능을 비활성화 상태로 표시해서 stun 함수가 호출될 수 없게 하기 위한 템플릿은 다음과 같이 작성할 수 있다.

```
<div ng-controller='DeathrayMenuController'>
    <ul>
        <li class='menu-disabled-{{isDisabled}}' ng-click='stun()'>기절시키기</li>
        ...
    </ul>
</div>
```

앞의 템플릿에 있는 isDisabled 속성에 대한 값은 다음과 같이 컨트롤러에서 지정하면 된다.

```
function DeathrayMenuController($scope) {
    $scope.isDisabled = false;

    $scope.stun = function() {
        // 대상을 기절시킨 후, 메뉴를 다시 사용할 수 없게 함
        $scope.isDisabled = 'true';
    };
}
```

기절시키기 메뉴 항목에 지정한 class 속성에는 menu-disabled-에 $scope.isDisabled의 값을 덧붙인 문자열이 할당된다. 초기에는 $scope.isDisabled의 값이 false이므로 클래스 속성 값은 menu-disabled-false이다. 그런데 이 클래스 선택자에 대해서는 CSS 스타일 명령을 작성하지 않았으므로 아무런 영향이 미치지 않는다. 그러다가 $scope.isDisabled의 값이 true가 되면 menu-disabled-true 클래스 선택자에 대해 작성한 스타일 명령이 적용되어 텍스트 색상이 회색으로 바뀐다. style="{{표현식}}" 같은 형태로 인터폴레이션과 인라인 스타일을 병용해도 같은 식으로 적용된다.

이 방법은 기발하긴 하지만 클래스명 지정이 표현식 맵핑 단계를 거쳐 간접적으로 이뤄진다는 단점이 있다.[03] 이 예제는 짧아서 과정을 이해하기가 쉽지만, 조금만 복잡해져도 템플릿과 자바스크립트를 이해하고 CSS를 그에 맞게 작성하기가 힘들어진다.

이런 문제점을 해소하기 위해 Angular에는 ng-class 지시어와 ng-style 지시어가 있다. 두 지시어에는 표현식을 지정해야 하며, 다음 중 하나의 값으로 산출된다.

- 빈칸 구분자를 사용한 클래스명 문자열
- 클래스명 배열
- 불값에 대응되는 클래스명 맵

사용자에게 보여줄 에러와 경고를 애플리케이션 헤더의 정상 위치에 표시해야 한다고 가정하자. ng-class 지시어를 사용하면 다음과 같이 작성할 수 있다.

소스 코드 2-5 errorsAndWarnings.html

```
.error {
    background-color: red;
}

.warning {
    background-color: yellow;
}
<div ng-controller='HeaderController'>
    ...
    <div ng-class='{error: isError, warning: isWarning}'>{{messageText}}</div>
    ...
    <button ng-click='showError()'>에러 시연하기</button>
```

03 역자 주_ 직접 클래스명 값을 지정하지 않고 표현식을 지정하였으므로, 표현식을 산출하는 과정이 하나 더 필요하다는 뜻이다.

```
    <button ng-click='showWarning()'>경고 시연하기</button>
</div>
function HeaderController($scope) {
    $scope.isError = false;
    $scope.isWarning = false;

    $scope.showError = function() {
        $scope.messageText = '에러입니다!';
        $scope.isError = true;
        $scope.isWarning = false;
    };

    $scope.showWarning = function() {
        $scope.messageText = '경고에 불과합니다. 계속 진행하세요!';
        $scope.isWarning = true;
        $scope.isError = false;
    };
}
```

테이블에서 선택된 행에 형광펜 효과를 주는 등의 실용적인 기능도 구현할 수 있다. 식당 명부를 만드는 중인데, 사용자가 클릭하는 행에 형광펜 효과를 적용해야 한다고 하자. 그러면 먼저 다음과 같이 CSS에서 형광펜 효과가 적용된 행에 대한 스타일을 지정해야 한다.

```
.selected {
    background-color: lightgreen;
}
```

그런 다음 템플릿 코드에서 ng-class 지시어 값으로 {selected: $index==selected

Row} 표현식을 지정한다. 그러면 모델의 selectedRow 속성이 ng-repeat의 $index와 일치할 때 클래스 속성에 selected가 할당된다. 어느 행을 사용자가 클릭했는지에 대해 컨트롤러에 통보하기 위해 다음과 같이 ng-click 지시어도 지정하자.

```
<table ng-controller='RestaurantTableController'>
    <tr ng-repeat='restaurant in directory' ng-click='selectRestaurant($index)'
    ng-class='{selected: $index==selectedRow}'>
        <td>{{restaurant.name}}</td>
        <td>{{restaurant.cuisine}}</td>
    </tr>
</table>
```

자바스크립트에는 다음과 같이 가상의 식당을 지정하고 selectRow 함수를 작성하자.

소스 코드 2-6 selectedRow.html

```
function RestaurantTableController($scope) {
    $scope.directory = [{name:'푸짐푸짐 암소구이', cuisine:'바비큐'},
                        {name:'미스터 그린의 청정 샐러드', cuisine:'샐러드'},
                        {name:'파인 피시 하우스', cuisine:'해산물'}];

    $scope.selectRestaurant = function(row) {
        $scope.selectedRow = row;
    };
}
```

2.3.7 src 속성과 href 속성을 사용할 때 주의할 점

<img> 태그나 <a> 태그에 데이터를 바인딩할 때, src 속성이나 href 속성에는 {{ }}

문법이 제대로 동작하지 않는다. 웹 브라우저가 공격적으로 이미지를 다른 콘텐츠와 번갈아 가며 로딩하므로 Angular가 데이터 바인딩 요청을 낚아챌 겨를이 없기 때문이다. 〈img〉 태그의 표준 문법대로 작성한다면 다음과 같을 것이다.

```
<img src="/images/cats/{{favoriteCat}}">
```

그러나 앞에 설명한 이유 때문에 다음과 같이 src 속성 대신 ng-src 속성을 사용해야 한다.

```
<img ng-src="/images/cats/{{favoriteCat}}">
```

마찬가지로 〈a〉 태그도 다음과 같이 href 속성 대신 ng-href 속성을 사용해야 한다.

```
<a ng-href="/shop/category={{numberOfBalloons}}">이러쿵저러쿵</a>
```

2.3.8 표현식

템플릿에 표현식을 사용하는 목적은 두 가지다. 첫째, 표현식을 사용하면 템플릿, 로직, 데이터 간에 연결고리를 만들 수 있다. 둘째, 표현식을 사용하면 애플리케이션 로직이 템플릿을 엿볼 수 없다.

지금까지 Angular 지시어에 전달된 표현식을 주로 데이터 원시 타입 참조에 사용해왔다. 그러나 표현식에는 그 외에도 훨씬 다양한 기능이 있다. 단순 계산(+, -, /, *, %), 비교(==, !=, 〉, 〈, 〉=, 〈=), 논리 연산(&&, ||, !), 비트 연산(\^, &, |)을 수행할 수도 있다. 컨트롤러 안의 $scope에 지정한 함수를 호출할 수도 있고, 배열과

객체 표기([ ], { }, .)를 참조할 수도 있다.

다음 코드는 표현식의 다양한 기능을 보여준다.

```
<div ng-controller='SomeController'>
   <div>{{recompute() / 10}}</div>
   <ul ng-repeat='thing in things'>
      <li ng-class='{highlight: $index % 4 >= threshold($index)}'>
         {{otherFunction($index)}}
      </li>
   </ul>
</div>
```

앞의 코드에서 첫 번째 표현식 'recompute() / 10'은 문법은 틀리지 않지만 이런 식으로 템플릿에 로직을 삽입하는 것은 바람직하지 않으므로 사용하지 않는 것이 좋다. 뷰와 컨트롤러의 역할을 분리해야 왜 그렇게 했는지 이해하기도 쉽고 테스트하기도 쉽다.

Angular 표현식으로도 상당히 많은 것을 할 수는 있지만, 자바스크립트의 eval() 함수를 통해 산출되는 것이 아니라 Angular에 들어 있는 사용자 정의 파서를 통해 산출되므로 훨씬 제한적이다.

Angular의 사용자 정의 파서에는 for문이나 while문 등의 루프 구조체, if-else나 throw 같은 제어순서 연산자, ++나 -같은 데이터 증감 연산자가 들어 있지 않다. 따라서 이런 종류의 연산자가 필요한 작업은 컨트롤러 안에서 처리하거나 지시어를 사용해야 한다.

표현식이 자바스크립트에 비해 여러모로 제약은 많지만, undefined와 null에 대한 융통성은 더 높다. NullPointerException 에러를 통지하는 대신 템플릿은 그

냥 아무 것도 렌더링하지 않는다. 이런 장점 때문에 개발자는 아직 지정하지 않은 모델 값을 안심하고 사용할 수 있으며, 값이 입력되는 시점에 즉시 UI에 그 값이 표시된다.

2.3.9 UI와 컨트롤러의 기능을 분리

컨트롤러가 애플리케이션에서 하는 역할은 다음과 같이 세 가지다.

- 애플리케이션의 모델 안에 초기 상태를 설정한다.
- $scope를 통해 모델과 함수를 뷰(UI 템플릿)에 공개한다.
- 모델의 다른 부분이 변경되거나 어떤 조치를 취하는지를 감시한다.

두 절에 걸쳐서 이미 많은 예제를 살펴보았다. 마지막 예제는 잠시 후에 살펴보겠다. 어쨌든 컨트롤러의 개념적인 목표는 사용자들이 뷰를 조작할 때 그들의 요구를 실행할 코드나 로직을 제공하는 것이다.

컨트롤러를 간결하고 관리하기 쉽게 유지하려면, 뷰 안에 있는 기능 영역마다 컨트롤러를 하나씩 작성하는 것이 좋다. 뷰 안에 메뉴가 있다면 MenuController를 작성하고 탐색 가능한 브레드크럼이 있다면 BreadCrumbController를 작성하는 것이다.

지금까지의 설명으로 감은 잡았겠지만 더 확실히 이해할 수 있도록 설명하자면, 컨트롤러는 자신이 관리하는 DOM의 특정 부분과 연결되어 있다. 컨트롤러를 DOM 노드와 연결하는 주된 방법은 두 가지다. 첫 번째 방법은 ng-controller 속성에 컨트롤러를 선언해서 템플릿 안에 지정하는 것이다. 두 번째 방법은 루트를 통해 동적 로딩이 가능한 DOM 템플릿 조각(소위 뷰)에 컨트롤러를 연결하는 것이다.

뷰와 루트에 대해서는 이 장의 뒷부분을 참고하면 된다.

UI를 구성하는 구획이 복잡해도 코드를 간결하고 유지보수하기 쉽게 만드는 방법

은 다음과 같다. 우선 컨트롤러 안에 다른 컨트롤러를 넣는 식으로 포함관계를 띤 겹 컨트롤러를 만든다. 그리고 상속 트리를 통해 모델과 함수를 공유할 수 있게 하면 된다. 이렇게 컨트롤러를 겹겹이 계층화하는 것은 쉽다. 그냥 DOM 요소에 컨트롤러를 지정하고 그 안에 있는 DOM 요소에 다른 컨트롤러를 지정하면 된다.

```
<div ng-controller="ParentController">
   <div ng-controller="ChildController">...</div>
</div>
```

겹 컨트롤러라는 표현과 달리 실제로는 컨트롤러가 겹치는 게 아니라 스코프가 겹친다. 겹 컨트롤러로 전달되는 $scope는 부모 컨트롤러의 $scope에서 프로토타입을 통해 상속받는다. 앞의 코드를 예로 들면, ChildController로 전달되는 $scope는 ParentController로 전달되는 $scope의 모든 속성에 접근할 수 있는 것이다.

2.3.10 스코프 객체를 사용해 모델 데이터를 발행

컨트롤러로 전달되는 $scope 객체는 모델 데이터를 뷰에 공개할 때 사용된다. 애플리케이션 안에 다른 데이터가 있을 수도 있는데, Angular는 스코프를 통해 이 절에서 설명한 속성들에 접근할 수 있다면 그 데이터를 모델의 일부로 간주한다. 스코프는 모델 수정을 위해 참조하는 식별 가능한 콘텍스트(범위, 환경)라고 생각하면 이해하기 쉽다.

앞에서 살펴본 대부분의 예제에서는 스코프를 '$scope.count = 5' 같이 직접적으로 지정했는데, 템플릿 자체에서 모델을 설정하는 간접적인 스코프 지정 방식도 있다. 스코프를 간접적으로 지정하는 방법은 다음과 같다.

1. 표현식을 통한 방법 - 표현식은 해당 요소와 연결된 컨트롤러의 스코프 범위 안

에서 실행되므로, 표현식 안에서 속성을 지정하는 방식이든 컨트롤러 스코프의 속성을 지정하는 방식이든 마찬가지다. 예를 들면 다음과 같다.

```
<button ng-click='count=3'>count에 3을 할당</button>
```

앞의 코드는 다음과 같이 작성해도 결과는 같다.

```
<div ng-controller='CountController'>
    <button ng-click='setCount()'>count에 3을 할당</button>
</div>
```

이 템플릿과 함께 CountController를 다음과 같이 정의해야 한다.

```
function CountController($scope) {
    $scope.setCount = function() {
        $scope.count = 3;
    }
}
```

2. 폼 인풋에 ng-model 지시어를 사용하는 방법 - 표현식을 사용하는 방법과 마찬가지로 ng-model 지시어에 모델을 지정하는 방법도 상위 요소에 있는 컨트롤러의 스코프 내에서만 유효하다. 다만, 이 방법을 사용하면 폼 필드 상태와 지정한 모델 간에 양방향 데이터 바인딩 관계가 형성된다.

2.3.11 $watch를 사용해 모델 변경사항을 관찰

쓰임새가 제일 많은 스코프 함수는 $watch이다. $watch 함수는 모델의 일부가 변경되면 그 사실을 통지한다. 객체의 각 속성과 함수의 산출 결과뿐 아니라 자바스크립트 함수로 산출되거나 속성을 통해 접근할 수 있는 거의 모든 것을 감시할

수 있다. $watch 함수의 시그너처는 다음과 같다.

```
$watch(watchFn, watchAction, deepWatch)
```

각 매개변수에 대한 설명은 다음과 같다.

watchFn

이 매개변수는 감시하려는 모델의 현재 값을 반환하는 Angular 함수나 표현식을 나타내는 문자열이다. 이 표현식은 상태가 변경되지 않아도 여러 번 호출될 때마다 산출 작업이 수행될 수밖에 없기 때문에 오류가 있는지 반드시 확인해야 하며, 감시하는 표현식이 리소스 소모가 적도록 간결해야 한다. Angular 표현식에 문자열을 전달했다면 그 문자열은 호출이 행해진 스코프 내에서 사용 가능한 객체들을 이용해 산출된다.

watchAction

이 매개변수는 watchFn이 변경되면 호출되는 함수나 표현식이다. 이 함수는 형태상 watchFn의 새 값과 기존 값을 받으며 스코프 참조도 받는다. 이 함수의 시그너처는 function(newValue, oldValue, scope)이다.

deepWatch

이 매개변수는 옵션이며 불값을 나타낸다. 이 매개변수에 true를 전달하면 Angular는 감시 대상 객체 안의 모든 속성 중 하나라도 변경됐는지 검사한다. 단순 값이 아닌 배열의 원소나 객체의 속성을 낱낱이 감시해야 할 때 이렇게 true를 전달하면 된다. 다만, Angular가 배열이나 객체를 일일이 탐색해야 하므로 원소나 속성이 많으면 리소스 소모가 크다.

$watch 함수는 변경에 대한 통지를 더 이상 받지 않게 되면, 등록된 리스너를 해

제하는 함수를 반환한다.

어떤 속성의 변경을 감시한 후 등록된 리스너를 해제하려면 다음과 같이 작성하면 된다.

```
...
var dereg = $scope.$watch('someModel.someProperty', callbackOnChange());
...
dereg();
```

1장에서 살펴본 미완성 장바구니 예제를 지금 완성해보자. 고객이 장바구니에 담은 상품의 전체 금액이 $100 이상이면 $10 할인을 적용해야 한다고 가정하자. 그럼 템플릿은 다음과 같이 작성하면 된다.

```
<div ng-controller="CartController">
   <div ng-repeat="item in items">
      <span>{{item.title}}</span>
      <input ng-model="item.quantity" />
      <span>{{item.price | currency}}</span>
      <span>{{item.price * item.quantity | currency}}</span>
   </div>
   <div>전체 금액: {{totalCart() | currency}}</div>
   <div>할인: {{bill.discount | currency}}</div>
   <div>결제 금액: {{subtotal() | currency}}</div>
</div>
```

그리고 CartController 컨트롤러 함수는 다음과 같이 작성하면 된다.

소스 코드 2-7 orderDiscountWatch.html

```
function CartController($scope) {
    $scope.bill = {};

    $scope.items = [{ title : '페인트 그릇', quantity : 8, price : 3.95 },
                    { title : '땡땡이 리본', quantity : 17, price : 12.95 },
                    { title : '공깃돌', quantity : 5, price : 6.95 }];

    $scope.totalCart = function() {
        var total = 0;
        for ( var i = 0, len = $scope.items.length; i < len; i++) {
            total = total + $scope.items[i].price * $scope.items[i].quantity;
        }

        return total;
    }

    $scope.subtotal = function() {
        return $scope.totalCart() - $scope.bill.discount;
    };

    function calculateDiscount(newValue, oldValue, scope) {
        $scope.bill.discount = newValue > 100 ? 10 : 0;
    }

    $scope.$watch($scope.totalCart, calculateDiscount);
}
```

CartController 함수의 맨 밑에 $watch 함수를 사용해서 totalCart() 함수의 값이 변하는지 감시하게끔 지정했다. totalCart()는 이 구매의 전체 금액을 합산하는 함수다. 이 함수의 반환 값이 변하면 $watch 함수는 calculateDiscount() 함수

를 호출하게 되므로 의도대로 할인이 적용된다. 전체 금액이 $100 이상이면 $10 할인액이 할당되고 그렇지 않으면 $0 할인액이 할당된다.

할인 기능을 추가한 장바구니 템플릿을 웹 브라우저에서 열면 그림 2-4와 같은 화면이 표시된다.

그림 2-4 할인이 적용된 장바구니(orderDiscountWatch.html)

2.3.12 watch() 함수의 성능 고려사항

앞의 예제는 정확하게 실행되지만 성능 저하 문제가 일어날 가능성이 있다. 디버거에서 totalCart() 함수에 중단점을 삽입하면, 페이지를 렌더링하기까지 totalCart() 함수가 여섯 번이나 호출된다. 이 애플리케이션은 간단해서 성능 차이가 느껴지진 않지만 복잡한 애플리케이션이라면 어떤 함수를 불필요하게 여섯 번이나 실행한다는 것은 문제다.

왜 하필 여섯 번인가? 이 중에서 3회는 다음 항목마다 한 번씩 실행되므로 쉽게 찾아낼 수 있다.

- 템플릿의 {{totalCart() | currency}} 부분
- subtotal() 함수
- $watch() 함수

Angular는 앞의 3회 실행을 전부 반복하므로, 총 6회가 실행되는 것이된다. Angular

가 이렇게 하는 이유는 모델에 발생하는 중간적인 변화[04]가 끝나 모델이 고착됐는지 확인하기 위해서다. Angular는 이것을 확인하기 위해 감시 대상 속성을 전부 복사해서 현재 값과 비교하여, 변화가 있는지 검사한다. 사실상 Angular는 변화가 완전히 끝났음을 확인하기 위한 이러한 전이적 변화 검사를 최대 열 번까지밖에 실행하지 못 한다. 즉, 중간의 변화가 10회를 초과해 일어나면 Angular는 에러를 표시하고 종료된다. 만약 이런 상황이 벌어진다면, 수정해야 할 종속물 루프가 있을 가능성이 있다.

지금 당장은 이것이 걱정거리겠지만, 이 책을 다 읽었을 때쯤이면 아무 문제가 아닐 수도 있다. Angular는 이제까지 자바스크립트로 데이터 바인딩을 구현해야 했지만, 필자는 TC39 위원회와 함께 저수준 네이티브 라이브러리인 Object.observe() 개발에 착수했다. 이것만 완성되면 Angular는 Object.observe()를 자동으로 사용하게 되며, 어디서든 네이티브 언어 속도의 데이터 바인딩이 가능해진다.

다음 장에서 설명하겠지만 Angular에는 Batarang('바터랭'이라고 읽는다)이라는 훌륭한 Chrome 디버깅 확장 기능이 있는데, Batarang은 리소스 소모가 큰 데이터 바인딩 부분에 자동으로 형광펜 표시를 넣는다.

이 문제에 대한 해결책은 두 가지다. 하나는 items 배열에 발생하는 변경을 감시하고 $scope에 있는 속성들로 전체 금액, 할인, 결제 금액을 재계산하는 $watch 함수를 작성하는 것이다.

그러기 위해서는 다음과 같이 이러한 속성들을 사용하도록 템플릿을 수정해야 한다.

```
<div>전체 금액: {{bill.total | currency}}</div>
<div>할인: {{bill.discount | currency}}</div>
<div>결제 금액: {{bill.subtotal | currency}}</div>
```

04 역자 주_ 중간중간의 변화단계가 완전히 마무리되어, 모델이 안정된 상태가 되었다는 의미다.

그런 다음 자바스크립트 부분의 코드를 다음과 같이 작성하자. 먼저 컨트롤러 함수 안에 items 배열을 감시하고, 배열에 변경이 생기면 전체 금액 계산 함수를 호출하게 하는 것이다.

소스 코드 2-8 orderDiscountWatch-corrected.html

```
function CartController($scope) {
    $scope.bill = {};

    $scope.items = [{ title : '페인트 그릇', quantity : 8, price : 3.95 },
                    { title : '땡땡이 리본', quantity : 17, price : 12.95 },
                    { title : '공깃돌', quantity : 5, price : 6.95 }];

    var calculateTotals = function() {
        var total = 0;
        for ( var i = 0, len = $scope.items.length; i < len; i++) {
            total = total + $scope.items[i].price * $scope.items[i].quantity;
        }
        $scope.bill.totalCart = total;
        $scope.bill.discount = total > 100 ? 10 : 0;
        $scope.bill.subtotal = total - $scope.bill.discount;
    };

    $scope.$watch('items', calculateTotals, true);
}
```

앞의 코드에서 보듯이 $watch 함수에는 items 배열을 문자열로 전달했다. 이것이 가능한 이유는 $watch 함수가 인자로 함수와 문자열을 받을 수 있기 때문이다. 만약 문자열을 전달하면 $watch 함수의 결과는 $watch 함수가 호출된 $scope의 유효범위 내에 있는 표현식을 통해 산출될 것이다.

이 방법은 애플리케이션에 잘 들어맞을 수도 있다. 그러나 items 배열을 감시하는 것이므로 Angular는 개발자를 대신해서 비교할 수 있는 items 배열의 사본을 만들어야 한다. 구성 원소가 많아 items 배열이 길 때는 Angular가 페이지를 산출할 때마다 bill의 속성만 재계산하는 편이 성능면에서 유리할지도 모른다. 그렇게 하려면 다음과 같이 속성들을 재계산하는 watchFn만을 매개변수로 받는 $watch 함수를 작성해야 한다.

소스 코드 2-9 orderDiscountWatch-corrected2.html

```
$scope.$watch(function() {
    var total = 0;
    for ( var i = 0; i < $scope.items.length; i++) {
        total = total + $scope.items[i].price * $scope.items[i].quantity;
    }
    $scope.bill.totalCart = total;
    $scope.bill.discount = total > 100 ? 10 : 0;
    $scope.bill.subtotal = total - $scope.bill.discount;
});
```

■ 여러 대상을 감시하기

여러 개의 속성이나 객체를 감시해서 변경이 있을 때마다 한 함수를 실행해야 한다면 어떤 방법이 있을까? 기본적인 방법은 두 가지다.

- 대상들을 배열이나 객체로 만들고 deepWatch 매개변수에 true를 전달한다.
- 속성들의 값을 접합해서 그 접합된 값을 감시한다.

스코프 내에 a와 b라는 두 속성이 든 하나의 객체가 있는데 두 속성을 감시해서 변경이 발생하면 callMe() 함수를 실행해야 한다고 가정하자. 그러면 첫 번째 방법을 사용하면 다음과 같이 작성할 수 있다.

```
$scope.$watch('things.a + things.b', callMe(...));
```

물론 a 속성과 b 속성은 서로 다른 두 객체에 있어도 상관없고, 속성이 몇 개여도 상관없다. 감시할 속성이 많을 때는 로직을 표현식으로 구현할 게 아니라 접합된 값을 반환하는 함수를 작성하는 편이 나을 것이다.

두 번째 방법은 things 객체에 든 모든 속성을 감시해야 한다고 할 때 다음과 같이 작성하면 된다.

```
$scope.$watch('things', callMe(...), true);
```

앞의 코드에서 세 번째 인자로 전달한 true는 객체 내의 모든 속성을 낱낱이 검사해서 그 중 하나라도 변경되면 callMe() 함수를 호출하라고 Angular에게 명령한다.

2.4 모듈을 사용해 종속물 체계화하기

조금이라도 복잡한 애플리케이션에서는 코드의 기능을 여러 책임 영역으로 체계화할 방법을 알아내기가 어렵다. 적절한 데이터와 함수에 접근 가능하게 하는 코드는 컨트롤러를 사용해서 뷰 템플릿의 어디에 어떻게 넣으면 되는지 앞서 살펴보았다. 하지만 개발자가 애플리케이션을 보조하는 데 필요한 나머지 코드는 어디에 넣을 것인가? 가장 확실한 위치는 컨트롤러에 든 함수의 내부일 것이다.

컨트롤러에 든 함수 안에 넣는 방식은 간단한 애플리케이션과 앞서 살펴본 예제들에는 알맞지만 실무 애플리케이션에서는 금세 관리할 수 없게 된다. 결국 컨트롤러는 개발자에게 필요한 온갖 기능이 버려지는 쓰레기 처리장이 될 것이므로 알아보기 힘들고 수정하기도 어려울 가능성이 높다.

모듈을 사용할 때가 왔다. 애플리케이션 안에서 여러 종속물을 모듈이라는 하나의

기능 영역으로 묶을 수 있고, 그 종속 모듈을 '종속물 주입'이라는 패턴을 사용해 자동으로 가져올 수 있다. 애플리케이션에 특화된 서비스들을 제공한다는 뜻에서 보통은 이러한 종속물을 서비스라고 부른다.

예컨대, 쇼핑 웹 사이트에서 컨트롤러가 서버에서 판매 상품 목록을 가져와야 한다면 서버에서 상품을 가져오는 기능의 객체가 필요하다. 이 객체를 'Items'라고 하자. 이 Items 객체에는 또 XHR이나 웹소켓을 통해 서버의 데이터베이스와 통신하는 기능도 필요하다.

모듈을 사용하지 않고 이러한 기능을 구현하려면 다음과 같이 작성해야 할 것이다.

```
function ItemsViewController($scope) {
    // 서버에 요청함
    ...

    // 응답을 Item 객체로 파싱함
    ...

    // $scope의 Items 배열을 설정해서 뷰가 표시할 수 있게 함
    ...
}
```

이렇게 해도 의도한 결과는 얻을 수 있지만 다음과 같은 문제점이 있다.

- 서버에서 Items 객체를 가져와야 하는 또 다른 컨트롤러가 있을 경우엔 이 코드를 복사해야 한다. 그렇게 되면 유지보수가 부담스러워진다. 스키마나 다른 것을 수정할 때마다 똑같은 코드를 몇 군데에서 일일이 수정해야 하기 때문이다.
- 서버 인증, 파싱의 복잡도 등과 같은 다양한 요인으로 인해 컨트롤러 객체의 기능 경계를 미루어 짐작하기 어렵고 코드를 이해하기란 더욱 어렵다.

- 이런 코드를 단위 테스트하려면 실제로 서버를 구동하든지 모크mock(또는 '목'이라고 읽음) 데이터를 반환하게끔 XMLHttpRequest를 '멍키 패칭monkey patching'[05]해야 한다. 서버를 구동해야 하기 때문에 테스트는 매우 느려지고 설정하기가 까다롭고 테스트에 깨짐이 자주 발생한다. 루트를 멍키 패치하면 속도와 깨짐의 문제가 동시에 해결되지만, 테스트와 테스트 중간에 패치한 객체는 반드시 언패치해야 하며 데이터에 정확한 네트워크 전송 가능 형식을 지정해야 하므로 더욱 복잡하고 불안정해진다. 게다가 이 형식이 변경될 때마다 테스트를 수정해야 한다.

모듈과 모듈에서 얻어지는 종속물 주입을 이용하면 다음과 같이 컨트롤러 작성이 훨씬 더 간단해진다.

```
function ShoppingController($scope, Items) {
    $scope.items = Items.query();
}
```

그렇다면 Items 객체는 어디서 가져올까? 당연히 앞의 코드에 필요한 Items 객체를 서비스로 정의해야 한다.

서비스는 애플리케이션의 기능을 보조하는 데 필요한 작업들을 수행하는 싱글턴singleton(단일 인스턴스) 객체다. Angular에는 웹 브라우저의 위치를 주고받는 $location, 위치(URL) 변경에 따라 뷰를 전환하는 $route, 서버와 통신하는 $http 등 다양한 서비스가 있다.

자신의 애플리케이션에만 특화된 모든 작업을 수행하는 '개발자 정의 서비스'를 작성할 수도 있다. 필요하다면 서비스는 어느 컨트롤러든 서로 공유할 수 있다. 그러

05 역자 주_ 원본 소스 코드를 변경하지 않고 동적 언어의 런타임 코드를 확장하거나 수정하는 방법이다.

므로 컨트롤러끼리 송수신해서 상태를 공유해야 할 때는 서비스를 사용하는 것이 적절하다. 개발자가 작성하는 서비스의 이름은 마음대로 정해도 되지만, Angular에 내장된 서비스는 앞에 $가 붙어 있으므로 내장 서비스와 충돌하는 것을 방지하려면 서비스 이름 앞에 $를 붙이지 않는 것이 좋다.

서비스는 모듈 객체의 API를 사용해서 정의한다. 범용 서비스를 작성할 수 있는 함수는 다음 표에 정리한 것처럼 세 가지가 있는데 복잡도와 기능이 각기 다르다.

함수명	정의
provider(이름, 객체 또는 생성자())	설정 가능한 서비스며 생성 로직이 복잡하다. 객체를 전달하려면 객체에는 해당 서비스의 인스턴스를 반환하는 $get이라는 이름의 함수가 있어야 한다. 객체를 전달하지 않으려면, Angular가 호출되는 순간 인스턴스를 생성하는 생성자 함수를 전달해야 한다.
factory(이름, $읽기함수())	설정 불가능한 서비스며 생성 로직이 복잡하다. 해당 서비스의 인스턴스를 반환하는 함수를 지정해야 한다. 이것은 프로바이더(이름, { $get: $읽기함수() })와 같다고 생각해도 된다.
service(이름, 생성자())	설정 불가능한 서비스며 생성 로직이 간단하다. 프로바이더에 생성자를 전달하는 방식과 마찬가지로 Angular는 이 생성자 함수를 호출해서 서비스 인스턴스를 생성한다.

provider() 객체를 사용하는 방식의 설정 옵션에 대해서는 나중에 설명하겠다. 일단 지금은 앞의 Items 예제에 factory()를 사용하는 예를 살펴보자.

```
// 쇼핑 뷰에 사용할 모듈을 선언
var shoppingModule = angular.module('ShoppingModule', []);

// 서버 데이터베이스에 Items 인터페이스를 생성하는 서비스 팩토리를 설정
shoppingModule.factory('Items', function() {
```

```
    var items = {};
    items.query = function() {
        // 실무 애플리케이션이었다면, 이 데이터를 서버에서 받아오게 했겠지만,
        // 이건 예제이므로.간단히 지정한 값을 반환하게 했다.
        return [
            {
                title : '페인트 그릇',
                description : '페인트를 담아 쓸 수 있는 용기',
                price : 3.95
            }, {
                title : '땡땡이 리본',
                description : '물방울 무늬가 자수된 리본',
                price : 2.95
            }, {
                title : '공깃돌',
                description : '공기 놀이를 위한 오색 조약돌',
                price : 6.95
            }
        ];
    };
    return items;
});
```

Angular가 ShoppingController를 생성하면, ShoppingController는 $scope와 앞의 코드에 정의한 새 Items 서비스를 전달한다. 이것은 매개변수 이름을 대조함으로써 이뤄진다. Angular는 ShoppingController 클래스의 함수 시그너처를 보고 클래스가 Items 객체를 요청하는지 판단한다. 앞에서 Items 객체를 서비스로 정의했으므로 ShoppingController 클래스는 Items 객체가 있는 위치를 안다.

이러한 종속물을 문자열로 찾아보면 컨트롤러 생성자 등 함수에 전달할 수 있는 인자는 순서에 무관함을 알 수 있다.

```
function ShoppingController($scope, Items) {...}
```

따라서 앞의 코드는 다음과 같이 작성해도 상관없다.

```
function ShoppingController(Items, $scope) {...}
```

매개변수 순서가 바뀌었지만 결과는 같다.

앞에서 정의한 모듈을 템플릿에 적용하려면 다음과 같이 ng-app 지시어를 사용해서 그 모듈의 이름을 지정하면 된다.

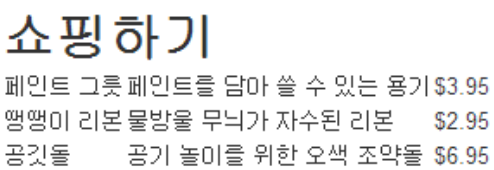
```
<html ng-app='ShoppingModule'>
```

예제를 마무리하기 위해 나머지 템플릿을 다음과 같이 작성하자.

```
<body ng-controller="ShoppingController">
    <h1>쇼핑하기</h1>
    <table>
        <tr ng-repeat="item in items">
            <td>{{item.title}}</td>
            <td>{{item.description}}</td>
            <td>{{item.price | currency}}</td>
        </tr>
    </table>
</body>
```

웹 브라우저에서 이 애플리케이션은 그림 2-5와 같이 표시된다.

그림 2-5 쇼핑 품목



2.4.1 나에게 필요한 모듈의 수

서비스 자체는 종속물을 가질 수 있기에 개발자는 Module API를 사용하여 필요한 종속물을 정의할 수 있다.

애플리케이션 대부분은 자신이 작성한 코드 전체에 필요한 모듈을 간단히 하나만 작성하고 그 안에 종속물을 전부 집어넣어도 괜찮다. 타사 라이브러리의 서비스나 지시어를 사용할 때는 서비스나 지시어에 필요한 모듈이 라이브러리에 함께 들어 있기 마련이므로 그걸 사용하면 된다. 애플리케이션이 타 라이브러리의 서비스나 지시어에 의존할 경우, 그 서비스나 지시어를 애플리케이션 모듈의 종속물이라고 한다.

예컨대, 전달하는 인자에 모듈 SnazzyUIWidgets와 SuperDataSync를 포함시키려면 애플리케이션의 모듈은 다음과 같이 선언하면 된다.

```
var appMod = angular.module('app', ['SnazzyUIWidgets', 'SuperDataSync']);
```

2.5 필터를 사용해 데이터를 형식화하기

필터를 사용하면 템플릿 인터폴레이션 안에 사용자에게 표시할 데이터의 변환 방식을 선언할 수 있다. 필터 사용 문법은 다음과 같다.

```
{{ 표현식 | 필터명 : 매개변수1 : ...매개변수N }}
```

여기서 표현식 자리에는 Angular 표현식 아무 것이나 넣을 수 있고, 필터명 자리에는 사용할 필터의 이름을 넣으면 된다. 그리고 매개변수 자리에는 콜론을 구분자로 사용해서 필터에 지정할 매개변수를 나열하면 된다. 매개변수 자체는 올바른 Angular 표현식이면 어느 것이든 사용할 수 있다.

Angular에는 앞서 살펴보았던 currency를 비롯해 몇몇 필터가 들어 있다.

```
{{12.9 | currency}}
```

앞의 코드는 웹 브라우저에서 다음과 같은 문자열로 표시된다.

```
$12.90
```

이 선언문은 컨트롤러나 모델에 넣는 것이 아니라 뷰에 넣어야 한다. 왜냐하면 숫자 앞의 달러 기호는 인간이 보고 이해하는 데에만 소용이 있을 뿐 숫자 처리에 사용되는 로직에는 필요가 없기 때문이다. Angular에 내장된 필터는 date, number, uppercase 등 다양하다. 필터 뒤에 파이프 기호를 사용해서 다른 필터 바인딩을 덧붙일 수도 있다. 예를 들면, 앞의 예제를 소수점 뒤에 자릿수가 없게끔 형식화하려면 다음과 같이 number 필터를 덧붙이면 된다. number 필터는 소수점 수를 받아서 매개변수 자리까지 반올림한다.

```
{{12.9 | currency | number:0 }}
```

앞의 코드로 표시되는 결과는 다음과 같다.

```
$13
```

Angular 내장 필터만 사용할 수 있는 것은 아니다. 개발자 정의 필터도 간단히 작성해서 쓸 수 있다. 그 예로, 제목 요소의 문자열에 제목 대소문자 표기법[06]을 적용하는 필터는 다음과 같이 작성하면 된다.

소스 코드 2-10 titleCaseFilter.html

```
var homeModule = angular.module('HomeModule', []);
homeModule.filter('titleCase', function() {
    var titleCaseFilter = function(input) {
        var words = input.split(' ');
        for ( var i = 0; i < words.length; i++) {
            words[i] = words[i].charAt(0).toUpperCase() + words[i].slice(1);
        }
        return words.join(' ');
    };
    return titleCaseFilter;
});
```

```
<body ng-app='HomeModule' ng-controller="HomeController">
    <h1>{{pageHeading | titleCase}}</h1>
</body>
```

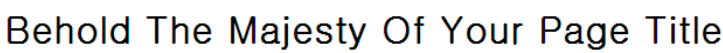
그리고 다음과 같이 컨트롤러를 통해 pageHeading을 모델 변수로 삽입하자.

06 역자 주_ Title-Case(제목 대소문자 표기법) : 보통 제목이나 간판에 많이 쓰이는 형식처럼 영문 문자열의 각 단어 첫 글자를 대문자로 표기하는 방식이다.
예) Education News And University Jobs
Camel-Case(낙타 대소문자 표기법) : 보통 프로그래밍 언어의 함수명에 많이 쓰이는 형식으로, 붙여 쓰는 영문구에서 맨 앞 글자를 제외하고 각 의미의 최소단위나 단어의 첫 글자를 대문자로 표기하는 방식이다.
예) createNewImage

```
function HomeController($scope) {
    $scope.pageHeading = 'behold the majesty of your page title';
}
```

그러면 결과는 웹 브라우저에 그림 2-6과 같이 표시된다.

그림 2-6 titleCase 필터를 지정해서 제목 대소문자 표기법을 적용한 모습



2.6 라우트 서비스와 $location을 사용해 뷰를 변경하기

Ajax 애플리케이션은 최초의 요청 시에만 HTML 페이지를 로딩하고 그 후의 요청에는 DOM 내부의 영역을 업데이트할 뿐이라는 점에서 엄밀히 따지면 싱글 페이지 애플리케이션이지만, 보통은 상황에 따라 사용자에게 보이거나 감추는 여러 개의 하위 페이지 뷰가 있다.

이렇게 상황에 따라 보이거나 감추는 작업을 Angular의 $route 서비스를 사용하여 자동으로 처리할 수 있다. 루트를 사용하면, 웹 브라우저가 이동할 URL에 따라 Angular가 템플릿을 로딩해서 표시하고 템플릿에 콘텍스트를 제공하는 컨트롤러의 인스턴스를 생성하게끔 지정할 수 있다.

애플리케이션에 루트를 작성하려면 다음의 의사 코드에서 보듯이 $routeProvider 서비스에서 설정 블록을 통해 함수를 호출하면 된다.

```
var someModule = angular.module('someModule', [...모듈 종속물 배열...])
someModule.config(function($routeProvider) {
    $routeProvider.
        when('url', {controller:aController, templateUrl:'/path/to/template'}).
```

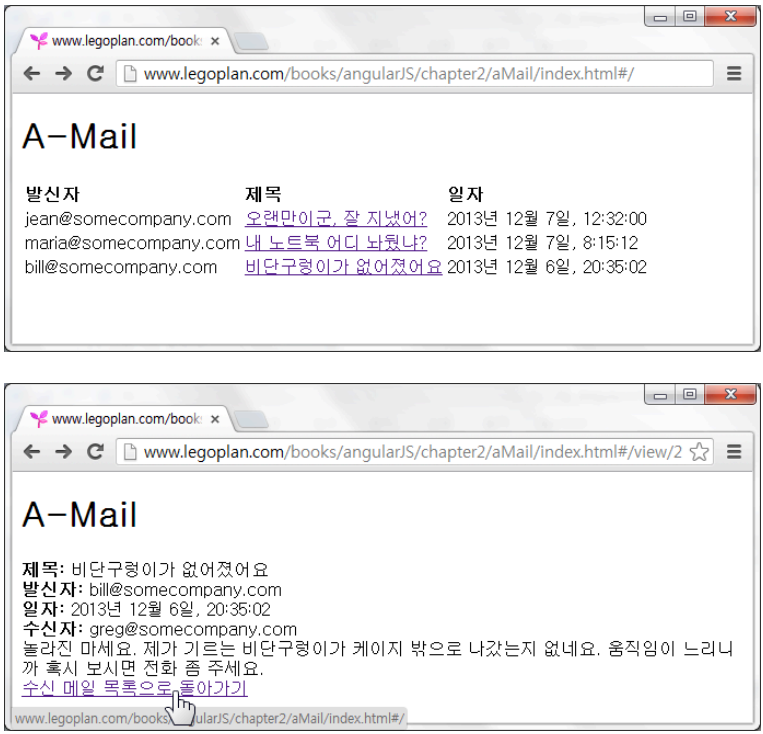
```
        when(...애플리케이션에 필요한 각종 매핑 및 지정...).
        ...
        otherwise(...그 외의 경로일 때 이동할 경로를 지정...);
)};
```

앞의 코드는 웹 브라우저의 URL이 특정 URL로 변경되면 Angular에 /path/to/template 경로에 있는 템플릿을 로딩하고 그 템플릿의 루트 요소를 aController 컨트롤러와 연결하도록 명령한다. 결과적으로 ng-controller=aController라고 지정한 것과 같다.

마지막 줄의 otherwise() 함수 호출 명령은 when 함수에서 대조하는 경로 이외의 경로일 때 지정한 경로로 이동할 것을 명령한다. 그럼 실제 예제에 이 코드를 활용하여 지메일이나 핫메일 정도는 거뜬히 능가할 이메일 애플리케이션을 만들어보자. 제작할 애플리케이션의 이름을 A-Mail이라고 하고 시작은 간단하게 하자. 먼저 날짜, 제목, 발신자와 함께 이메일 목록이 표시될 메인 뷰가 있어야 한다. 그리고 이메일 하나를 클릭하면 메일 본문이 표시되는 새로운 뷰가 하나 필요하다.

NOTE_ 웹 브라우저는 보안이 제대로 지원되지 않기에 자신이 손수 코드를 작성해보려면 로컬(file://)이 아니라 웹 서버에 올려서 테스트해야 한다. 파이썬이 설치되어 있다면 작업 중인 디렉터리에서 python -m SimpleHTTPServer 8888 명령을 실행해서 이 코드를 테스트해도 된다.

그림 2-7 이메일 목록 뷰와 내용 뷰(A-Mail/index.html)



메인 템플릿에는 약간 다른 작업을 넣자. 처음 로딩되는 페이지에 전부 넣는 것이 아니라, 뷰를 집어넣을 레이아웃 템플릿만 작성하자. 그리고 레이아웃 템플릿에는 메뉴 같이 뷰가 전환돼도 계속 표시되는 것들을 전부 배치하자. 이 절에서는 그냥 애플리케이션 이름을 나타내는 제목 요소만 표시해 보자. 그리고 ng-view 지시어를 사용해서 Angular에게 뷰가 나타나야 할 위치를 알려주자.

index.html

```
<html ng-app="AMail">
```

```
    <head>
        <script src="src/angular.js"></script>
        <script src="src/controllers.js"></script>
    </head>
    <body>
        <h1>A-Mail</h1>
        <div ng-view></div>
    </body>
</html>
```

뷰 템플릿은 앞 코드의 셸 안에 삽입될 것이므로, HTML 문서 안에 직접 작성하면 된다. 이제 이메일 목록을 표시하기 위해 다음의 list.html 코드와 같이 ng-repeat 지시어를 사용하여 이메일 항목을 반복시켜 테이블 안에 렌더링하자.

list.html

```
<table>
    <tr>
        <td><strong>발신자</strong></td>
        <td><strong>제목</strong></td>
        <td><strong>일자</strong></td>
    </tr>
    <tr ng-repeat='message in messages'>
        <td>{{message.sender}}</td>
        <td><a href='#/view/{{message.id}}'>{{message.subject}}</a></td>
        <td>{{message.date}}</td>
    </tr>
</table>
```

사용자가 제목을 클릭해서 특정 이메일 항목(message)을 볼 수 있게 했다. URL을

message.id에 데이터 바인딩했으므로 id=1인 이메일 항목을 클릭하면 사용자의 웹 브라우저는 /#/view/1 주소로 이동된다. 이메일 내용 뷰의 컨트롤러 안에 이러한 딥링크 URL 기반의 내비게이션을 사용해서 특정 이메일 항목을 내용 뷰에 표시할 수 있게 하자.

이러한 이메일 내용 뷰를 작성하려면 하나의 메시지 객체에 들어 있는 속성들을 표시하는 템플릿을 다음 detail.html과 같이 작성해야 한다.

detail.html

```
<div><strong>제목:</strong> {{message.subject}}</div>
<div><strong>발신자:</strong> {{message.sender}}</div>
<div><strong>일자:</strong> {{message.date}}</div>
<div>
    <strong>수신자:</strong>
    <span ng-repeat='recipient in message.recipients'>{{recipient}}</span>
</div>
<div>{{message.message}}</div>
<a href='#/'>수신 메일 목록으로 돌아가기</a>
```

이제 컨트롤러를 사용하여 앞서 작성한 템플릿을 연결해야 하므로, 다음의 controller.js 코드에서 보듯이 컨트롤러와 템플릿을 호출하는 URL들을 가지고 $routeProvider를 설정하자.

controllers.js

```
var aMailServices = angular.module('AMail', []);

function emailRouteConfig($routeProvider) {
```

```
    $routeProvider.when('/', {
        controller : ListController,
        templateUrl : 'list.html'
    }).when('/view/:id', {
        controller : DetailController,
        templateUrl : 'detail.html'
    }).otherwise({
        redirectTo : '/'
    });
}

aMailServices.config(emailRouteConfig);
messages = [
    {
        id : 0,
        sender : 'jean@somecompany.com',
        subject : '오랜만이군, 잘 지냈어?',
        date : '2013년 12월 7일,  12:32:00',
        recipients : [ 'greg@somecompany.com' ],
        message : '야, 우리 언제 점심이나 같이 먹으면서 밀린 얘기나 하지 않으련?'
                + '올해 공동작업 할 것도 많고 말이야.'
    },
    {
        id : 1,
        sender : 'maria@somecompany.com',
        subject : '내 노트북 어디 놔뒀냐?',
        date : '2013년 12월 7일,  8:15:12',
        recipients : [ 'greg@somecompany.com' ],
        message : '네가 쓰고 내 책상 서랍에 넣어놨을 줄 알았는데 보니까 거기 없다.'
                + '혹시 노트북 쓰고 어디에 뒀어?'
    },
    {
```

```
        id : 2,
        sender : 'bill@somecompany.com',
        subject : '비단구렁이가 없어졌어요',
        date : '2013년 12월 6일,  20:35:02',
        recipients : [ 'greg@somecompany.com' ],
        message : '놀라진 마세요. 제 비단구렁이가 케이지 밖으로 나갔는지 없거든요.'
                + '움직임이 느리니까 혹시 보시면 전화 좀 주세요.'
    },
];

function ListController($scope) {
    $scope.messages = messages;
}

function DetailController($scope, $routeParams) {
    $scope.message = messages[$routeParams.id];
}
```

이것으로 개인 애플리케이션 기본 구조와 다양한 뷰를 작성해봤다. 뷰 전환은 URL 변경을 통해 이뤄진다. 그래서 사용자는 [앞으로] 버튼과 [뒤로] 버튼을 사용할 수 있다. 게다가 실제 HTML 페이지는 하나뿐이지만, 사용자는 애플리케이션 안에 있는 뷰 링크를 '즐겨찾기'에 추가하거나 이메일로 전송할 수 있다.

2.7 서버와 통신하기

앞 절의 코드는 이제 그만 가지고 놀자. 실무 애플리케이션은 주로 실제 서버와 통신을 한다. 모바일 앱과 성장세인 크롬 데스크톱 애플리케이션은 예외일지 몰라도, 클라우드나 다른 사용자들과의 실시간 채팅 기능을 넣을 생각이든 아니든 상관없이 나머지 애플리케이션에는 서버와의 통신이 아마도 필수적일 것이다.

서버와의 통신을 위해 Angular에는 $http 서비스가 있다. $http 서비스에는 서버와의 통신을 간단히 구현할 수 있게 해주는 수많은 추상 함수가 들어 있다. $http 서비스는 vanilla HTTP, JSONP, CORS를 지원한다. $http 서비스에는 JSON의 취약성과 XSRF를 방지하기 위한 보안 규정이 들어 있다. $http 서비스를 사용하면 요청 데이터와 응답 데이터를 손쉽게 변환할 수 있으며, 간단한 캐시 기능도 구현할 수 있다.

로컬이 아닌 서버에서 쇼핑몰 사이트의 상품을 가져와야 한다고 가정해보자. 서버 처리 코드를 작성하는 것은 이 책의 범주를 벗어나므로 그냥 /products에 질의하면 JSON으로 상품 목록을 반환하는 서비스를 만들었다고 상상만 해보자.

서버가 반환한 응답이 다음과 같다고 하자.

```
[
  {
     "id": 0,
     "title": "페인트 그릇",
     "description": "페인트를 담아 쓸 수 있는 용기",
     "price": 3.95
  },
  {
     "id": 1,
     "title": "땡땡이 리본",
     "description": "물방울 무늬가 자수된 리본",
     "price": 12.95
  },
  {

     "id": 2,
     "title": "공깃돌",
```

```
        "description": "공기 놀이를 위한 오색 조약돌",
        "price": 6.95
    }
    ... 기타 등등 ...
]
```

그러면 질의는 다음과 같이 작성하면 될 것이다.

```
function ShoppingController($scope, $http) {
    $http.get('/products').success(function(data, status, headers, config) {
        $scope.items = data;
    });
}
```

그리고 템플릿은 다음과 같이 작성해야 할 것이다.

```
<body ng-controller="ShoppingController">
    <h1>쇼핑하기!</h1>
    <table>
        <tr ng-repeat="item in items">
            <td>{{item.title}}</td>
            <td>{{item.description}}</td>
            <td>{{item.price | currency}}</td>
        </tr>
    </table>
    </div>
</body>
```

앞에서 배웠지만 서버와 통신하는 기능은 컨트롤러 간에 공유할 수 있는 서비스에

게 처리를 위임하는 것이 장기적으로 더 나을 것이다. 〈AngularJS 활용편〉에서 이러한 구조와 다양한 $http 함수를 살펴볼 것이다.

2.8 지시어로 DOM 수정하기

지시어는 HTML 문법을 확장하며, 지시어를 사용해서 기능과 DOM 변환을 사용자 정의 요소와 속성에 연결할 수 있다. 지시어를 사용해 재사용 가능한 UI 컴포넌트를 작성하고, 애플리케이션을 설정하며, UI 템플릿에서 하고자 하는 무엇이든지 할 수 있다.

Angular의 내장 지시어를 사용해서 애플리케이션을 작성해도 되지만, 자신에게 맞춤화된 지시어를 직접 작성해서 사용해야 할 일도 많을 것이다. 내장 지시어와는 다른 방식으로 웹 브라우저 이벤트를 처리해야 할 때나, DOM을 수정해야 할 때는 지시어를 직접 작성하게 된다. 이런 코드는 컨트롤러나 서비스, 애플리케이션의 다른 부분이 아니라 자신이 작성한 지시어 안에 넣어야 한다.

서비스와 마찬가지로 지시어도 모듈 객체의 API를 사용해 다음 코드와 같이 directive() 함수를 호출하여 정의한다. 여기서 directiveFunction은 지시어의 기능이 정의된 팩토리 함수에 해당한다.

```
var appModule = angular.module('appModule', [...]);
appModule.directive('directiveName', directiveFunction);
```

지시어 팩토리 함수를 작성하는 방법은 깊이가 있으므로 뒤에 따로 다루겠다. 그래도 맛보기는 하고 넘어가야 하니 간단한 예를 하나 살펴보자.

HTML5에는 'autofocus'라는 유용한 속성이 있다. autofocus 속성은 키보드 포커스를 인풋 요소에 위치시킨다. 그래서 autofocus 속성을 사용하면, 애플리케이

션 시작 시 사용자가 인풋 요소를 클릭하지 않고도 키보드로 인풋 요소에 입력할 수 있다. 이것은 자바스크립트를 코딩할 필요 없이 autofocus 속성만 지정하면 웹 브라우저가 해야 할 일을 지정할 수 있다는 점에서 아주 탁월한 기능이다.

그러나 인풋 요소가 아닌 〈a〉 요소나 〈div〉 요소에 포커스를 두고 싶을 땐 어쩔 것인가? 또한, HTML5를 지원하지 않는 웹 브라우저에서도 autofocus 속성이 가능하게 하려면 어떻게 해야 할까? 그럴 땐 다음과 같이 지시어 함수를 사용하면 된다.

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

appModule.directive('ngbkFocus', function() {
    return {
        link : function(scope, element, attrs, controller) {
            element[0].focus();
        }
    };
});
```

앞의 지시어 함수는 link 함수가 지정된 지시어 설정 객체를 반환한다. link 함수는 자신이 속한 스코프 참조, DOM 요소, 해당 지시어에 전달된 모든 속성의 배열, DOM 요소의 컨트롤러를 가져온다(컨트롤러가 있는 경우에만). 이 절의 예제에서는 DOM 요소를 가져오고 focus() 메서드를 호출하기만 하면 된다. 앞에서 작성한 지시어는 다음과 같이 사용하면 된다(다음 예제 파일은 talkingToServers 폴더 안에 있다).

index.html

```
<html lang='ko' ng-app='app'>
```

```
...Angular 프레임워크 및 다른 라이브러리 스크립트 파일을 인클루딩할 것...
<body ng-controller="SomeController">
    <button ng-click="clickUnfocused()">포커스가 없습니다</button>
    <button ngbk-focus ng-click="clickFocused()">여기에 포커스가 있습니다!
        </button>
    <div>{{message.text}}</div>
</body>
</html>
```

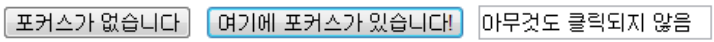
controllers.js

```
function SomeController($scope) {
    $scope.message = {
        text : '아무것도 클릭되지 않음'
    };

    $scope.clickUnfocused = function() {
        $scope.message.text = '포커스 없는 버튼 클릭됨';
    };

    $scope.clickFocused = function() {
        $scope.message.text = '포커스 있는 버튼 클릭됨';
    }
}
var appModule = angular.module('app', [ 'directives' ]);
```

로딩된 페이지에는 라벨이 “여기에 포커스가 있습니다!”라고 적힌 버튼이 포커스 강조를 표시하는 형광 테두리와 함께 표시된다. 이때 키보드에서 스페이스바나 엔터 키를 누르면 클릭이 발생하고 ng-click에 지정한 함수가 호출된다. 호출된 함수는 〈div〉 요소의 텍스트에 “포커스 있는 버튼이 클릭됨”이라는 문자열이 할당된

다. 이 예제를 웹 브라우저에서 열면 그림 2-8과 같은 초기 화면이 표시된다.

그림 2-8 포커스 지시어



2.9 사용자 입력이 올바른지 검사하기

Angular는 〈form〉 요소에 싱글 페이지 애플리케이션에 적합한 몇 가지 탁월한 기능을 자동 보강한다. 그러한 보강의 일환으로, Angular는 폼 안에 인풋의 유효 상태를 선언할 수 있는 기능과 폼의 모든 요소가 유효할 때만 전송 가능하게 하는 기능이 있다.

예컨대 성명과 이메일 주소는 반드시 입력해야 하고 나이는 입력하지 않아도 되는 회원 가입 폼을 제작하려면, 사용자 입력 내용을 서버로 전송하기 전에 검사를 실시하면 된다. 이 예제를 웹 브라우저에서 열면 그림 2-9와 같은 화면이 뜬다.

그림 2-9 폼 유효성 검사(formValidation.html)

사용자가 이름 입력란에 텍스트를 입력하고, 입력한 이메일 주소는 적절한 형식이며, 나이를 입력해야만 폼은 유효하다.

다음과 같이 Angular의 표현식을 〈form〉 요소와 각종 인풋 요소에 사용하면, 이

모든 기능을 템플릿 안에서 처리할 수 있다.

```
<h1>가입하기</h1>
<form name='addUserForm'>
   <div ng-show='message'>{{message}}</div>
   <div>이름: <input ng-model='user.first' required /></div>
   <div>성씨: <input ng-model='user.last' required /></div>
   <div>이메일: <input type='email' ng-model='user.email' required /></div>
   <div>나이: <input type='number' ng-model='user.age' ng-maxlength='3'
      ng-min='1' /></div>
   <div><button>전송</button></div>
</form>
```

앞의 코드를 보면, 세 필드의 유효성 검사를 위해 required 속성을 사용했으며 이메일 〈input〉 요소의 type 속성에 HTML5의 email 값과 number 값을 지정했다. 이 방법은 Angular와 잘 연동되며, HTML5를 지원하지 않는 웹 브라우저에서 Angular는 같은 기능을 하는 지시어를 사용해서 이러한 갭을 메운다.

이제 전송 기능을 담당하는 컨트롤러를 추가하자. 먼저 컨트롤러를 참조하게끔 〈form〉 요소를 다음과 같이 수정해두자.

```
<form name='addUserForm' ng-controller="AddUserController">
```

컨트롤러 안에서 $valid 속성을 통해 폼의 유효 상태에 접근할 수 있다. Angular는 폼의 모든 인풋이 유효하면 $valid 속성에 true 값을 할당한다. $valid 속성을 사용하면 폼 입력이 완료되지 않은 상태에선 전송 버튼을 비활성화하는 등의 유용한 작업들을 수행할 수 있다.

유효하지 않은 상태에선 폼 전송이 이뤄지지 않도록 하려면, 다음과 같이 전송 버튼에 해당하는 요소에 ng-disabled 지시어를 지정하면 된다.

```
<button ng-disabled='!addUserForm.$valid'>전송</button>
```

끝으로 컨트롤러는 사용자에게 가입이 성공적으로 완료되었음을 알려야 한다. 이렇게 해서 완성된 템플릿 코드는 다음과 같다.

```
<h1>가입하기</h1>
<form name='addUserForm' ng-controller="AddUserController">
   <div ng-show='message'>{{message}}</div>
   <div>이름: <input name='firstName' ng-model='user.first' required
/></div>
   <div>성씨: <input ng-model='user.last' required /></div>
   <div>이메일: <input type='email' ng-model='user.email' required /></div>
   <div>나이: <input type='number' ng-model='user.age' ng-maxlength='3'
      ng-min='1' /></div>
   <div><button ng-click='addUser()'
      ng-disabled='!addUserForm.$valid'>전송</button></div>
</form>
```

그리고 컨트롤러는 다음과 같다.

```
function AddUserController($scope) {
   $scope.message = '';

   $scope.addUser = function() {
      // 가입된 사용자를 데이터베이스에 실제로 저장하는 코드를
      // 여기에 스스로 작성해보자.
```

```
            $scope.message = '감사합니다, ' + $scope.user.first + '님. 가입을 축하
                             드립니다!';
        };
    }
```

2.10 다음 장에서 배울 내용

지금까지 두 개의 장에 걸쳐 Angular 프레임워크에서 주로 사용되는 기능에 대해 알아보았다. 설명한 기능마다 미처 다루지 못한 부가적인 사항도 많다. 다음 장에서는 Angular의 일반적인 개발 순서에 대해 살펴보겠다.

3 | AngularJS 프레임워크로 개발하기

이제 AngularJS를 구성하는 요소들에 대해서는 조금 감이 올 것이다. 앞 장에서 사용자에게 입력받은 데이터를 어떻게 애플리케이션으로 가져오는지, 텍스트를 어떻게 표시하는지, 유효성 검사와 필터링과 DOM 수정을 통해 어떻게 그럴싸한 작업을 수행하는지 알아보았다. 그런데 그것들을 어떻게 취합할까?

이 장에서는 AngularJS 애플리케이션을 어떤 식으로 구성하는지 전반적으로 알아볼 것이다. 따라서 실제 애플리케이션 자체를 다루지는 않는다. AngularJS의 다양한 기능이 사용된 샘플 애플리케이션은 4장에서 살펴보겠다.

이 장에서 다룰 내용은 다음과 같다.

- 신속한 개발을 위해 AngularJS 애플리케이션을 구성하는 방법
- AngularJS 애플리케이션을 실행시킬 서버를 구동하는 방법
- Karma를 사용해서 단위 테스트와 시나리오 테스트를 작성하고 실행하는 방법
- 제품 배포를 위해 AngularJS 애플리케이션을 컴파일하고 최소화하는 방법
- Batarang을 사용해서 AngularJS 애플리케이션을 디버깅하는 방법
- 새 파일 작성에서부터 애플리케이션 실행과 테스트에 이르기까지 개발 흐름을 단순화하는 방법
- AngularJS 프로젝트를 종속물 관리 라이브러리인 RequireJS와 연동하는 방법

3.1 프로젝트의 구성

프로젝트의 구조를 잡는 데에는 Yeoman[01]을 사용하길 권한다. Yeoman은 AngularJS 애플리케이션에 부트스트랩을 실시하는 데 필요한 파일들을 자동으로 생성한다.

01 '요우먼'이라고 읽는다. http://yeoman.io/

Yeoman은 다양한 프레임워크와 클라이언트 사이드[02] 라이브러리로 구성된 탄탄한 도구다. Yeoman을 사용하면, 부트스트랩과 애플리케이션 개발에 필수적인 일부 루틴 작업이 자동화되어 신속한 개발 환경이 확보할 수 있다. 이 장 전체에 걸쳐 Yeoman을 설치하고 사용하는 방법에 대해 설명할 텐데, 일단은 이러한 작업을 수동으로 수행하기 위해 Yeoman 명령에 대해 간략히 살펴보고 넘어가자.

뒷부분에서는 Yeoman을 사용하지 않고 작업하는 방법에 대해서도 다룰 것이다. 왜냐하면 Yeoman은 윈도우 환경 컴퓨터에서 약간 문제가 있는데다 설정하기도 조금 까다롭기 때문이다.

Yeoman을 사용하지 않는 개발자를 위해 GitHub를 통해 제공하는, 이 책의 코드 파일 중 chapter3/sample-app 폴더에 든 샘플 애플리케이션의 구조도 살펴볼 것이다. 3장의 sample-app 애플리케이션은 Yeoman을 사용해 구조를 잡았으므로 권장할만한 구조로 돼 있다. 이 애플리케이션을 구성하는 파일을 카테고리로 분류하면 다음과 같다.

자바스크립트 소스 파일

chapter3/sample-app/app/scripts 폴더 안에는 자바스크립트 소스 코드 파일을 전부 모아뒀다. 그 중에서 메인 파일인 app/scripts/app.js는 Angular 모듈과 애플리케이션의 각종 루트를 설정한다.

chapter3/sample-app 폴더 안에는 app/scripts/controllers 폴더가 있는데, 이 폴더엔 개별 컨트롤러 파일이 들어 있다. 컨트롤러는 스코프에 액션을 제공하고 데이터를 발행한다. 이 스코프가 나중에 뷰에 표시되며, 대체로 뷰와 일대일 대응된다.

지시어, 필터, 서비스 파일도 app/scripts 폴더에 들어 있다.

---

02 네트워크의 한 방식인 클라이언트/서버 구조의 클라이언트 쪽에서 행해지는 처리를 말한다.

HTML Angular 템플릿 파일

Yeoman이 생성하는 AngularJS의 구성 템플릿은 전부 app/views 폴더에 들어 있다. 이 폴더에는 대개 app/scripts/controllers 폴더와 일대일 대응된다.

중요한 Angular 템플릿 파일이 하나 더 있다. 그것은 메인 템플릿인 app/index.html 파일이다. 이 파일은 AngularJS에 내장된 소스 파일과 개발자 애플리케이션의 소스 파일을 전부 불러와서 표시하는 밑바탕 역할을 한다.

새로운 자바스크립트 파일을 작성했으면 반드시 index.html 파일에 연결해야 하고, 메인 모듈과 루트도 업데이트해야 한다. Yeoman을 사용하면 이 작업이 자동으로 이뤄진다.

자바스크립트 라이브러리 종속물

Yeoman에는 자바스크립트 소스의 의존 라이브러리를 모아놓은 app/scripts/vendor 폴더가 있다. 애플리케이션에 Underscore나 SocketIO를 사용하고 싶은가? 문제 없다. 그냥 Underscore나 SocketIO 같은 의존 라이브러리 파일을 vendor 폴더에 넣고 index.html 파일 안에서 불러온 후, 애플리케이션의 필요한 부분에서 라이브러리의 함수를 참조하면 된다.

정적 리소스

작업 마지막에 HTML 애플리케이션을 작성할 것이므로, 당연히 필요한 CSS와 이미지 종속물을 애플리케이션과 함께 제공해야 한다.

그러한 CSS와 이미지 종속물을 제공하는 폴더는 app/styles과 app/img이다. 필요한 종속물을 이 두 폴더에 추가한 후 애플리케이션에서 정확한 상대 경로를 사용해 참조하면 된다.

NOTE_ Yeoman은 app/img 경로를 기본으로 생성하지 않는다.

단위 테스트

테스트는 상당히 중요한데, AngularJS를 사용하면 테스트하기가 아주 쉬워진다. test/spec 폴더는 테스트 측면에서 app/scripts 폴더와 일대일로 대응되어야 한다. 각 파일마다 단위 테스트 코드가 들어 있는 스펙spec 파일이 일대일로 있어야 한다. 시드(스펙 파일)는 test/spec/controllers 폴더에 원본 컨트롤러와 같은 이름으로 각 컨트롤러 파일의 스텁을 생성한다. 이러한 스펙 파일은 컨트롤러의 각 기능에 대한 상세 정보가 기술되어 있는 Jasmine 프레임워크 방식으로 되어있다.

통합 테스트

AngularJS는 라이브러리 안에 클라이언트 서버 간(end-to-end) 테스트 기능이 내장돼 있다. 모든 클라이언트 서버 간 테스트는 Jasmine 스펙 형식으로 tests/e2e 폴더에 저장된다.

생성된 간단한 HTML 파일도 하나 있는데, 이 파일은 웹 브라우저에서 단독으로 열어서 수동으로 테스트할 수 있다. Yeoman은 이 파일에 대한 스텁은 아직 생성하지 않지만 단위 테스트와 방식은 비슷하다.

NOTE_ Yeoman은 tests/folder 경로를 기본으로 생성하지 않는다.

NOTE_ 클라이언트 서버 간 테스트 코드가 Jasmine 같아 보일 수도 있지만, 사실은 그렇지 않다. 클라이언트 서버 간 테스트 코드는 나중에 Angular Scenario Runner에 의해 비동기적으로 실행된다. 따라서 Jasmine 테스트에서 사용할 수 있는 repeater 값에 대한 console.log 함수 같은 기능은 사용할 수 없다.

설정 파일

설정 파일은 두 개가 있다. 첫 번째는 karma.conf.js 파일인데, 이 파일은

Yeoman에 의해 자동 생성되며 단위 테스트 실행에 사용된다. 두 번째는 Yeoman이 생성하지 않는 karma.e2e.conf.js 파일이며 시나리오 테스트를 실행하는 데 사용된다. RequireJS와의 통합을 다른 이 장의 끝 절에 간단한 샘플 파일을 수록했다. 3.10절에서 RequireJS와 통합하는 방법에 대해서 간단한 예제로 설명할 것이다. 환경설정에는 Karma를 이용해 단위 테스트를 실행할 때, 사용할 종속물과 파일이 상세히 기술된다. Yeoman은 기본적으로 Karma 서버를 9876 포트로 실행한다.

아마도 애플리케이션 실행 방법, 단위 테스트 방법, 앞서 설명한 각 부분을 코딩하는 방법이 궁금할 것이다.

나중에 필요한 부분에서 각각 설명할 것이니 걱정할 필요 없다. 이 장에서는 프로젝트와 개발 환경을 구성해야 한다. 그래야 유용한 코드를 작성하기 시작했을 때 개발이 빠르게 진척되기 때문이다. 무슨 코드를 작성해야 하는지, 그렇게 작성한 코드를 완성된 유용한 애플리케이션으로 어떻게 엮어내는지에 대해서는 4장에서 알아본다.

3.1.1 도구

AngularJS는 웹 페이지를 실제로 개발할 수 있게 도와주는 도구모음의 일부에 지나지 않는다. 이 절에서는 IDE(통합 개발 환경)에서부터 테스트 러너와 디버거에 이르기까지 효율적이고 신속한 개발을 보장하는 각종 도구를 알아볼 것이다.

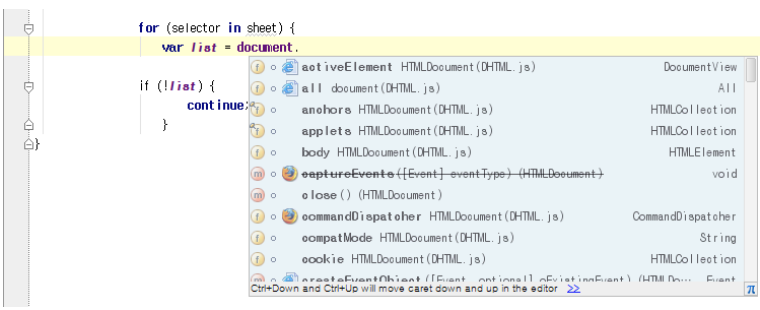
3.1.2 각종 IDE

소스 코드를 편집하려면 어떻게 해야 하는지부터 살펴보자. 시중에 유료/무료 자바스크립트 에디터는 널려 있다. Emacs나 Vi가 자바스크립트로 개발할 때 최선의 선택이던 시절로부터 엄청난 발전이 있었다. 요즘은 IDE에 구문채색이나 자동완성을 비롯한 무수한 기능이 들어 있으니 사용해보면 편리함을 느낄 것이다. 그렇다

면 과연 어느 IDE를 사용할 것인가?

필자는 WebStorm[03]을 추천한다. 30일 시험판으로 사용해볼 수 있는 유료 IDE이긴 하지만, 요즘에는 JetBrains가 제작한 WebStorm이 가장 다양한 기능을 지닌 웹 개발 플랫폼이다. WebStorm에는 원래 타입 구별 언어에만 사용할 수 있던 각종 기능이 들어 있다. 나열하자면 그림 3-1 같은 웹 브라우저별 코드완성, 코드 탐색, 구문 채색, 에러 채색, 다양한 라이브러리와 프레임워크의 뛰어난 기능 등이다. 게다가 자바스크립트를 크롬에서 실행하면서 IDE에서 바로 디버깅할 수 있는 아주 깔끔한 통합 기능이 갖춰져 있다.

그림 3-1 WebStorm에서 브라우저별로 제시되는 자동완성 코드



AngularJS 개발에 WebStorm을 진지하게 고려해야 하는 가장 큰 이유는 WebStorm이 AngularJS 플러그인을 갖춘 몇 안 되는 IDE 중 하나기 때문이다. AngularJS 플러그인은 HTML 템플릿 안의 AngularJS HTML 태그에 자동완성 기능을 제공한다. 또한, 멋진 기능 중 하나는 '라이브 템플릿'이라는 개념이다. 라이브 템플릿은 개발자가 처음부터 작성할 필요가 없도록 공통적인 코드 부분을 미리 만들어 놓은 템플릿이다.

03 http://www.jetbrains.com/webstorm/

```
directive('$directiveName$', function factory($injectables$) {
    var directiveDefinitionObject = {
        $directiveAttrs$
        compile: function compile(tElement, tAttrs, transclude) {
            $END$
            return function (scope, element, attrs) {
            }
        }
    };
    return directiveDefinitionObject;
});
```

따라서 WebStorm을 사용하면 코드를 위와 같이 작성할 필요 없이 그냥 다음과 같이 작성하면 된다.

```
ngdc
```

그리고 탭 키를 누르면 같은 결과를 얻을 수 있다. 이것은 AngularJS 플러그인이 제공하는 많은 기능 중 하나에 불과하다.

3.2 작성한 애플리케이션 실행하기

작업물을 실행하는 방법에 대해 알아보기 위해, 웹 브라우저에서 애플리케이션이 돌아가는 모습을 직접 보자. 애플리케이션이 돌아가는 원리에 대해 제대로 감을 잡으려면, HTML 파일과 자바스크립트 파일을 올려둘 수 있는 웹 서버가 있어야 한다. 애플리케이션을 실행하는 두 가지 방법이 있다. 첫 번째는 아주 간단한 방법으로, Yeoman을 사용해 애플리케이션을 실행시키는 것이다. 두 번째는 Yeoman을 사용하지 않는 방법인데 이건 조금 복잡하다.

3.2.1 Yeoman으로 실행

Yeoman을 이용하면 간단히 웹 서버를 구동하고, 정적인 AngularJS 관련 파일을 전부 실행할 수 있다. Yeoman으로 서버를 구동하려면 다음 명령을 실행한다.

```
yeoman server
```

그러면 서버가 구동되고 웹 브라우저가 열리면서 AngularJS 애플리케이션의 메인 페이지가 뜬다. 그리고 소스 코드를 수정할 때마다 웹 브라우저가 새로고침된다.

3.2.2 Yeoman을 사용하지 않고 실행

Yeoman을 사용하지 않고 실행하려면 모든 파일을 제공할 웹 서버를 설정해야 한다. 웹 서버를 설정하는 쉬운 방법을 모르거나 웹 서버 제작에 시간들이기 싫으면, ExpressJS를 사용해서 간단한 웹 서버를 즉시 작성할 수 있다. 그냥 다음과 같이 Node에서 npm install -g express 명령만 실행하면 된다.

소스 코드 3-1 chapter3/sample-app/web-server.js

```
var express = require("express"), app = express(),
    port = parseInt(process.env.PORT, 10) || 8080;

app.configure(function() {
    app.use(express.methodOverride());
    app.use(express.bodyParser());
    app.use(express.static(dirname + '/'));
    app.use(app.router);
});

app.listen(port);
console.log('애플리케이션을 제공하는 위치가 이제 http://localhost:' + port
```

```
        + '/app 으로 설정됐습니다');
```

앞의 web-server.js 파일을 Node를 이용해 실행하려면 다음 명령을 입력한다.

```
node web-server.js
```

그러면 8080 포트나 자신이 선택한 포트로 서버가 구동된다. 아니면 애플리케이션이 들어 있는 폴더에서 Python을 사용해 다음과 같은 명령을 실행해도 된다.

```
python -m SimpleHTTPServer
```

어느 방법으로 하든 서버 설정이 완료되고 작동 중이면, 다음 주소로 이동한다.

```
http://localhost:[포트번호]/app/index.html
```

웹 브라우저에서 앞의 주소로 이동하면 방금 작성한 애플리케이션이 뜰 것이다. 변경사항이 적용된 모습을 보려면 Yeoman과는 달리 반드시 수동으로 웹 브라우저를 새로고침해야 한다.

3.3 AngularJS로 테스트하기

앞에서도 언급했지만, 다시 강조하건대 테스트는 필수며 AngularJS를 사용하면 적절한 단위 테스트와 통합 테스트를 손쉽게 작성할 수 있다. AngularJS는 다양한 테스트 실행 도구에서 원활히 실행되지만, 필자는 Karma[04]가 가장 안정적이고 견고하며 개발자의 모든 요구에 부응하는 매우 빠른 테스트 실행 도구를 갖췄다고 확신한다.

04 https://github.com/vojtajina/karma/

3.3.1 Karma

Karma의 주요 개발 목적은 테스트 주도 개발TDD, test-driven development 흐름을 단순하고 빠르며 재미있게 만드는 것이다. Karma는 NodeJS와 SocketIO를 사용해서 여러 웹 브라우저에서 엄청나게 빠른 속도로 코드를 실행하고 테스트할 수 있다. 다행히 NodeJS와 SocketIO에 대해서는 알지 못해도 된다. Karma에 대한 자세한 내용은 웹 사이트[05]를 참고하자.

여기서 잠깐_ TDD

TDD는 코드를 구현하기 전에 테스트를 먼저 작성하는, 개발 수명주기를 뒤집는 애자일AGILE 방법론이다. 그리고 이런 각종 테스트는 단순히 검사 도구로만 사용되는 것이 아니라 개발의 속도를 높여준다. TDD의 원칙은 다음과 같다.

- 코드는 반드시 통과해야 할 결함 테스트failing test가 있을 때에만 작성한다.
- 테스트를 통과시키는 데 필요한 최소한의 코드만을 작성한다.
- 매 단계마다 중복된 코드를 삭제한다.
- 모든 테스트를 통과했으면, 결함 테스트는 그다음 필수 기능에 추가한다.

이러한 간단한 원칙을 준수함으로써 다음과 같은 장점이 확보된다.

- 코드가 조직적으로 발전해 나가므로, 작성한 모든 코드 행이 각각 뚜렷한 목적을 지닌다.
- 코드가 매우 모듈화되고, 응집력이 높으며, 재사용 가능한 상태로 유지된다. 코드를 테스트할 수 있어야 하기 때문이다.
- 향후의 깨짐이나 버그를 방지하는 일련의 종합적인 테스트들을 제공할 수 있다.
- 테스트가 명세 역할을 하며, 나중에 필요하거나 수정해야 할 때 문서 역할도 한다.

AngularJS에서 앞에 나열한 항목들이 사실임을 확인했으며, 전체 AngularJS 코드베이스codebase가 바로 TDD를 이용해 개발됐다. 자바스크립트처럼 컴파일되지 않는 동적 언어에서 일련의 단위 테스트를 작성하면 나중에 문제 발생 가능성이 줄어든다는 점은 확실하다.

그렇다면 이러한 장점을 Karma에서 어떻게 얻을 것인가? 우선 NodeJS를 컴퓨터에

05 https://github.com/vojtajina/karma/

설치해야 한다. NodeJS를 설치하면 NPM(Node Package Manager, 노드 패키지 관리자)가 들어 있는데, 이것을 사용하면 NodeJS에 호환되는 수많은 라이브러리를 손쉽게 관리하고 설치할 수 있다.

NodeJS와 NPM 설치가 완료됐으면 다음과 같이 손쉽게 Karma를 설치할 수 있다.

```
sudo npm install -g karma
```

이것으로 Karma를 시작할 준비가 됐다. Karma를 시작하려면 다음에 설명할 3단계를 실시하자.

환경설정 파일 만들기

애플리케이션 뼈대를 Yeoman을 사용해서 작성했다면, Karma 설정 파일이 이미 있을 것이다. 만약 없다면 애플리케이션 디렉터리의 base 폴더에서 다음 명령을 실행하면 된다.

```
karma init
```

앞의 명령을 터미널 콘솔에서 실행시키면 karma.conf.js라는 이름으로 환경설정 파일이 생성된다. 이 파일의 이름은 원하는 대로 수정할 수 있지만 karma.conf.js가 가장 많이 사용하는 기본 이름이니 이 이름을 사용하면 된다.

Karma 서버 시작하기

다음 명령을 실행하면 된다.

```
karma start [환경설정_파일의_옵션_경로]
```

Karma 서버가 9876 포트에서 시작된다. 9876 포트가 기본인데, 앞 단계에서 karma.conf.js 파일을 수정해서 포트 번호를 바꿔도 된다. Karma는 웹 브라우저를 띄우고 웹 브라우저를 자동으로 캡처하면서, 콘솔에서 다른 웹 브라우저 캡처에 필요한 명령들도 전부 보여줄 것이다. 너무 게을러서 이걸 못 하겠다는 사람은, 그냥 다른 웹 브라우저나 기기에서 "http://localhost:9876"로 이동하면 다수의 웹 브라우저에서 테스트 실행을 시작할 수 있다.

> **NOTE_** Karma는 시작할 때 일반적으로 많이 사용하는 웹 브라우저(Firefox, Chrome, IE, Opera, PhantomJS)를 자동으로 캡처할 수 있다. 물론 다른 웹 브라우저도 캡처할 수 있다. URL을 탐색할 수 있는 어떠한 기기에서든 Karma를 실행할 수가 있다. 따라서 아이폰이나 안드로이드 기기의 웹 브라우저를 띄우고 "http://machinename:9876"을 탐색하면, 이 주소가 실제로 있다면 모바일 기기에서도 테스트를 실행할 수 있다.

테스트 실행하기

다음 명령을 실행하면 된다.

```
karma run
```

이것으로 카르마가 실행된다. 이제 명령을 실행하면 그 결과가 콘솔에 제대로 출력될 것이다. 아주 쉽지 않은가?

3.4 단위 테스트

AngularJS를 사용하면 단위 테스트를 손쉽게 작성할 수 있으며, 테스트를 Karma에서처럼 Jasmine 방식으로 작성할 수도 있다. Jasmine을 필자는 기능 위주의 개발 프레임워크라고 부르며, 이러한 프레임워크를 사용하여 자신의 코드가 어떠한 원리로 기능을 수행해야 할지 나타내는 명세를 작성할 수 있다. Jasmine 코드는

다음과 같다.

```
describe("MyController:", function() {
    it("to work correctly", function() {
        var a = 12;
        var b = a;

        expect(a).toBe(b);
        expect(a).not.toBe(null);
    });
});
```

보다시피 Jasmine 코드 형식은 아주 알아보기가 쉽다. 코드 대부분이 일반 영어 문장을 읽는 것과 다를 바 없기 때문이다. 그리고 expect문 같이 아주 다양하고 강력한 대조 기능을 제공하며, setUp과 tearDowns이라는 xUnit의 주요 구성 함수가 들어 있다. 두 함수는 각 테스트 케이스 전후에 실행된다.

AngularJS에는 몇 가지 유용한 모의 테스트와 테스트 함수가 들어 있어서 서비스, 컨트롤러, 필터를 자신의 단위 테스트 안에 바로 작성할 수 있다. 또한 HttpRequest 같은 객체를 모사할 수 있다(이에 대해서는 〈AngularJS 활용편〉을 참고하기 바란다).

Karma를 자신의 개발 작업흐름에 보다 쉽게 통합할 수 있으며, 작성한 코드에 대한 보다 빠른 피드백을 얻을 수 있다.

IDE와의 통합

Karma는 아직 기능이 많은 최신의 모든 IDE를 통합할 수 있는 플러그인이 없지만, 따지고 보면 사실 필요하지도 않다. 단지 사용하는 IDE 안에서 karma start와 karma run을 실행하는 단축 명령만 추가하면 된다. 이를 위해서는 보

통 자신이 선택한 에디터에 따라 실행하고자 하는 간단한 스크립트를 추가하든지 직접적인 셸 명령을 추가하면 된다. 물론 명령의 실행이 완료될 때마다 결과가 표시된다.

변경될 때마다 테스트 실행하기

저장을 누를 때마다 엄청나게 짧은 순간에 모든 테스트가 실행되고 즉시 그 결과를 볼 수 있기를 수많은 TDD 개발자는 꿈꾼다. 이 기능은 AngularJS와 Karma를 병용하면 아주 쉽게 구현할 수 있다. Karma 환경설정 파일(karma.conf.js)에는 autoWatch 플래그가 들어 있다. autoWatch 플래그의 값을 true로 지정하면, Karma는 감시 중인 파일(소스 코드와 테스트 코드)이 변경될 때마다 테스트를 실행한다. 그럼 IDE 내에서 karma start 명령을 수행하면 어떻게 될까? Karma 실행으로 인한 결과를 IDE 내에서 바로 확인할 수 있다. 따라서 개발자는 무슨 문제가 있는지 보기 위해 콘솔과 터미널을 오가지 않아도 된다.

3.5 종단간 테스트와 통합 테스트

애플리케이션의 덩치가 커지고 개발자가 눈치챌 겨를도 없이 매우 빨라짐에 따라, 애플리케이션이 의도대로 돌아가는지를 수동으로 테스트하기란 만만치 않게 되었다. 무엇보다 새 기능을 추가할 때마다, 새로운 기능이 제대로 실행되는지 확인해야 하지만, 그와 동시에 기존의 기능이 여전히 잘 실행되는지, 버그나 퇴보는 없는지도 확인해야 한다. 여러 웹 브라우저에 새로운 기능을 제공하려면, 테스트해야 할 경우의 수가 얼마나 폭발적으로 늘어날지는 쉽게 짐작할 수 있다.

AngularJS에는 애플리케이션에 연동할 수 있는 Scenario Runner가 있어서 그러한 엄청난 테스트 작업을 줄여준다.

Scenario Runner를 통해 개발자는 애플리케이션을 Jasmine 스타일의 문법으로

기술할 수 있다. 앞에 설명한 단위 테스트에서와 마찬가지로 기능에 대한 일련의 describe와 그 기능의 개별 기능을 설명하는 각각의 it을 작성해야 한다. 항상 그렇듯이 테스트를 호출할 때 각 명세 앞뒤에 수행될 공통적인 기능을 넣을 수 있다.

애플리케이션이 결과를 필터링하는 모습을 관찰하는 샘플 테스트는 다음과 같이 작성할 수 있다.

```
describe('결과를 검색', function() {
    beforeEach(function() {
        browser().navigateTo('http://localhost:8000/app/index.html');
    });
    it('결과를 필터링 함', function() {
        input('searchBox').enter('jacksparrow');
        element(':button').click();
        expect(repeater('ul li').count()).toEqual(10);
        input('filterText').enter('Bees');
        expect(repeater('ul li').count()).toEqual(1);
    });
});
```

이 테스트를 실행하는 방법은 두 가지다. 어느 방법을 사용하든 애플리케이션을 제공할 웹 서버를 시동해야 한다. 웹 서버를 시동하는 방법은 앞에 있는 내용을 참고하기 바란다. 웹 서버를 시동했으면 다음 중 하나의 방법을 사용하자.

1. 자동 실행 방법: Karma는 현재 Angular 시나리오 테스트의 실행 기능을 지원한다. Karma 환경설정 파일을 작성하자.
   a. 환경설정 파일의 files 섹션에 ANGULAR_SCENARIO & ANGULAR_SCENARIO_ADAPTER를 추가한다.
   b. 서버로의 요청을 테스트 파일이 들어 있는 적절한 폴더로 재전송하는 proxies

섹션을 다음과 같이 추가한다.

```
proxies = {'/': 'http://localhost:8000/test/e2e/'};
```

c. Karma의 소스 파일이 테스트에 방해되지 않도록 다음과 같이 Karma 루트를 추가한다.

```
urlRoot = '/_karma_/';
```

그다음에 "http://localhost:9876/_karma_"로 가서 Karma 서버를 캡처하면 된다. 개발자는 Karma를 이용해 막힘 없이 테스트를 실행할 수 있다.

2. 수동 실행 방법: 테스트를 수동으로 실행하면 웹 서버에 있는 간단한 페이지를 열어서 모든 테스트를 실행하면서 볼 수 있다. 그 절차는 다음과 같다.

   a. 간단한 runner.html 파일을 작성하자. 이 파일에는 Angular 라이브러리를 구성하는 angular-scenario.js 파일을 연결한다.
   b. Scenario 스위트 안에 작성한 명세가 들어 있는 모든 자바스크립트 파일을 연결한다.
   c. 웹 서버를 구동하고 runner.html 파일을 웹 브라우저에서 연다.

타사의 통합 또는 웹 브라우저 서버 간 테스트 실행 도구가 아닌 Angular Scenario Runner를 사용해야 하는 이유는 무엇일까? Scenario Runner를 사용하면 다음과 같은 몇 가지 커다란 장점이 있기 때문이다.

AngularJS 기민성

Angular Scenario Runner는 이름에서 미루어 짐작할 수 있듯이, Angular를 위해 만들어졌다. 그래서 Angular와 적절하게 동작하며, 바인딩 같은

AngularJS의 다양한 요소를 잘 파악하고 인식한다. 텍스트 입력이 필요하다면? 바인딩 값을 검사해야 한다면? 리피터의 상태를 확인해야 한다면? 이 모든 것은 Scenario Runner를 사용하면 가능하다.

임의의 대기 시간이 없음

Angular 기민성은 Angular가 서버에 이뤄지는 모든 XHR[XmlHttpRequest] 에도 기민함을 뜻한다. 따라서 페이지가 로딩되기까지 임의의 시간 동안 대기해야 하는 사태를 피할 수 있다. Scenario Runner는 페이지가 로딩된 시점을 알고 있으므로, 페이지 로딩 대기 중에 타임아웃으로 인해 테스트가 실패할 가능성 등이 있는 Selenium 테스트에 비해 훨씬 결정성이 높다.[06]

디버깅 기능

Scenario 테스트가 실행되는 동안 자신의 코드를 보면서 자바스크립트를 분석하고 원할 때 테스트를 일시중지/재개할 수 있다면 좋지 않겠는가? Angular Scenario Runner를 사용하면 이 모든 것뿐 아니라 더한 기능도 가능하다.

3.6 컴파일

자바스크립트에서 컴파일은 주로 코드의 최소화를 뜻하지만, Google Closure Library를 사용하여 약간은 진정한 의미의 컴파일도[07] 가능하다. 하지만 그렇게 훌륭하게 잘 작성되었고 이해하기 쉬운 코드를 도대체 왜 번거롭게 코드로 변환해야 할까?

한 가지 이유는 사용자에게 빠르고 응답성이 좋은 애플리케이션을 만들려는 목적

---

06 역자 주_ '결정성(deterministic)이 높다'란 의미다.
입력 값에 따라 출력 값이 결정된다. 입력이 일정하면 출력도 같게 나온다. 즉, 도중의 변수로 인한 예상치 못 한 결과가 나올 확률이 낮음을 뜻한다.

07 자바스크립트의 컴파일은 주로 코드를 최소화하는 것인데 반해, 구글의 클로저 라이브러리 사용하면 어느 정도 진정한 의미의 컴파일이 가능하단 뜻이다.

때문이다. 클라이언트 측 애플리케이션이 몇 년 사이에 급부상한 주된 이유가 바로 그 때문이다. 그리고 애플리케이션 작동을 빠르게 할수록 응답률도 높아진다.

자바스크립트 코드를 최소화하는 이유가 바로 응답률을 높이기 위해서다. 코드가 간결할수록 페이로드가 줄어들어 사용자 브라우저로의 파일 전송 속도가 빨라진다. 이것은 파일 크기가 장애가 되는 모바일 앱에선 특히 중요하다.

자신의 애플리케이션에 작성한 AngularJS 코드를 최소화하는 방법이 몇 가지 있는데, 각 방법은 효과가 각각 다르다.

기본적이고 간단한 최적화

기본적이고 간단한 최적화를 위해서는 코드에 사용한 모든 변수를 최소화하되 속성은 최소화하지 말아야 한다. 이것을 소위 'Closure Compiler'를 통과하는 간단한 최적화라고 한다. 이 최적화로는 파일 크기가 눈에 띄게 줄어들진 않지만 오버헤드가 아주 적다는 장점이 있다.

이것이 가능한 이유는 컴파일러(Closure[08]나 UglifyJS[09])가 템플릿이 참조하는 속성명을 바꾸지 않기 때문이다. 따라서 템플릿은 이 최적화 이후에도 계속 문제없이 돌아가며 단지 지역변수와 매개변수의 이름만 변경된다.

Google Closure를 사용하면 이 최적화는 다음과 같이 간단히 호출만 하면 된다.

```
java -jar closure_compiler.jar --compilation_level SIMPLE_OPTIMIZATIONS
--js path/to/file.js
```

고급 최적화

고급 최적화는 가능한 거의 모든 함수의 이름을 변경하기 때문에 좀 더 복잡하

08 https://developers.google.com/closure/compiler/
09 https://github.com/mishoo/UglifyJS

다. 이 수준으로 최적화를 하려면 externs 파일을 사용해서 이름을 변경하면 안 되는 함수, 변수, 속성을 명시적으로 지정하는 식으로 컴파일러를 상당히 밀착 감시해야 한다. 이름을 변경하면 안 되는 것들은 주로 템플릿이 사용하는 함수와 속성이다.

컴파일러는 externs 파일을 사용해서 지정하지 않은 모든 것의 이름을 변경한다. 적절히 완료되면 자바스크립트 파일 크기가 확연히 줄어들지만, 이를 위해서는 코드가 변경될 때마다 externs 파일을 업데이트하는 등 엄청난 양의 작업이 필요하다.

한 가지 명심할 점은 코드를 최소화하려면 종속물 주입의 선언 형태(컨트롤러에 $inject 속성을 지정)를 사용해야 한다는 것이다.

다음과 같이 작성하면 소용 없다.

```
function MyController($scope, $resource) {
    // 이런저런 기능
}
```

앞의 코드 대신에 다음 중 한 가지 방식으로 작성해야 한다.

```
function MyController($scope, $resource) {
    // 이런저런 기능
}

MyController.$inject = [ ' $scope ' , ' $resource ' ];
```

아니면 모듈을 사용해서 다음과 같이 작성하자.

```
myAppModule.controller( ' MyController ' , [ ' $scope ' ,
                                        ' $resource ' ,
                                       function($scope, $resource) {
    // 이런저런 기능
}]);
```

이것은 개발자가 원래 요청하려던 서비스나 변수 중 어느 것이 모호하거나 압축됐는지를 AngularJS가 알아낼 수 있는 유일한 방법이다.

> **NOTE_** 나중에 코드를 컴파일할 때 버그가 생기지 않도록 항상 배열 형태의 주입[09]을 사용하는 것이 바람직하다. 나중에 머리를 쥐어뜯으면서 $e 변수의 프로바이더(일부 서비스의 최소화되고 모호화된 버전)가 어째서 갑자기 없어진 건지 알아내려고 해봐야 소용없다.

3.7 그 밖의 유용한 도구

이 절에서는 개발 흐름에 도움되고 생산성을 향상시켜주는 도구들을 살펴보자. 여기서 소개할 도구들은 Batarang을 사용한 디버깅부터 Yeoman을 사용한 실질적 코딩과 개발까지 가능하다.

3.7.1 디버깅 도구

자바스크립트를 코딩할 때 웹 브라우저에서 코드를 디버깅하는 일은 당연한 일이 되었다. 이 사실을 빨리 받아들일수록 개발자로서는 더욱 편해진다. 다행히 Firebug가 없던 당시와는 달리 모든 것이 엄청나게 발전했다. 이제 웹 브라우저 대부분에는 코드를 단계적으로 검토하고, 에러를 분석하며, 애플리케이션 상태를 파악할 수 있는 기능이 있다(크롬과 IE에는 Developer Tools가 있으며, 파이어폭스

10 역자 주_ 매개변수로 전달하는 데 뒷부분에 배열을 전달하는 것으로, 4장에 주입할 때 배열 형태로 작성하는 예제 코드가 있다.

와 크롬에서는 Firebug를 사용할 수 있다).

다음은 애플리케이션을 디버깅할 때 도움되는 몇 가지 추가된 조언이다.

- 디버깅하기 전에는 반드시 자신의 소스 코드와 모든 종속물을 최소화하지 않은 것으로 준비하자. 그래야 변수명을 더욱 알아보기 쉽고 행 번호와 실제 유용한 정보 및 디버깅 기능을 사용할 수 있다.
- 소스 코드를 HTML 파일에 인라인으로 넣지 말고 개별 자바스크립트 파일로 유지하자.
- 중단점breakpoint을 사용하면 편리하다. 중단점을 설정하면 애플리케이션, 모델, 주어진 시점 사이의 모든 것의 상태를 검사할 수 있다.
- 'Pause on all exceptions(모든 예외 발생 시에 중단)' 옵션은 요즘의 웬만한 개발자 도구에 내장돼 있으며 아주 유용하다. 디버거는 예외가 발생하면 중지되고 예외를 발생시킨 행을 형광펜 효과로 강조한다.

3.7.2 Batarang

Batarang[11]은 크롬의 확장기능으로, AngularJS 기능을 구글 크롬에 내장된 Developer Tools 솔루션에 넣은 것이다. Batarang을 크롬으로 추가하면, 크롬의 Developer Tools 패널에 AngularJS라는 탭이 하나 추가된다.

AngularJS 애플리케이션의 현 상태가 어떤지 궁금해한 경험이 있는가? 각각의 모델, 스코프scope, 변수에 현재 무엇이 들어 있을까? 내가 작성한 애플리케이션의 성능은 어떨까? 아직 이런 궁금증을 겪어보지 않았더라도 언젠가는 하게 될 것이라 장담한다. 이런 것이 궁금할 때 Batarang을 사용하면 그 모든 궁금증을 풀 수 있다.

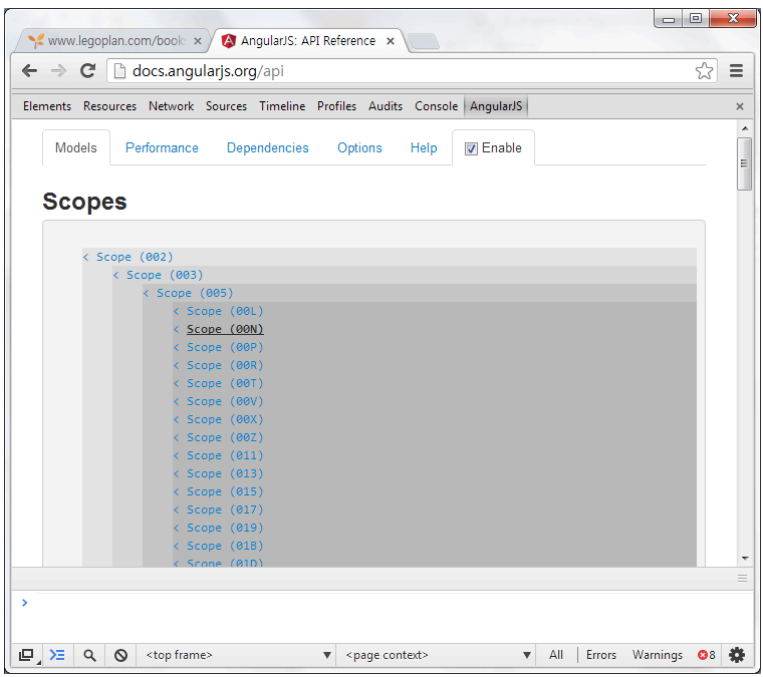
Batarang에는 주요 기능 4가지가 있다.

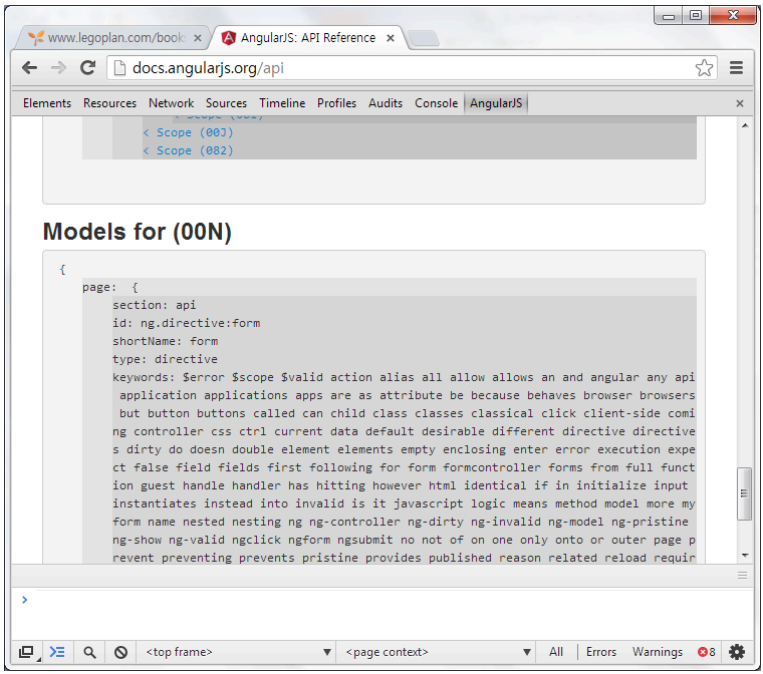
11 http://bit.ly/batarangjs

Model 탭

Batarang을 사용하면 스코프를 자세히 관찰할 수 있다. 그림 3-2와 같이 스코프가 어떤 식으로 포함관계를 형성하고 있는지, 모델이 스코프에 어떻게 연결되어 있는지를 확인할 수 있다. 심지어 스코프의 실시간 변경과 애플리케이션에 반영된 결과도 볼 수도 있다. 굉장히 편리한 도구다.

그림 3-2 Batarang의 모델 트리

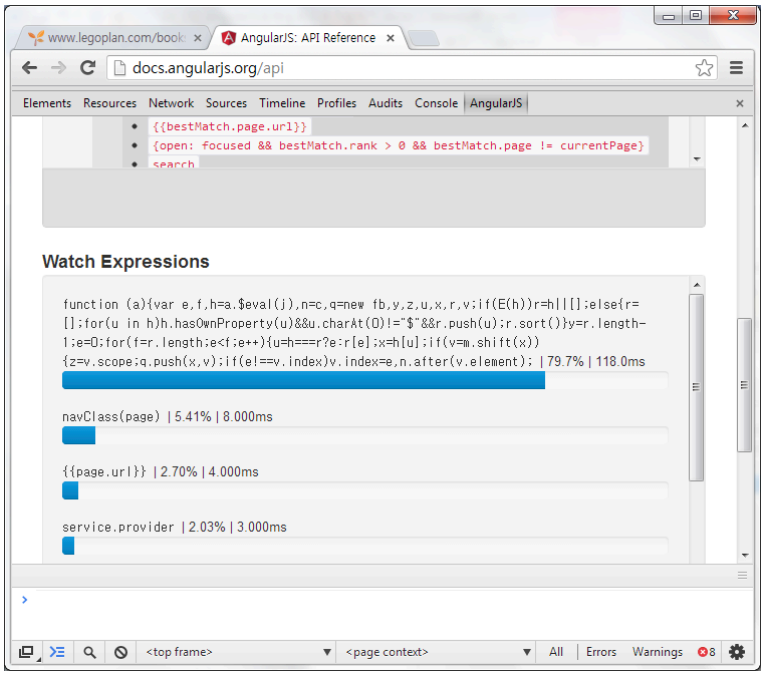




Performance 탭

Performance 탭은 반드시 따로 띄워야 한다. 왜냐하면 일부 특수한 자바스크립트 코드를 애플리케이션에 주입하기 때문이다. Performance 탭에서는 그림 3-3과 같이 각종 스코프와 모델을 보면서, 각 스코프에 있는 모든 감시 표현식의 성능을 산출할 수 있다. Performance 탭에서는 애플리케이션을 사용하면서 업데이트를 할 수도 있으므로 실시간으로 디버깅이 가능하다.

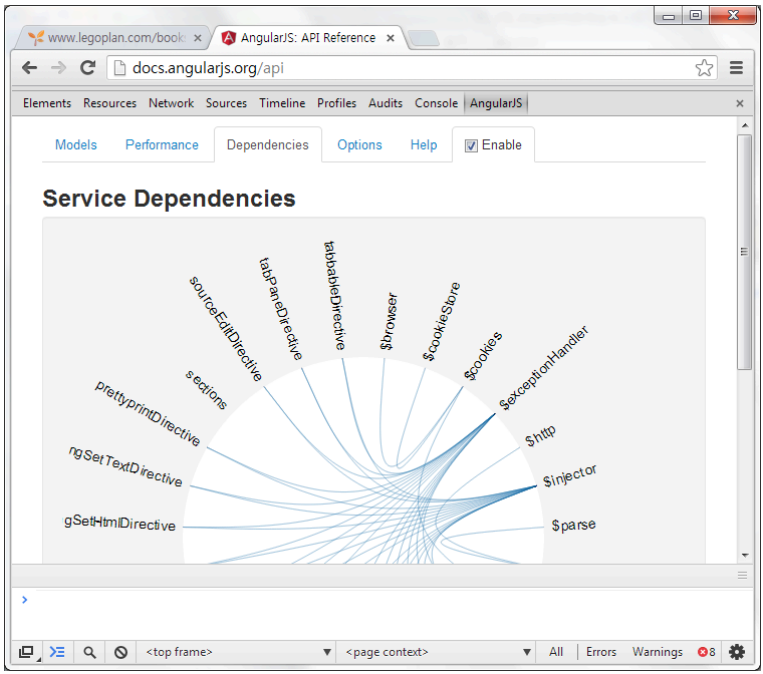
그림 3-3 Batarang의 Performance 탭



서비스 종속물

간단한 애플리케이션은 컨트롤러와 서비스에 종속물이 한두 개 정도면 충분할 것이다. 그러나 실무에 사용하는 애플리케이션이라면, 서비스 종속물이 너무 많아서 적절한 도구를 사용하지 않고는 관리하기 힘들 수도 있다. 바로 이럴 때 Batarang을 사용하면 이런 간격이 메워진다. Batarang은 그림 3-4와 같이 간결하고 깔끔한 방식으로 서비스 종속물 차트를 시각적으로 표시해주기 때문이다.

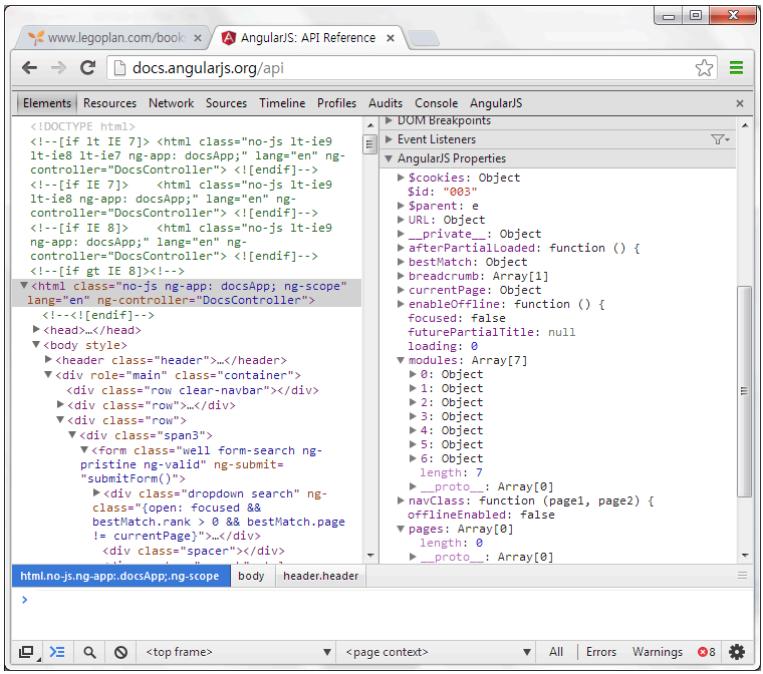
그림 3-4 Batarang의 서비스 종속물 관계 차트



요소 속성과 콘솔 접근

AngularJS 애플리케이션의 HTML 템플릿 코드를 디버깅할 때, Elements 탭에서 AngularJS Properties 섹션을 찾을 수 있다. 이 탭에서는 해당 요소의 스코프에 연결된 모델을 검사할 수 있다. 그리고 해당 요소의 스코프를 콘솔에 공개할 수도 있어서, 콘솔의 $scope 변수를 통해 접근할 수 있다. 이것은 그림 3-5를 보면 알 수 있다.

그림 3-5 Batarang에서 AngularJS 속성 보기



3.8 개발 흐름을 최적화하는 도구 Yeoman

웹 애플리케이션을 개발할 때 작업 흐름 최적화를 보조해주는 도구가 몇 가지 있다. 앞부분에서 설명했던 Yeoman이 바로 그런 도구며, 다음과 같은 놀라운 기능이 있다.

- 매우 빠른 스캐폴딩scaffolding
- 내장된 미리보기 서버
- 통합 패키지 관리
- 깔끔한 빌드 처리
- PhantomJS를 사용한 단위 테스트

Yeoman은 AngularJS와 다양한 측면에서 적절하게 연동되는데, 바로 그 이유 때문에 모든 AngularJS 프로젝트에 Yeoman을 사용하길 권한다. Yeoman을 사용하면 개발의 어떤 면이 편의해지는지 살펴보자.

3.8.1 Yeoman 설치하기

Yeoman의 설치 절차는 다소 복잡하지만, 설치를 간편하게 해주는 스크립트를 사용하면 된다.

매킨토시나 리눅스 사용자는 다음 명령을 실행하면 된다.

```
curl -L get.yeoman.io | bash
```

그리고 출력되는 설명에 따라 Yeoman을 설치하면 된다.

윈도우 사용자이거나 다른 문제에 부딪혔을 땐 https://github.com/yeoman/yeoman/wiki/Manual-Install에 나와있는 설명을 참고하자.

3.8.2 AngularJS 프로젝트 시작하기

앞에서 설명했지만 아무리 간단한 AngularJS 프로젝트라도 템플릿, 기본 컨트롤러, 라이브러리 종속물에서부터 체계적인 구조 형성에 필요한 모든 것에 이르기까지 상당량의 준비작업이 필요하다. 개발자가 그 작업을 수동으로 할 수도 있겠지만, 기왕이면 Yeoman을 사용해서 자동으로 처리하자.

프로젝트 폴더를 생성하자. Yeoman은 프로젝트명과 같은 이름의 폴더를 생성한다. 그리고 다음과 같은 명령을 실행하자.

```
yeoman init angular
```

앞의 명령은 "3.1 프로젝트의 구성"에서 설명했던 구조를 생성하며, 여기엔 루트와 단위 테스트 등을 렌더링하는 뼈대가 포함된다.

3.8.3 서버 구동하기

Yeoman을 사용하지 않는다면 프론트엔드 코드를 제공할 HTTP 서버를 만들어야 한다. 그러나 Yeoman을 사용하면 몇몇 장점이 더해져서 미리 설정된 내장 서버가 생성된다. 서버를 시작하는 명령은 다음과 같다.

```
yeoman server
```

앞의 명령으로 인해 코드를 제공할 웹 서버가 시작됨과 동시에, 웹 브라우저가 자동으로 열리고 애플리케이션을 수정할 때마다 웹 브라우저가 새로고침된다.

3.8.4 새 루트, 뷰, 컨트롤러 추가하기

Angular에 새 루트를 추가하는 과정은 다음과 같다.

- New Controller 자바스크립트 파일을 index.html 안에 링크시킨다.
- AngularJS 모듈에 정확한 루트를 추가한다.
- 템플릿 HTML 파일을 작성한다.
- 단위 테스트를 추가한다.

이 모든 과정을 Yeoman에서는 다음 명령을 사용해 한 단계만으로 처리할 수 있다.

```
yeoman init angular:route 루트명
```

따라서 "yeoman init angular:route home"을 실행하면 Yeoman은 다음과 같은 작업을 자동 수행한다.

- 컨트롤러 뼈대인 home.js 파일을 app/scripts/controllers 폴더에 생성한다.
- 명세 뼈대인 home.js 파일을 test/specs/controllers 폴더에 생성한다.
- 템플릿인 home.html 파일을 app/views 폴더에 추가한다.
- 메인 애플리케이션 모듈(app/scripts/app.js)에 홈 루트를 연결한다.

이 모든 과정이 단 하나의 명령으로 이뤄진다.

3.8.5 테스트하기

Karma를 사용하면 테스트를 시작하고 실행하기가 얼마나 쉬운지 앞에서 살펴봤다. 마지막에 단위 테스트 전부를 실행하기 위해 두 개의 명령이 필요했다.

Yeoman을 사용하면 더 쉽게 할 수 있다. Yeoman을 사용해서 파일을 생성할 때마다 내용을 채울 테스트 스텁이 자동 생성된다. Karma를 설치했으면 Yeoman을 사용해 다음과 같이 간단한 명령으로 테스트를 실행할 수 있다.

```
yeoman test
```

3.8.6 프로젝트 빌드하기

애플리케이션의 제품화 버전을 제작하는 일은 힘겨울 수도 있고, 그렇지 않더라도 최소한 여러 단계를 거쳐야 한다. Yeoman을 사용하면 개발자는 다음과 같은 것이 가능하므로 그 복잡한 단계 중 일부는 줄어든다.

- 자스크립트를 한 파일에 모두 몰아넣을 수 있다.
- 파일의 버전을 관리할 수 있다.
- 이미지를 최적화할 수 있다.
- 애플리케이션 캐시 매니페스트를 생성할 수 있다.

이 모든 장점은 다음과 같은 하나의 명령만으로 얻을 수 있다.

```
yeoman build
```

Yeoman은 매니페스트를 아직 지원하지 않지만 조만간 가능해질 것이다.

3.9 AngularJS와 RequireJS를 통합하기

개발 환경을 제대로 설정하는 일은 빠르면 빠를수록 훨씬 더 쉬워진다. 개발 단계 후반부에서 개발 환경을 바꾸려면 더 많은 파일을 수정해야 하기 때문이다. 종속물 관리와 배포 패키지 제작이 웬만큼 규모가 큰 프로젝트에선 제일 큰 골치덩어리다.

자바스크립트를 사용해 자신의 개발 환경을 설정하기란 상당히 힘들다. Ant 빌드를 관리하고, 파일을 집중화하는 스크립트를 작성하며, 스크립트를 최소화하는 등의 작업이 필요하기 때문이다. 다행히 최근에 RequireJS 같은 도구가 등장했다. RequireJS를 사용하면 자바스크립트 종속물을 정의하고 관리할 수 있으며, 그런 종속물을 더 간단한 빌드 과정에 연결할 수도 있다. 코드가 실행되기 전에 모든 종속물이 로딩됐는지 확인하는 이러한 비동기적 로딩 관리 도구를 사용하면 실질적인 기능 개발에만 전념할 수 있다.

다행히 AngularJS는 RequireJS와 손쉽게 연동되어, 둘의 장점을 취할 수 있다. 이러한 목적으로 체계적이고 따라하기 쉬운 샘플을 설정해보자.

프로젝트 구성을 보자. 앞서 설명했던 뼈대와 비슷한데 차이가 조금 있다.

1. app: 이 폴더에는 사용자에게 표시되는 모든 애플리케이션 코드가 들어 있다. HTML, 자바스크립트, CSS, 부속 라이브러리 등이다.
   a. /styles: CSS/LESS 파일이 전부 들어 있다.

b. /images: 프로젝트 이미지가 들어 있다.

c. /scripts: 주요 AngularJS 코드베이스다. 이 폴더엔 부트스트랩 코드와 RequireJS와의 주된 통합기능도 들어 있다.

- /controllers: AngularJS 컨트롤러가 들어 있다.
- /directives: AngularJS 지시어가 들어 있다.
- /filters: AngularJS 필터가 들어 있다.
- /services: AngularJS 서비스가 들어 있다.

d. /vendor: 필요한 라이브러리(Bootstrap, RequireJS, jQuery)가 들어 있다.

e. /views: 프로젝트의 뷰와 컴포넌트에 해당하는 HTML이 들어 있다.

2. config: 단위 테스트와 시나리오 테스트에 사용할 Karma 환경설정 파일이 들어 있다.
3. test: 애플리케이션의 단위 테스트와 시나리오(통합) 테스트가 들어 있다.
   a. /spec: app 디렉터리 안의 자바스크립트 폴더의 구조를 미러링하는 단위 테스트가 들어 있다.
   b. /e2e: 서버-클라이언트 간 시나리오 명세가 들어 있다.

우선 RequireJS가 로딩하는 main.js 파일이 필요한데, 이 파일은 app 폴더에 있다. main.js 파일이 로딩되면 나머지 모든 종속물이 로딩되기 시작한다. 여기서 살펴볼 자바스크립트 프로젝트는 필자가 작성할 코드에 jQuery와 트위터 부트스트랩을 사용한다.

소스 코드 3-2 RequireJS 환경설정을 정의하는 requirejs-app/app/scripts/main.js 파일

```
require.config({
    paths : {
        angular : 'vendor/angular.min',
        jquery : 'vendor/jquery',
        domReady : 'vendor/require/domReady',
```

```
        twitter : 'vendor/bootstrap',
        angularResource : 'vendor/angular-resource.min'
    },
    shim : {
        'twitter/js/bootstrap' : {
            deps : [ 'jquery/jquery' ]
        },
        angular : {
            deps : [ 'jquery/jquery', 'twitter/js/bootstrap' ],
            exports : 'angular'
        },
        angularResource : {
            deps : [ 'angular' ]
        }
    }
});

require([ 'app',
    // 이것은 트위터 부트스트랩이 아니라
    // AngularJS 부트스트랩이다.
    'bootstrap', 'controllers/mainControllers',
    'services/searchServices', 'directives/ngbkFocus'
    // 개발자가 추가하는 개별 컨트롤러, 서비스, 지시어, 필터 파일을
    // 여기에서 불러와야 한다.이것은 수작업으로 관리해야 한다.
], function(angular, app) {
    'use strict';

    app.config([ '$routeProvider', function($routeProvider) {
        // 여기에 루트를 정의할 것
    } ]);
});
```

그다음 app.js 파일을 정의하자. 이 파일은 AngularJS 애플리케이션을 정의하며 필자가 정의하는 모든 컨트롤러, 서비스, 필터, 지시어에 종속됨을 알린다. RequireJS 종속물 열거 부분에 언급된 파일들은 잠시 후에 살펴보자.

RequireJS 종속물 열거는 자바스크립트의 블록 import문이라고 생각하면 된다. 즉, 블록 안의 함수는 열거된 모든 종속물이 로딩될 때까지 실행되지 않는다.

그리고 불러올 지시어, 서비스, 필터를 RequireJS에게 개별적으로 전달하지 않아도 된다. 왜냐하면 이 프로젝트는 그런 구조로 되어있지 않기 때문이다. 컨트롤러, 서비스, 필터, 지시어마다 모듈이 하나씩 있으므로 종속물만 가지고도 컨트롤러, 서비스, 필터를 충분히 정의할 수 있다.

소스 코드 3-2 AngularJS 애플리케이션을 정의하는 app/scripts/app.js 파일

```
define(['angular', 'angularResource', 'controllers/controllers',
        'services/services', 'filters/filters',
        'directives/directives'], function (angular) {
    return angular.module( ' MyApp ' , ['ngResource', 'controllers', 'services',
                                        'filters', 'directives']);
});
```

DOM이 준비되기까지 대기했다가(RequireJS의 domReady 플러그인을 사용해서) AngularJS에 진행하라고 명령하는 bootstrap.js 파일은 미리 준비되어 있다.

```
// AngularJS에게 진행하고 DOM이 모두 로딩되면 부트스트랩을 수행하라고
// 명령하는 app/scripts/bootstrap.js 파일
define(['angular', 'domReady'], function(angular, domReady) {
    domReady(function() {
        angular.bootstrap(document, [ ' MyApp ' ]);
    });
```

```
});
```

부트스트랩을 애플리케이션과 분리하면 또 한가지 장점을 얻을 수 있다. 즉, 테스트할 때는 mainApp 대신 모크 객체인 mockApp를 사용할 수도 있다. 예를 들어, 개발에 사용하는 서버가 불안정하다면 모든 $http 요청을 가짜 데이터로 대체하는 페이크 객체인 fakeApp을 작성하면 걱정 없이 개발할 수 있다. 그렇게 하면 fakeBootstrap과 fakeApp을 애플리케이션으로 끼워 넣을 수 있다.

이제 app 폴더에 들어 있는 메인 index.html 파일의 내용은 다음과 같을 것이다.

```
<!DOCTYPE html>
<html>
<!-- AngularJS의 부트스트랩을 수동으로 실시할 것이므로 ng-app 지시어를 여기에
추가하면 안된다. -->
<head>
   <title>내 AngularJS 애플리케이션</title>
   <meta charset="utf-8" />

   <link rel="stylesheet" type="text/css" href="styles/bootstrap.min.css">
   <link rel="stylesheet" type="text/css"
        href="styles/bootstrap-responsive.min.css">

   <link rel="stylesheet" type="text/css" href="styles/app.css">

</head>
<body class="home-page" ng-controller="RootController">

   <div ng-view></div>

   <script data-main="scripts/main" src="lib/require/require.min.js"></script>
```

```
</body>
</html>
```

이제 js/controllers/controllers.js 파일을 살펴보자. 이 파일은 다음과 같이 js/directives/directives.js, js/filters/filters.js, js/services/services.js 파일과 거의 비슷할 것이다.

```
define([ 'angular' ], function(angular) {
    'use strict';
    return angular.module('controllers', []);
});
```

RequireJS 종속물을 구성한 방식 때문에 모든 파일은 반드시 Angular 종속물이 완전히 로딩된 후에 실행된다. 파일들 각각은 AngularJS 모듈을 정의한다. 이렇게 정의된 모듈은 그 정의에 추가하는 개별 컨트롤러, 지시어, 필터, 서비스에 의해 사용된다.

2장의 focus 지시어와 마찬가지로, 지시어 정의를 살펴보자.

소스 코드 3-3 ngbkFocus.js

```
define([ 'directives/directives' ], function(directives) {
    directives.directive(ngbkFocus, [ '$rootScope', function($rootScope) {
        return {
            restrict : 'A',
            scope : true,
            link : function(scope, element, attrs) {
                element[0].focus();
            }
        };
```

```
    } ]);
});
```

지시어 자체는 꽤 단순하지만 어떤 일이 일어나는지 좀 더 자세히 살펴보자. ngbkFocus.js 파일에 사용되는 RequireJS의 shim을 보면 ngbkFocus.js 파일이 모듈 선언 파일인 directives/directives.js를 사용함을 알 수 있다. ngbkFocus.js 파일은 주입된 directives 모듈을 사용해 자체적인 지시어 선언을 추가한다. 여러 개의 지시어를 사용하도록 선택할 수도 있고 파일 당 하나만 사용하게 선택할 수 있다. 이것은 전적으로 개발자 마음이다.

중요한 참고사항 하나는, 서비스를 끌어오는 컨트롤러가 있으면(RootController가 UserService를 사용하여 UserService가 주입된다고 가정) 다음과 같이 RequireJS에도 반드시 파일 종속물을 정의해야 한다.

```
define(['controllers/controllers', 'services/userService'],
    function(controllers) {
        controllers.controller('RootController', ['$scope', 'UserService',
        function($scope, UserService) {
            // 여기에 필요한 기능을 작성
        };
    }]);
});
```

이것이 기본적으로 전체 소스 폴더 구조가 형성되는 원리다. 그러나 이 구조가 테스트에 어떤 식으로 영향을 미칠까? 좋은 질문이다. 때마침 그 답을 설명할 참이었으니까.

다행히 Karma는 RequireJS를 지원한다. 따라서 Karma 최신 버전만 설치하면

된다. 설치 명령은 npm install -g karma이다. Karma를 설치했으면 단위 테스트를 위한 Karma 환경설정도 약간 변경된다. 앞에서 정의했던 프로젝트 구조에서 실행할 단위 테스트를 설정하는 방법은 다음과 같다.

소스 코드 3-4 config/karma.conf.js

```
// 파일을 가져올 때 사용할 기준 경로를 지정
// (이 예제에선 프로젝트의 루트임)
basePath = '../';

// 브라우저에 로딩할 파일과 패턴을 배열로 정의
files = [
    JASMINE,
    JASMINE_ADAPTER,
    REQUIRE,
    REQUIRE_ADAPTER,

    // !! RequireJS의 'paths' 설정에 들어 있는 모든 라이브러리를
    // 여기에 넣음 (included: false).
    // 이 파일들은 전부 테스트 실행에 필수적이지만,
    // Karma는 이 파일들을 로딩하지 않도록 직접적으로 명령 받는다.
    // 왜냐하면 메인 모듈이 로딩될 때 RequireJS가 로딩할 것이기 때문이다.
    {pattern: 'app/scripts/vendor/**/*.js', included: false},

    // !! 모든 소스와 테스트 모듈 (included: false)
    {pattern: 'app/scripts/**/*.js', included: false},
    {pattern: 'app/scripts/*.js', included: false},
    {pattern: 'test/spec/*.js', included: false},
    {pattern: 'test/spec/**/*.js', included: false},

    // !! 테스트 메인은 모듈을 맨 나중에 필요로 한다.
    'test/spec/main.js'
```

```
];

// 배제할 파일 배열을 정의
exclude = [];

// 사용할 테스트 결과 리포터를 정의
// 가질 수 있는 값: dots || progress
reporter = 'progress';

// 웹 서버 포트
port = 8989;

// 클라이언트 실행 포트
runnerPort = 9898;

// 리포터와 로그 출력에 색상 가용화/불용화
colors = true;

// 로그 수준
logLevel = LOG_INFO;

// 파일을 감시하다가 파일이 변경될 때마다 테스트를 실행하는 기능의 가용화/불용화
autoWatch = true;

// 다음 브라우저를 시작함. 현재 사용 가능한 브라우저:
// - Chrome
// - ChromeCanary
// - Firefox
// - Opera
// - Safari
// - PhantomJS
// - IE 윈도우일 경우
browsers = ['Chrome'];
```

```
// 연속 통합 모드
// true를 지정하면, 브라우저를 캡처하고 테스트를 실행한 후 종료한다
singleRun = false;
```

앞의 코드에 사용한 종속물 정의 형식이 약간 다르다. 'included: false'는 매우 중요하다. REQUIRE_JS와 어댑터에도 종속물을 추가했다. 이 모든 것이 돌아가게 하기 위해 마지막으로 테스트를 구동하는 main.js 파일을 다음과 같이 작업해야 한다.

소스 코드 3-5 test/spec/main.js

```
require.config({
    // !! Karma가 제공하는 파일은 '/base' 경로에 들어 있다.
    // (이 예제에서는 프로젝트 루트인 /자신의프로젝트/app/js 폴더에 들어 있다)
   baseUrl: '/base/app/scripts',
   paths: {
      angular: 'vendor/angular/angular.min',
      jquery: 'vendor/jquery',
      domReady: 'vendor/require/domReady',
      twitter: 'vendor/bootstrap',
      angularMocks: 'vendor/angular-mocks',
      angularResource: 'vendor/angular-resource.min',
      unitTest: '../../../base/test/spec'
   },
   // shim을 사용해 AMD 이외의 라이브러리를 로딩하는 예
   // (Backbone, jQuery와 마찬가지로)
   shim: {
      angular: {
         exports: 'angular'
      },
      angularResource: { deps:['angular']},
```

```
        angularMocks: { deps:['angularResource']}
    }
});

// DOM이 모두 준비되면 karma를 시작
require([
    'domReady',
    // 개별 테스트 파일 각각을 이 목록에 추가해야 실행된다.
    // 다시 말하지만, 이것은 수동으로 관리해야 한다.
    'unitTest/controllers/mainControllersSpec',
    'unitTest/directives/ngbkFocusSpec',
    'unitTest/services/userServiceSpec'
    ], function(domReady) {
    domReady(function() {
        window.__karma__.start();
    });
});
```

그럼 이 설정을 가지고 다음 명령을 실행할 수 있다.

```
karma start config/karma.conf.js
```

그리고 테스트를 실행하면 된다. 물론 단위 테스트를 작성할 땐 이것과는 약간 다르다. RequireJS가 지원되는 모듈이기도 해야 하므로 다음 샘플 테스트를 살펴보자.

소스 코드 3-6 test/spec/directives/ngbkFocus.js

```
define([ 'angularMocks', 'directives/directives', 'directives/ngbkFocus' ],
    function() {
        describe('ngbkFocus Directive', function() {
```

```
            beforeEach(module('directives'));

            // 이것들은 각 명세(각각의 it() 함수)에 앞서 초기화되고 재사용된다.
            var elem;
            beforeEach(inject(function($rootScope, $compile) {
                elem = $compile('<input type="text" ngbk-focus>')($rootScope);
            }));

            it('즉시 포커스를 획득해야 함', function() {
                expect(elem.hasClass('focus')).toBeTruthy();
            });
        });
    }
);
```

모든 테스트는 다음 작업을 수행하게 된다.

1. angular, angularResource을 사용할 수 있는 angularMocks 을 끌어온다.
2. 고수준 모듈(지시어용 directives, 컨트롤러용 controllers 등)을 끌어온 후, 실제로 테스트하는(loadingIndicator) 개별 파일을 끌어온다.
3. 테스트에 다른 서비스나 컨트롤러가 필요하다면, RequireJS 종속물도 정의한 후 AngularJS에게 그에 대한 정보를 전달해야 한다.

이러한 방법은 어느 테스트에든 사용할 수 있다. 다행히 RequireJS 방식은 서버-클라이언트 간 테스트에 전혀 영향을 미치지 않으므로, 이제껏 해왔던 방법으로 그냥 하면 된다. 애플리케이션을 실행하는 서버가 http://localhost:8000에서 실행된다는 가정하에, 샘플 환경설정 파일은 다음과 같다.

```
// 파일을 가져올 때 사용될 기준 경로를 지정
```

```
// (이 파일에선 프로젝트의 루트로 지정)
basePath = '../';

// 브라우저에 로딩할 파일 및 패턴 배열을 정의
files = [ ANGULAR_SCENARIO, ANGULAR_SCENARIO_ADAPTER, 'test/e2e/*.js' ];

// 배제할 파일 배열을 정의
exclude = [];

// 테스트 결과 리포터를 정의
// 지정 가능한 값: dots || progress
reporter = 'progress';

// 웹 서버 포트
port = 8989;

// 클라이언트 실행 포트
runnerPort = 9898;

// 리포터와 로그 출력 색상 가용화/불용화
colors = true;

// 로그 수준
logLevel = LOG_INFO;

// 파일을 감시하다가 변경이 발생할 때마다 테스트를 실행하는 기능을 가용화/불용화
autoWatch = true;

urlRoot = '/_karma_/';

proxies = {
    '/' : 'http://localhost:8000/'
};
```

```
// 시작할 브라우저를 지정
browsers = [ 'Chrome' ];

// 연속 통합 모드
// 값을 true로 지정하면 브라우저를 캡처하고 테스트를 실행한 후 종료된다.
singleRun = false;
```

4 | AngularJS 애플리케이션 분석하기

2장에서는 AngularJS의 주로 사용되는 기능 몇 가지를 알아보았고, 3장에서는 개발을 위한 구조를 어떻게 잡아야 하는지 알아보았다. 이 장에서는 앞장에서처럼 개별 기능을 깊게 설명하지 않고, 간단한 실무 애플리케이션을 살펴볼 것이다. 샘플 예제를 가지고 지금까지 설명했던 모든 요소가 실제로 어떻게 조합되어 실무 애플리케이션을 형성하는지 감을 잡을 수 있을 것이다.

애플리케이션 전체를 앞부분과 중간 부분에서 다루지 않고, 일단 애플리케이션의 한 부분만 소개한 후 중요하거나 관련된 부분들은 차근차근 설명하겠다.

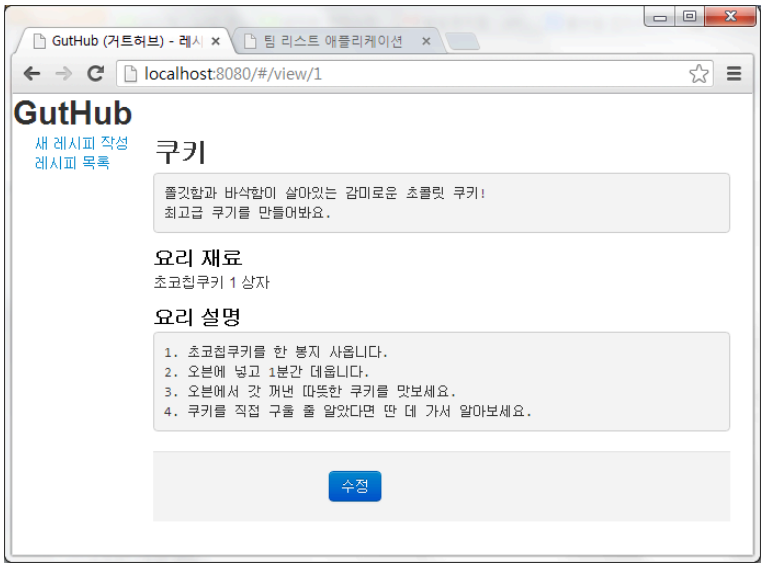
4.1 애플리케이션

이 장에서 제작해볼 GutHub는 간단한 레시피 관리 애플리케이션이다. GutHub는 아주 맛있는 레시피를 저장할 수 있게 설계돼 있다. 이 애플리케이션을 통해 AngularJS 애플리케이션의 다양한 부분을 엿볼 수 있을 것이다. GutHub 애플리케이션의 특징은 다음과 같다.

- 2단 레이아웃이다.
- 내비게이션 바가 왼쪽에 있다.
- 새 레시피를 작성할 수 있다.
- 기존 레시피 목록을 탐색할 수 있다.

메인 뷰는 오른쪽에 있으며 URL에 따라 레시피 목록, 선택한 레시피의 상세 설명, 새 레피시를 추가하거나 기존 레시피를 편집할 수 있는 폼 중 하나로 내용이 바뀐다. 이 애플리케이션의 모습은 그림 4-1과 같다.

그림 4-1 GutHub – 간단한 레시피 관리 애플리케이션



이 애플리케이션의 완전한 소스 코드는 GitHub 저장소의 chapter4/guthub에서 받을 수 있다.

4.2 모델, 컨트롤러, 템플릿의 관계

GutHub 애플리케이션에 본격적으로 들어가기에 앞서, 우선 애플리케이션의 세 부분이 어떤 식으로 연동되는지 알아보고 각 부분을 어떻게 생각하면 될지 알아보보자.

"모델은 사실이다." 이 문장을 몇 회 되뇌어보자. 완전한 애플리케이션은 어느 뷰들을 표시할지, 뷰들에 무엇을 표시할지, 무엇을 저장할지 등과 같은 모든 사안이 모델 위주로 결정되고 진행된다. 따라서 작성하는 객체에는 어떤 속성이 들어갈지, 어떤 식으로 서버에서 가져오고 서버로 저장할지 등 모델에 대해 생각할 시간을

가져야 한다. 데이터 바인딩을 통해 뷰는 자동으로 업데이트되므로 항상 중점은 모델에 둬야 한다.

컨트롤러에는 모델을 어떻게 가져올지, 모델에서 어떤 종류의 작업들을 수행할지, 뷰가 모델에서 얻어야 할 정보가 어떤 종류인지, 필요한 정보를 얻으려면 모델을 어떻게 변형해야 할지 같은 비즈니스 로직이 들어간다. 유효성 검사하기, 서버 호출하기, 뷰를 올바른 데이터로 부트스트랩하기, 그리고 그 사이의 거의 모든 작업이 컨트롤러에 들어간다.

템플릿은 모델이 표시될 방식, 사용자가 애플리케이션을 조작할 방식을 표현한다. 템플릿의 기능은 거의 다음과 같은 것들로 한정된다.

- 모델을 표시한다.
- 클릭, 인풋 필드 등 사용자가 애플리케이션을 조작할 방식을 정의한다.
- 애플리케이션에 스타일을 입히고, 일부 요소를 언제(예: 마우스가 올라왔을 때) 어떻게(예: 보일지 감출지) 표시할지 판단한다.
- 입력 및 출력 데이터를 필터링하고 형식화한다.

Angular의 템플릿은 꼭 모델-뷰-컨트롤러 설계 패러다임을 구성하는 뷰는 아니다. 실행되는 템플릿이 컴파일을 거치면 뷰가 되는데, 이 뷰는 템플릿과 모델의 혼합물이다.

템플릿 안에는 비즈니스 로직이나 기능이 절대로 들어가선 안된다. 비즈니스 로직이나 기능 정보는 반드시 컨트롤러에만 넣어야 한다. 템플릿을 간결하게 유지하면 관심사를 적절히 분리할 수 있으며, 단위 테스트만 가지고 대부분의 코드를 테스트할 수 있다. 템플릿은 시나리오 테스트를 사용해서 테스트해야 한다.

그런데 DOM 조작은 어디로 갔을까? DOM 조작은 사실 컨트롤러나 템플릿에 들어가지 않는다. DOM 조작은 AngularJS 지시어에 들어 있다. 그러나 간혹 코드 중복

을 피하기 위해 DOM 조작이 들어 있는 서비스를 통해 사용될 수도 있다. GutHub 예제에서도 그러한 예를 다루겠다.

각설하고 곧장 애플리케이션 제작을 실습해보자.

4.3 모델

이 애플리케이션에서는 모델을 아주 간단히 작성하도록 하자. 여러 레시피가 있고, 그 레시피들은 Guthub 애플리케이션 전체에서 하나뿐인 모델 객체다. 나머지는 전부 모델 객체에서 시작한다.

각 레시피에는 다음과 같은 속성이 있다.

- 서버에 저장될 때의 ID
- 이름
- 간략한 설명
- 조리법
- 특집 레시피인지 아닌지 여부
- 양, 단위, 이름으로 구성된 재료 배열

이게 전부이다. 매우 간단하다. 애플리케이션의 모든 구성요소는 이 간단한 모델을 기반으로 한다. 사용할 샘플 레시피 코드는 다음과 같다. 이것은 그림 4-1에서 본 레시피다.

```
{
   "id": "1",
   "title": "쿠키",
   "description": "쫄깃함과 바삭함이 살아있는 감미로운 초콜릿 쿠키." +
      " 최고급 쿠기를 만들어 봐요.",
   "ingredients": [
      {
         "amount": "1",
         "amountUnits": "봉지",
```

```
            "ingredientName": "초코칩쿠키"
        }
    ],
    "instructions": "1. 초코칩쿠키를 한 봉지 사옵니다.\n" +
        "2. 오븐에 넣고 1분간 데웁니다.\n" +
        "3. 오븐에서 갓 꺼낸 따뜻한 쿠키를 맛보세요.\n" +
        "4. 쿠키를 직접 구울 줄 알았다면 딴 데 가서 알아보세요."
}
```

다음 절에서 이 간단한 모델을 토대로 더 복잡한 UI 기능을 어떻게 구현하는지 살펴보자.

4.4 서비스, 지시어, 컨트롤러

이제 필자는 맛있는 Guthub 애플리케이션의 살코기를 살짝 베어 물었다. 우선 지시어와 서비스 코드를 살펴보고 그것들이 무슨 기능을 하는지 알아본 후, 이 애플리케이션에 필요한 각종 컨트롤러를 살펴보자.

4.4.1 서비스

소스 코드 4-1 app/scripts/services/services.js

```
'use strict';
var services = angular.module('guthub.services', [ 'ngResource' ]);

services.factory('Recipe', [ '$resource', function($resource) {
    return $resource('/recipes/:id', {
        id : '@id'
    });
} ]);
```

```
services.factory('MultiRecipeLoader', [ 'Recipe', '$q', function(Recipe, $q) {
    return function() {
        var delay = $q.defer();
        Recipe.query(function(recipes) {
            delay.resolve(recipes);
        }, function() {
            delay.reject('레시피를 가져올 수 없습니다');
        });
        return delay.promise;
    };
} ]);

services.factory('RecipeLoader', [
    'Recipe',
    '$route',
    '$q',
    function(Recipe, $route, $q) {
        return function() {
            var delay = $q.defer();
            Recipe.get({
                id : $route.current.params.recipeId
            }, function(recipe) {
                delay.resolve(recipe);
            }, function() {
                delay.reject($route.current.params.recipeId
                    + '의 레시피를 가져올 수 없습니다');
            });
            return delay.promise;
        };
    }
]);
```

먼저 서비스부터 살펴보자. "2.4 모듈을 사용해 종속물 체계화하기"에서 서비스에 대해 간략히 소개했다. 여기서는 좀 더 구체적으로 알아보자.

Guthub의 서비스 파일 안에서 AngularJS 서비스 3개를 인스턴스화하자.

레시피 서비스는 하나인데, 이 서비스는 Angular Resource라는 것을 반환한다. Angular Resource는 REST 기반의 서버를 겨냥하는 REST 기반의 리소스다. Angular Resource는 하위 계층의 $http 서비스를 캡슐화해서 개발자는 자신의 코드 안에서 객체만을 처리할 수 있다.

return $resource라는 한 줄의 코드만 있으면 이제 어느 컨트롤러에든 recipe를 인자로 넣을 수 있고, 결국 그 인자는 컨트롤러에 주입된다. 당연한 이야기지만 $resource는 guthub.services 모듈에 있는 종속물이다. 게다가 각 recipe 객체에는 다음과 같은 메서드가 들어 있다.

- Recipe.get()
- Recipe.query()
- Recipe.delete()
- Recipe.save()
- Recipe.remove()

NOTE_ 애플리케이션을 IE에서 작업할 때 Recipe.delete를 사용할 생각이라면, 그 명령 대신 Recipe[delete]() 형태로 호출해야 한다. IE에는 delete가 키워드로 내장돼 있기 때문이다.

앞에 나열한 메서드 중에서 query를 제외한 나머지는 전부 단일 레시피에 사용 가능하다. query() 메서드는 기본으로 레시피 배열을 반환한다. 코드에서 리소스를 반환하는 return $resource 행은 다음과 같은 비교적 고급 기능도 수행한다.

1. REST 기반의 리소스에 지정된 URL 안의 :id는 중요하다. Recipe.get() 같은 질의를 할 때 id 필드를 통해 객체를 전달하면, id 필드의 값이 URL 끝에 추가

된다. 즉, Recipe.get({id: 15})를 호출하면 /recipe/15가 호출된다.

2. 두 번째 객체는 어떨까? {id: @id}는? 골백번 설명하느니 코드 한 줄을 보는 편이 빠르니 다음의 간단한 예를 보자.

한 recipe 객체가 있는데 그 안에 id를 비롯한 필수 정보가 미리 들어 있다고 가정하자. 그러면 다음과 같은 코드만으로 그 필수 정보를 저장할 수 있다.

```
// existingRecipeObj에 id(값이 13이라고 하자)를 포함해
// 필요한 필드가 전부 들었다고 가정할 때
var recipe = new Recipe(existingRecipeObj);
recipe.$save();
```

앞의 코드는 /recipe/13에 POST를 요청한다. @id는 객체에서 id 필드를 선택해서 id 매개변수로 전달하라고 /recipe/13에게 명령한다. @id를 사용함으로써 몇 줄의 코드가 줄어드는 장점도 얻을 수 있다.

apps/scripts/services/services.js 파일에는 다른 서비스가 두 개 더 있다. 두 서비스 모두 로더인데, 하나는 단일 레시피를 로딩하는 RecipeLoader이고 다른 하나는 모든 레시피를 목록 형태로 로딩하는 MultiRecipeLoader이다. 두 로더 서비스는 루트를 연결할 때 사용된다. RecipeLoader와 MultiRecipeLoader는 둘 다 내부적으로 기능이 아주 비슷하다. 두 서비스의 흐름은 다음과 같다.

1. 지연 객체(deferred object) $q를 생성한다(이것은 비동기 함수들을 연쇄시킬 때 사용하는 AngularJS의 promise 인터페이스다).
2. 서버를 호출한다.
3. 서버가 값을 반환하면 지연 객체를 가져온다.
4. AngularJS의 루트 메커니즘에 사용될 promise 인터페이스를 반환한다.

여기서 잠깐_ AngularJS 세계의 약속(promise)

promise는 반환된 객체나 미래 시점에 채워질 객체를 처리하는 인터페이스이며, 기본적으로 비동기 작업이다. 내부적으로 promise는 then() 함수가 들어 있는 객체다. promise 인터페이스의 장점을 알아보기 위해 사용자의 현재 프로필을 가져오는 다음 예제 코드를 보자.

```
var currentProfile = null;
var username = 'something';

fetchServerConfig(function(serverConfig) {
    fetchUserProfiles(serverConfig.USER_PROFILES, username,
        function(profiles) {
            currentProfile = profiles.currentProfile;
        }
    );
});
```

그런데 앞의 코드와 같은 방식을 사용하면 다음과 같은 몇 가지 문제가 있다.

1. 작성한 코드에 들여쓰기가 난무한다. 특히 여러 개의 호출을 연쇄시켜야 한다면 더 심해진다.
2. 각 단계마다 수동으로 처리하지 않으면 콜백과 함수 간에 보고되는 에러를 놓칠 가능성이 높다.
3. 제일 안쪽에 있는 currentProfile을 사용해 수행할 로직을 직접적으로든 별도의 함수를 사용해서든 캡슐화해야 한다.

promise를 사용하면 이러한 문제가 없다. 방법을 설명하기 전에 앞의 상황을 promise로 구현하면 어떻게 되는지 코드부터 살펴보자.

```
var currentProfile =
    fetchServerConfig().then(function(serverConfig) {
    return fetchUserProfiles(serverConfig.USER_PROFILES, username);
}).then(function(profiles) {
    return profiles.currentProfile;
```

```
}, function(error) {
    // fetchServerConfig나 fetchUserProfiles에 들어 있는
    // 에러를 여기에서 처리
});
```

---

promise를 사용했더니 다음과 같은 장점이 생겼다.

1. 여러 개의 함수 호출을 연쇄시킬 수 있어서 끔찍한 들여쓰기 지옥을 피할 수 있다.
2. 연쇄된 함수 중에서 앞에 있는 함수 호출이 반드시 완료돼야만 그다음 함수가 호출된다.
3. 각 then() 함수는 두 인자(둘 다 함수임)를 받는다. 첫 번째 인자는 성공 시에 호출될 콜백 함수고, 두 번째 인자는 에러 시에 호출될 핸들러 함수다.
4. 연쇄된 함수 중에 에러가 있으면 그 에러가 에러 핸들러 함수의 나머지를 통해 전파된다. 따라서 어느 콜백에 들어 있는 어떤 에러든 간에 맨 마지막에 처리된다.

resolve 함수와 reject 함수는 무엇인지 궁금하지 않은가? AngularJS의 deferred는 promise를 작성하는 수단이다. deferred에서 resolve를 호출하면 promise가 충족되는(성공 핸들러가 호출됨) 반면, reject를 호출하면 promise의 에러 핸들러가 호출된다.

나중에 루트를 연결할 때 이 절을 다시 참고하길 바란다.

4.4.2 지시어

이제 애플리케이션이 사용하는 지시어를 살펴보자. 애플리케이션에는 다음과 같이 두 개의 지시어가 사용된다.

butterbar

butterbar 지시어를 지정한 요소는 페이지가 로딩 정보를 표시할 땐 보이다가 루트가 변경되면 감춰진다. butterbar 지시어는 루트 변경 메커니즘에 연결되어 페이지의 상태에 따라 자신이 지정된 요소를 자동으로 보였다감췄다를 한다.

focus

focus 지시어는 특정 인풋 필드나 요소에 포커스를 둘 때 사용한다.

다음 코드를 보자.

소스 코드 4-2 app/scripts/directives/directives.js

```
var directives = angular.module('guthub.directives', []);

directives.directive('butterbar', [ '$rootScope', function($rootScope) {
    return {
        link : function(scope, element, attrs) {
            element.addClass('hide');

            $rootScope.$on('$routeChangeStart', function() {
                element.removeClass('hide');
            });

            $rootScope.$on('$routeChangeSuccess', function() {
                element.addClass('hide');
            });
        }
    };
} ]);

directives.directive('focus', function() {
    return {
        link : function(scope, element, attrs) {
            element[0].focus();
        }
    };
});
```

앞의 지시어는 속성이 link 하나뿐인 객체를 반환한다. 〈AngularJS 활용편〉에서 개발자 정의 지시어를 작성하는 방법에 대해 자세히 설명하겠지만, 지금은 일단 다음과 같은 내용만 알고 넘어가도 된다.

1. 지시어는 두 단계를 거친다. 첫 번째 단계는 컴파일 단계로, DOM 요소에 연결된 모든 지시어가 발견된 후 처리된다. 모든 DOM 조작 역시 이 컴파일 단계 동안에 일어난다. 이 단계의 마지막엔 링크 함수가 생성된다.
2. 두 번째 단계는 링크 단계로, 앞에서 사용한 적이 있다. 링크 단계에서는 이전에 생성된 DOM 템플릿이 스코프에 연결된다. 그리고 모든 감시 함수와 리스너 함수는 필요에 따라 추가되므로 스코프와 요소 간의 동적 바인딩이 형성된다. 결국 스코프에 관련된 모든 것은 이 링크 단계에서 이뤄진다.

그럼 지시어는 어떤 식으로 작동하는지 살펴보자. butterbar 지시어를 다음과 같이 사용할 수 있다.

```
〈div butterbar〉로딩 중 표시할 문구...〈/div〉
```

butterbar 지시어는 기본적으로 요소를 미리 감춘 후, 루트 스코프에 두 개의 감시를 붙인다. 그러다가 루트 변경이 시작될 때마다 butterbar 지시어는 요소의 클래스를 변경해서 요소를 보여주고, 루트 변경에 성공할 때마다 butterbar 지시어는 요소를 감춘다.

여기서 $rootScope를 butterbar 지시어 안에 어떻게 주입했는지도 주의 깊게 봐야 한다. 모든 지시어는 AngularJS 종속물 주입 시스템에 직접 연결돼 있어서, 서비스 혹은 서비스에 필요한 무엇이든 주입할 수 있다.

마지막으로 눈여겨봐야 할 것은 butterbar 지시어가 지정된 div 요소와 연동될 API다. jQuery에 익숙한 사람들은 addClass나 removeClass 같은 jQuery 방식

의 문법에 기뻐할 것이다. AngularJS는 jQuery의 호출 부분 중 일부를 구현함으로써, 모든 AngularJS 프로젝트에서 jQuery를 종속물로 반드시 사용하지 않아도 된다. 프로젝트에 완전한 jQuery 라이브러리를 사용할 경우에는 AngularJS가 내장된 jQlite 대신에 완전한 jQuery 라이브러리를 사용하게 된다는 것을 알아둬야 한다.

두 번째 지시어인 focus는 그보다 훨씬 간단하다. 단순히 현재 요소에서 focus() 메서드를 호출한다. 어떤 인풋 요소에든지 다음과 같이 focus 지시어 속성을 추가하면 focus() 메서드를 호출할 수 있다.

```
<input type="text" focus></input>
```

페이지가 로딩되면 focus 지시어를 지정한 요소에 즉시 포커스가 가게 된다.

4.4.3 컨트롤러

지금까지 지시어와 서비스를 다뤘으니 이번에는 컨트롤러에 대해 알아보자. Guthub 애플리케이션에는 컨트롤러가 5개 사용된다. 컨트롤러 5개가 app/scripts/controllers/controllers.js 파일 하나에 전부 들어 있지만, 하나씩 나눠서 살펴보자. 우선 Guthub 시스템에서 모든 레시피 목록을 표시하는 List 컨트롤러는 다음과 같다.

```
app.controller('ListCtrl', ['$scope', 'recipes',
   function($scope, recipes) {
      $scope.recipes = recipes;
   }
]);
```

List 컨트롤러에 있어서 아주 중요한 사항이 하나 있다. 생성자 안에서 List 컨트롤러는 서버로 가서 레시피들을 가져오는 작업을 하지 않는다. 그저 이미 가져와 있던 레시피들을 전달받는다. 왜 그렇게 되는지 궁금할 것이다. 이 장의 루트 부분에서 그 답을 설명할 텐데, 힌트를 주면 앞에서 보았던 MultiRecipeLoader 서비스와 관계가 있다. 단지 그렇다는 사실만 염두에 두자.

List 컨트롤러를 작성했는데, 나머지 컨트롤러도 그와 기본적으로 거의 비슷하지만 중요한 부분들을 설명하면서 하나씩 차례로 구현하겠다.

```
app.controller('ViewCtrl', ['$scope', '$location', 'recipe',
   function($scope, $location, recipe) {
      $scope.recipe = recipe;
      $scope.edit = function() {
         $location.path('/edit/' + recipe.id);
      };
   }
]);
```

View 컨트롤러에서 중요한 부분은 스코프에서 공개하는 edit 함수다. 필드를 보이고 숨기거나 그와 비슷한 뭔가를 하는 대신, View 컨트롤러는 AngularJS를 이용해서 힘든 작업을 수행한다. edit 함수는 URL을 해당 레시피의 수정 페이지로 변경하는 기능만 수행하고, 나머지 작업은 전부 AngularJS가 수행한다. AngularJS는 URL이 변경됐음을 인식해서 해당 뷰를 로딩한다. 이것은 같은 레시피인데 수정 모드일 뿐이다.

그럼 이번에는 Edit 컨트롤러를 보자.

```
app.controller('EditCtrl', [ '$scope', '$location', 'recipe',
```

```
    function($scope, $location, recipe) {
        $scope.recipe = recipe;

        $scope.save = function() {
            $scope.recipe.$save(function(recipe) {
                $location.path('/view/' + recipe.id);
            });
        };

        $scope.remove = function() {
            delete $scope.recipe;
            $location.path('/');
        };
    }
]);
```

앞의 코드에는 스코프에서 Edit 컨트롤러가 공개하는 메서드를 저장하고 삭제하는 기능이 새로 추가됐다.

스코프 상의 save 함수는 예상대로 저장 작업을 수행한다. 현재 수정한 레시피를 저장하고, 저장이 완료되면 해당 레시피의 뷰 화면으로 되돌아간다. 콜백 함수는 이 상황에서 작업이 완료됐을 때 어떠한 작업을 실행하거나 수행할 때 사용하면 적합하다.

레시피를 저장할 수 있는 방법은 두 가지다. 첫 번째 방법은 코드에서처럼 $scope.recipe.$save() 함수를 실행해 저장 작업을 수행하는 것이다. recipe는 애초에 RecipeLoader가 반환하는 리소스 객체이기 때문에 이 방법만 가능하다.

레시피를 저장하는 두 번째 방법은 다음과 같다.

```
    Recipe.save(recipe);
```

remove 함수도 레시피를 스코프에서 제거하고 메인 페이지로 이동시키기만 하면 되므로 간단하다. remove 함수를 추가로 호출하는 것이 별로 어려운 일은 아니지만, remove 함수는 레시피를 서버에서 실제로 제거하지 않는다.

그다음으로 New 컨트롤러를 보면 다음과 같다.

```
app.controller('NewCtrl', [ '$scope', '$location', 'Recipe',
    function($scope, $location, Recipe) {
        $scope.recipe = new Recipe({
            ingredients : [ {} ]
        });

        $scope.save = function() {
            $scope.recipe.$save(function(recipe) {
                $location.path('/view/' + recipe.id);
            });
        };
    }
]);
```

New 컨트롤러는 Edit 컨트롤러와 거의 똑같다. 실제로 두 컨트롤러가 하나로 합쳐진 것을 예제에서 볼 수 있을 것이다. 한가지 큰 차이점은 New 컨트롤러는 처음에 새로운 레시피(리소스)를 생성한다는 점이다. Edit 컨트롤러에선 기존에 있던 레시피이므로 remove 함수가 필요하지만 New 컨트롤러는 새로 작성하는 레시피이므로 remove 함수가 필요 없다. 나머지 부분인 save 함수는 공통적으로 들어 있다.

끝으로 Ingredients 컨트롤러가 있는데 Ingredients 컨트롤러는 특수하다. 어떻게 왜 특수한지 설명하기 전에 일단 코드부터 살펴보자.

```
app.controller('IngredientsCtrl', [ '$scope', function($scope) {
    $scope.addIngredient = function() {
        var ingredients = $scope.recipe.ingredients;
        ingredients[ingredients.length] = {};
    };
    $scope.removeIngredient = function(index) {
        $scope.recipe.ingredients.splice(index, 1);
    };
} ]);
```

앞에서 살펴봤던 모든 컨트롤러는 UI의 특정 뷰에 연결돼 있었다. 그러나 Ingredients 컨트롤러는 그렇지 않다. Ingredients 컨트롤러는 상위 계층에는 필요하지 않은 특정 기능을 캡슐화하기 위해 수정 페이지에 사용되는 자식 컨트롤러다. 자식 컨트롤러의 특성상 Ingredients 컨트롤러는 부모 컨트롤러의 스코프를 상속받는다. 부모 컨트롤러는 Edit 컨트롤러와 New 컨트롤러다. 따라서 Ingredients 컨트롤러는 부모 컨트롤러에 있는 $scope.recipe에 접근할 수 있다.

컨트롤러 자체는 그다지 중요하거나 고유한 기능을 수행하진 않는다. 단지 새로운 요리 재료를 레시피의 ingredients 배열에 추가하거나, 특정 재료를 해당 레시피의 ingredients 배열에서 제거할 뿐이다.

이렇게 해서 컨트롤러는 모두 작성됐다. 이제 자바스크립트는 루트를 설정하는 부분만 작성되면 된다.

소스 코드 4-3 app/scripts/controllers/controllers.js

```
var app = angular.module('guthub', [ 'guthub.directives', 'guthub.services' ]);

app.config([ '$routeProvider', function($routeProvider) {
    $routeProvider.when('/', {
        controller : 'ListCtrl',
        resolve : {
            recipes : function(MultiRecipeLoader) {
                return MultiRecipeLoader();
            }
        },
        templateUrl : '/views/list.html'
    }).when('/edit/:recipeId', {
        controller : 'EditCtrl',
        resolve : {
            recipe : function(RecipeLoader) {
                return RecipeLoader();
            }
        },
        templateUrl : '/views/recipeForm.html'
    }).when('/view/:recipeId', {
        controller : 'ViewCtrl',
        resolve : {
            recipe : function(RecipeLoader) {
                return RecipeLoader();
            }
        },
        templateUrl : '/views/viewRecipe.html'
    }).when('/new', {
        controller : 'NewCtrl',
        templateUrl : '/views/recipeForm.html'
    }).otherwise({
```

```
            redirectTo : '/'
        });
    } ]);
```

앞서 예고했던 대로, 이제 resolve 함수들이 사용된 부분까지 왔다. 코드의 앞부분은 Guthub AngularJS 모듈뿐 아니라 애플리케이션에 필요한 루트와 템플릿도 설정한다.

앞의 코드는 이미 작성한 지시어와 서비스를 연결한 후 애플리케이션에 사용할 각종 루트를 지정한다. 각 루트마다 URL에 사용될 컨트롤러, 로딩할 템플릿을 지정하고 옵션으로 resolve 객체를 지정하자.

이 resolve 객체는 각 해독키가 일치해야만 루트를 표시하도록 AngularJS에게 명령한다. 레시피 목록을 로딩하거나 개별 레시피를 로딩해야 하므로, 서버에서 반드시 응답이 온 후에 페이지를 표시하게 해야 한다. 따라서 레피시(또는 개별 레시피)가 있음을 루트 프로바이더에게 먼저 알리고, 그런다음 그 레시피를 어떻게 가져올지 명령해야 한다.

링크는 앞의 "4.4.1 서비스"에서 정의했던 두 개의 서비스 MultiRecipeLoader와 RecipeLoader로 되돌아간다. resolve 함수가 AngularJS의 promise를 반환하면 AngularJS는 promise를 모두 가져올 때까지 처리하지 않고 대기한다. 즉, 서버가 응답할 때까지 기다린다.

이렇게 promise를 가져와 처리한 결과는 생성자 메서드에 인자로 전달되는데, 그 인자는 객체의 필드에 해당하는 매개변수명이다.

끝으로 otherwise 함수는 일치하는 루트가 없을 때 실행되며 기본 URL로 이동시킨다.

NOTE_ Edit 컨트롤러의 루트와 New 컨트롤러의 루트는 둘 다 똑같은 템플릿 URL인 views/recipeForm.html로 이동시킨다. 어떻게 된 것일까? 이건 레시피 수정 템플릿을 재사용한 것이다. 수정 레시피 템플릿 안에는 어느 컨트롤러에 연결돼 있는지에 따라 다른 요소들이 표시된다.

이것으로 컨트롤러까지 모두 살펴보았으니 다음 절에서 템플릿 부분을 살펴보자. 그리고 이 절에서 살펴본 컨트롤러들과 어떤 식으로 연결되는지, 최종 사용자에게 표시될 내용을 어떻게 관리하는지에 대해서도 알아보자.

4.5 템플릿

그럼 이번에는 메인 템플릿인 index.html 파일을 살펴보자. 메인 템플릿은 Guthub 단일 페이지 애플리케이션의 기준이며, 나머지 뷰는 전부 이 템플릿 콘텍스트의 내부에 로딩된다.

```
<!DOCTYPE html>
<html lang="ko" ng-app="guthub">
<head>
<title>GutHub (거트허브) - 레시피를 작성하고 공유하세요</title>
<script src="scripts/vendor/angular.min.js"></script>
<script src="scripts/vendor/angular-resource.min.js"></script>
<script src="scripts/directives/directives.js"></script>
<script src="scripts/services/services.js"></script>
<script src="scripts/controllers/controllers.js"></script>
<link href="styles/bootstrap.css" rel="stylesheet">
<link href="styles/guthub.css" rel="stylesheet">
</head>
<body>
    <header>
        <h1>GutHub</h1>
```

```
    </header>

    <div butterbar>로딩 중입니다...</div>

    <div class="container-fluid">
      <div class="row-fluid">
        <div class="span2">
          <!--Sidebar-->
          <div id="focus">
            <a href="/#/new">새 레시피 작성</a>
          </div>
          <div>
            <a href="/#/">레시피 목록</a>
          </div>
        </div>
        <div class="span10">
          <div ng-view></div>
        </div>
      </div>
    </div>
  </body>
  </html>
```

앞의 템플릿 코드에는 중요한 요소가 다섯 개 있다. 그 중 대부분은 2장에서 이미 설명했다. 그럼 하나씩 차례로 살펴보자.

ng-app

ng-app 모듈을 GutHub로 지정했다. GutHub는 angular.module 함수 안에 부여했던 모듈 이름이다. 이런 식으로 AngularJS는 두 모듈을 연결한다.

script 태그

script 태그는 애플리케이션에 AngularJS를 로딩한다. AngularJS를 사용하는 모든 자바스크립트 파일이 로딩되기 전에 AngularJS 파일부터 로딩이 이뤄져야 한다. 원칙적으로는 AngularJS를 사용하는 자바스크립트 파일은 body 요소 맨 끝에서 이뤄져야 한다.

Butterbar

처음으로 사용한 '개발자 정의 지시어'다. 앞에서 butterbar 지시어를 정의한 목적은 어떤 요소에 사용해서 루트가 변경되면 그 요소를 보이고 변경이 완료되면 감추려는 것이었다. 즉, 형광펜 효과로 테두리가 강조된 요소의 텍스트('로딩 중입니다...')는 필요할 때만 보여진다.

링크의 href 속성 값

href 속성은 GutHub 단일 페이지 애플리케이션의 각종 페이지에 링크를 건다. href 속성이 # 기호를 어떤 식으로 사용하면 페이지가 새로고침되지않는지, 그리고 어떻게 현재 페이지를 기준으로 상대적인지 눈여겨보자. AngularJS는 페이지가 새로고침되지 않는 한 URL을 감시하다가, 필요할 때 routes 안에 정의했던 당당히 자루한 루트를 마법처럼 변경시킨다.

ng-view

마법의 끝은 ng-view 지시어에서 이뤄진다. 앞의 컨트롤러 절에서 루트를 정의했다. 그 정의 안에 각 루트별 URL, 루트에 연결된 컨트롤러, 템플릿을 지정했다. AngularJS는 루트 변경을 감지하면 템플릿을 로딩하고, 그 템플릿에 컨트롤러를 부착한 후, ng-view를 템플릿의 내용으로 치환한다.

한가지 눈에 띄는 것은 ng-controller 태그가 없다는 점이다. 애플리케이션 대부분은 외부 템플릿에 어떤 식의 MainController든지 연결돼 있게 마련이다. ng-

controller 지시어는 보통 가장 보편적으로 body 태그에 지정된다. 그런데 여기서는 ng-controller 지시어를 사용하지 않았다. 그 이유는 외부 템플릿 어디에도 스코프를 참조해야 하는 AngularJS의 내용물이 들어 있지 않기 때문이다.

그럼 이제 각 컨트롤러와 연결된 템플릿을 하나씩 살펴보자. 우선 '레시피 목록' 템플릿은 다음과 같다.

소스 코드 4-4 chapter4/guthub/app/views/list.html

```
<h3>레시피 목록</h3>
<ul class="recipes">
   <li ng-repeat="recipe in recipes">
      <div><a ng-href="/#/view/{{recipe.id}}">{{recipe.title}}</a></div>
   </li>
</ul>
```

정말이지 레시피 목록 템플릿은 아주 진부하다. 이 템플릿에서 짚고 넘어갈 부분이라곤 두 군데뿐이다. 첫 번째는 아주 평범하게 사용된 ng-repeat 지시어다. ng-repeat 지시어는 스코프에서 모든 레시피를 가져다가 반복시킨다.

두 번째는 href 속성 대신 사용된 ng-href 지시어다. ng-href 지시어는 단지 AngularJS가 로딩되는 시간 동안 지저분한 링크가 표시되는 것을 방지하기 위함이다. ng-href를 사용하면 지저분한 링크가 절대로 사용자에게 표시되지 않는다. URL이 정적일 땐 몰라도 동적일 때는 반드시 ng-href를 사용하자.

컨트롤러는 어디 있는지 당연히 궁금할 것이다. 이 템플릿에는 ng-controller 지시어와 Main Controller가 정의되어 있지 않다. 이 템플릿에서는 루트 맵핑이 이뤄진다. 앞에서 설명했는데 / 루트는 목록 템플릿 페이지로 이동시키고 그 목록 템플릿에 List 컨트롤러를 연결했다. 그러므로 어느 참조든 변수와 그 비슷한 것을 대

상으로 이뤄지면 그 참조는 List 컨트롤러 스코프 내에 있는 것이다.

이제 개별 레시피를 보여주는 '레시피 보기' 템플릿을 살펴보자.

소스 코드 4-5 chapter4/guthub/app/views/viewRecipe.html

```
<h2>{{recipe.title}}</h2>

<div>{{recipe.description}}</div>

<h3>요리 재료</h3>
<span ng-show="recipe.ingredients.length == 0">재료가 없습니다</span>
<ul class="unstyled" ng-hide="recipe.ingredients.length == 0">
   <li ng-repeat="ingredient in recipe.ingredients">
     <span>{{ingredient.ingredientName}}</span>
     <span>{{ingredient.amount}}</span>
     <span>{{ingredient.amountUnits}}</span>
   </li>
</ul>

<h3>요리 설명</h3>
<div>{{recipe.instructions}}</div>

<form ng-submit="edit()" class="form-horizontal">
   <div class="form-actions">
      <button class="btn btn-primary">수정</button>
   </div>
</form>
```

이 템플릿은 깔끔하고 간결하며 절제되어 있다. 이 템플릿에서 주목할 부분은 두 군데다.

첫 번째는 평범하게 사용된 ng-repeat 지시어다. 모든 레시피는 다시 View 컨트롤러의 스코프 내에 있다. View 컨트롤러는 페이지가 표시되기 전에 resolve 함수에 의해 로딩된다. 그래서 깨진 페이지나 로딩이 덜 된 상태의 페이지는 사용자에게 보여질 가능성이 전혀 없다.

두 번째는 폼에 지정된 ng-submit 지시어다. ng-submit 지시어는 폼이 전송되면 scope 상의 edit() 함수를 호출하라고 명령한다. 폼 전송은 코딩을 통해 함수가 연결되지 않은 어느 버튼이든(여기서는 Edit 버튼) 클릭되면 이뤄진다. 다시 말하지만 AngularJS는 영리해서 모듈, 루트, 컨트롤러가 참조하는 스코프를 인식해서 올바른 메서드를 적시에 호출한다.

끝으로 '레시피 수정 폼' 템플릿을 살펴보자. 셋 중에 가장 복잡한 템플릿이다.

소스 코드 4-6 chapter4/guthub/app/views/recipeForm.html

```
<h2>레시피 수정하기</h2>
<form name="recipeForm" ng-submit="save()" class="form-horizontal">
    <div class="control-group">
        <label class="control-label" for="title">제목:</label>
        <div class="controls">
            <input ng-model="recipe.title" class="input-xlarge"
                id="title" focus required>
        </div>
    </div>

    <div class="control-group">
        <label class="control-label" for="description">설명:</label>
        <div class="controls">
            <textarea ng-model="recipe.description" class="input-xlarge"
                id="description"></textarea>
        </div>
```

```
</div>

<div class="control-group">
    <label class="control-label" for="ingredients">재료:</label>
    <div class="controls">
        <ul id="ingredients" class="unstyled"
            ng-controller="IngredientsCtrl">
          <li ng-repeat="ingredient in recipe.ingredients">
            <input ng-model="ingredient.ingredientName" class="input-small">
            <input ng-model="ingredient.amount" class="input-mini">
            <input ng-model="ingredient.amountUnits" class="input-mini">
               <button type="button" class="btn btn-mini"
                   ng-click="removeIngredient($index)">
                   <i class="icon-minus-sign"></i> 삭제
               </button></li>
          <button type="button" class="btn btn-mini"
              ng-click="addIngredient()">
              <i class="icon-plus-sign"></i> 추가
          </button>
        </ul>
    </div>
</div>

<div class="control-group">
    <label class="control-label" for="instructions">조리법:</label>
    <div class="controls">
        <textarea ng-model="recipe.instructions"
            class="input-xlarge" id="instructions"></textarea>
    </div>
</div>

<div class="form-actions">
    <button class="btn btn-primary"
```

```
            ng-disabled="recipeForm.$invalid">저장</button>
        <button type="button" ng-click="remove()" ng-show="!recipe.id"
            class="btn">삭제</button>
    </div>
</form>
```

코드가 길어보이지만 당황하지 말자. 실제로 분석해보면 별로 복잡하지 않다. 사실 대부분이 레시피 수정을 위한 수정 가능한 인풋 필드를 표시하는 간단하고 반복적인 코드다.

- focus 지시어는 첫 번째 인풋 필드에 추가된다. 그게 바로 레시피 제목을 수정하는 인풋 필드다. focus 지시어를 지정하면 사용자가 페이지를 탐색할 때 제목 필드에 포커스가 있게 되어, 사용자는 머뭇거림 없이 이 필드에 제목을 입력할 수 있다.
- ng-submit 지시어는 앞의 예제에서와 거의 비슷하게 사용되므로 자세한 설명은 하지 않겠다. 한가지 차이점은 레시피의 상태를 저장하고 수정 과정이 끝났음을 알린다. ng-submit 지시어는 Edit 컨트롤러에 있는 save() 함수에 연결된다.
- ng-model 지시어는 해당 필드의 각종 인풋 박스와 텍스트 에어리어를 모델에 바인딩한다.
- 이 페이지에서 매우 중요하므로 꼭 이해해야 할 부분 중 하나는, 바로 재료 목록 부분에 지정된 ng-controller 지시어다. 시간을 내어 이 부분의 원리를 이해하도록 하자.
- 표시될 재료 목록을 보면 컨테이너 태그가 ng-controller에 연결되어 있다. 그래서 〈ul〉 태그 전체가 Ingredients 컨트롤러 스코프에 속한다. 그렇다면 템플릿의 실질적 컨트롤러인 Edit 컨트롤러는 어떻게 된 것일까? 결론만 말하면 Ingredients 컨트롤러는 Edit 컨트롤러의 자식 컨트롤러로 생성되고, Edit

컨트롤러의 스코프를 상속받는다. 그래서 Edit 컨트롤러에서 recipe 객체에 접근할 수 있는 것이다.

- 그리고 Edit 컨트롤러는 addIngredient() 메서드를 추가하는데, 이 메서드는 형광펜 효과로 강조되는 ng-click 요소에 사용된다. ng-click은 〈ul〉 태그의 스코프 내에서만 접근할 수 있다. 그렇게 할 이유가 있을까? 관심사를 분리할 수 있는 최선의 방법이기 때문이다. 템플릿의 99%가 아무 상관도 없으면 Edit 컨트롤러에 왜 addIngredients() 메서드가 있어야 할까? 자식 컨트롤러와 하위 컨트롤러는 그렇게 정확하고 절제된 작업에 적합하며, 개발자로 하여금 비즈니스 로직을 보다 관리하기 쉬운 덩어리들로 쪼갤 수 있게 한다.
- 나머지 중요한 부분은 폼 유효성 검사 컨트롤들이다. AngularJS에서는 특정 폼 필드를 '필수 기입'으로 설정하는 일이 간단하다. 앞의 코드에서처럼 그냥 해당 인풋 요소에 required 지시어만 지정하면 되기 때문이다. 그런데 이번에는 required 지시어로 무엇을 할까?
- 궁금증을 풀기 위해 일단 '저장' 버튼 부분으로 건너뛰자. 저장 버튼 요소에 ng- disabled 지시어가 있고 값이 recipeForm.$invalid로 지정돼 있다. recipeForm 은 앞서 선언했던 폼의 이름이다. AngularJS는 recipeForm에 특수한 변수 $valid와 $invalid를 추가한다. 두 변수를 사용해 폼 요소를 제어할 수 있다. AngularJS는 모든 필수 요소를 관찰하고 그 상태에 맞게 $valid와 $invalid 변수를 업데이트한다. 따라서 만약 레시피 제목이 비어있다면 recipeForm.$invalid의 값은 true로 변경되고 $valid 값은 false로 변경되며, '저장' 버튼이 즉시 불용화된다.

인풋의 최대 길이와 최소 길이를 지정할 수도 있고, 인풋 필드를 유효하게 만드는 정규표현식 패턴을 지정할 수도 있다. 게다가 특정 조건에 부합할 때만 특정 에러 메시지를 표시하기 위해 적용할 수 있는 고급 사용법도 있다. 다음의 간단한 예제를 통해 살펴보자.

```
<form name="myForm">
    사용자명: <input type="text" name="userName"
        ng-model="user.name" ng-minlength="3">
    <span class="error"
        ng-show="myForm.userName.$error.minlength">이름이 너무 짧습니다!</
span>
</form>
```

앞의 예제는 ng-minlength 지시어를 사용해서 사용자명에 최소한 3개 이상의 문자를 입력하도록 조건을 부여했다. 이 폼은 스코프 안의 각 이름이 붙은 인풋으로 구성되며(이 예제에는 userName뿐임), 각 인풋마다 $error 객체와 인풋 자체가 유효한지 아닌지를 알리는 $valid 태그가 지정된다. $error 객체에는 나중에 required, minlength, maxlength, pattern 중에서 어떤 에러가 있는지 없는지가 포함된다.

$error 객체는 앞 예제에서처럼 사용자가 저지른 입력 에러의 종류에 따라 선택적으로 에러 메시지를 표시할 때 사용한다. 이제 다시 원래의 '레시피 수정 폼' 템플릿으로 돌아가자.

끝으로 맨 마지막에 쓰인 ng-click을 보자. ng-click 지시어는 레시피를 삭제하는 버튼에 붙어 있다. 레시피 삭제 버튼은 레시피가 아직 저장되지 않았을 때만 표시된다. 보통은 ng-hide="recipe.id"를 작성하는 것이 더 적합하긴 하지만, 때로는 ng-show="!recipe.id"라고 작성하는 것이 시맨틱 측면에서 더 바람직할 때도 있다. 이 코드는 레시피에 id가 없으면 버튼을 보이고, 레시피에 id가 있으면 버튼을 감춘다.

4.6 테스트

드디어 컨트롤러와 함께 작성할 테스트를 살펴볼 시간이다. 이 절에서는 코드의 어

느 부분에 어떤 종류의 테스트를 작성해야 할지, 그리고 테스트를 실제로 어떻게 작성해야 하는지에 대해 알아보자.

4.6.1 단위 테스트

가장 우선적으로 해야할 중요한 테스트는 단위 테스트다. 단위 테스트는 개발한 컨트롤러, 지시어, 서비스가 올바른 구조로 작성됐는지 검사하고, 의도한대로 동작하는지를 시험한다.

개별 단위 테스트에 앞서, 모든 컨트롤러 단위 테스트를 아우르는 테스트 하니스 test harness부터 살펴보자.

```
describe('Controllers', function() {
    var $scope, ctrl;
    // 테스트 안에 모듈을 지정해야 한다
    beforeEach(module('guthub'));
    beforeEach(function() {
        this.addMatchers({
            toEqualData: function(expected) {
                return angular.equals(this.actual, expected);
            }
        });
    });

    describe('ListCtrl', function() {....});
    // 여기에 다른 컨트롤러도 작성할 것
});
```

앞의 테스트 하니스는(아직도 필자는 Jasmine을 사용해서 기능적인 방식으로 이 테스트들을 작성한다) 다음과 같은 기능을 한다.

1. 접근 가능한 스코프와 컨트롤러를 전역적으로 생성해서(적어도 이 테스트 명세의 목적에 맞게), 컨트롤러마다 새로운 변수를 생성할 필요가 없다.
2. 애플리케이션(GutHub)에 사용되는 모듈을 초기화한다
3. equalData라는 특수 대조 함수를 추가한다. equalData 함수를 사용하면 기본적으로 $resource 서비스나 REST 기반의 호출을 통해 반환되는 recipes 같은 리소스 객체에서 어설션assertion을 수행할 수 있다.

> **NOTE_** ngResource 반환 객체에서 어설션을 수행해야 할 때마다 equalData라는 특수 대조 함수를 추가하자. 왜냐하면 ngResource 반환 객체에는 일반 expect equal 호출이 불가능한 메서드가 몇 개 추가로 들어 있기 때문이다.

앞의 하니스를 완성했으면 이제 List 컨트롤러의 단위 테스트를 살펴보자.

```
describe('ListCtrl', function() {
    var mockBackend, recipe;
    // _$httpBackend_는 $httpBackend와 같다. 주입된 변수와 지역 변수를
    // 구별하기 위해 다르게 표기했을 뿐이다.
    beforeEach(inject(function($rootScope, $controller, _$httpBackend_, Recipe) {
        recipe = Recipe;
        mockBackend = _$httpBackend_;
        $scope = $rootScope.$new();
        ctrl = $controller('ListCtrl', {
            $scope : $scope,
            recipes : [ 1, 2, 3 ]
        });
    }));

    it('레시피 목록이 있어야 함', function() {
        expect($scope.recipes).toEqual([ 1, 2, 3 ]);
```

```
    });
});
```

List 컨트롤러는 GutHub 애플리케이션에서 가장 간단한 컨트롤러다. List 컨트롤러의 생성자는 레시피 목록을 받아서 스코프에 저장하는 기능만 한다. List 컨트롤러에 대한 테스트를 작성할 수도 있지만 어리석은 일같다. 그래도 필자는 테스트를 했다. 테스트해서 나쁠 건 없으니까!

더 중요한 부분은 MultiRecipeLoader 서비스다. MultiRecipeLoader 서비스는 레시피 목록을 서버에서 가져와서 인자로 전달하는 역할을 한다($route 서비스를 통해 제대로 연결될 경우).

```
describe('MultiRecipeLoader', function() {
    var mockBackend, recipe, loader;
    // _$httpBackend_는 $httpBackend와 같다. 주입된 변수와 지역 변수를
    // 구별하기 위해 다르게 표기했을 뿐이다.
    beforeEach(inject(function(_$httpBackend_, Recipe, MultiRecipeLoader) {
        recipe = Recipe;
        mockBackend = _$httpBackend_;
        loader = MultiRecipeLoader;
    }));

    it('레시피 목록을 로딩해야 함', function() {
        mockBackend.expectGET('/recipes').respond([ { id : 1 }, { id : 2 } ]);
        var recipes;
        var promise = loader();

        promise.then(function(rec) {
            recipes = rec;
        });
```

```
        expect(recipes).toBeUndefined();
        mockBackend.flush();
        expect(recipes).toEqualData([ { id : 1 }, { id : 2 } ]);
    });
});
// 여기에 다른 컨트롤러도 작성할 것
```

테스트 안에 모크 HttpBackend를 연결해서 MultiRecipeLoader를 테스트하자. HttpBackend는 angular-mocks.js 파일에 들어 있으며, 테스트가 실행될 때 포함된다. HttpBackend를 beforeEach 메서드에 주입하기만 하면 의도대로 설정이 자동 시작된다. 더욱 뜻 깊은 두 번째 테스트를 보면, server GET의 기댓값으로 recipes 호출을 지정했는데, recipes 호출 결과 단순 객체 배열이 반환된다. 그다음 새로운 사용자 정의 대조 함수를 사용해서 단순 객체 배열이 정확히 반환된 값인지 확인한다. 서버에서 지금 응답을 반환할 모크 백엔드에 명령하는 모크 백엔드의 flush() 호출을 보자. 그 메커니즘을 사용하면 컨트롤 흐름을 테스트하면서 서버가 응답을 반환하기 전후에 GutHub 애플리케이션이 어떻게 처리하는지 볼 수 있다.

View 컨트롤러는 건너뛰자. 대부분이 스코프에 edit() 메서드를 추가하는 것만 빼고는 List 컨트롤러와 똑같기 때문이다. View 컨트롤러는 테스트하기가 꽤 간단하다. 테스트 안에 $location를 주입하고 값을 검사하면 된다.

다음으로 Edit 컨트롤러를 살펴보자. Edit 컨트롤러에는 단위 테스트해야 할 중요한 부분이 두 군데 있다. resolve 함수는 앞서 보았던 것과 비슷하므로 같은 식으로 테스트하면 된다. save() 메서드와 remove() 메서드를 어떻게 테스트하면 될지 살펴보자. 두 메서드에 대한 테스트는 다음과 같다. 하니스는 앞 예제와 같다고 가정하자.

```
describe('EditController', function() {
    var mockBackend, location;
    beforeEach(inject(function($rootScope, $controller, _$httpBackend_,
           $location, Recipe) {
        mockBackend = _$httpBackend_;
        location = $location;
        $scope = $rootScope.$new();

        ctrl = $controller('EditCtrl', {
            $scope : $scope,
            $location : $location,
            recipe : new Recipe({
                id : 1,
                title : 'Recipe'
            })
        });
    }));

    it('레시피를 저장해야 함', function() {
        mockBackend.expectPOST('/recipes/1', {
            id : 1,
            title : 'Recipe'
        }).respond({
            id : 2
        });

        // 테스트 동안 변경되게 하기 위해 뭔가를 지정
        location.path('test');

        $scope.save();
        expect(location.path()).toEqual('/test');
```

```
        mockBackend.flush();

        expect(location.path()).toEqual('/view/2');
    });

    it('레시피를 삭제해야 함', function() {
        expect($scope.recipe).toBeTruthy();
        location.path('test');
        $scope.remove();

        expect($scope.recipe).toBeUndefined();
        expect(location.path()).toEqual('/');
    });
});
```

첫 번째 테스트는 save() 함수에 대한 검사다. 특히 처음 저장할 때 객체와 함께 서버에 POST 요청이 이뤄져야 하고, 서버가 응답하면 위치가 새로 저장한 객체의 레시피 보기 페이지로 변경돼야 한다.

두 번째 테스트는 훨씬 간단하다. 스코프에서 remove() 함수를 호출하면 현재의 레시피가 삭제된 후 메인 페이지로 이동한다. 이 부분은 테스트 안에 $location 서비스를 주입해서 값을 검사하면 쉽게 할 수 있다.

컨트롤러에 대한 나머지 단위 테스트 역시 앞에 설명한 것과 매우 비슷하므로 넘어가자. 그 단위 테스트들은 기본적으로 다음과 같은 몇 가지 작업을 기반으로 한다.

- 컨트롤러(또는 스코프)가 초기화 끝 단계에서 확실히 올바른 상태에 도달하게 한다.
- 단위 테스트 안에 모크 벡엔드를 사용해서 올바른 서버 호출이 이뤄지는지, 서버 호출 동안과 서버 호출이 끝난 후에 스코프가 올바른 상태에 도달하는지 확인

한다.

- 컨트롤러가 확실히 올바른 상태로 설정되게 하기 위해 AngularJS 종속물 주입 프레임워크를 이용해 컨트롤러에 연결된 요소와 객체로 핸들을 가져온다.

4.6.2 시나리오 테스트

단위 테스트를 만족스럽게 작성하고 나면, 안락의자에 기대 앉아 담배나 한 대 피우면서 오늘 일은 이것으로 끝이라고 외치고픈 유혹이 들 것이다. 그러나 AngularJS 개발자의 작업은 시나리오 테스트를 실행하기 전까지 끝난 것이 아니다. 단위 테스트에 성공하면 자바스크립트 코드의 모든 작은 부분이 의도대로 돌아가겠지만, 아직 템플릿이 로딩되는지, 컨트롤러에 제대로 연결되는지, 템플릿 여기저기를 클릭했을 때 제대로 된 작업이 수행되는지 등을 확인해야 한다.

AngularJS에서 시나리오 테스트는 다음과 같은 작업을 자동으로 수행한다.

- 애플리케이션을 로딩한다.
- 특정 페이지를 탐색한다.
- 여기저기를 클릭하고 이런저런 텍스트를 입력한다.
- 올바른 동작이 발생하는지 확인한다.

그럼 '레시피 목록' 페이지에 대해서는 시나리오 테스트가 어떤 식으로 이뤄질까? 우선 실제 테스트를 시작하기 전에 약간의 준비작업을 해야 한다.

시나리오 테스트가 제대로 되려면 GutHub 애플리케이션에서 요청을 받을 준비가 되어 있고 레시피 목록을 저장하고 가져올 수 있는 웹 서버 작업이 필요하다. 메모리에 저장된 레시피 목록을 사용하는 코드나(레시피 $resource를 삭제한 후 JSON 객체 덤프로 변경), 앞 장에서 보았던 웹 서버를 재사용하고 변경하는 코드나, Yeoman을 사용하는 코드를 부담 갖지 말고 수정하자.

서버를 가동 중이고 GutHub 애플리케이션을 서버에 올렸으면, 이제 다음 테스트를 작성해서 실행하자.

```
describe('GutHub App', function() {
    it('레시피 목록을 표시해야 함', function() {
        browser().navigateTo('/index.html');
        // 기본 GutHub 레피시 목록에는 두 개의 레시피가 있다
        expect(repeater('.recipes li').count()).toEqual(2);
    });
});
```