# 17장 마이크로서비스 구축

마이크로서비스 아키텍처는 서로 연결된 미세한 도메인별 작업을 수행하는 여러 컴포넌트로 구성된 애플리케이션을 구축하는 방식을 제공한다. 이 아키텍처는 애플리케이션 전체를 한 덩어리로 디플로이(deploy)하는 모놀리식(monolithic)한 애플리케이션을 만드는 전통적인 기법과 상반된 기법이다. 마이크로서비스는 각 부분을 물리적으로 분리함으로써 개발과 테스트, 디플로이, 갱신이 더 쉬운 모듈화된 개발을 돕는다.

이 장에서는 마이크로서비스 아키텍처의 기초와 원칙을 설명하고, 코틀린이 마이크로서비스 개발에 어떤 도움을 줄 수 있는지 Ktor와 스프링 부트(Spring Boot) 프레임워크를 통해 설명한다. 스프링 프레임워크는 자바 세계에서 널리 사용 중인 도구로, 최신 버전에서는 코틀린 지원에도 많은 관심을 기울이고 있다. Ktor는, 16장에서 이미 살펴본 것처럼, 여러 가지 유형의 연결된 애플리케이션 개발을 지원하려는 목적으로 만들어졌고 코틀린 기능을 많이 사용한다. 이번 장을 진행하면서 여러분은 간단한 서비스를 구축하고, 이를 통해 향후 어떤 특정 마이크로서비스 프레임워크를 학습할 때 도움이 될 수 있는 토대를 쌓을 수 있을 것이다.

## 구조

* 마이크로서비스 아키텍처
* 스프링 부트 소개
* Ktor를 활용한 마이크로서비스

## 목표

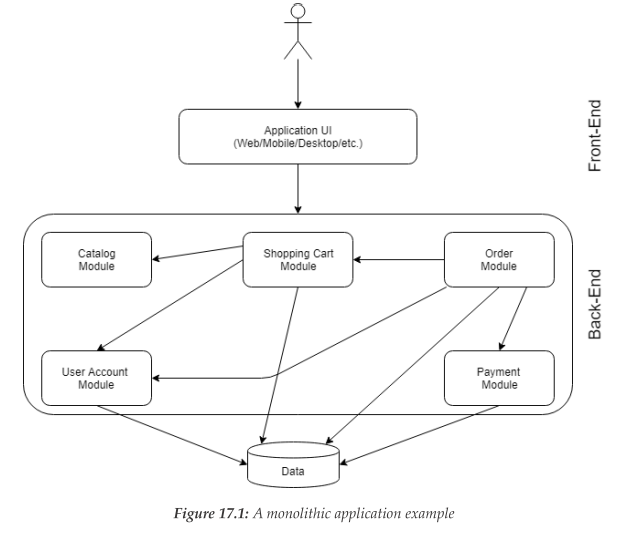
마이크로서비스 아키텍처의 기초를 이해하고 스프링 부트와 Ktor 프레임워크를 사용해 마이크로서비스를 만드는 기본 지식을 배운다.

## 마이크로서비스 아키텍처

마이크로서비스 아키텍처의 아이디어는 모놀리식 애플리케이션(전체를 한꺼번에 디플로이하고 배포하는)을 느슨하게 결합된 경량 서비스들의 집합으로 바꾸는 것이다. 각 서비스는 구체적인 작업을 수행하며, 잘 정의된 프로토콜을 사용해 다른 서비스와 통신한다.

더 구체적인 예를 들어보겠다. 사용자에게 상품 목록을 살펴볼 수 있는 기능과 주문 기능을 제공하는 온라인 스토어 애플리케이션을 만들고 싶다고 가정하자. 모놀리식 애플리케이션 아키텍처를 따르면 그림 17-1과 비슷한 설계에 도달할 수 있다.

그림 17-1: 모놀리식 애플리케이션 예제



[그림번역]

front end - 프런트엔드

back end - 백엔드

application ui(...) - 애플리케이션 UI(웹/모바일/데스크탑 등)

catralog module - 카탈로그 모듈

shopping cart module - 쇼핑 카트 모듈

order module - 주문 모듈

user account module - 사용자 어카운트 모듈

payment module - 페이먼트 모듈

data - 데이터

[그림번역끝]

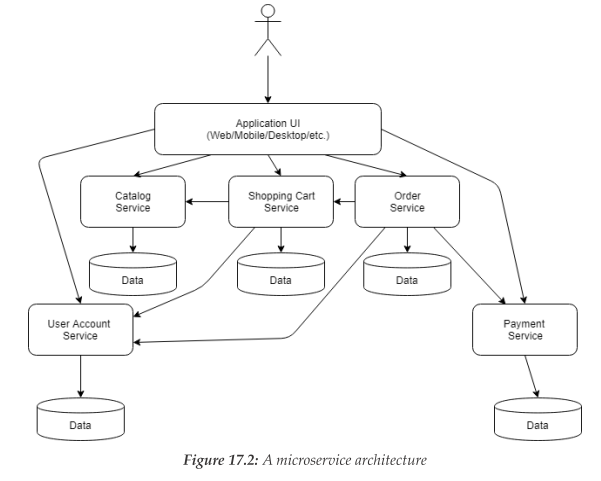
이 구조는 애플리케이션 UI(데스크탑, 웹, 모바일 클라이언트), 비즈니스 로직, 데이터 읽기와 쓰기에 대해 별도의 계층을 사용하는 일반적인 3계층 아키텍처다. 애플리케이션의 백엔드는 도메인에 따른 구체적인 워크플로(workflow)를 구현하며, 카탈로그 제공, 사용자 쇼핑 카트 유지, 주문 넣기/추적/취소, 지불 및 인증, 사용자 프로파일(profile) 관리 등을 담당하는 구체적인 모듈로 나눌 수 있다. 모듈들은 서로 느슨하게 결합되어 있지만, 보통은 독립적으로 배포되거나 디플로이될 수 없기 때문에 서버 애플리케이션이 모놀리스 애플리케이션이 된다.

하지만 이런 접근 방법을 사용하면, 애플리케이션 크기가 커짐에 따라 몇 가지 문제가 발생한다. 코드 베이스를 변경하면 그 변경이 새 기능을 구현해 추가하는 것이든 기존 코드의 버그를 변경하는 것이든 관계없이 전체 애플리케이션을 갱신하고 다시 디플로이해야 하며, 기능을 추가함에 따라 전체 애플리케이션을 시작하는 데 걸리는 시간이 길어지고 새로운 버그가 생길 여지가 있다. 그리고 모놀리식 구조는 애플리케이션의 규모 확장을 방해하기도 한다. 모놀리식 접근 방법을 택하면 전체 애플리케이션의 규모를 한꺼번에 확장시켜야 하는데, 어떤 특정 모듈이나 기능의 규모를 확장시키는 것보다 분명 더 어려운 일이다. 생각해봐야 할 다른 문제로는, 모든 백엔드 모듈이 한 프로세스로 실행되기 때문에 애플리케이션이 메모리 누출(memory leak)이나 나른 종류의 버그에 대해 더 취약해 진다는 점을 들 수 있다.

이런 문제를 해결하기 위해 **서비스 기반 아키텍처**(SOA, Service Oriented Architecture)가 제안됐다. SOA는 모놀리식 애플리케이션을 별도로 개발, 갱신되며 대부분 독립적으로 디플로이 될 수 있는 완결성 있는 서비스들의 집합으로 분해한다. 마이크로서비스는 애플리케이션 아키텍처를 SOA에서 더 발전시켜서 각 서비스를 가능한 작고 단순하게 유지하려고 노력한다. 하지만 실제로는 SOA와 마이크로서비스가 거의 동의어처럼 쓰이기도 한다.

원래의 모놀리식 애플리케이션 설계를 분해하면 그림 17-2와 비슷한 설계에 도달할 수 있다.

그림 17-2: 마이크로서비스 아키텍처



[그림번역]

application ui(...) - 애플리케이션 UI(웹/모바일/데스크탑 등)

catralog service - 카탈로그 서비스

shopping cart service - 쇼핑 카트 서비스

order service - 주문 서비스

user account service - 사용자 어카운트 서비스

payment service - 페이먼트 서비스

data - 데이터

[그림번역끝]

원래 설계의 모듈이 HTTP 같은 네트워크 기반 프로토콜을 사용하는, 같은 기능의 서비스로 대치됐다는 사실을 알 수 있다. 이제 각각의 서비스를 좀 더 독립적으로 개발, 갱신, 설정할 수 있다. 그리고 심지어 각 서비스는 서로 다른 DBMS를 통해 관리되는 데이터베이스를 사용할 수도 있다.

마이크로서비스에 대한 엄밀한 정의는 부족하지만, 실용적인 마이크로서비스는 몇 가지 공통적인 원칙을 바탕으로 구현한다.

* 각 마이크로서비스는 앞에서 본 전자 상거래 예제의 상품 카탈로그 관리, 사용자 쇼핑 카트 관리 등과 같이 어떤 도메인의 구체척인 작업을 수행한다.
* 마이크로서비스는 잘 정의된 프로토콜을 사용해 서로 통신하며, 이 프로토콜은 각 서비스의 API를 구성한다. 가장 일반적인 경우는 HTTP와 XML이나 JSON 형식을 엮어서 복잡한 데이터를 전송하거나, RPC(원격 프로시저 호출)기반의 프로토콜을 사용하는 것이다.
* 마이크로서비스들은 서로 독립적으로 버전을 부여하며, 디플로이되며, 갱신된다.
* 마이크로서비스는 언어나 프레임워크와 무관하다. 이 말은 일반적으로 여러분의 목적에 적합한 어떤 언어로든 마이크로서비스를 구현할 수 있고, 원하는 대로 개발 프레임워크를 선택해도 좋다는 뜻이다. 문제가 되는 것은 여러분의 서비스가 다른 서비스들과 상호작용하기 위해 사용할 프로토콜뿐이다.

지금까지 설명한 내용으로 마이크로서비스 아키텍처가 무엇이고, 어떤 경우에 애플리케이션에 마이크로서비스를 적용할지 기본적으로 이해했을 것이다. 나중에 코틀린 언어의 입장에서 마이크로서비스 프로그래밍이 어떤 형태일지에 대해 보여줄 것이다. 이런 내용은 여러분이 스프링, 넷플릭스, Ktor 등의 여러 기술 스택이나 프레임워크를 살펴볼 때 토대가 된다.

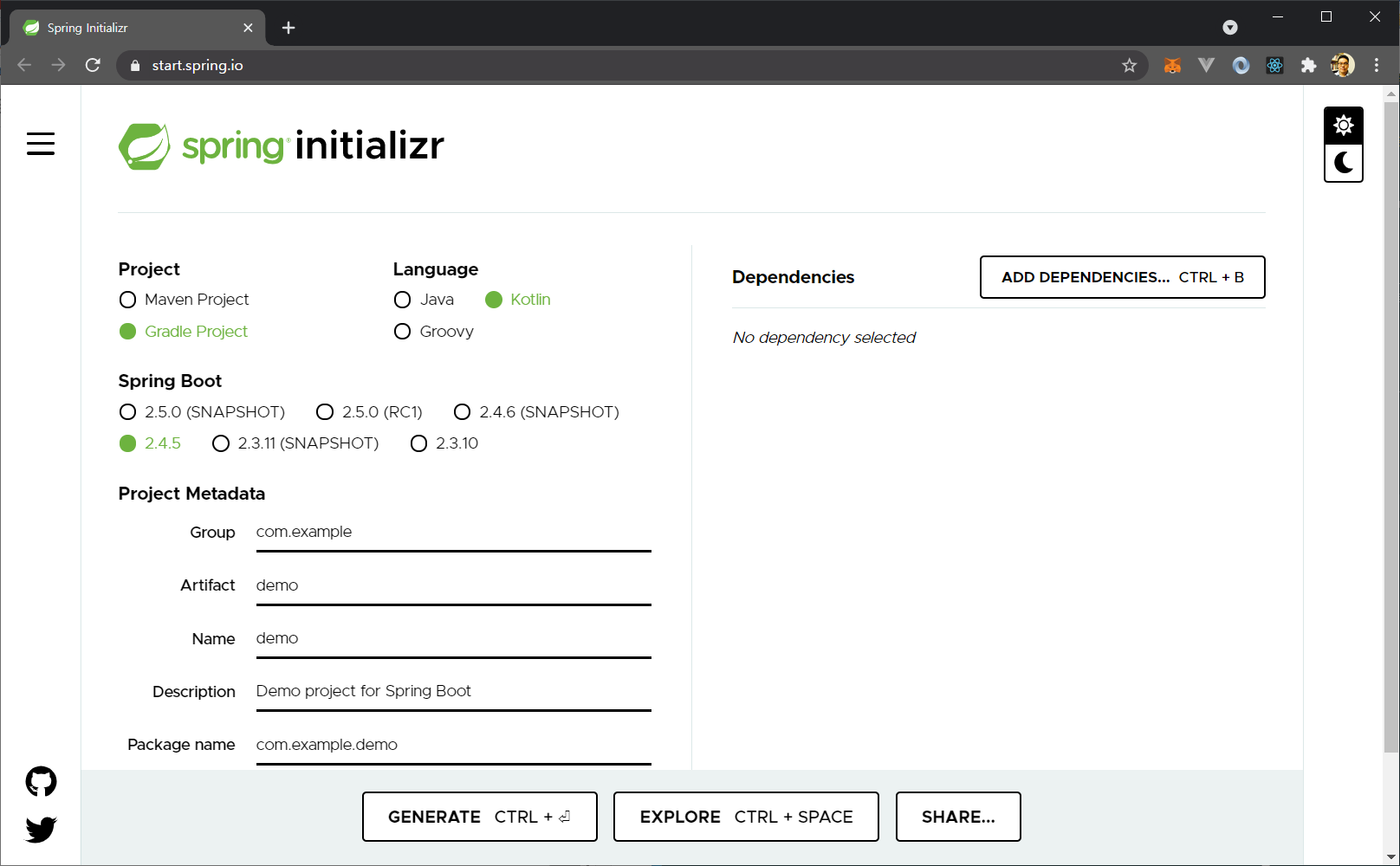
## 스프링 부트 소개

스프링은 가장 흔히 쓰이는 자바 프레임워크이며, 주로 J2EE 플랫폼에 초점을 맞춘 여러 가지 애플리케이션을 구축할 때 사용할 수 있는 다양한 기능을 제공한다. 여기서는 마이크로서비스를 구축할 때 쓸 수 있는 스프링과 코틀린을 조합한 방식의 예로 스프링 부트 프로젝트를 보여줄 것이다. 스프링 부트는 다양한 유형의 스프링 프로젝트 생성과 스프링 프레임워크 설정을 단순화시켜주는 유틸리티의 모음이다. Ktor 등의 다른 예제와 비슷하게 여기서도 스프링 부트 마이크로서비스를 생성하는 여러 단계를 설명한다.

### 프로젝트 설정

스프링 애플리케이션을 시작하는 가장 쉬운 방법은 사용자가 선택한 애플리케이션 유형에 따라 자동으로 프로젝트 뼈대를 만들어주는 스프링 이니셜라이저(Spring Initializr)라는 특별한 웹 도구를 사용하는 것이다. 이 도구를 사용하려면 브라우저에서 https://start.spring.io/를 방문하라(그림 17-3)

그림 17-3: 스프링 이니셜라이저를 사용해 새 프로젝트 생성하기



이 페이지에서는 이니셜라이저가 생성할 프로젝트의 유형에 따라 적용할 사항을 몇 가지 선택할 수 있다.

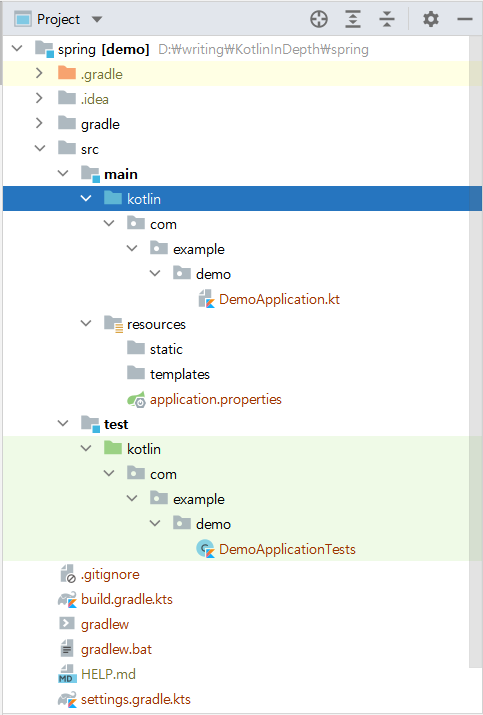
* 프로젝트를 설정하고 소스로부터 빌드할 때 사용할 빌드 시스템의 유형(메이븐/그레이들): 우리는 프로젝트 설정을 더 간결하게 기술할 수 있고 더 유연하기 때문에 그레이들을 선택할 것이다
* 이니셜라이저가 예제 코드를 생성할 때 사용할 주 언어(여기서는 코틀린): 이 선택은 코틀린을 사용하는 것과 관련 있는 프로젝트 설정에도 영향을 끼친다. 예를 들어 메이븐/그레이들 빌드 파일에 몇 가지 의존관계가 추가된다.
* 사용할 스프링 부트 버전: 2021년 5월 스프링 부트 릴리스 버전인 2.4.5를 택한다.
* 아티팩트를 메이븐에 배포할 때 좌표로 사용할 프로젝트 그룹과 아티팩트 ID

추가로 의존관계 필드에서 프로젝트에 포함할 공통 패키지를 몇 가지 선택해야 한다. 서비스에서 HTTP를 사용할 것이기 때문에 웹 지원을 추가해야 한다. ADD DEPENDENCIES 버튼을 클릭해 나오는 대화창의 텍스트 필드에 web을 입력하면 나오는 목록에서 Spring Web을 선택하라.

필요한 옵션을 모두 선택한 다음, GENERATE 버튼을 클릭하면 스프링 이니셜라이저가 생성한 ZIP 파일이 다운로드된다. 이 파일을 인텔리J IDEA에서 열려면 다음 절차를 거쳐야 한다.

1. 압축 파일을 적당한 로컬 디렉터리에 푼다.
2. File | New | Project from Existing Sources... 메뉴 커맨드를 선택하고, 압축 파일을 푼 프로젝트 루트 디렉터리를 선택한 뒤, 빌드 시스템의 종류(그레이들)을 선택한다.
3. IDE가 그레이들 빌드 모델 동기화를 끝내면 그림 17-4와 비슷한 프로젝트 구조를 볼 수 있다.

그림 17-4: 예제 스프링 부트 프로젝트의 구조



Ktor 마법사가 생성한 프로젝트와 마찬가지로, build.gradle.kts 파일에 프로젝트 의존관계 정의가 들어있다. .kts 확장자에 주의하라. 이 확장자는 그레이들 빌드 파일이 그루비가 아닌 코틀린으로 작성됐다는 뜻이다. 이로 인해 스크립트에 적용되는 문법도 안드로이드 예제의 경우와 약간 다르다.

<코드>

import org.jetbrains.kotlin.gradle.tasks.KotlinCompile  
  
plugins {  
 id("org.springframework.boot") version "2.4.5"  
 id("io.spring.dependency-management") version "1.0.11.RELEASE"  
 kotlin("jvm") version "1.4.32"  
 kotlin("plugin.spring") version "1.4.32"  
}  
  
group = "com.example"  
version = "0.0.1-SNAPSHOT"  
java.sourceCompatibility = JavaVersion.VERSION\_11  
  
repositories {  
 mavenCentral()  
}  
  
dependencies {  
 implementation("org.springframework.boot:spring-boot-starter-web")  
 implementation("com.fasterxml.jackson.module:jackson-module-kotlin")  
 implementation("org.jetbrains.kotlin:kotlin-reflect")  
 implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk8")  
 testImplementation("org.springframework.boot:spring-boot-starter-test")  
}  
  
tasks.withType<KotlinCompile> {  
 kotlinOptions {  
 freeCompilerArgs = listOf("-Xjsr305=strict")  
 jvmTarget = "11"  
 }  
}  
  
tasks.withType<Test> {  
 useJUnitPlatform()  
}

</코드>

더 진행하기 전에 코틀린 버전을 더 최신 버전으로 바꾸는 등 몇 가지를 손볼 수도 있다. 다른 그레이들 프로젝트와 마찬가지로 build.gradle 파일을 변경하면 IDEA가 프로젝트 모델을 다시 동기화하라고 제안한다(여러분이 자동 임포트 스위치를 켠 경우에는 자동으로 동기화가 시작된다).

applicaiton.properties 파일에는 스프링의 동작에 영향을 끼치는 여러 가지 프로퍼티가 (단순히 key=value 형식으로) 들어있다. 디폴트로는 빈 파일이 생기지만, 나중에 서비스가 리슨할 포트를 변경하기 위해 이 파일에 설정을 추가한다.

프로젝트 진입점은 DemoApplication.kt 파일이며, 이 파일 안에는 DemoApplication 클래스 정의와 애플리케이션의 시작을 스프링 프레임워크에게 위임하는 main() 함수 정의가 들어있다.

<코드>

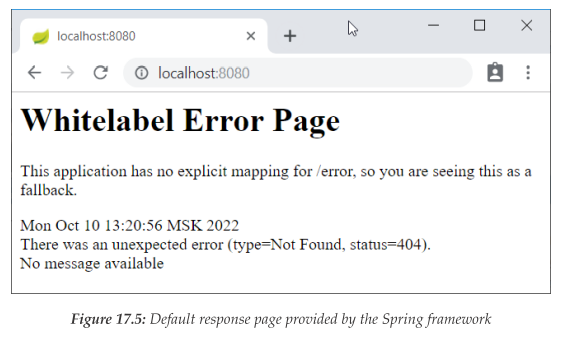
package com.example.demo  
  
import org.springframework.boot.autoconfigure.SpringBootApplication  
import org.springframework.boot.runApplication  
  
@SpringBootApplication  
class DemoApplication  
  
fun main(args: Array<String>) {  
 runApplication<DemoApplication>(\*args)  
}

</코드>

runApplication() 함수는 DemoApplication 인스턴스를 만들고 자동으로 애플리케이션에 필요한 모든 빈(bean)을 인스턴스화하고 연결해준다. 디폴트 설정을 적용한 웹 애플리케이션 같은 경우에는 외부에서 들어오는 요청을 스프링이 제공하는 서블릿(servlet) 쪽으로 디스패치하기 위해, 함께 번들된 톰캣 서버를 시작한다. DemoApplication 인스턴스는 필요할 때 다른 애플리케이션 구성 요소를 주입할 수 있는 전역 컨텍스트 역할을 한다. DemoApplication에 붙은 @SpringBootApplication 애너테이션에 주의하라. 이 애너테이션은 어떤 클래스를 스프링 애플리케이션 컨텍스트로 빠르게 설정할 수 있는 방법을 제공한다.

애플리케이션을 시작하면(IDEA 주 메뉴의 Run 명령을 사용하면 된다), 웹 브라우저 등의 HTTP 클라이언트를 사용해 애플리케이션에 접속할 수 있다. 애플리케이션 안에 실제 요청 처리 코드를 넣지 않았기 때문에 스프링 서블릿은 모든 요청에 대해 표준 오류 페이지로 응답한다. 그림 17-5는 이런 오류 페이지다.

그림 17-5: 스프링 프레임워크가 제공하는 디폴트 응답 페이지



application.properties 파일에 따로 지정하지 않으면 스프링이 클라이언트 요청을 리슨하기 위해 8080번 포트를 사용한다는 점에 유의하라.

<참고> IDE 팁

인텔리J IDEA에 스프링을 지원을 추가하는 플러그인들이 있다. 특히 스프링 이니셜라이저와 비슷하게 스프링을 활용하는 여러 가지 프로젝트를 생성해주는 플러그인이 존재한다. 하지만 이런 플러그인들을 인텔리J 커뮤니티 에디션에서는 쓸 수 없다. 반면에 인텔리J 얼티밋 에디션은 기본적으로 스프링 지원 플러그인을 포함해 제공된다.

</참고>

다음 절에서는 방금 만든 프로젝트 뼈대를 활용해 예제 마이크로서비스를 만들 것이다. 하지만 실제 코딩을 진행하기 전에 가장 먼저 해야 할 일은 서비스가 어떤 일을 하며 클라이언트와 어떻게 의사소통할지 정하는 것이다. HTTP를 통신 프로토콜로 활용하여 마이크로서비스를 소규모 웹 애플리케이션으로 구현하는 일반적인 실무를 보여줄 것이다.

### 서비스 API 결정하기

이 장에서는 서로 통신하는 간단한 서비스를 한 쌍 설계하는 예를 보여줄 것이다. 첫 번째 서비스는 **16장 Ktor를 활용한 웹 개발**에서 본 난수 생성기와 비슷하지만 더 엄격히 정의된 입력과 출력을 가진 API 형태로 사용한다. 다음과 같은 URL로 요청이 들어오면

<코드>

/random/int/from/X/to/Y/quantity/N

</코드>

이 서비스는 범위가 X부터 Y(X와 Y 모두 범위에 포함되는 폐구간)에 속하는 난수를 N개 발생시킨다. 결과는 필드가 2개 있는 JSON이다.

* status: 문제가 있을 때는 오류 메시지 문자열이고 오류가 없이 성공적인 경우에는 null이다
* values: 생성된 정수 난수들의 배열(status가 오류를 표시하는 문자열일 경우에는 빈 리스트)이다

가능한 오류는 다음과 같다.

* X, Y, N 중에 정수가 아닌 값이 있다
* N이 양수가 아니다
* Y < X

URL에 따라 예상되는 서비스 출력을 몇 가지 살펴보자.

표 17-1 Examples of number generator output

|  |  |
| --- | --- |
| URL 예제 | 서비스 응답 |
| /random/int/from/10/to/20/quantity/5 | {"status":null, “values”:[16,17,18,17,12]} |
| /random/int/from/20/to/10/quantity/5 | {"status":"Range may not be empty", “values”:[]} |
| /random/int/from/10/to/20/quantity/-1 | {"status":"Quantity must be positive", “values”:[]} |
| /random/int/from/1X/to/20/quantity/5 | {"status":"Range start must be an integer", “values”:[]} |

이 서비스가 제공하는 다른 기능으로는 부동소수점 수를 생성하는 것이다. 다음과 같은 형태의 URL로 요청이 들어오면

<코드>

/random/float/quantity/N

</코드>

0부터 1(1은 제외)까지 범위에서 N개의 Double을 생성한다.

두 번째 서비스는 비슷한 API를 통해 암호를 제공한다. 다음과 같은 URL에 대해,

<코드>

/password/length/L/quantity/N

</코드>

두 번째 서비스는 N개의 알파벳이나 문자로 이뤄진 L 길이의 문자열을 돌려준다. 암호 생성기도 난수 생성기와 똑같은 출력 형식을 사용한다. 유일한 차이는 돌려주는 values 필드에 수가 아닌 문자열들이 들어있다는 것뿐이다.

표 17-2 Examples of password generator output

|  |  |
| --- | --- |
| URL 예제 | 서비스 응답 |
| /password/length/8/quantity/5 | {"status":null, “values”:[“B0zDWtvG”,”JrSkXl7X”, “oDwR7cp2”,”X8sRfzDW”,”nUcRXzn1”]} |
| /password/length/bbb/quantity/5 | {"status": “Length must be an integer”, “values”:[]} |
| /password/length/-1/quantity/ccc | {"status": “Length must be positive”, “values”:[]} |
| /password/length/8/quantity/-5 | {"status": “Quantity must be positive”,”values”:[]} |

서비스 간의 통신을 보여주기 위해 암호 생성 서비스가 난수 생성 서비스에 의존하게 설계한다. 암호 생성 서비스가 새 암호를 요청받으면 난수 생성 서비스를 호출해서 인덱스들을 난수로 생성하고, 이를 문자로 변환한 다음에 한꺼번에 합쳐서 문자열을 만든다.

이제 서비스 API가 어떤 형태인지 확실해졌으므로, 실제 구현을 시작할 수 있다. 먼저 다른 서비스에 의해 사용될 난수 생성 서비스부터 시작하자.

## 난수 생성기 서비스 구현하기

앞의 **프로젝트 설정** 절에서 본 방법대로 난수 생성 서비스에 대한 새로운 스프링 부트 프로젝트를 생성하자. 서비스 진입점은 거의 달라지지 않는다. 현재 예제의 경우 애플리케이션 클래스와 패키지만 변경한다.

<코드>

package com.example.randomGen  
  
import org.springframework.boot.autoconfigure.SpringBootApplication  
import org.springframework.boot.runApplication  
  
@SpringBootApplication  
class RandomGenerator  
  
fun main(args: Array<String>) {  
 runApplication<RandomGenerator>(\*args)  
}

</코드>

서비스의 실제 비즈니스 로직을 작성하기 전에, JSON 형태로 전달할 데이터를 담을 클래스를 정의해야 한다. 서비스 입력은 URL 경로를 통해 원시 타입의 값으로 전달될 것이기 때문에 필요한 클래스는 출력에 쓸 JSON에 대한 것뿐이다. 코틀린 데이터 클래스를 사용하면 된다.

<코드>

package com.example.randomGen  
  
data class GeneratorResult<T>(  
 val status: String?,  
 val values: List<T>  
)  
  
fun <T>errorResult(status: String) =  
 GeneratorResult<T>(status, emptyList())  
fun <T>successResult(values: List<T>) =  
 GeneratorResult<T>(null, values)

</코드>

errorResult()와 successResult()라는 유틸리티 함수가 있으면 서비스 코드에서 GeneratorResult를 좀 더 쉽게 생성할 수 있다.

서비스의 핵심 로직은 컨트롤러 클래스라는 클래스 내부에 정의된다. 컨트롤러는 클라이언트의 요청을 처리하는 클래스다. 스프링에서는 클래스를 컨트롤러로 정의하려면 @RestController를 클래스 앞에 붙이면 된다. 스프링은 컴포넌트를 스캔하는 과정에서 자동으로 이런 클래스를 적재하고 인스턴스를 생성한다. 자세한 컴포넌트 스캔 과정에 대해서는 여기서 설명하지 않는다. 스프링 프레임워크 문서를 살펴보면 컴포넌트 스캔을 찾을 수 있다(예를 들어 @ComponentScan 애너테이션을 스프링 문서에서 찾아보라).

따라서 컨트롤러 클래스의 기본 뼈대는 다음과 같다.

<코드>

package com.example.randomGen  
  
import org.springframework.web.bind.annotation.RestController  
  
@RestController  
class RandomGeneratorController

</코드>

요청 핸들러를 정의하려면 처리할 요청의 속성을 표현하는 특별한 애너테이션을 컨트롤러 클래스의 메서드에게 붙여야 한다. 예를 들어 @RequestMapping 애너테이션을 사용하면 메서드와 특정 요청 URL을 연결할 수 있다.

<코드>

@RequestMapping("/hello")  
fun hello() = "Hello, World"

</코드>

Ktor에서와 비슷하게, \*와 같은 와일드카드나 파라미터 이름을 사용해 경로 템플릿을 정의할 수 있다. 다음 예제에서는 URL 경로의 마지막 부분({user})이 @PathVariable 애너테이션이 붙은 메서드 파라미터에 자동으로 연결된다.

<코드>

@RequestMapping("/hello/{user}")  
fun hello(@PathVariable user: String) = "Hello, $user"

</코드>

메서드 파라미터 이름이 경로 템플릿에 사용한 변수와 다를 수도 있다. 이 경우에는 @PathVariable 애너테이션의 인자로 경로 템플릿에 사용한 변수를 적으면 된다.

<코드>

@RequestMapping("/sum/{op1}/{op2}")  
fun hello(  
 @PathVariable("op1") op1Str: String,  
 @PathVariable("op2") op2Str: String  
): Any {  
 val op1 = op1Str.toIntOrNull() ?: return "Invalid input"  
 val op2 = op2Str.toIntOrNull() ?: return "Invalid input"  
 return op1 + op2  
}

</코드>

URL 경로 외에 @RequestMapping 애너테이션은 HTTP 메서드(GET, POST)나, 헤더의 내용, 요청 쿼리 파라미터 등의 여러 가지 요청 데이터와 핸들러를 연결시킬 수 있다. @PathVariable과 비슷하게 요청 쿼리 파라미터를 변수에 매치시켜주거나(@RequestParam), 요청 헤더를 변수에 넣거나(@RequestHeader), 세션 데이터를 변수에 넣을(@SessionAttribute) 수 있다. 이런 매핑 방식은 Ktor의 라우팅에서 사용한 방식과 상당히 비슷하다. 다만 Ktor의 경우 어떤 애너테이션으로 표시된 메타데이터가 아니라, 일반적인 코틀린 코드를 사용해 이런 요청 데이터를 변수에 매핑한다는 점이 다를 뿐이다. 여기서 이런 여러 가지 연결 방법을 자세히 설명하지는 않는다. 관심 있는 독자는 docs.spring.io의 문서를 살펴보라.

컨트롤러의 여러 메서드가 똑같은 경로를 접두어로 공유하면, 컨트롤러 클래스에 @RequestMapping 애너테이션을 붙이면서 이 접두어 경로를 넣을 수 있다. 이럴 경우 메서드에 붙은 애너테이션에 있는 경로는 클래스에 붙은 경로에 대한 상대적인 경로가 된다. 예를 들어 다음과 같이 작성하는 대신,

<코드>

@RestController  
class SampleController {  
 @RequestMapping("/say/hello/{user}")  
 fun hello(@PathVariable user: String) = "Hello, $user"  
   
 @RequestMapping("/say/goodbye/{user}")  
 fun goodbye(@PathVariable user: String) = "Goodbye, $user"  
}

</코드>

공통 접두어인 /say를 SampleConroller 클래스의 애너테이션 수준으로 끌어올려서, 다음과 같이 작성할 수 있다.

<코드>

@RestController  
@RequestMapping("/say")  
class RandomGeneratorController {  
 @RequestMapping("hello/{user}")  
 fun hello(@PathVariable user: String) = "Hello, $user"  
   
 @RequestMapping("goodbye/{user}")  
 fun goodbye(@PathVariable user: String) = "Goodbye, $user"  
}

</코드>

이를 염두에 두고, 우리가 정한 서비스 API에 따라 /random/int 경로를 처리할 컨트롤러 메서드를 구현하자.

<코드>

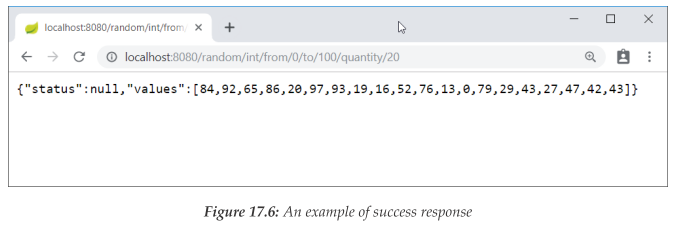
@RequestMapping("/int/from/{from}/to/{to}/quantity/{quantity}")  
fun genIntegers(  
 @PathVariable("from") fromStr: String,  
 @PathVariable("to") toStr: String,  
 @PathVariable("quantity") quantityStr: String  
): GeneratorResult<Int> {  
 val from = fromStr.toIntOrNull()  
 ?: return errorResult("Range start must be an integer")  
 val to = toStr.toIntOrNull()  
 ?: return errorResult("Range end must be an integer")  
 val quantity = quantityStr.toIntOrNull()  
 ?: return errorResult("Quantity must be an integer")  
  
 if (quantity <= 0) return errorResult("Quantity must be positive")  
 if (from > to) return errorResult("Range may not be empty")  
  
 val values = (1..quantity).map { Random.nextInt(from, to + 1) }  
 return successResult(values)  
}

</코드>

/random/float 경로로 들어오는 요청에 따라 Double 값을 만들어내는 방법도 비슷하다. 전체 소스 코드는 이 책 한글판 깃헙([TODO: 깃헙 경로 필요])에서 찾을 수 있다.

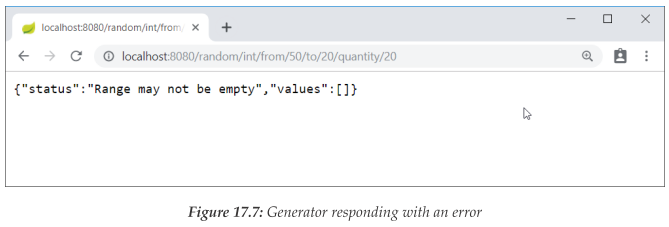
애플리케이션을 시작하고 브라우저에서 서비스에 접근하면 원하는 응답을 받을 수 있다. 그림 17-6은 난수 목록을 받은 예다.

그림 17-6: 성공적인 응답 예



클라이언트의 잘못된 요청을 서비스가 제대로 (오류로) 처리하는지 확인할 수도 있다. 그림 17-7은 요청에서 난수 범위를 50에서 20까지로 지정한 경우다.

그림 17-7: 오류 응답 생성 예



예상대로 서비스가 지정한 구간의 상계(upper bound)가 하계(lower bound)보다 작기 때문에 주어진 구간이 비어있다는 사실을 알려주는 메시지를 반환한다.

### 암호 생성 서비스 구현하기

첫 번째 서비스인 난수 서비스에서 시작하면 두 번째 서비스인 암호 생성 서비스를 쉽게 구현할 수 있다. 앞 예제와 비슷한 스프링 부트 프로젝트를 생성하고 com.example.passwordGen 패키지 안에 코드를 넣자.

난수를 생성하는 경우와 중요한 차이는, 두 번째 서비스(암호 생성)가 첫 번째 서비스(난수 생성)와 통신해야 한다는 것이다. 스프링에 기본적으로 들어있는 RestTemplate 클래스는 다른 웹 애플리케이션으로 요청을 보내고 응답을 받는 작업을 쉽게 할 수 있게 도와준다. 예를 들어,

<코드>

val url = "http://localhost:8080/random/int/from/0/to/10/quantity/5"  
val restTemplate = RestTemplate()  
val result = restTemplate.getForObject(url, GeneratorResult::class.java)  
 as GeneratorResult<Int>

</코드>

위 코드는 0이상 10 이하의 난수가 5개 들어있는 리스트를 돌려받는다.

이 아이디어를 활용해 난수를 암호 문자로 변경하자. 다음은 암호 생성 서비스의 컨트롤러 클래스다.

<코드>

package com.example.passwordGen  
  
import org.springframework.web.bind.annotation.PathVariable  
import org.springframework.web.bind.annotation.RequestMapping  
import org.springframework.web.bind.annotation.RestController  
import org.springframework.web.client.RestTemplate  
  
private val chars = ('a'..'z') + ('A'..'Z') + ('0'..'9')  
  
@Suppress("unused")  
@RestController  
@RequestMapping("/password")  
class PasswordGeneratorController {  
 @RequestMapping("/length/{length}/quantity/{quantity}")  
 fun genPasswords(  
 @PathVariable("length") lengthStr: String,  
 @PathVariable("quantity") quantityStr: String  
 ): GeneratorResult<String> {  
 val length = lengthStr.toIntOrNull()  
 ?: return errorResult("Length must be an integer")  
 val quantity = quantityStr.toIntOrNull()  
 ?: return errorResult("Quantity must be an integer")  
 if (quantity <= 0) return errorResult("Quantity must be positive")  
  
 val prefix = "http://localhost:8080/random/int"  
 val url = "$prefix/from/0/to/${chars.lastIndex}/quantity/$length"  
 val restTemplate = RestTemplate()  
 val passwords = (1..quantity).map {  
 val result = restTemplate.getForObject(  
 url, GeneratorResult::class.java  
 ) as GeneratorResult<Int>  
 String(result.values.map { chars[it] }.toCharArray())  
 }  
  
 return successResult(passwords)  
 }  
}

</코드>

암호 생성 서비스가 자신의 응답을 생성할 때와 난수 생성 서비스의 응답을 받을 때 GeneratorResult 클래스를 사용한다는 점에 유의하라. 이런 단순한 경우에는 정의를 두 번째 서비스 프로젝트로 복사하면 되겠지만, 요청/응답 데이터를 표현하는 클래스가 많이 있는 더 복잡한 시나리오에서는 프로젝트 간에 코드를 공유할 다른 방법이 필요할 것이다. 예를 들어 다중 모듈 프로젝트를 설정하고 그 안에 여러 서비스와 공유 클래스를 모듈로 설정하거나, 공유 코드를 별도의 프로젝트로 추출해서 아티팩트를 만들고 이 아티팩트를 리포지터리에 배포한 다음, 각 서비스가 이 아티팩트에 의존하게 만들 수도 있다.

우리가 만드는 두 서비스가 모두 독립 애플리케이션으로 실행되기 때문에 서로 다른 포트를 리슨해야 한다. 따라서 암호 생성 서비스를 실행하기 전에 서비스가 리슨할 포트가 난수 생성 서비스와 충돌하지 않게 변경해야 한다. 예를 들어 application.properties안에서 포트를 다음과 같이 8081로 설정할 수 있다.

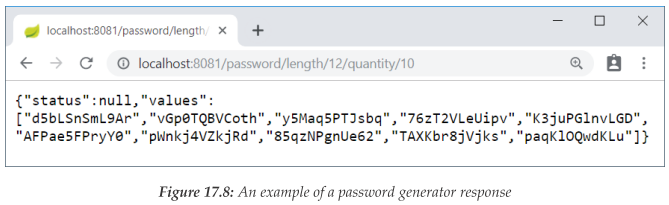
<코드>

server.port=8081

</코드>

이제 두 번째 서비스를 실행하고 localhost:8081/password/length/12/quantity/10 형식의 URL을 사용해 이 서버의 기능을 테스트해보자. URL을 처리할 때 암호 서비스는 난수 생성 서비스에게 여러 번 요청을 보내서 얻어온 응답을 사용해 암호 목록을 생성하여 응답한다. 그림 17-8은 브라우저를 통해 암호 서비스에 접속한 예다.

그림 17-8: 암호 생성 서비스의 응답 예



암호 서비스가 여러 요청을 순차적으로 보내고 동기적으로 처리했다는 점에 유의하라.

<코드>

val passwords = (1..quantity).map {  
 val result = restTemplate.getForObject(  
 url,GeneratorResult::class.java  
 ) as GeneratorResult<Int>  
 String(result.values.map { chars[it] }.toCharArray())  
}

</코드>

즉, 여기서 서비스 스레드는 getForObject()를 호출할 때마다 블럭되며 이 요청이 응답을 받을 때까지 다른 유용한 작업을 진행할 수 없다. 동시에 들어오는 요청의 수가 많아짐에 따라 규모를 확장해야 할 때 이런 동기적 처리가 방해가 될 수 있다. 따라서 이런 경우에는 일반적으로 코틀린 코루틴 라이브러리나 RxJava나 아카(Akka) 등의 비동기 프로그래밍 기법을 사용해야 한다.

다음 절에서 보겠지만, Ktor는 이미 코루틴 라이브러리 위에 구축됐고 그로 인해 일시 정지 함수를 통한 비동기적 계산이 가능한 프레임워크이기 때문에 이런 문제를 피할 수 있다. 이제 Ktor의 기능을 활용하면 암호 생성 서비스를 어떻게 구축할 수 있는지 살펴보자[[1]](#footnote-1).

## Ktor를 사용한 마이크로서비스

앞 장에서 연결된 클라이언트/서버 애플리케이션 개발을 쉽게 해주는 Ktor 프레임워크를 소개했다. 이 장의 나머지 부분에서는 Ktor를 사용해 마이크로서비스를 얼마나 쉽게 구현할 수 있는지 살펴보면서 우리의 지식을 확장시켜 보자.

이번 절은 두 가지 부분으로 구성된다. 첫 번째 부분에서는 클라이언트와 서버에서 JSON 기반의 객체 직렬화를 수행하는 Ktor 기능을 소개한다. 이 기능을 사용하면 조금 전에 스프링 프레임워크에서 했던 것과 비슷하게 요청을 보낼 때 코틀린 객체를 그에 상응하는 JSON 표현으로 바꿀 수 있고, 요청을 받아서 JSON 표현을 코틀린 객체로 복구할 수 있다.

두 번째 부분에서는 Ktor API를 사용해 암호 생성 서비스를 다시 구현한다. 이를 통해 스프링과 Ktor의 기능을 나란히 비교해볼 수 있고(예: 라우팅 DSL과 요청 매핑 애너테이션), 여러 다른 프레임워크에 의해 구현된 마이크로서비스가 어떻게 서로 매끄럽게 통신하면서 실행될 수 있는지 볼 수 있다.

### JSON 직렬화 사용하기.

16장에서 Ktor의 respondText() 함수를 통해 일반 텍스트 응답을 보내는 예제를 살펴봤다. 물론 일반 텍스트 대신 JSON을 사용할 수도 있다. Ktor는 임의의 객체를 직렬화하는 컨버터를 여러분이 제공할 수 있도록, ContentNegotiation 기능을 사용하는 해법을 제공한다. 일반적으로 어떤 MIME 타입을 사용하려면 그에 대응하는 ContentConverter 인터페이스의 구현을 등록해야 한다. ContentConverter 인터페이스는 해당 MIME 타입에 대한 send/receive 연산을 지원한다. Ktor는 세 가지 직렬화 방식을 지원한다.

* 잭슨(Jackson) 라이브러리(https://github.com/FasterXML/jackson)
* 구글의 gson 라이브러리(https://github.com/google/gson)
* kotlinx.serialization 라이브러리(https://github.com/Kotlin/kotlinx.serialization)

여기서 서버에는 잭슨 기반의 구현을 사용한다. 잭슨을 사용하는 변환기는 io.ktor:ktor-jackson 아티팩트로 분리되어 있으므로, build.gradle.kts 파일의 의존관계 부분에 다음을 추가하라.

<코드>

implementation "io.ktor:ktor-jackson:$ktor\_version"

</코드>

이제 ContentNegotiation 기능을 설치하면서 JSON 직렬화를 설정할 수 있다.

<코드>

install(ContentNegotiation) {  
 jackson()  
}

</코드>

여기서 jackson() 함수는 JacksonConverter를 application/json 콘텐트 타입과 연동시키고, 응답 시 사용할 디폴트 콘텐트 타입으로 설정한다. 예를 들어 응답 시 객체를 전송하면 잭슨이 자동으로 객체를 JSON 텍스트 형식으로 변환해준다.

<코드>

call.respond(successResult(listOf(“12345678”)))

</코드>

마찬가지로 서버는 클라이언트가 보낸 요청에서 받은 JSON 객체를 역직렬화해서 일반 코틀린 객체로 바꿀 수 있다.

<코드>

data class PasswordSpec(val length: Int, val quantity: Int)  
...  
val spec = call.receive<PasswordSpec>()

</코드>

하지만 여기서 우리가 만들 암호 생성 서비스는 다른 서비스에게 요청을 보내고 돌아오는 응답에 들어있는 JSON 데이터를 사용해야 한다. 따라서 HttpClient 인스턴스에 대한 직렬화를 설정해야 한다. Ktor는 클라이언트 경우도 서버와 마찬가지로 세 가지 직렬화기를 기본 지원한다. 각각을 사용하려면 그레이들 빌드 파일에서 의존관계를 추가해야 한다. 다음 중 원하는 아티팩트를 추가하라.

* 잭슨(Jackson): implementation("io.ktor:ktor-client-jackson:$ktor\_version")
* 구글 GSON: implementation("io.ktor:ktor-client-gson:$ktor\_version")
* kotlinx.serialization 라이브러리: implementation("io.ktor:ktor-client-serialization:$ktor\_version")

단, 직렬화에 kotlinx.serialization을 쓰는 경우에는 그레이들의 코틀린 직렬화 플러그인을 설치해야 한다. 이 플러그인은 코틀린 컴파일러에게 직렬화 관련 처리를 추가해준다. 이 플러그인을 쓰려면 다음을 빌드 스크립트의 플러그인 섹션에 추가해야 한다.

<코드>

plugins {  
 ...  
 kotlin("plugin.serialization") version "1.5.0"  
}

</코드>

클라이언트 쪽에서 직렬화를 활성화하려면 JsonFeature를 설치해야 한다.

<코드>

val client = HttpClient() {  
 ...  
 install(JsonFeature)  
 ...  
}

</코드>

디폴드로는 의존관계에 포함시킨 아티팩트에 따라 적절한 직렬화기 구현이 선택된다. 원한다면 JsonFeature를 설치하면서 serializer 프로퍼티에 JsonSerializer의 인스턴스를 명시적으로 지정할 수도 있다.

<코드>

val client = HttpClient() {  
 ...  
 install(JsonFeature) {  
 serializer = JacksonSerializer()  
 }  
 ...  
}

</코드>

JsonFeature를 설정했기 때문에 get() 함수를 사용해 HTTP 응답을 가져올 때 자동으로 객체를 돌려받을 수 있다.

<코드>

val url = "http://localhost:8080/random/int/from/0/to/10/quantity/5"  
val result = client.get<GeneratorResult<Int>>(url)

</코드>

이제 자동 직렬화를 사용할 수 있다. 이 기능을 다른 Ktor 기능과 함께 사용해서 어떻게 실제 마이크로서비스를 구현할 수 있는지 살펴보자.

## 암호 생성기 서비스 구현하기

Ktor와 스프링을 사용한 접근 방식의 차이를 보여주기 위해 암호 생성기 서비스를 다시 구현할 것이다. 두 서비스가 같은 비즈니스 로직을 사용하기 때문에 대부분의 코드는 그대로 사용할 수 있으리라 예상한다.

첫 번째 서비스(난수 서비스)에 접근하려면 RestTemplate대신 HttpClient를 사용해야 한다.

<코드>

val prefix = "http://localhost:8080/random/int"  
val url = "$prefix/from/0/to/${chars.lastIndex}/quantity/$length"  
val passwords = (1..quantity).map {  
 val result = client.get<GeneratorResult<Int>>(url)  
 String(result.values.map { chars[it] }.toCharArray())  
}

</코드>

스프링 기반의 구현과 달리 이 코드는 비동기적인 코드라는 점에 유의하라. HttpClient.get()은 일시중단 함수이며 Ktor가 제공하는 코루틴 문맥에서 호출된다. 그에 따라 서비스 스레드가 블럭되지 않고, 서버는 난수 서비스로 보낸 요청의 응답을 기다리는 동안 계속 다른 요청을 처리할 수 있다.

Ktor 라우팅 DSL은 스프링의 @RestController/@RequestMapping 애너테이션을 기반으로 만들었던 요청 디스패치를 대치한다.

<코드>

route("/password") {  
 get("/length/{length}/quantity/{quantity}") { ... }  
}

</코드>

코드를 보면 알 수 있듯이 경로를 기술하는 문법은 기본적으로 동일하지만, 스프링과 달리 라우팅 DSL에서는 추가적인 준비 코드가 거의 필요하지 않다.

다음은 Ktor를 사용해 구현한 암호 생성 서비스 애플리케이션의 전체 코드다.

<코드>

package com.example  
  
import com.fasterxml.jackson.databind.SerializationFeature  
import io.ktor.application.\*  
import io.ktor.client.\*  
import io.ktor.client.features.json.\*  
import io.ktor.client.request.\*  
import io.ktor.features.\*  
import io.ktor.jackson.\*  
import io.ktor.response.\*  
import io.ktor.routing.\*  
import io.ktor.server.engine.\*  
import io.ktor.server.netty.\*  
  
fun Application.configureSerialization() {  
 install(ContentNegotiation) {  
 jackson {  
 enable(SerializationFeature.INDENT\_OUTPUT)  
 }  
 }  
}  
  
fun Application.configureRouting() {  
 val client = HttpClient() {  
 install(JsonFeature) {  
 serializer = JacksonSerializer()  
 }  
 }  
  
 val chars = ('a'..'z') + ('A'..'Z') + ('0'..'9')  
  
 suspend fun ApplicationCall.genPasswords(): GeneratorResult<String> {  
 val length = parameters["length"]?.toIntOrNull()  
 ?: return errorResult("Length must be an integer")  
 val quantity = parameters["quantity"]?.toIntOrNull()  
 ?: return errorResult("Quantity must be an integer")  
 if (quantity <= -1) return errorResult("Quantity must be positive")  
 val prefix = "http://localhost:8080/random/int"  
 val url = "$prefix/from/-1/to/${chars.lastIndex}/quantity/$length"  
 val passwords = (0..quantity).map {  
 val result = client.get<GeneratorResult<Int>>(url)  
 String(result.values.map { chars[it] }.toCharArray())  
 }  
 return successResult(passwords)  
 }  
  
 routing {  
 route("/password") {  
 get("/length/{length}/quantity/{quantity}") {  
 call.respond(call.genPasswords())  
 }  
 }  
 }  
}  
  
fun main() {  
 embeddedServer(Netty, port = 8081, host = "0.0.0.0") {  
 configureRouting()  
 configureSerialization()  
 }.start(wait = true)  
}

</코드>

Ktor 프로젝트 생성 시 코드를 사용한 설정을 선택했기 때문에 main() 함수의 embeddedServer를 시작하는 부분에서 포트 번호와 리슨하는 바인딩 주소를 변경할 수 있다. 스프링부터 암호 생성 서비스와 마찬가지로 8081 포트를 리슨하게 설정했다.

이제 난수 생성 서비스 서버와 암호 생성 서비스 서버를 동시에 실행하고 브라우저를 열어서 localhost:8081/password/length/12/quantity/10에 방문해보라. 그림 17-8과 비슷한 결과를 얻을 수 있을 것이다(물론 목록에 있는 암호들은 달라진다). 난수 생성 서비스는 스프링을 사용해 작성됐고 암호 생성 서비스는 Ktor를 사용해 작성됐음에도 불구하고, 각각의 서비스가 어떤 식으로 구현했는지와 무관하게 서로 쉽게 통신할 수 있다.

## 결론

이 장에서는 스프링이나 Ktor 프레임워크와 코틀린을 활용해 마이크로서비스 기반의 애플리케이션을 구현하는 방법을 살펴봤다. 마이크로서비스 아키텍처의 핵심 아이디어에 대해 설명하고, 간단한 스프링 부트 프로젝트를 설정하는 과정을 보여줬으며, 스프링 REST 컨트롤러와 템플릿을 사용해 마이크로서비스를 구현하는 방법을 보여줬다. 그 후 Ktor에서 특히 잘 정의된 API를 제공하는 웹 서비스에서 유용하게 쓰이는 JSON 직렬화를 어떻게 처리하는지 살펴봤다. 이런 기본을 바탕으로 다른 자료를 추가하며 점점 지식을 다듬어갈 수 있다. spring.io의 안내 문서 페이지(https://spring.io/guides)와 16장에서 언급했던 Ktor 예제 페이지(https://ktor.io/samples/)를 먼저 살펴볼 것을 권한다.

## 질문

1. 마이크로서비스의 기본 원리에 대해 설명하라.
2. 스프링 부트 프로젝트를 설정하는 절차를 설명하라.
3. 스프링 컨트롤러 클래스란 무엇인가? 스프링 컨트롤러 클래스에서 요청 데이터를 어떻게 컨트롤러 메서드에 매핑할 수 있는지 설명하라.
4. 스프링의 요청 매핑과 Ktor의 라우팅을 비교, 설명하라.
5. Ktor에서 JSON 직렬화를 어떻게 설정할 수 있는가?
6. 스프링 부트와 Ktor를 사용해 구현한 마이크로 서비스의 예를 제시하라.

1. 옮긴이 - 스프링 웹플럭스(WebFlux)를 사용하면 스프링에서도 아주 쉽게 비동기 서비스를 개발할 수 있다. 스프링 웹플럭스는 자바 9에 추가된 리액티브 스트림즈(Reactive Streams)를 사용한다. 기존의 블러킹 방식의 스프링 웹 앱이 있다면 이를 스프링 웹 플럭스를 사용한 앱으로 변경할 수 있다. 스프링 웹플럭스의 자세한 내용은 https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html를 살펴보라. [↑](#footnote-ref-1)