Hanbit eBook
Realtime 41

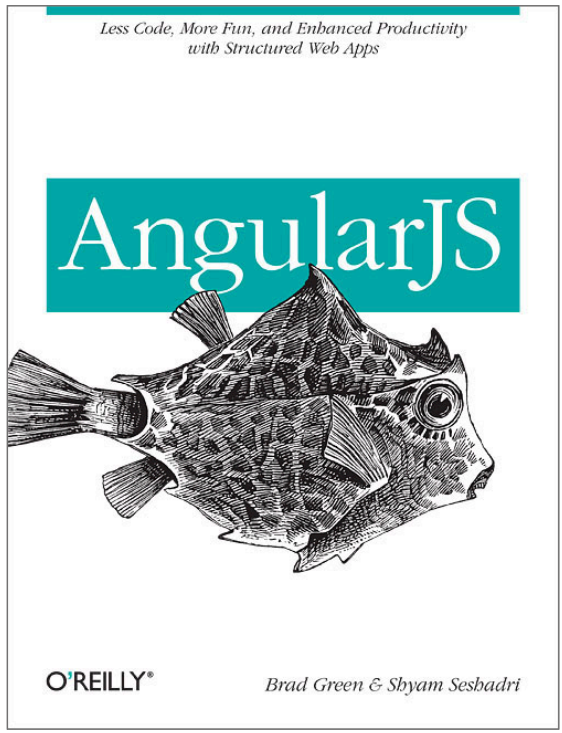
MVC 패턴을 구현하는 자바스크립트 프레임워크

AngularJS 활용편

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 활용편

MVC 패턴을 구현하는 자바스크립트 프레임워크 AngularJS **활용편**

초판발행 2013년 9월 13일

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 978-89-6848-643-2 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 그린**Brad Green

구글 AngularJS 프로젝트팀에서 엔지니어 관리자를 맡고 있으며, 접근성과 지원 공학을 총괄 감독한다. 구글에 입사하기 전에는 인터넷 기업의 마케팅 도구를 만들어 팔던 AvantGo 사에서 초창기 모바일 웹 개발자로 근무하다가, 출장요식업에 뛰어들어 고단한 몇 년을 보냈다. 대학을 졸업하고 NeXT Computer 사에서 스티브 잡스의 지휘 아래 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 세샤드리**Shyam 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 Indian School of Business에서 MBA 과정을 마쳤다. 샤이엄은 대학 졸업 후 첫 직업으로 구글에서 다수의 프로젝트를 진행했다. 그중에는 AngularJS가 처음으로 사용된 구글 피드백(Google 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), 『AngularJS 기초편』(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 메소드 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_ 고지은

졸업하고 나서 알고 싶은 게 더 많은, 컴퓨터공학을 전공한 취업준비생이다. 앞으로 나아가기 위해 다양한 전공 책들에 싸여 지내고 있다.

베타테스터_ 김광남

프로그래머로서, 많은 사람이 무료로 사용할 수 있는 프로그램을 만드는 게 삶의 목표다. 게임을 개발하다 현재는 노래방 회사에서 PC, SmartPhone, TV용 노래방을 개발하고 있다. “나의 아내 효성아! 언제나 사랑한다.”

베타테스터_ 김종호

스타트업을 통해 새로운 가치를 만들길 원하는 학생이다. C/C++을 사용하며 지루해 했으나 안드로이드 개발을 통해 프로그래밍에 눈을 떴다. 지금은 파이썬과 다양한 웹 기술을 이용하여 웹 서비스를 만들어서 재미있는 시도를 하고 있다. 서버와 플랫폼에도 관심이 많으며 풀 스택 개발자가 되기를 희망한다.

베타테스터_ 박정춘

새로운 기술과 프로그래밍 언어에 관심이 많은 백엔드 개발자다. 최근에 오픈소스와 함수형 프로그래밍에 빠져 있으며, 좋은 E-Commerce 서비스를 만들기 위해 노력하고 있다.

베타테스터_ 한상곤

리눅스에서 개발자 생활을 시작하여, HTML과 Spring 영역까지 뚜벅 뚜벅 걸어왔다. 어떻게 왔는지, 어디로 갈지는 모르지만 매일 매일 새로운 것을 설계하고 만드는 재미에 푹 빠져있다. 요즘은 Golang과 파이썬에 빠져서 허우적 거리고 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 내려받을 수 있다.

- https://github.com/shyamseshadri/angularjs-book (영문 버전)
- http://www.hanb.co.kr/exam/2643 (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 내려받을 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

들어가기 전에 | 필수 용어와 개념 정리

종속물 주입(Dependency Injection)[01]

종속물 주입은 하드코딩한 필요 기능(depencency, 이것은 '결합도'가 아님)을 제거/해제하거나 변경할 수 있도록 적용하는 소프트웨어 디자인 패턴이다.

예컨대 플러그인을 동적으로 로딩하거나 테스트 단계에서 스텁 객체나 목mock 객체를 선택 또는 제품화 단계에서 실제 객체를 선택하는 작업에 종속물 주입 패턴을 적용하면 수월해진다. 종속물 주입 패턴은 대상의 요건을 파악해서 종속 요소(객체, 값 등)를 대상에 자동으로 주입한다. 종속물 조회dependency lookup라는 패턴도 있는데, 이 패턴은 종속물 주입을 위한 순과정과 역과정에 해당한다.

> **NOTE_** 종속물 조회는 호출 객체가 컨테이너 객체에 특정 이름이나 특정 타입으로 된 객체를 요청하는 패턴이다.

종속물 주입 패턴은 최소한 다음 세 요소로 구성된다.

- 종속된 소비 객체consumer
- 구성요소의 종속물(인터페이스 콘트랙트로 정의됨)을 선언하는 코드
- 주입 객체injector[02]

종속 객체 안에는 작업 수행에 필요한 소프트웨어 구성요소를 작성한다. 주입 객체는

---

01 출처: Wikipedia의 Dependency Injection 페이지
http://en.wikipedia.org/wiki/Dependency_Injection

02 '프로바이더'나 '컨테이너'라고도 하며, 이 객체 안에 종속물을 넣어서 주입한다. 이 객체는 요청을 받으면 주어진 종속 인터페이스를 구현하는 클래스를 인스턴스화한다.

어떤 구체 클래스[03]가 종속 객체의 요건에 부합하는지를 판단하여, 조건에 부합하는 구체 클래스를 종속 객체에 제공한다.

기존 방식으로 소프트웨어를 개발할 때는 어떤 구체 클래스를 사용할지를 종속 객체가 스스로 판단했다. 그러나 종속물 주입 패턴에서는 주입 객체에 이 판단이 위임되므로, 주입 객체는 종속 콘트랙트 인터페이스를 적절하다고 판단한 구체 클래스로 교체할 수 있다. 이것은 컴파일이 아니라 런타임 시에 이뤄진다.

이 책에서 '종속물 주입'이라는 용어를 사용한 이유

dependency: 종속물. 기본적으로는 '종속 객체'를 뜻하지만 언어와 문맥에 따라 종속 객체, 종속 라이브러리, 종속 함수, 종속 리소스, 종속 모듈 등 현재 객체의 기능 수행에 필요하여 컨테이너로 캡슐화해 주입할 수 있는 모든 대상을 지칭할 수 있다.

기존에는 dependency를 '의존성'이나 '의존관계'로 번역했는데, 이러한 용어는 매우 막연하고 두리뭉실한 뜬구름 같아서 용어의 의미를 제대로 설명할 수 없었다. 프로그래밍의 흐름은 대충 알지만 이런 용어의 개별적인 개념을 정확하게 알지 못하고 넘어간다면, 후에 더욱 복잡한 기술로 진화하고 많은 파생 개념이 생길수록 혼란이 심해질 것이다.

'의존성'이라고 하면 의존하는 '성질'을 뜻한다. 따라서 상위 코드에 꼭 필요한 부품과도 같은 하위 코드를 '의존성'이라고 표현해서는 그 의미를 제대로 전달할 수 없다. 이것은 IT 기술 분야에서 용어를 모든 문맥에 가장 부합하게끔 고민을 거쳐 선택했다기보다는, 마치 영어사전을 뒤졌을 때 나오는 단순한 사전적 의미 중 하나를 고른 듯한 성의 없는 용어 선택이 아닐 수 없다. 종속되는 객체나 모듈은 주입될

03 추상 클래스의 반대개념으로 일반적인 클래스를 말한다. 인스턴스를 만들 수 있는 클래스로, 추상 메소드를 가질 수 없다.

대상으로서 구체적인 실체이지 '성질'이나 '관계'가 아니다. 종속물을 주입하면 의존성이나 의존 관계가 형성되는 것이지, 의존성이나 의존 관계 자체를 주입하는 것이 아니다. 그리고 '종속 객체'라는 용어를 사용하면 종속된 관계의 대상이 객체만으로 한정돼버리기 때문에 적절하지 않다. 문맥과 기술에 따라 달라지겠지만, 적어도 이 책에서는 dependency가 단지 종속 객체뿐 아니라 종속 모듈, 서비스, 함수 등을 통칭하기 때문에 포괄적인 뜻을 담은 '종속물'로 번역했다. 물론 자바스크립트에서도 모든 것(함수, 모듈, 서비스 등)을 아우르는 개념이 객체이므로 종속 객체라는 표현도 괜찮기는 하지만 설명할 때는 모듈, 서비스라고 표기하지 '모듈 객체', '서비스 객체'라는 식으로 표기하지 않기 때문에 상위 개념으로 한정되는 느낌을 주지 않기 위해서 '종속물'을 택한 것이다.

1. **이것을 '의존-'으로 번역하면 안 되는 이유:** dependency를 의존, 의존성, 의존관계, 의존물 등으로 번역하면 'dependent ~'라는 표현이 자주 병행 등장하는데 이것을 일관성에 따라 '의존하는 ~'으로 번역해야 하므로 잘못된 뜻이 돼버린다. 왜냐하면 dependent function은 종속된 함수인데, 오히려 이 함수를 '의존하는 함수'로 해석하게 되므로 틀린 의미가 된다. 반면에 '종속된 함수'라고 해석하면 의미가 맞으므로 모든 문맥에서 일관되게 올바른 뜻을 가지려면 반드시 '종속'으로만 번역해야 하는 것이다. 그리고 개념을 설명하는 영상이나 도표 자료를 검색해 보면 dependency injection 개념을 설명할 때는 항상 벤다이어그램을 사용해 포함관계를 나타내는데, 이를 보더라도 역시 '종속'이 더욱 적절하다. 이것을 거부감 없이 이해하려면 기존에 널리 쓰이던 '의존성 주입'이라는 용어의 편견을 떨쳐야 한다.

2. **이것을 '부속-'으로 번역하면 안 되는 이유:** '부속'과 '종속'은 의미상 비슷하다. 그러나 다음과 같은 미묘하고도 중대한 의미적 차이가 있다.
   - 부속: 주된 사물이나 기관에 딸려서 붙음(공간적/위치적 소속과 기능적인 구성요소임을 동시에 뜻함).

- 종속: 자주성이 없이 주가 되는 것에 딸려 붙음(공간적/위치적 뉘앙스에 비해 주종 관계에서 필요에 따라 주가 이용할 수 있는 종의 역할을 한다는 의미가 더 강하다).

따라서 '종속'은 주(주입 객체)가 있어야 호출되며 종속된 종(종속 객체)은 주에게 필요한 기능을 제공한다는 뜻에서 '종속'을 사용하는 것이 올바르다. 물론, '부속'에도 미약하나마 기능적인 구성요소라는 의미가 있지만, '부속'은 '종속'에 비해 의존적인 관계 형성에 있어 훨씬 약한 뉘앙스를 풍기는 낱말이므로 '종속'을 택했다.

3. injection을 '삽입'으로 번역하는 것이 바람직하지 않은 이유: 국어사전에 있는 의미를 그대로 생각하면 injection은 액체나 사상 등을 주입할 때 쓰는 말이므로, 종속물을 현 객체로 집어넣는 이 상황에서 '삽입(insertion)'이 사전적 의미로 보아 바른 용어 선택인 것처럼 보일 수 있다. 그러나 IT 분야의 각종 개념을 설명하는 표현이나 용어를 정할 때는 '메타포' 기법을 사용한다. 메타포란 현실 세계(일상, 제품, 상식, 각 학문 분야 등)에서 흔히 쓰이는 낱말과 표현에 빗대는 것을 말한다. 메타포에 따른 용어와 표현을 사용하면 개발자는 어떠한 개념이든 쉽게 이해할 수 있기 때문이다. 중요한 것은, 어차피 영어권 국가에서 만든 용어고 새로운 개념 및 용어가 현재에는 없지만 미래의 어느 순간 기존 용어에서 분리/파생되거나 신생될 수 있기에 영문 용어를 실정에 맞게 의역하되 가능하면 일대일로 대응시켜 정립하는 것이 바람직하다. 미래엔 insertion이라는 용어가 새로 생겨나 병용될 수도 있기 때문에 injection을 굳이 사전적 의미로만 고집해 '삽입'으로 번역할 것이 아니라 상위 객체에 종속 객체를 주사기로 밀어 넣는 것에 빗댄 메타포에 의한 용어를 채택하는 것이 좋다. 그렇게 하면 dependency injection이라는 영문 용어를 보고 '주입'을 바로 떠올릴 수 있고 '종속물 주입'이라고 하면 곧바로 'dependency injection'을 떠올릴 수 있다. 만약 '종속물 삽입'이라고 용어를 정한다면 그걸 들은 사람이 영문을 생각해내

야 할 상황(외국인 개발자와 협업 등)이 닥쳤을 때 'dependency insertion이던가?' 하는 혼동을 초래할 수도 있다. '삽입'을 주장하기 위해 '객체'를 굳이 '고체'라고 여길 필요는 없다(심지어 현실의 물질도 아니다). 관점에 따라 객체는 '액체'일 수도 있고 '기체'일 수도 있다. 핵심은 결코 '고체'로 한정할 근거가 없다는 데 있다. 용어를 정하기 전에는 반드시 비슷한 여러 단어 중에서 그 단어를 원어민 필자가 사용한 데에는 나름의 이유가 있게 마련이라는 생각을 해봐야 한다.

참고) dependency injection 패턴에는 구성원인 injector 개념이 함께 등장한다. injector는 '주입 객체'로 번역해야 하며, 프로바이더(프로바이더 객체)나 컨테이너(컨테이너 객체)라고도 한다.

프라미스 인터페이스와 $q 서비스 객체

프라미스

프라미스는 비동기적으로 수행되는 작업의 결과를 대변하는 객체와 교류하는 인터페이스다. 프라미스promise는 하나의 작업 완료로 반환된 결과값을 대변한다. 프라미스는 세 가지(unfulfilled:미이행, fulfilled:이행완료, failed:실패) 중 하나의 상태에 처할 수 있다. 프라미스가 fulfilled(이행완료) 상태나 failed(실패) 상태면 프라미스의 값은 절대로 변해선 안 되며, 자바스크립트에서의 값이나 기본타입/객체 ID와 마찬가지로 변할 수 없다. 프라미스의 '변할 수 없는' 특징 덕에 리스너에 예상치 못한 기능 변화가 생길 수 있는 부작용이 예방되며, 프라미스를 다른 함수에 전달해도 호출 객체에 영향이 미치지 않는다. 이것은 '호출하는 객체'의 변수가 '호출되는 객체'에 의해 변경될 염려 없이 기본 타입을 함수에 전달할 수 있는 것과도 같다.

함수가 값을 반환하는 것이 불가능하거나 차단하지 않으면, 예외가 통지될 때 그 함수는 값 대신 프라미스를 반환하면 된다. 프라미스는 함수가 최종적으로 제공할

가능성이 있는 반환 값이나 통지 예외를 대변하는 인터페이스 객체다. 지연을 극복하기 위해 프라미스를 원격 객체의 프록시로 사용할 수도 있다.

$q 객체

$q: 크리스 코우얼Kris Kowal의 q 도구[04]에 들어 있는 프라미스 API와 지연 API에서 착안해 만든 AngularJS의 서비스 객체 중 하나지만, AngularJS의 $q 객체는 크리스 코우얼의 q와는 차이가 있다. $q 서비스의 종속물은 $rootScope 서비스다.

$q 객체의 메소드로는 다음과 같은 것들이 있다.

1. all(프라미스배열) 메소드

   모든 입력 프라미스가 해독될 때 해독되는 다수의 프라미스를 하나의 프라미스로 묶는 메소드다.

   - 매개변수: { Array.〈프라미스〉 } 프라미스 배열

   반환값: { 프라미스 } - 값 배열로 해독될 하나의 프라미스를 반환한다. 사용되는 값 배열의 각 값은 같은 인덱스에 위치한 프라미스에 일대일 대응된다. 프라미스 중 어느 하나라도 '거부됨'으로 해독되면 반환되는 프라미스 역시 '거부됨' 상태(이벤트)로 해독된다.

2. defer( ) 메소드

   나중에 끝마칠 작업을 대변하는 지연 객체를 생성한다.

   - 반환값: { Deferred } - deferred의 새로운 인스턴스를 반환한다.

04 https://github.com/kriskowal/q

3. reject(사유) 메소드

'특정 사유로 거부됨'으로 해독될 프라미스를 생성한다. reject API는 프라미스 체인에서 '거부됨'을 전달할 때 사용한다. 프라미스 체인의 맨 끝에 있는 프라미스를 처리할 때는 신경 쓸 필요가 없다.

지연/프라미스 구조를 이미 잘 알고 있는 try/catch/throw 구조와 비교하면, reject 메소드를 자바스크립트의 throw 키워드라고 생각하면 된다. 그리고 에러 통지는 throw문 대신에 reject 메소드를 통해 생성된 '거부됨' 상태를 반환하는 식으로 현재의 프라미스에서 파생된 프라미스에 전송한다.

```
promiseB = promiseA.then(function(result) {
    // 성공할 경우의 코드를 여기 넣자 - 기능을 수행하고
    // promiseB를 기존 값이나 새 값으로 해독할 것
    return result;
}, function(reason) {
    // 에러할 경우의 코드를 여기 넣자 - 가능하면 에러를 처리하고
    // promiseB를 newPromiseOrValue로 해독하든지,
    // promiseB에 '거부됨'을 전달할 것
    if (canHandle(reason)) {
        // 에러를 처리하고 복원하는 코드를 여기 넣을 것
        return newPromiseOrValue;
    }
    return $q.reject(reason);
});
```

- 매개변수: { 사유 } - 거부 사유를 대변하는 상수, 메시지, 예외, 객체 중 하나다.
- 반환값: { 프라미스 } - '특정 사유로 거부됨'으로 이미 해독이 완료된 프라미스를 반환한다.

4. when(값) 메소드

값이나 then 메소드가 들어 있는 프라미스로 산출될 가능성이 있는 객체를 $q 객체 안에 캡슐화한다. when 메소드는 프라미스가 들어 있을지 없을지 불확실한 객체를 처리할 때나 프라미스의 출처를 신뢰할 수 없을 때 사용하면 적절하다.

- 매개변수: { 값 } - 값 또는 프라미스다.
- 반환값: { 프라미스 } - 매개변수로 전달한 값이나 프라미스에 대한 프라미스를 반환한다.

1 | 서버와 통신하기

『AngularJS 기초편』(한빛미디어, 2013)에서는 AngularJS 애플리케이션을 구성하는 방법, AngularJS의 다양한 부분이 서로 연동되는 원리, AngularJS를 사용한 템플릿의 동작 원리를 알아보았다. 그 내용만으로도 세련된 애플리케이션을 만들 수는 있지만, 대부분은 클라이언트 측 애플리케이션에 그쳤다. 『AngularJS 활용편』에서는 『AngularJS 기초편』에서 배운 내용을 바탕으로 확장된 기능을 배울 것이다. AngularJS에 대한 기초적인 지식이 없다면, 『AngularJS 기초편』을 꼭 학습하길 바란다.

『AngularJS 기초편』에서 $http 서비스를 사용한 서버 측 통신에 대해서 설명했는데, 이 장에서는 $http 서비스를 응용해 실무 애플리케이션을 제작하는 방법에 대해 알아볼 것이다.

AngularJS를 사용해 서버와 통신하는 방법(추상화의 최하위 계층 수준에서 그리고 래퍼wrapper를 이용한 방법)을 알아보자. 또한, AngularJS의 내장 캐시 시스템을 사용해 애플리케이션의 실행 속도를 높이는 방법에 대해서도 알아볼 것이다. SocketIO를 사용해 AngularJS 실시간 애플리케이션을 개발하는 방법은 지시어로 SocketIO를 래핑(캡슐화)하는 4장 예제를 참고하면 되므로, 이 장에서는 SocketIO에 대해서는 다루지 않겠다.

1.1 $http 서비스를 사용한 통신

AJAX 애플리케이션에서 XMLHttpRequest를 사용해 서버에 요청하는 일반적인 절차는 XMLHttpRequest 객체로 핸들을 가져와서 요청을 서버에 보내고 에러 코드를 검사한 후 최종적으로 서버에서 받은 응답을 처리하는 것이다. 이 과정을 코드로 작성하면 다음과 같다.

```
var xmlhttp = new XMLHttpRequest();

xmlhttp.onreadystatechange = function() {
    if (xmlhttp.readystate == 4 && xmlhttp.status == 200) {
        var response = xmlhttp.responseText;
    } else if (xmlhttp.status == 400) {  // 또는 4 계열의 어떤 상태 코드든
                                         // 상관없음.
        // 여기에 적절한 에러 처리 코드를 넣자.
    }
};

// 연결 설정
xmlhttp.open("GET", "http://myserver/api", true);

// 요청 보내기
xmlhttp.send();
```

앞의 코드에서 보듯이 XMLHttpRequest를 사용한 서버 요청 방식은 간단하고 일반적이면서 자주 반복되는 작업임에도 작성해야 할 코드는 복잡하다. 반복적으로 해야 할 작업이라면 래퍼를 작성하거나 라이브러리를 사용하는 것이 적절할 때가 많다.

AngularJS의 XHR API(XMLHttpRequest)는 프라미스Promise 인터페이스를 따른다. XHR은 비동기 메소드 호출이므로 서버가 언제 응답을 반환할지 정확한 날짜와 시간을 알 수 없다(즉시 반환하기도 한다). 프라미스 인터페이스는 서버의 응답이 처리될 방식을 확실히 보증하므로, 프라미스는 예측 가능한 방식으로 사용될 수 있다.

서버에서 사용자 정보를 가져오는 코드를 작성해보자. XHR API가 /api/user에서 이용가능하고 URL 매개변수로 id를 받는다면, AngularJS의 코어 $http를 사용한 XHR 요청은 다음과 같이 작성할 수 있다.

```
$http.get('api/user', { params : { id : '5' }
}).success(function(data, status, headers, config) {
    // 여기엔 성공했을 때 수행할 작업을 넣자.
}).error(function(data, status, headers, config) {
    // 여기엔 에러 처리를 넣자.
});
```

jQuery 개발 경험이 있다면 AngularJS와 jQuery가 비동기 요청을 주고받는 방식이 비슷하다는 것을 알 수 있을 것이다.

앞의 예제에 사용한 $http.get 메소드는 AngularJS의 코어 $http 서비스에 들어 있는 편의 메소드convenience method 중 하나다. 마찬가지로, AngularJS를 이용해 같은 URL 매개변수와 POST 데이터를 사용한 POST 요청을 하려면 다음과 같이 작성하면 된다.

```
var postData = {
    text : '긴 blob 타입의 문자열'
};
// 다음 행이 params를 통해 URL에 덧붙여져서,
// /api/user?id=5로의 포스트 요청이 만들어진다.
var config = {
    params : {
        id : '5'
    }
};
$http.post('api/user', postData, config).success(
        function(data, status, headers, config) {
            // 여기엔 성공 시에 수행할 작업을 넣자.
        }).error(function(data, status, headers, config) {
    // 여기엔 에러 처리를 넣자.
```

```
});
```

다음에 나열한 것 외에도 일반적인 요청 타입에는 대부분 이와 비슷한 편의 메소드가 있다.

- GET
- HEAD
- POST
- DELETE
- PUT
- JSONP

1.1.1 부가 요청 옵션 설정

다음과 같은 상황이라면 내장된 표준 요청 옵션만으로는 충분치 않다.

- 요청에 권한 부여 헤더를 추가할 때
- 요청에 대한 캐시 처리 방식을 변경할 때
- 보내는 요청이나 받는 응답을 일정한 설정 방식으로 변환할 때

위와 같은 상황에서는 환경설정을 위한 옵션 객체를 통해 요청을 설정할 수 있다. 앞의 예제에서는 config 객체를 사용해서 URL 옵션 매개변수를 지정했다. 그런데 앞에서 살펴본 GET 방식과 POST 방식에도 편의 메소드가 있다. 그러한 편의 메소드의 호출 방법은 다음과 같다.

```
$http(config)
```

앞의 메소드를 호출하는 기본적인 형식을 의사 코드pseudo-code로 나타내면 다음과 같다.

```
$http({
    method: 문자열,
    url: 문자열,
    params: 객체,
    data: 문자열 또는 객체,
    headers: 객체,
    transformRequest: function transform(data, headersGetter) 또는 함수 배열,
    transformResponse: function transform(data, headersGetter) 또는 함수 배열,
    cache: 부울값 또는 캐시 객체
    timeout: 숫자,
    withCredentials: 부울값
});
```

GET과 POST를 비롯한 편의 메소드 안에서 메소드가 설정되므로 개발자가 직접 지정하지 않아도 된다. config 객체는 $http.get 메소드와 $http.post 메소드에 마지막 인자로 전달되므로 편의 메소드 중 어느 것을 사용해도 config 객체를 활용할 수 있다. 다음과 같은 키를 설정한 config 객체를 전달하면 이미 이뤄진 요청을 수정할 수 있다.

method

GET이나 POST 같이 HTTP 요청 타입을 나타내는 문자열

url

요청되는 리소스의 절대 URL이나 상대 URL을 나타내는 문자열

params

다음과 같이 URL 매개변수로 변환될 키와 값을 나타내는 문자열-문자열 객체(정확히 말하면 맵)

```
[{키1: '값1', 키2: '값2'}]
```

앞의 코드는 다음과 같이 변환되어 URL 뒤에 붙는다.

```
?키1=값1&키2=값2
```

값에 문자열이나 숫자 대신 객체를 사용하면 그 객체는 JSON 문자열로 변환된다.

data

요청 메시지 데이터로써 보낼 문자열이나 객체

timeout

요청이 타임아웃 처리되기 전의 밀리 초 단위의 대기 시간

다음 절에서 설정할 수 있는 옵션 몇 가지를 더 설명하겠다.

1.1.2 HTTP 헤더 지정

AngularJS에는 보내는 모든 요청에 적용되는 기본 헤더가 있는데, 그중 일부는 다음과 같다.

```
1. Accept: application/json, text/plain,
2. X-Requested-With: XMLHttpRequest
```

특수 헤더를 설정하고 싶을 땐 다음과 같은 두 가지 방법을 이용하면 된다.

보내는 모든 요청에 헤더를 적용하는 첫 번째 방법은 특수 헤더를 AngularJS의 기본 헤더에 포함시키는 것이다. 이러한 기본 헤더는 $httpProvider.defaults.headers 환경설정 객체에서 설정한다. 그리고 설정 단계는 보통 애플리케이션을 설정하는 config 부분에서 이뤄진다. 따라서 모든 GET 요청을 대상으로 '정보 수집 금지(DO NOT TRACK)'를 설정하되 Requested-With 헤더를 모든 요청에서 삭제하려면 다음과 같이 작성하면 된다.

```
angular.module('MyApp',[]).config(function($httpProvider) {
    // AngularJS의 기본 헤더인 X-Request-With를 삭제.
    delete $httpProvider.default.headers.common['X-Requested-With'];
    // 모든 GET 방식 요청을 대상으로 'DO NOT TRACK'(정보 수집 금지)를 지정
    $httpProvider.default.headers.get['DNT'] = '1';
});
```

특정 요청들만을 대상으로 할 때는 특수 헤더를 설정한다. 다만 기본 헤더를 사용하지 않으려면 특수 헤더를 config 객체에 포함시켜 $http 서비스로 전달하면 된다. 개발자 정의 헤더는 다음과 같이 URL 매개변수와 함께 두 번째 매개변수에 넣어 GET 요청에 전달한다.

```
$http.get('api/user', {
    // Authorization 헤더를 설정. 실제 애플리케이션이라면,
    // 서비스에서 인증 토큰을 가져와야 할 것이다.
    headers: {'Authorization': 'Basic Qzsda231231'},
    params: {id: 5}
}).success(function() {
    // 여기엔 성공하면 수행할 코드를 넣자.
});
```

애플리케이션 내부에서 인증을 처리하는 방법을 보여주는 완전한 예제는 4장을 참고하기 바란다.

1.1.3 응답을 캐시에 저장

AngularJS에는 HTTP GET 요청에 즉시 사용할 수 있는 간단한 캐시 시스템이 있다. 이 캐시 시스템은 모든 요청에 사용되지 않게 기본으로 설정되어 있는데, 다음과 같이 작성하면 요청에 캐시를 사용하도록 설정할 수 있다.

```
$http.get('http://server/myapi', {
    cache: true
}).success(function() {
    // 여기엔 성공하면 처리할 작업을 넣자.
});
```

앞의 코드처럼 캐시를 사용하도록 설정하면 다음과 같은 장점이 있다.

첫째, AngularJS는 서버에서 받은 응답을 캐시에 저장해 두었다가 같은 URL로 요청을 다시 보낼 때 캐시에 저장해둔 응답을 반환한다.

둘째, 캐시는 신속하고 영리해서 동시에 한 URL에 여러 요청을 해도 하나의 요청만 서버로 전송하며, 그 요청에 대한 응답만 반환한다.

그러나 이런 점은 사용성 관점과 상충할 수도 있다. 사용자 입장에선 이전의 결과가 일단 표시된 후에 새로운 결과가 불쑥 표시되기 때문이다. 예를 들어, 사용자가 어떤 항목을 막 클릭하려던 찰나에 그 항목이 갑자기 변경될 수도 있다. 캐시가 보낸 응답이든 서버가 보낸 응답이든 비동기적인 성질은 같으므로, 당연히 코드는 처음 요청을 수행했을 때와 똑같은 동작을 할 수밖에 없다.

1.1.4 요청 변환과 응답 변환

AngularJS는 $http 서비스를 통해 이뤄지는 모든 요청과 응답을 대상으로 다음과 같은 기본적인 변환을 적용한다.

요청 변환

요청된 config 객체의 data 속성에 객체가 들어 있으면, 그 객체를 JSON 형식으로 직렬화한다.

응답 변환

XSRF 접두어가 붙어 있으면 그 접두어를 떼어낸다. JSON 응답이 감지되면 그 응답을 JSON 파서로 역직렬화한다.

변환이 적용되는 것을 원치 않거나 자신의 변환을 추가하려면 config 객체 안에 함수를 넣어 전달하면 된다. 이 함수는 HTTP 요청/응답의 내용, 헤더를 가져와서 직렬화하고 수정한 후 응답한다. 이런 config 함수는 transformRequest 키와 transformResponse 키를 사용해 설정한다. 두 키는 모듈의 config 함수에 들어 있는 $httpProvider 서비스를 사용해 설정한다.

그렇다면 transformRequest 키와 transformResponse 키는 언제 사용할까? jQuery 작업 처리 방식에 더 적합하게 설정된 서버가 있다고 하자. 이 서버는 POST 요청 데이터를 JSON 형식 {key1: val1, key2: val2}이 아니라 문자열 형식 {key1=val1&key2=val2}을 기대한다. 요청 시마다 받은 문자열을 목적에 따라 수정하거나 transformRequest 호출을 개별적으로 추가할 수도 있지만, 필자는 보내는 모든 호출을 JSON 형식에서 문자열 형식으로 변환되도록 범용 transformRequest를 추가하였다. 작성한 코드는 다음과 같다.

```
var module = angular.module('myApp');

module.config(function($httpProvider) {
    $httpProvider.defaults.transformRequest = function(data) {
        // jQuery의 param 메소드를 이용해서
        // JSON 데이터를 문자열 형식으로 변환하자.
        return $.param(data);
    };
});
```

1.2 단위 테스트 실시

앞에서 $http 서비스의 사용법과 다양한 설정 방법을 알아보았다. 그러나 단위 테스트를 작성하고 테스트가 원활히 돌아가는지 확인하려면 어떻게 해야 할까?

계속 말하지만 AngularJS는 테스트를 염두하고 설계된 프레임워크이므로, 단위 테스트에서 직접 올바른 요청이 이뤄지는지 여부를 테스트하고 응답이 처리되는 시점과 방식까지도 제어할 수 있는 목 백엔드mocked backend가 있다.

그럼, 서버에 요청하고 데이터를 가져와서 해당 스코프scope에서 데이터가 뷰view에 특정한 형식으로 표시되게 하는 컨트롤러를 어떻게 단위 테스트하면 되는지 알아보자.

NamesListCtrl은 하나의 용도로 사용하는 아주 간단한 컨트롤러다. 이 용도는 필자가 작성한 names API를 호출해서 해당 스코프에 모든 이름을 저장하는 것이다.

```
function NamesListCtrl($scope, $http) {
    $http.get('http://server/names', {params: {filter: 'none'}}).
        success(function(data) {
            $scope.names = data;
        }
    );
}
```

앞의 코드를 어떻게 단위 테스트할까? 단위 테스트는 다음과 같은 사항이 가능해야 한다.

- NamesListCtrl는 모든 종속물을 검색해서 제대로 주입할 수 있어야 한다.
- NamesListCtrl는 로딩되는 즉시 서버에 요청해서 이름을 가져와야 한다.
- NamesListCtrl는 받은 응답을 해당 스코프의 names 변수에 저장해야 한다.

테스트 안에 컨트롤러를 생성하고, 그 안에 스코프와 가짜 HTTP 서비스를 주입하는 방법도 있지만, AngularJS가 제품 코드에 적용하는 방식대로 테스트를 생성하는 방법이 좋다. 이 방법은 훨씬 복잡하지만 사용하는 것을 권장한다. 그 이유는 다음 코드를 보면 알 수 있다(소스 코드 주석을 잘 살펴보기 바란다).

```
describe('NamesListCtrl', function(){
    var scope, ctrl, mockBackend;

    // AngularJS가 다음을 테스트 안에 주입한다.
    beforeEach(inject(function(_$httpBackend_, $rootScope, $controller) {
        // 다음은 가상의 백엔드이므로, 요청과 서버의 응답을 제어할 수 있다.
        mockBackend = _$httpBackend_;

        // 컨트롤러가 생성되면 다음 호출이 시작될 것이므로
        // 컨트롤러 생성에 앞서 예상 형식을 지정.
        mockBackend.expectGET('http://server/names?filter=none').
            respond(['Brad', 'Shyam']);
        scope = $rootScope.$new();

        // AngularJS가 제품화 코드에 적용하는 방식대로 컨트롤러를 생성.
        ctrl = $controller(PhoneListCtrl, {$scope: scope});
    }));

    it('로딩되면 서버에서 이름을 가져와야 함', function() {
        // 초기에 이 요청은 응답을 반환하지 않았음.
        expect(scope.names).toBeUndefined();

        // 현재 보내는 모든 요청에 대한 응답을 반환하라고 가상 백엔드에게 명령
        mockBackend.flush();

        // 이제 names를 스코프 상에서 지정해야 함.
```

```
        expect(scope.names).toEqual(['Brad', 'Shyam']);
    });
});
```

1.3 REST 기반의 리소스 사용하기

$http 서비스는 하위 수준으로 구현되어 있어 XHR 요청을 할 수 있으면서도 많은 부분을 제어할 수 있고 유연성도 뛰어나다. 그러나 대체로 개발자가 처리하는 것은 개인에 대한 세부 속성, 개인 객체, 신용카드 객체 같이 특정 속성과 메소드로 캡슐화된 객체와 객체 모델이다.

이렇게 수정할 수 있는 것이 제한적일 경우, 객체 모델을 파악해서 표현하는 자바스크립트 객체를 작성할 수 있다면 좋지 않을까? 상태는 서버에 그대로 유지된 채 save나 update 같은 자바스크립트 객체의 속성들만 수정할 수 있다면 어떨까?

$resource를 사용하면 이러한 기능을 만들 수 있다. AngularJS 리소스를 사용하면 객체 모델을 정의할 수 있으며 다음과 같은 것들을 지정할 수 있다.

- 리소스의 서버 측 URL
- XHR 요청에서 주로 볼 수 있는 매개변수의 타입
- 객체 모델의 특정 기능(예: 신용카드의 charge( ) 메소드)과 비즈니스 로직을 캡슐화하는 부가 메소드(get, save, query, remove, delete 메소드는 기본적으로 들어 있다)
- 응답의 예상 타입(배열 또는 객체)
- 헤더

NOTE_ Angular 리소스는 어떤 경우에 사용하나?

Angular 리소스는 서버 측이 REST 방식으로 동작할 때만 사용해야 한다. 이 장의 끝부분에서 살펴볼 신용카드 예제는 다음과 같은 특징이 있다.

1. /user/123/card로 GET 요청을 보내면 123이라는 사용자의 신용카드 목록이 반환된다.
2. /user/123/card/15로 GET 요청을 보내면 123이라는 사용자의 ID가 15인 신용카드가 반환된다.
3. POST 데이터 안의 신용카드 정보와 함께 /user/123/card로 POST 요청을 보내면 123 사용자의 새 신용카드가 생성된다.
4. POST 데이터 안의 신용카드 정보와 함께 /user/123/card/15로 POST 요청을 보내면 123 사용자의 ID가 15인 신용카드의 정보가 보낸 정보로 업데이트된다.
5. /user/123/card/15로 DELETE 요청을 보내면 123 사용자의 ID가 15인 신용카드가 삭제된다.

개발자의 필요에 따라 서버에 질의할 수 있는 객체를 제공할 뿐만 아니라, $resource를 사용하면 반환 객체를 가지고 마치 영구 저장된 데이터 모델처럼 작업하고, 수정하고, 영구 저장되게 요청할 수도 있다.

ngResource는 별도의 옵션 모듈이다. 따라서 ngResource 모듈을 사용하려면 다음과 같이 해야 한다.

- 작업하는 기본 자바스크립트 파일 안에 angular-resource.js 파일을 인클루딩한다.
- 모듈 종속물 선언 코드 안에 angular.module('myModule', ['ngResource'])와 같은 식으로 ngResource를 포함시킨다.
- 필요하면 inject $resource를 사용한다.

ngResource 메소드를 사용해 리소스를 생성하는 방법을 살펴보기 전에, 기본 $http 서비스를 사용하여 리소스를 생성하는 방법부터 알아보자. 신용카드 예제의 리소

스는 신용카드를 가져오고 질의, 저장할 수 있어야 하며 신용카드 결제도 가능해야 한다.

이러한 기능을 구현하는 첫 번째 방법은 다음과 같다.

```
myAppModule.factory('CreditCard', [ '$http', function($http) {
    var baseUrl = '/user/123/card';
    return {
        get : function(cardId) {
            return $http.get(baseUrl + '/' + cardId);
        },
        save : function(card) {
            var url = card.id ? baseUrl + '/' + card.id : baseUrl;
            return $http.post(url, card);

        },
        query : function() {
            return $http.get(baseUrl);
        },
        charge : function(card) {
            return $http.post(baseUrl + '/' + card.id, card, {
                params : {
                    charge : true
                }
            });
        }
    };
} ]);
```

아니면, 두 번째 방법으로 다음과 같이 애플리케이션 전반에 리소스를 반영하는 Angular 서비스를 간단히 작성할 수도 있다.

```
myAppModule.factory('CreditCard', ['$resource', function($resource) {
    return $resource('/user/:userId/card/:cardId',
        {userId: 123, cardId: '@id'},
        {charge: {method:'POST', params:{charge:true}, isArray:false});
}]);
```

이제 AngularJS 주입 객체에 CreditCard를 요청할 때마다 Angular 리소스를 가져오는데, 이 리소스엔 기능이 시작되는 메소드 몇 개가 기본으로 들어 있다. Angular 리소스에 포함된 각 메소드가 무엇이고 어떤 방식으로 동작하는지 표 1-1로 정리했다. 이것을 참고하면 서버를 어떻게 설정해야 할지 알 수 있다.

표 1-1 신용카드 리소스

리소스 함수	방식	URL	예상 반환 타입
CreditCard.get({id: 11})	GET	/user/123/card/11	Single JSON
CreditCard.save({}, ccard)	POST	/user/123/card (post 데이터 ccard와 함께)	Single JSON
CreditCard.save({id: 11}, ccard)	POST	/user/123/card/11 (post 데이터 ccard와 함께)	Single JSON
CreditCard.query( )	GET	/user/123/card	JSON Array
CreditCard.remove({id: 11})	DELETE	/user/123/card/11	Single JSON
CreditCard.delete({id: 11})	DELETE	/user/123/card/11	Single JSON

좀 더 확실한 이해를 위해 신용카드 예제를 보자.

```
// CreditCard 서비스가 여기에 주입된다고 가정하자.

// GET 요청을 보내는 서버 /user/123/card에서 컬렉션을 가져올 수 있다.
```

```
var cards = CreditCard.query();

// 하나의 카드를 가져와서 콜백 함수에서도 사용할 수 있다.
CreditCard.get({ cardId : 456 }, function(card) {
    // 각 항목은 CreditCard의 인스턴스다.
    expect(card instanceof CreditCard).toEqual(true);
    card.name = "J. Smith";
    // GET 이외의 방식이 그 인스턴스에 매핑된다.
    card.$save();

    // 개발자 정의 메소드도 매핑된다.
    card.$charge({ amount : 9.99 });
    // 데이터 {id:456, number:'1234', name:'J. Smith'}와 함께
    // POST 요청을 /user/123/card/456?amount=9.99&charge=true로 보낸다.
});
```

기능의 대부분이 앞의 예제에 들어 있으므로, 중요한 부분만 하나씩 파헤쳐보자.

1.3.1 선언

$resource를 선언하려면 그냥 적절한 매개변수를 전달하면서 주입된 $resource 함수를 호출하면 된다. 어떻게 주입해야 할지 모른다면 『AngularJS 기초편』을 읽어보기 바란다.

$resource 함수는 필수 인자 1개와 옵션 인자 2개를 받는다. 필수 인자는 리소스의 URL이고, 옵션 인자는 기본 매개변수와 리소스에 설정하고자 하는 부가 동작이다.

URL 매개변수는 매개변수 형태로 표기돼 있다. 앞에 붙은 콜론(:)은 매개변수임을 나타낸다. 즉, :userId는 userId 매개변수의 텍스트로 대체되며, :cardId는 cardId 매개변수의 값으로 대체된다. URL 매개변수가 전달되지 않으면 URL 매개변수는 빈 문자열로 대체된다.

두 번째 매개변수는 각 요청과 함께 전달될 기본 매개변수를 처리한다. 이 예제에선 userId에 상수 123을 전달했다. cardId 매개변수는 더욱 중요하다. cardId가 @id라고 하자. 이것은 서버에서 반환된 객체를 사용하고 그 객체의 $save 같은 메소드를 호출하면 cardId 필드가 객체의 id 속성에서 선택될 것임을 나타낸다.

세 번째 인자는 개발자 정의 리소스를 공개할 각종 메소드다. 이에 대해서는 다음 절에서 자세히 알아볼 것이다.

1.3.2 개발자 정의 메소드

$resource를 호출할 때 전달할 세 번째 인자는 리소스에 공개하고자 하는 부가 메소드들이다.

신용카드 예제는 세 번째 인자로 charge 메소드를 전달했다. charge 메소드는 객체 안에서 공개할 메소드명을 키로 사용해서 설정하면 된다. 이 설정을 위해서는 GET이나 POST 같은 요청 방식, 요청 안에 포함시켜 전달해야 하는 매개변수(신용카드 예제에선 charge=true), 반환된 결과가 배열인지 아닌지 여부(신용카드 예제에선 배열이 아님)를 지정해야 한다. 모두 지정했으면 CreditCard.charge( ) 함수를 언제든 호출해도 된다. 실생활에서는 사용자가 결제할 때마다 호출될 것이다.

1.3.3 콜백은 꼭 필요할 때만 사용하자

세 번째로 중요한 것은 리소스 호출의 반환 타입이다. CreditCard.query( ) 호출을 다시 살펴보자. 콜백 안에 신용카드를 지정하지 않고 신용카드의 변수를 직접 지정했음을 알 수 있다. 비동기 서버 요청으로 그 코드가 과연 돌아가기는 할까?

코드가 돌아가는지 아닌지를 걱정하는 것은 당연하지만, 코드는 실제로 정확하고, 잘 돌아간다. CreditCard.query( ) 호출 코드에서 AngularJS는 참조(예상 반환

타입에 따라 객체일 수도 있고 배열일 수도 있음)를 할당했는데, 이 참조는 나중에 서버 요청이 반환될 때 내용이 할당된다. 그 때까지 객체는 빈 상태로 유지된다.

AngularJS 애플리케이션에서 가장 일반적인 흐름은 서버에서 데이터를 가져와서 변수에 할당하고 그 변수를 템플릿에 표시하는 것이므로, 이러한 단축 방법도 괜찮다. 컨트롤러 코드에서는 단지 서버를 호출하고 적절한 스코프 변수에 반환 값을 할당하여 값이 반환되면 템플릿이 그 값을 표시하게 하면 된다.

이 방법은 원하는 비즈니스 로직이 반환 값을 이용해 실행되는 경우에는 통하지 않는다. 이럴 때는 CreditCard.get() 함수 안에서 쓰이는 콜백을 이용해야 한다.

1.3.4 간소화된 서버 측 작업

단축 반환 타입과 콜백 중 어느 것을 사용하든지 반환 객체에 대해 알아야 할 몇 가지가 있다.

반환 값은 기존의 평범한 자바스크립트 객체가 아니라 사실상 resource 타입의 객체다. 그래서 반환 값에는 서버가 반환하는 값 뿐만 아니라 $save()와 $charge() 같은 부수적인 기능도 들어 있다. 결과적으로 데이터 가져오기, 수정하기, 서버에 수정 내용 저장하기(CRUD 애플리케이션의 가장 일반적인 기능) 같은 서버 측 기능을 쉽게 수행할 수 있다.

1.3.5 ngResource 단위 테스트하기

ngResource는 캡슐화 모듈이며, $http 코어 AngularJS를 기반으로 한다. 따라서 단위 테스트 방법도 같다. 『AngularJS 기초편』의 $http 서비스에서 살펴봤던 단위 테스트 예제와 방법이 거의 다르지 않다. 단지 리소스에 의해 이뤄지리라 예상되는 최종 요청만 파악해서 가상의 $http 서비스에 전달하면 되고, 나머지는 전부 똑같다. 앞의 코드를 대상으로 한 테스트는 다음과 같다.

```
describe('신용카드 리소스', function() {
    var scope, ctrl, mockBackend;

    beforeEach(inject(function(_$httpBackend_, $rootScope, $controller) {
        mockBackend = _$httpBackend_;

        scope = $rootScope.$new();
        // CreditCard 리소스가 컨트롤러에 사용된다고 전제함.
        ctrl = $controller(CreditCardCtrl, {
            $scope : scope
        });
    }));

    it('신용카드 목록을 가져왔어야 함', function() {
        // CreditCard.query()에 대한 예상 호출 방식을 지정.
        mockBackend.expectGET('/user/123/card').respond([ {
            id : '234',
            number : '11112222'
        } ]);

        ctrl.fetchAllCards();

        // 처음에 이 요청은 응답을 반환하지 않았다.
        expect(scope.cards).toBeUndefined();

        // 가상의 백엔드에게 현재 보내는 모든 요청에 대한 응답을 반환하라고
        // 명령함.
        mockBackend.flush();

        // 이제 신용카드의 스코프를 설정함.
        expect(scope.cards).toEqualData([ {
            id : '234',
```

```
            number : '11112222'
        } ]);
    });
});
```

이 테스트는 간단한 $http 단위 테스트와 매우 유사한데 사소한 차이가 하나 있다. expect 함수에서 간단한 equals 메소드를 사용하지 않고 특수한 toEqualData 호출을 사용했다는 점이다. expect 함수는 기민해서 ngResource가 객체에 추가하는 부가 메소드를 무시한다.

1.4 $q와 프라미스

이제까지 AngularJS가 비동기 지연 API를 어떤 식으로 구현하는지 살펴봤다. AngularJS가 소속된 API를 구성하는 방식은 프라미스Promise 제안 문서를 바탕으로 한다. 프라미스 제안 문서에는 기본적으로 비동기 요청에 대해 다음과 같은 사항을 준수하도록 기술되어 있다.

- 비동기 요청은 반환 값 대신 프라미스를 반환해야 한다.
- 프라미스에는 then 함수가 있어야 한다. 이 then 함수는 두 개의 인자를 받는다. 첫 번째 인자는 resolved(해독됨) 이벤트나 success(성공) 이벤트를 처리하는 함수고, 두 번째 인자는 rejected(거부됨) 이벤트나 failure(실패) 이벤트를 처리하는 함수다. 두 함수는 결과나 거절 사유와 함께 호출된다.
- 결과가 해독되는 순간 바로 앞에서 말한 두 개의 콜백 함수 중 하나가 반드시 호출돼야 한다.

대개 지연 q 객체를 구현한 코드는 앞의 제안 문서에 기술된 방식을 따르지만, AngularJS의 $q 함수 구현 코드는 다음과 같은 이유로 다르다.

- $q 객체는 AngularJS를 완벽히 인식하므로 스코프 모델과 연동된다. 따라서

결과를 더욱 빠르게 전달할 수 있고 UI의 깜박임과 업데이트를 줄일 수 있다.

- AngularJS 템플릿 역시 $q 객체의 프라미스를 인식하므로, 템플릿은 결과값을 통지 받을 프라미스 대신에 템플릿 스스로를 결과값처럼 처리할 수 있다.
- AngularJS에는 흔히 수행되는 비동기 작업들에 필요한 기본적이고 가장 중요한 기능만 구현되어 있어 메모리 공간을 적게 차지하는 편이다.

그런데 이렇게까지 해서 대체 뭘 하려는 것일까? 비동기 함수를 사용하다가 맞닥뜨릴 수 있는 다음과 같은 일반적인 문제를 하나 살펴보자.

```
fetchUser(function(user) {
    fetchUserPermissions(user, function(permissions) {
        fetchUserListData(user, permissions, function(list) {
            // 여기엔 표시할 데이터 리스트로 기능을 수행하는 코드가 들어감.
        });
    });
});
```

앞의 코드는 자바스크립트로 코딩할 때 개발자들이 처하게 되는 '절망의 피라미드 pyramid of doom 패턴'[01]이다. 반환 값의 비동기적 특성이 프로그램의 동기적 요구와 대치하다가 결국 여러 함수가 포함되는 구조로, 현재 문맥을 파악하기가 훨씬 힘들어진다. 게다가 에러 처리에도 문제가 있다. 에러를 처리하는 최선의 방법은 무엇일까? 각 단계에서 에러를 처리할 것인가? 그러면 지저분해지기까지 한다. 이러한 앞의 코드의 문제점을 바로잡으려면 프라미스 제안 문서에 나온 then 메소드 개념을 활용해야 한다. then 메소드는 성공할 경우 실행할 함수와 에러가 날 경우 실행할 함수를 받는데, 두 함수를 체인화할 수 있다. 앞의 예제에 프라미스 API를 사용하면 다음과 같이 평탄화(포함 구조로 겹쳐진 함수들을 평이한 코드로 만드는 것)할

01 안티 패턴 중 하나로, 계속되는 포함관계로 코드가 복잡해져서 개발자에게 절망감을 준다고 하여 붙여진 이름이다.

수 있다.

```
var deferred = $q.defer();

var fetchUser = function() {
    // 비동기 호출 후에 응답 값을 전달하면서 deferred.resolve를 호출함.
    deferred.resolve(user);

    // 에러가 날 경우 다음을 호출함.
    deferred.reject('실패 사유');
}

// 마찬가지로, fetchUserPermissions과 fetchUserListData가 처리됨.
deferred.promise.then(fetchUser)
    .then(fetchUserPermissions)
    .then(fetchUserListData)
    .then(function(list) {
        // 여기엔 데이터 리스트를 사용한 기능 수행 코드를 넣자.
     }, function(errorReason) {
        // 어느 단계에든 에러가 있으면 한번에 처리하는 코드를 여기에 넣자.
     }
);
```

앞의 코드에서는 절망의 피라미드 문제가 해결됐고, 체인화[02]를 위한 스코프도 생겼으며, 에러 처리도 한 군데서 이뤄진다. AngularJS의 $q 서비스를 인클루딩해서 비동기 호출을 처리하려면 이 코드를 애플리케이션에 사용하면 된다. 게다가 이 방식으로 작성하면 응답을 가로채는 것도 가능하다.

02 마침표로 여러 함수를 일렬로 이어붙여, 관련 함수들을 실행하는 방법이다.