# 16장. Ktor를 활용한 웹 개발

웹 개발은 풍부한 UI와 복잡한 작업 흐름을 제공하는 완전한 엔터프라이즈 시스템부터 구체적인 작업을 처리하는 마이크로서비스에 이르기까지 현대 소프트웨어 엔지니어링에서 중요한 부분을 차지하고 있다. 대부분의 프레임워크는 자바 세계에서 웹 개발의 여러 가지 측면을 단순화하는 데 초점을 맞추고 있다. 이런 프레임워크는 자바/코틀린 상호운용성으로 인해 코틀린 환경에서도 쉽게 사용할 수 있다. 하지만 처음부터 코틀린을 지원하도록 설계된 도구도 알아야 한다. 이런 도구를 사용하면 코틀린 언어의 기능을 최대한 활용할 수 있어 개발자의 생산성을 극대화할 수 있다.

이 장에서는 코틀린에 특화된 프레임워크인 Ktor를 소개한다. 참고로 Ktor의 발음은 ‘케이토르’다(https://ktor.io/docs/faq.html#pronounce). Ktor는 여러 가지 클라이언트 서버 애플리케이션이 연결된 시스템의 개발을 편하게 도와주려는 목적으로 만들어진 프레임워크다. 이런 클라이언트나 서버 애플리케이션으로는 브라우저, 모바일 클라이언트, 웹 애플리케이션이나 서비스 등이 있다. 코루틴 라이브러리를 확장하기 때문에 Ktor는 비동기 통신을 강력하게 지원하며 사용하기도 쉽다. 분명히 16장에서 Ktor의 모든 기능을 소개하지는 못할 것이다. 따라서 이 책의 목적에 맞게 Ktor 기능 중 웹 개발과 연관된 작은 범위에 논의의 초점을 맞춘다. 특히 웹 개발 중에서도 서버 부분을 중심으로 클라이언트 요청을 디스패치하고, 요청 데이터를 얻고, 여러 가지 응답을 합성하는 방법에 대해 설명할 것이다. 이 책을 다 읽은 후에도 계속해서 https://ktor.io/나 다른 여러 자료를 통해 계속 Ktor를 알아가길 권장한다.

## 구조

* Ktor 프로젝트 설정하기
* 서버 기능
* 클라이언트 기능

## 목표

웹 클라이언트와 웹 서버 개발에 사용할 수 있는 Ktor의 기본 기능을 배운다.

## Ktor 소개

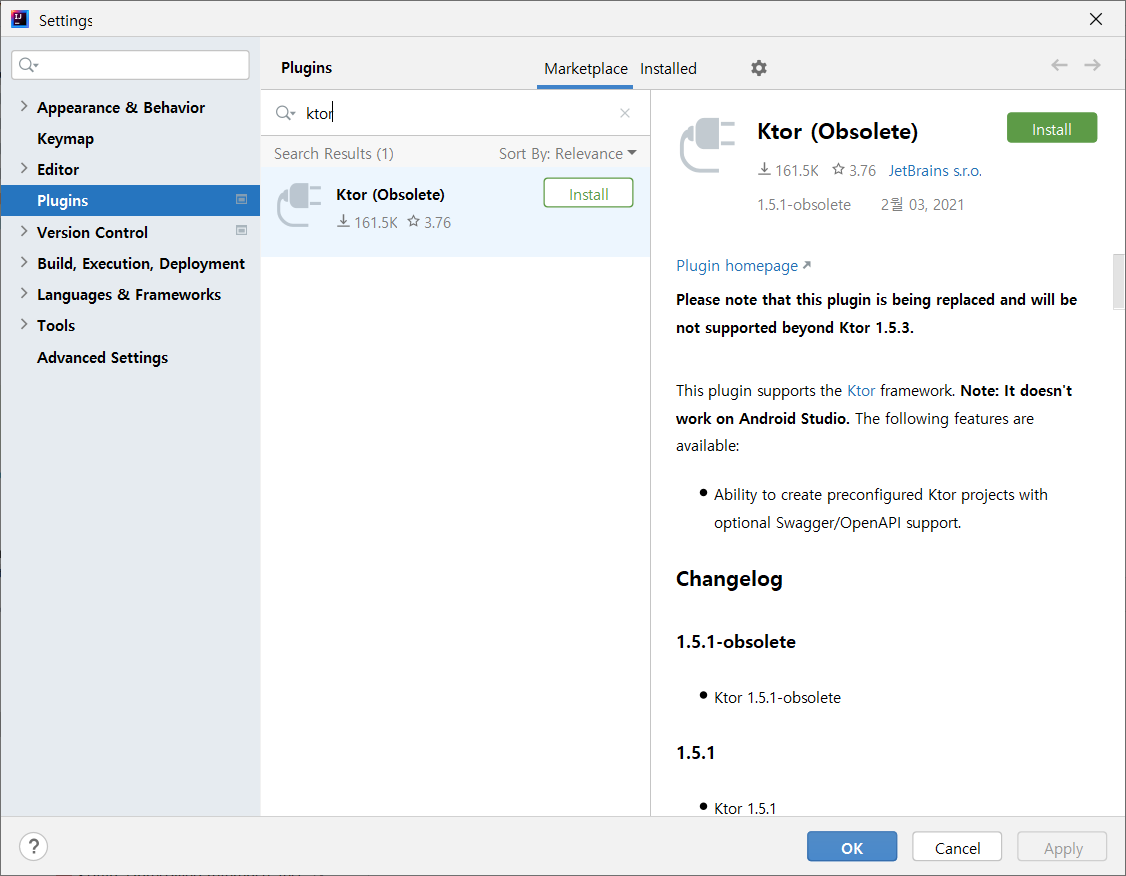
Ktor는 네트워크를 통해 서로 통신하는 클라이언트와 서버 애플리케이션을 개발하기 위한 프레임워크다. 하지만 Ktor의 주 응용 분야는 HTTP 프로토콜을 사용해 데이터를 교환하는 웹 애플리케이션 위주다. Ktor의 목표는 모든 유형의 연결이 필요한 애플리케이션을 구축하는 범용 다중 플랫폼 프레임워크가 되는 것이다. Ktor의 주 개발자이자 설계자는 젯브레인즈로 코틀린 언어 자체를 개발한 회사이기도 하다. J2EE 프레임워크와 비교할 때 Ktor의 핵심 기능은 다음과 같다.

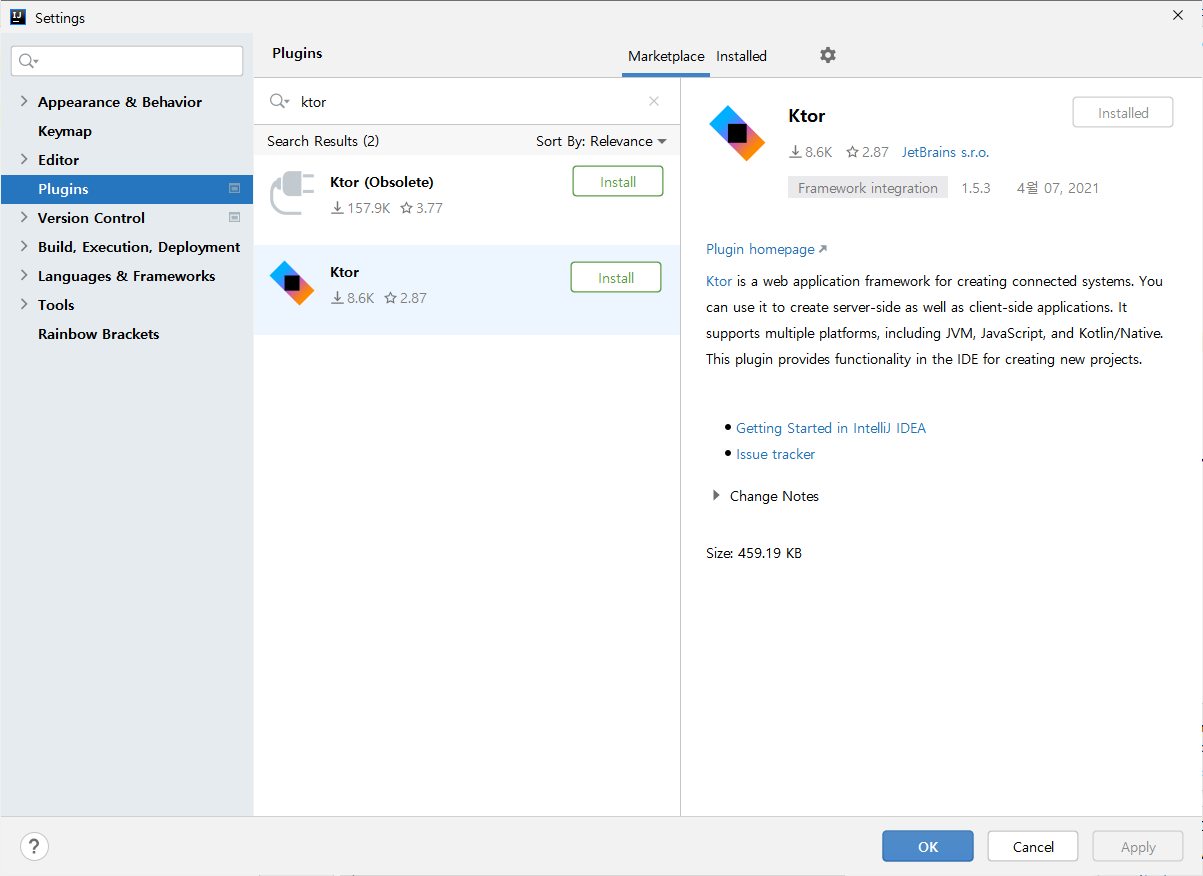
* 코틀린이 뒷받침하는 DSL을 통해 애플리케이션의 주요 측면(요청 라우팅(routing) 규칙이나 HTML 컨텐트 등)을 간결하고 선언적인 스타일로 기술하고 코드의 다른 부분과 쉽게 조합할 수 있다.
* 처음부터 코틀린 코루틴 라이브러리를 통해 효율적인 비동기 연산을 지원한다.

이 절에서는 Ktor를 간략히 살펴보고 인텔리J IDEA에서 Ktor를 설정할 때 필요한 기본 단계를 진행해본다. 프로젝트 마법사에서 Ktor를 지원해주는 특별한 인텔리J 플러그인을 사용할 것이다. 이 플러그인은 다음 단계를 따라 설치하라. 사용하는 Ktor 버전은 2021년 5월 현재 최신 버전인 1.5.4이다.

1. IDEA 주 메뉴에서 File | Settings ... 커맨드를 찾아서 왼쪽의 Plugins 섹션을 클릭하라. Plugin Manager 뷰가 표시된다.
2. 뷰의 맨 위 탭 중에 Marketplace를 선택하고, 그 바로 아래의 입력 텍스트 필드에 ktor를 입력하라. 검색 결과에서 Ktor 플러그인을 볼 수 있다(그림 16-1). 사용 금지된 버전을 쓰지 말고 새 버전을 사용하라.
3. Install 버튼을 클릭하면 IDE가 자동으로 플러그인을 다운로드해 설치한다. 설치가 완료되면 플러그인을 적용하기 위해 IDE를 재시작하라는 대화창이 뜬다. 대화창을 재시작하는 버튼을 눌러서 IDE를 재시작하라.

그림 16-1: 인텔리J IDEA Ktor 플러그인 설치하기





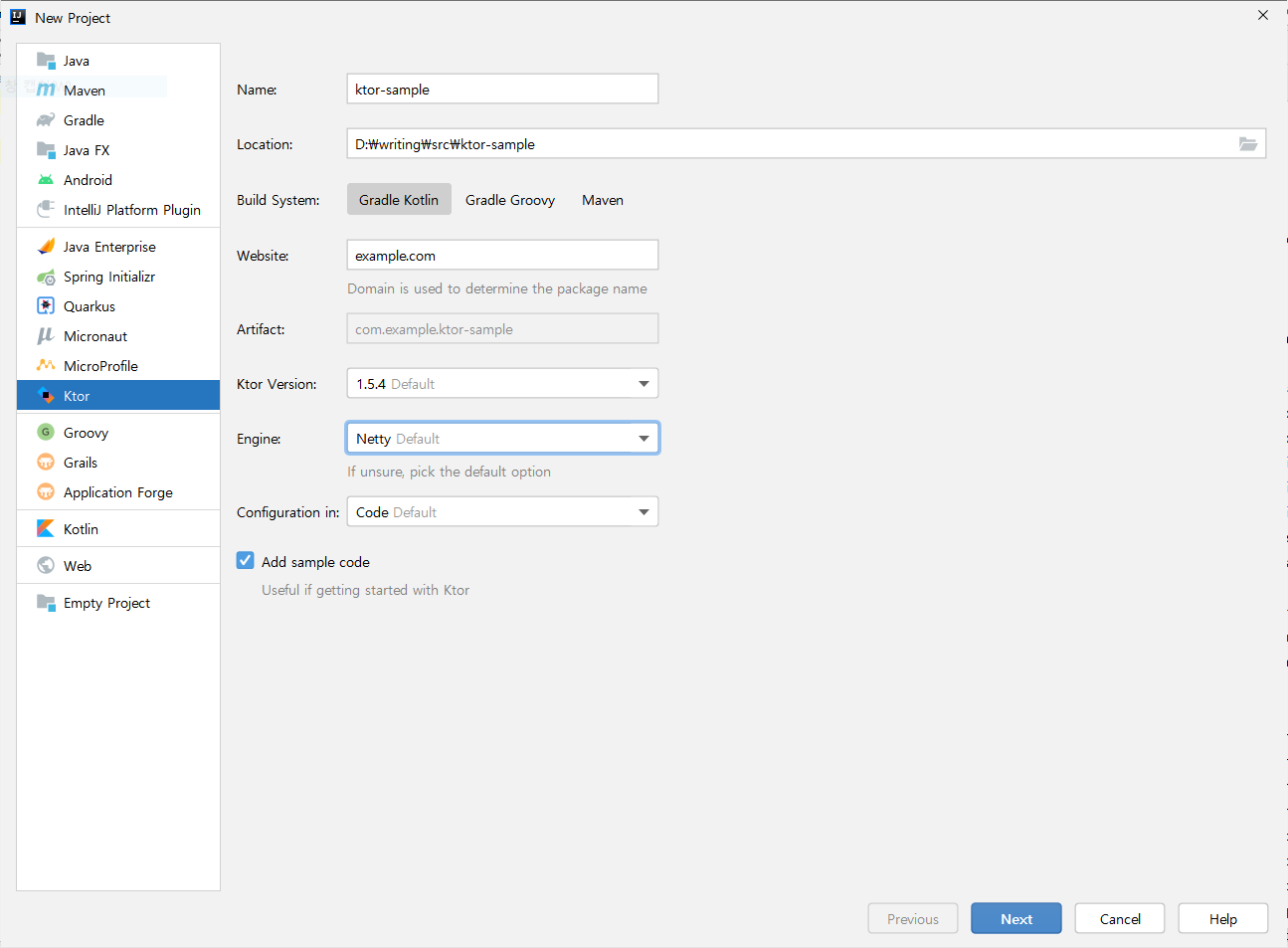
<참고> IDE 팁

대부분의 인텔리J 웹/J2EE 개발 플러그인과 달리 얼티밋(Ultimate) 에디션이 아니어도 Ktor 플러그인을 설치할 수 있기 때문에 인텔리J 커뮤니티 에디션에서도 Ktor 플러그인을 활용할 수 있다.

</참고>

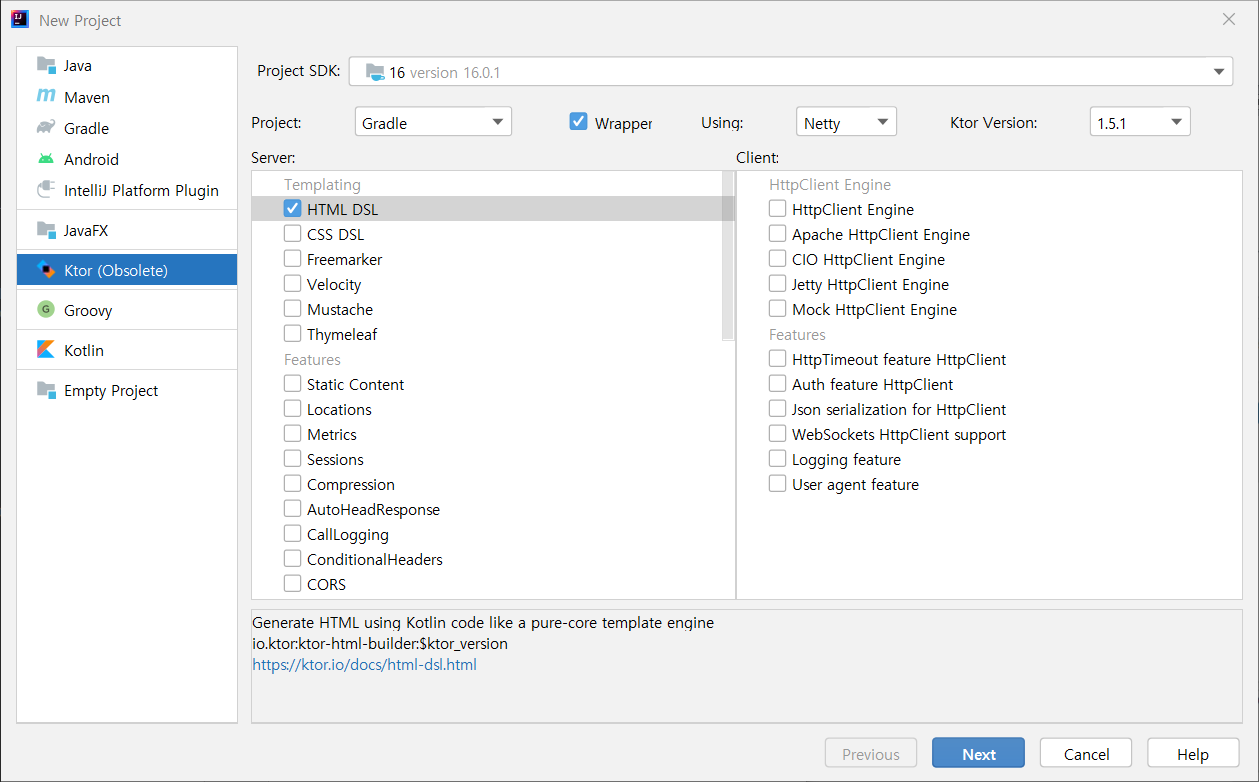
이제 프로젝트를 만들자. File | New | Project... 커맨드를 선택하고, New Project 대화창에서 Ktor 옵션을 선택하라. Ktor에서는 기본 클라이언트/서버 기능 외에도 다른 아티팩트에 의존하거나 설정 코드를 작성해야 하는 몇 가지 추가 기능을 끼워 넣어 사용할 수 있다. 마법사의 첫 번째 단계는 프로젝트 이름과 저장할 위치, 빌드 시스템(그레이들 코틀린, 그레이들 그루비(Groovy), 메이븐), 사용할 Ktor 버전과 서버 엔진(네티(Netty), 제티(Jetty), CIO, 톰캣(tomcat)) 등을 설정하는 단계다. 여기서는 디폴트 옵션을 그대로 사용하자.

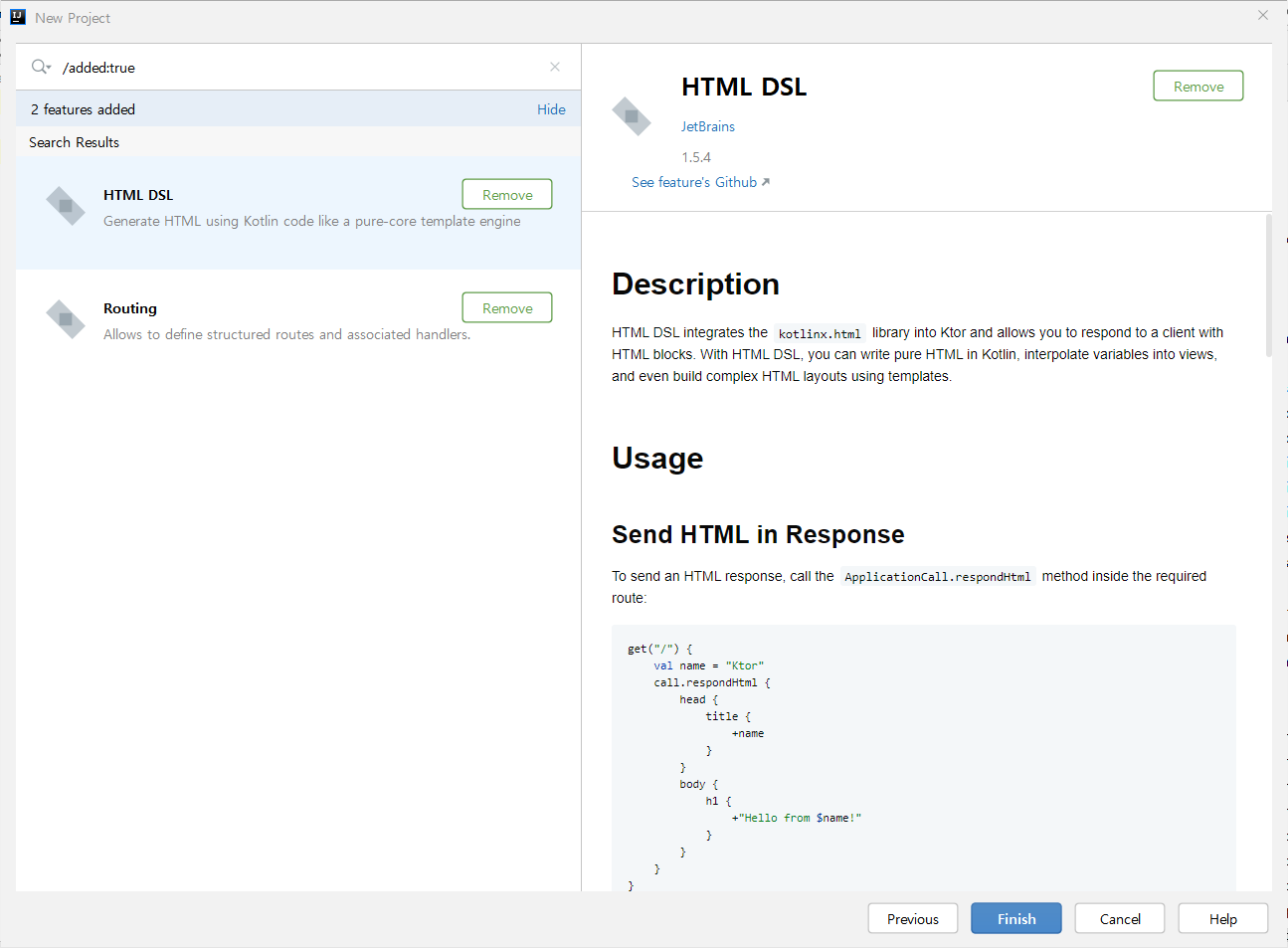
그림 16-2: Ktor 프로젝트 마법사 - 프로젝트 기본 설정



다음 단계는 그림 16-3처럼 애플리케이션에서 사용할 기능을 선택하는 단계다. 원하는 기능을 선택하면 된다.

그림 16-3: Ktor 프로젝트 마법사 - 애플리케이션에서 사용할 기능 추가

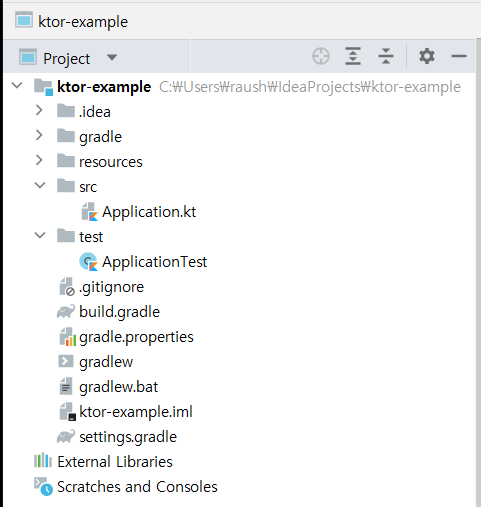


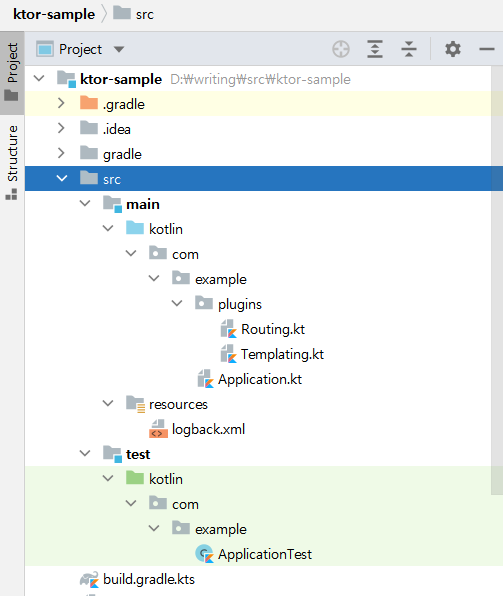


이 예제에서는 HTML DSL을 선택했다. HTML DSL을 선택하면 그림처럼 라우팅 기능도 함께 추가된다.

다음으로 **Finish**를 누르면 IDE가 프로젝트 구조를 생성하고 새 프로젝트를 연다. 프로젝트 영역에서 그림 16-4와 같은 파일을 볼 수 있다.

그림 16-4: 예제 Ktor 프로젝트의 구조





Kotr 애플리케이션은 기본적으로 main() 함수에서 embeddedServer 등의 적절한 서버나 클라이언트를 호출해서 실행하는 식으로 이뤄진다. 모든 설정은 프로그램 코드를 통해 이뤄지거나 HOCON(Human-Optimized Config Object Notation)을 통해 이뤄진다. Ktor가 권장하는 기본 설정을 사용하면 프로그램 코드를 통해 설정이 이뤄지는데, 이 부분을 다음에 표시한 Application.kt의 main() 함수 부분에서 볼 수 있다.

<코드>

package com.example  
  
import io.ktor.server.engine.\*  
import io.ktor.server.netty.\*  
import com.example.plugins.\*  
  
fun main() {  
 embeddedServer(Netty, port = 8080, host = "0.0.0.0") {  
 configureRouting()  
 configureTemplating()  
 }.start(wait = true)  
}

</코드>

네티를 사용하고 8080을 기본 포트로 하며, 0.0.0.0(모든 주소)을 바인딩한다는 사실을 쉽게 알 수 있다. 코드를 보면 프로젝트 설정 마법사에서 선택한 HTML DSL과 라우팅 기능에 맞춰 configureRouting()와 configureTemplating() 내부에서 DSL을 사용해 라우팅과 HTML 템플릿을 구성한다는 점도 추측할 수 있다.

라우팅 부분은 plugins 디렉터리의 Routing.kt에 들어있다.

<코드>

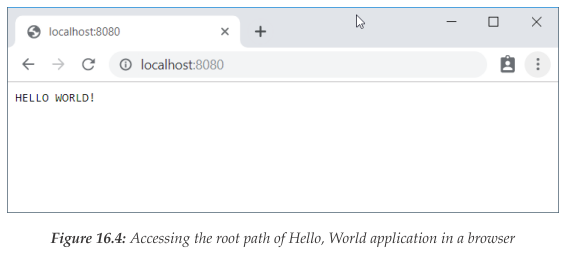
package com.example.plugins  
  
import io.ktor.routing.\*  
import io.ktor.http.\*  
import io.ktor.application.\*  
import io.ktor.response.\*  
import io.ktor.request.\*  
  
fun Application.configureRouting() {  
  
 routing {  
 get("/") {  
 call.respondText("Hello World!")  
 }  
 }  
}

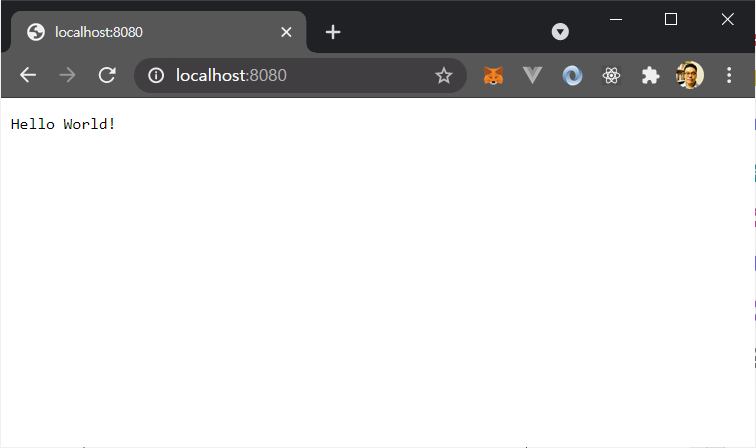
</코드>

라우팅 플러그인에는 루트(/)에 대한 요청이 오면 call.respondText("Hello World!")를 통해 Hello World!를 돌려주는 코드가 들어있다.

main() 함수를 실행하면 네티에 의해 서버가 실행되고, 8080 포트를 리슨한다. 웹 브라우저에서 localhost:8080을 주소창에 입력하면 그림 16-5 같은 모습을 볼 수 있다.

그림 16-5: 브라우저에서 Hello World! 애플리케이션의 루트(/) 경로에 접근한 경우





반면 HTML DSL을 사용한 템플릿은 plugins 디렉터리의 Templating.kt에 들어있다. 이 DSL은 여러 가지 HTML 태그에 대응하는 블록을 내포시키는 형태로 HTML 마크업을 표현한다.

<코드>

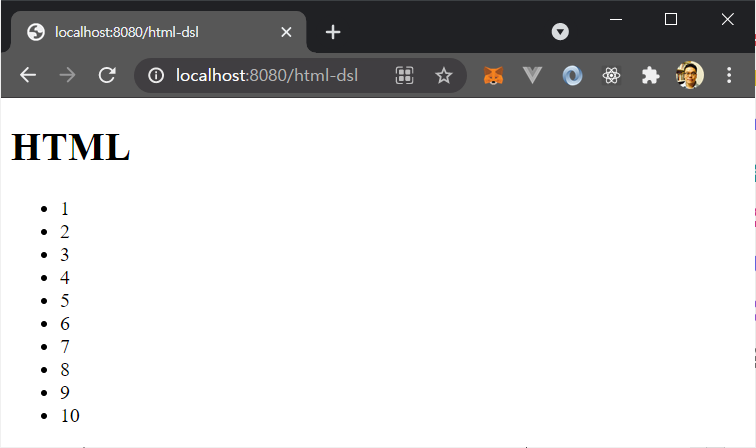
package com.example.plugins  
  
import io.ktor.html.\*  
import kotlinx.html.\*  
import io.ktor.application.\*  
import io.ktor.response.\*  
import io.ktor.request.\*  
import io.ktor.routing.\*  
  
fun Application.configureTemplating() {  
 routing {  
 get("/html-dsl") {  
 call.respondHtml {  
 body {  
 h1 { +"HTML" }  
 ul {  
 for (n in 1..10) {  
 li { +"$n" }  
 }  
 }  
 }  
 }  
 }  
 }  
}

</코드>

예제 코드는 /html-dsl 요청이 들어오면 10개짜리 번호가 붙지 않은 목록(<ul>)을 표시한다. 그림 16-6은 이 DSL이 렌더링(rendering)한 HTML 페이지다.

그림 16-6: 브라우저에서 /html-dsl 경로에 접근한 경우의 응답





앞으로는 HTML DSL과 라우팅 규칙을 자세히 살펴보고 다른 Ktor 기능도 함께 살펴본다.

IDE가 생성한 프로젝트에는 예제 서버 테스트가 들어있다. ApplicationTest 클래스를 열어 보자.

<코드>

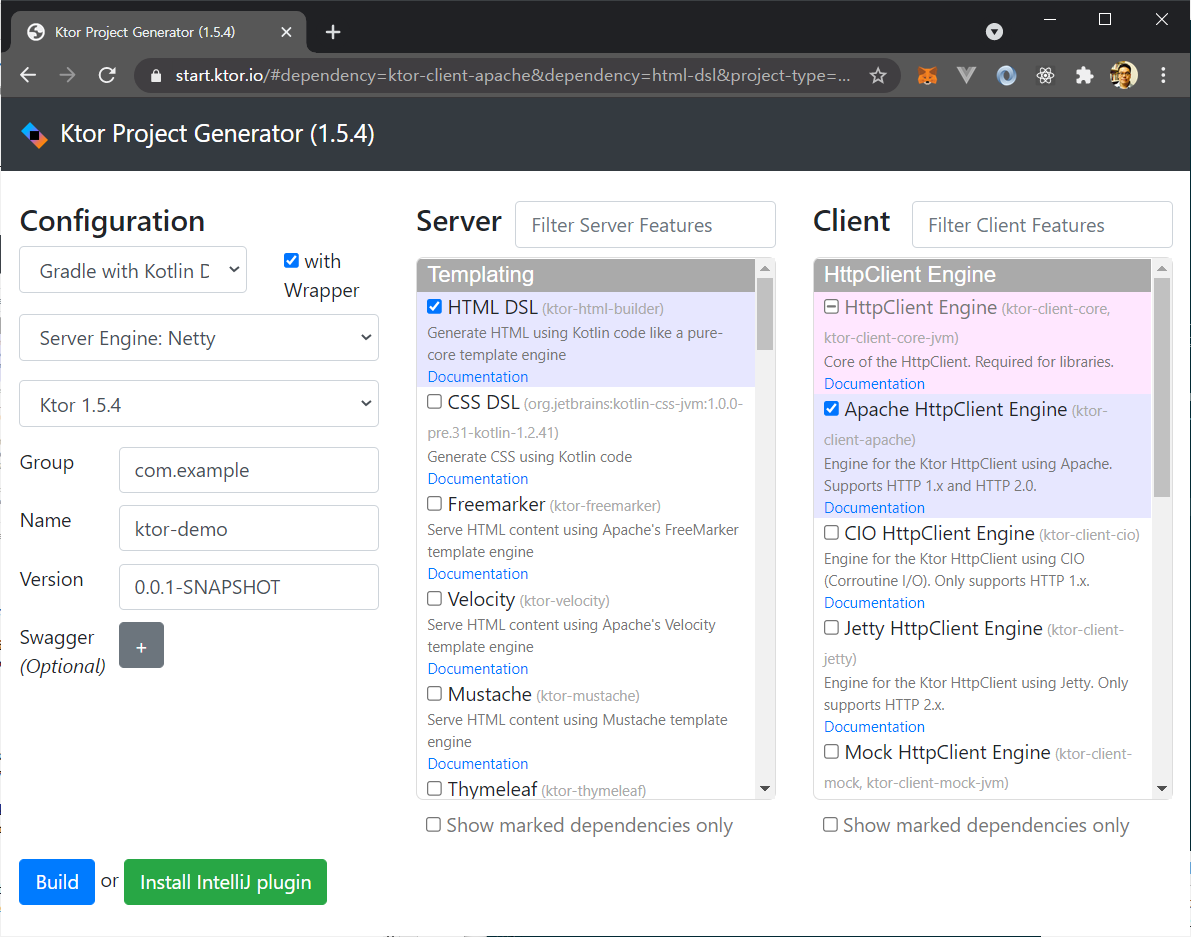
package com.example  
  
import io.ktor.routing.\*  
import io.ktor.http.\*  
import io.ktor.html.\*  
import kotlinx.html.\*  
import io.ktor.application.\*  
import io.ktor.response.\*  
import io.ktor.request.\*  
import com.example.plugins.\*  
import kotlin.test.\*  
import io.ktor.server.testing.\*  
  
class ApplicationTest {  
 @Test  
 fun testRoot() {  
 withTestApplication({ configureRouting() }) {  
 handleRequest(HttpMethod.Get, "/").apply {  
 assertEquals(HttpStatusCode.OK, response.status())  
 assertEquals("Hello World!", response.content)  
 }  
 }  
 }  
}

</코드>

이 코드는 프로젝트에 설정된 플러그인을 사용하는 테스트 애플리케이션을 사용하고(테스트를 위한 애플리케이션을 실행하는 withTestApplication()함수를 보라), 간단한 HTTP 요청을 루트 경로에 보낸 후, 응답으로 받은 결과를 검사한다. 이 클래스를 여러분이 서버의 여러 가지 동작을 검사하기 위해 필요한 테스트를 작성하는 시작점으로 삼을 수 있다.

인텔리J Ktor 플러그인 대신, https://start.ktor.io/의 온라인 프로젝트 생성기를 사용할 수도 있다. 웹 생성기 UI는 IDEA Ktor 플러그인과는 약간 다르게 기능 선택 부분이 서버와 클라이언트로 구분되어 있고 한 페이지만으로 구성된다. 그림 16-7은 한 가지 예다.

그림 16-7: start.ktor.io에 있는 Ktor 프로젝트 마법사



이것으로 프로젝트 설정에 대한 소개를 마친다. 앞으로는 Ktor 프레임워크의 클라이언트와 서버 쪽 기능에 대해 살펴본다.

## 서버 관련 기능

이 절에서는 웹 서버 애플리케이션 개발과 관련 있는 몇 가지 주제를 살펴본다. 대부분의 Ktor 기능은 ApplicationFeature라는 클래스를 상속한 끼워 넣을 수 있는 특성으로 되어 있다. 예를 들어 HTTP 송수신 시 데이터를 압축하는 Compression을 활성화하면 Application.configHTTP()라는 함수가 생기면서 그 안에서 압축 관련 방식 설정을 진행한다.

<코드>

fun main() {  
 embeddedServer(Netty, port = 8080, host = "0.0.0.0") {  
 configureRouting()  
 configureTemplating()  
 configureHTTP()  
 }.start(wait = true)  
}

</코드>

실제 기능 설정은 Application.configureHTTP() 확장 함수 안에서 install(Compression) 같이 해당 기능을 설치한다. 이 안에서 압축 방식과 우선순위를 설정한다.

<코드>

fun Application.configureHTTP() {  
 install(Compression) {  
 gzip {  
 priority = 1.0  
 }  
 deflate {  
 priority = 10.0  
 minimumSize(1024) // 적용 조건  
 }  
 }  
}

</코드>

이 장의 예제는 내장 HTTP 서버를 사용하기 때문에 독립 실행 프로그램으로 서버를 실행할 수 있다. 대부분의 경우 Ktor 애플리케이션을 아파치 톰캣(Apache Tomcat), 구글 앱 엔진(App Engine) 등의 웹/애플리케이션 컨테이너에 배포하게 된다. 애플리케이션을 이런 컨테이너에 설치하려면 애플리케이션 클래스와 애플리케이션이 의존하는 모든 의존관계와 컨테이너에 따른 설정 파일을 모아서 .jar나 .war 등의 아카이브(archive) 파일을 만들어야 한다. 컨테이너에 따른 자세한 배포 방법은 이 책의 범위를 벗어난다. 자세한 방법은 https://ktor.io/docs/deploy.html를 참조하라.

## 라우팅

라우팅 기능을 사용하면 HTTP 요청을 계층적인 패턴 매처로 이뤄진 시스템을 통해 구조적으로 처리할 수 있다. 라우팅 설정은 Application.configureRouting()이라는 확장함수 안에서 특별 DSL(영역 특화 언어)를 통해 표현된다.

<코드>

fun Application.configureRouting() {  
  
 routing {  
 get("/") {  
 call.respondText("Hello World!")  
 }  
 }  
}

</코드>

이 코드를 보면 routing()이라는 Application의 확장함수 내부에서 라우팅 설정이 이뤄지는데, 이 routing() 함수는 내부에서 라우팅 기능이 설치되지 않은 경우 install을 알아서 호출해준다.

<코드>

@ContextDsl  
public fun Application.routing(configuration: Routing.() -> Unit): Routing =  
 featureOrNull(Routing)?.apply(configuration) ?: install(Routing, configuration)

</코드>

따라서 원한다면 Compress를 설정할 때와 마찬가지로 다음과 같이 라우팅 설정을 진행할 수도 있다.

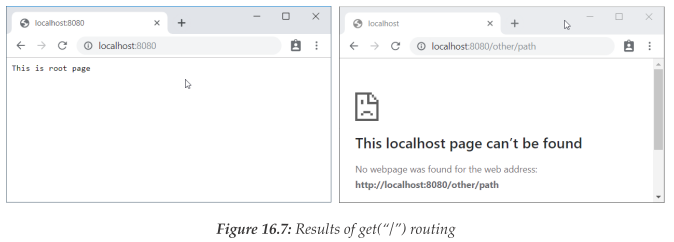
<코드>

fun Application.configureRouting() {  
 install(Routing) {  
 get("/") {  
 call.respondText("Hello World!")  
 }  
 }  
}

</코드>

가장 단순한 라우팅 시나리오는 주어진 URL 접두사(prefix)와 일치하는 HTTP GET 요청에 대한 핸들러를 서버가 실행하게 하는 get() 함수다. 예를 들어 바로 앞 예제는 사이트 루트("/")에 대한 요청이 들어오면 응답을 하지만, 다른 경로에 대한 요청은 HTTP 상태 코드 404(NOT FOUND)로 응답한다(그림 16-8).

그림 16-8: get("/") 라우팅 결과



Ktor는 모든 HTTP 동사에 대해 비슷한 함수를 제공한다. 즉, post(), put(), delete(), patch(), head(), options()를 제공한다.

라우팅 함수에서 경로의 특정 부분에 파라미터를 포함시킬 수도 있다. 나중에 애플리케이션 함수를 호출하면서 경로 파라미터의 값을 전달할 수 있다. 파라미터를 경로에 넣을 때는 중괄호({})를 사용한다. 예를 들어 다음 라우팅 경로는 /hello/로 시작하면서 hello바로 다음에 세그먼트(/로 구분되는 경로의 일부분)가 하나만 오는 경로와 매치된다.

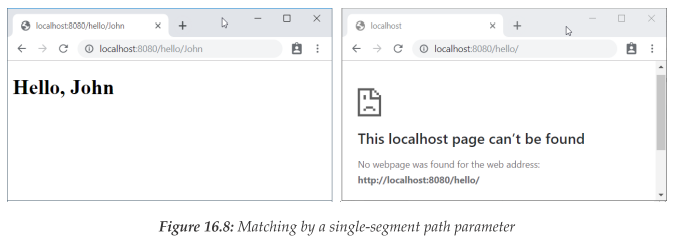
<코드>

routing {  
 get("hello/{userName}") {  
 call.respondHtml {  
 body {  
 h1 {+"Hello, ${call.parameters["userName"]}"}  
 }  
 }  
 }  
 }

</코드>

파라미터는 단 한 세그먼트에만 매치된다. 따라서 /hello나 /hello/Hyunsok/Oh는 매치가 되지 않는다(그림 16-9를 보라).

그림 16-9: 경로 중 한 세그먼트와 매치되는 파라미터로 매치시키는 경우



파라미터 값을 세그먼트와 매치시켜야 하지만 실제 사용하지 않는 경우, 파라미터 이름 대신 \*(와일드카드)를 사용할 수 있다.

<코드>

routing {  
 get("/hello/\*") {  
 call.respondHtml {  
 body {  
 h1 { +"Hello, World" }  
 }  
 }  
 }  
}

</코드>

방금 본 라우팅은 그 전에 본 라우팅과 같은 경로들과 매치되지만 경로 세그먼트의 값을 특정 파라미터에 저장하지는 않는다.

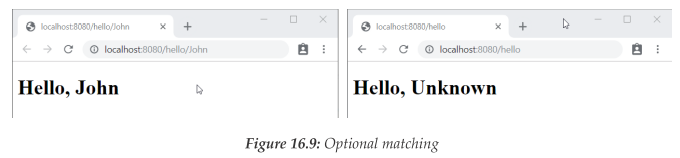
빈 경로 세그먼트와 매치될 수도 있는 선택적인 파라미터가 필요하다면 어떻게 할까? 파라미터 이름 바로 뒤에 ?를 추가하면 된다. 다음 라우팅은 /hello/John과 /hello에 모두 매치된다(그림 16-10).

<코드>

routing {  
 get("/hello/{userName?}") {  
 val userName = call.parameters["userName"] ?: "Unknown"  
 call.respondHtml {  
 body {  
 h1 { +"Hello, $userName" }  
 }  
 }  
 }  
}

</코드>

그림 16-10: 선택적인 매칭



파라미터 이름 뒤에 마침표를 세 개(...) 덧붙이면 경로의 뒤쪽에 있는 모든 세그먼트와 매치된다. 이 경우 Parameters 클래스의 getAll() 메서드를 사용해 모든 세그먼트를 List로 얻을 수 있다. 예를 들어 /calc/+/123/456 처럼 연산과 정수를 받아서 간단한 산술 연산을 수행하는 서비스를 정의해보자.

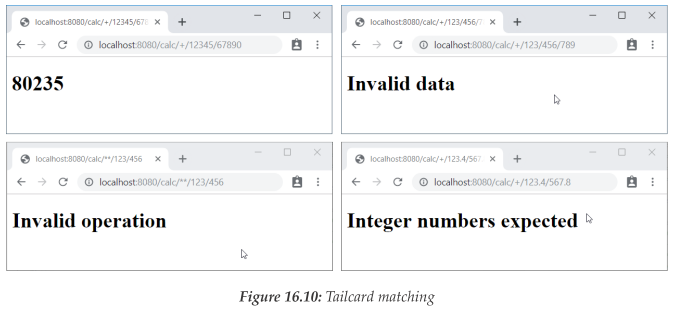
<코드>

routing {  
 get("/calc/{data...}") {  
 val data = call.parameters.getAll("data") ?: emptyList()  
 call.respondHtml {  
 body {  
 h1 {  
 if (data.size != 3) {  
 +"Invalid data"  
 return@h1  
 }  
 val (op, argStr1, argStr2) = data  
 val arg1 = argStr1.toBigIntegerOrNull()  
 val arg2 = argStr2.toBigIntegerOrNull()  
 if (arg1 == null || arg2 == null) {  
 +"Integer numbers expected"  
 return@h1  
 }  
 val result = when (op) {  
 "+" -> arg1 + arg2  
 "-" -> arg1 - arg2  
 "\*" -> arg1 \* arg2  
 "/" -> arg1 / arg2  
 else -> null  
 }  
 +(result?.toString() ?: "Invalid operation")  
 }  
 }  
 }  
 }  
}

</코드>

몇 가지 응답을 보자(그림 16-11). 왼쪽 위 그림은 제대로 정수와 연산자가 들어간 /calc/+/12345/67890 경로에 대한 성공적인 응답이다. 오른쪽 위 그림은 정수를 3개 넘긴 경우의 오류이고, 왼쪽 아래 그림은 \*\*라는 잘못된 연산자에 대한 오류다. 마지막으로 오른쪽 아래 그림은 정수가 아닌 값을 피연산자로 넘긴 경우다.

그림 16-11: ...를 사용한 매칭



...를 사용한 매핑이 빈 경로와도 매치된다는 사실을 알아두자. 즉, 이 예제의 get("/calc/{data...}") 핸들러는 /calc라는 경로에 대해서도 호출이 된다. 와일드카드 \*와 비슷하게, {...}를 사용하면 모든 남은 경로와 매치되지만 경로의 실제 내용은 무시할 수 있다. 예를 들어 get("/calc/{...}")라는 핸들러는 /calc로 시작하는 모든 경로와 매치된다(/calc 자체와도 매치됨).

get() 등의 함수를 사용해 전체 URL 경로를 매치시키는 방식 외에, Ktor는 트리 구조로 URL이나 요청 데이터 중 상당 부분을 차례대로 매치시키는 기능을 제공한다. 다음 예제를 보자.

<코드>

routing {  
 method(HttpMethod.Get) {  
 route("user/{name}") {  
 route("sayHello") {  
 handle {  
 call.respondText("Hello, ${call.parameters["name"]}")  
 }  
 }  
 route("sayBye") {  
 handle {  
 call.respondText("Bye, ${call.parameters["name"]}")  
 }  
 }  
 }  
 }  
}

</코드>

요청이 도작하면 서버는 요청 경로와 라우팅 규칙의 트리 루트가 일치하는지 검색한다. 여기서는 트리 루트 노드가 method(HttpMethod.Get)이므로, GET 동사와 매치된다. 클라이언트 요청이 이 조건에 부합된다면(GET 요청이라면) 서버는 트리 아래로 내려가면서 route("user/{name}") 규칙, 즉 주어진 접두어와 경로의 앞 부분이 일치하는지를 검사한다. 경로가 일치하면 서버는 한 단계 더 라우팅 규칙 트리를 내려가서 route("sayHello")나 route("sayBye") 규칙과 URL 경로의 나머지 부분이 일치하는지 비교한다. 규칙 트리에서 가장 낮은 수준에는 handle() 블록이 있다. 이 블록은 자신이 속한 가지와 일치하는 경로에 대해 응답을 생성한다.

HTTP 동사는 method() 함수를 사용하지 않고도 route() 호출에 지정할 수 있다. 사실 더 앞의 예제에서 본 get() 함수는 최상위 핸들러와 route() 블록을 사용한 코드를 줄여 쓴 것이다. 예를 들어 다음 코드는

<코드>

routing {  
 get("/hello/{userName}") {  
 call.respondText("Hello, ${call.parameters["userName"]}")  
 }  
}

</코드>

다음 코드와 같다.

<코드>

routing {  
 route("/hello/{userName}", HttpMethod.Get) {  
 handle {  
 call.respondText("Hello, ${call.parameters["userName"]}")  
 }  
 }  
}

</코드>

사용할 수 있는 매처가 route()와 method() 외에도 많다. 다음과 같은 빌더를 더 사용할 수 있다.

* header(name, value): 헤더 이름과 값을 매치시킨다
* param(name, value): 요청 쿼리 파라미터(query parameter[[1]](#footnote-1))의 이름과 값을 매치시킨다
* param(name): 요청 쿼리 파라미터의 이름만 매치시킨다. 매치되는 이름이 있으면 그 값을 name으로 (파라미터 맵에) 저장해준다
* optionalParam(name): 요청 쿼리 파라미터의 이름만 매치시킨다. (선택적이므로 지정한 이름에 해당하는 요청 파라미터가 없어도 매치된다.)

다음 예제는 action 파라미터의 값에 따라 적절한 응답을 선택한다.

<코드>

routing {  
 route("/user/{name}", HttpMethod.Get) {  
 param("action", "sayHello") {  
 handle {  
 call.respondHtml {  
 body { h2 { +"Hello, ${call.parameters["name"]}" } }  
 }  
 }  
 }  
 param("action", "sayBye") {  
 handle {  
 call.respondHtml {  
 body { h2 { +"Bye, ${call.parameters["name"]}" } }  
 }  
 }  
 }  
 }  
}

</코드>

예를 들어 /user/John?action=sayHello는 Hello, John라는 응답을, /user/John?action=sayBye는 Bye, John이라는 응답을 돌려준다. 다른 action 값은 매치되지 않기 때문에 404 응답 코드를 반환한다.

그림 16-12: 요청 쿼리 파라미터를 사용한 매칭



Ktor는 여러분이 직접 자신만의 매처를 작성할 수 있도록 지원한다. 따라서 라우팅 DSL을 확장할 수 있다. 이를 위해서는 RouteSelector를 상속한 셀렉터 클래스를 정의하고 적절한 빌더 함수를 Route 클래스에 대한 확장함수로 정의하면 된다. io.ktor.routing 패키지에 있는 RoutingBuilder.kt 파일을 살펴보면 다양한 구현 예제를 보고 좀 더 쉽게 자신만의 라우팅 규칙 빌더를 정의할 수 있을 것이다.

## HTTP 요청 처리

HTTP 서버가 주로 하는 일은 클라이언트 요청에 대해 적절한 응답을 돌려주는 것이다. Ktor는 이 작업을 단순화하기 위해 쉽게 사용할 수 있는 API를 제공해서 여러분이 URL 경로 컴포넌트, 파라미터, 헤더(공통 헤더와 커스텀 헤더), 여러 가지 유형의 요청 본문(body)에 쉽게 접근할 수 있게 해주는 동시에 응답 데이터를 편하게 생성할 수 있게 해준다. Ktor 프레임워크가 모든 저수준 세부 사항을 감춰주기 때문에, 여러분은 실제 요청을 처리하는 로직에만 집중할 수 있다.

이번 절부터는 Ktor의 기본 요청/응답 처리 능력에 대해 살펴볼 것이다. 앞에서는 라우팅 핸들러 안에서 responseText()나 respondHtml()를 써서 간단하게 응답을 생성하는 방법을 살펴봤다. 예를 들어 다음 코드를 보자.

<코드>

routing {  
 get(“/”) { call.respondText(“This is root page”) }  
}

</코드>

핸들러 안에서 사용 가능한 call 프로퍼티는 ApplicationCall 타입의 인스턴스로, 들어오는 요청과 생성된 응답을 묶어준다. 일반적인 경우로 어떤 텍스트를 기반으로 응답을 생성하는 방식을 들 수 있는데, 이런 경우를 responseText() 함수가 처리한다. 앞 예제에서는 간단한 일반 텍스트 본문을 응답으로 보낸다. 하지만 원한다면 다음 코드처럼 contentType을 사용해 MIME 타입을 지정할 수도 있다.

<코드>

call.respondText(“<h2>HTML Text</h2>”, ContentType.Text.Html

</코드>

이 코드는 HTML 기반의 응답을 돌려준다. 다른 방법으로 응답 텍스트를 일시중단된 람다를 통해 제공할 수도 있다.

<코드>

call.respondText(ContentType.Text.CSS) { "p { color: red; }" }

</코드>

또는 PrintWriter로 본문 텍스트를 합성할 수도 있다.

<코드>

call.respondTextWriter(ContentType.Text.Html) {  
 write("<head><title>Sample page</title><title>")  
 write("<body><h2>Sample page</h2></body>")  
}

</코드>

임의의 이진 데이터를 보내려면 String대신 ByteArray를 인자로 받는 respondBytes() 함수를 사용하면 된다.

<코드>

get(“/”) {  
 val data = “<h2>HTML Text</h2>”.toByteArray() // 디폴트인 UTF-8로 인코딩됨  
 call.respondBytes(data, ContentType.Text.Html)  
}

</코드>

respondFile()을 사용하면 서버에서 클라이언트로 파일을 전송할 수 있다.

<코드>

get("/download/{fileName}") {  
 val rootDir = File("contentDir")  
 val fileName = call.parameters["fileName"]!!  
 call.respondFile(rootDir, fileName)  
}

</코드>

본문 내용을 지정하는 기능 외에도, 다음과 같은 call.response의 프로퍼티를 사용해 헤더 데이터를 지정할 수도 있다.

* status(code: HttpStatusCode): HTTP 응답 상태를 지정한다
* header(name: String, value: String): 주어진 헤더를 HTTP 응답에 추가한다

Ktor는 ApplicationCall.respondRedirect(url: String, permanant: Boolean=false) 함수의 두 번째 인자 값에 따라 상태 301(Moved Permanently)이나 302(Found)를 사용한 HTTP 리디렉션(redirection)을 지원한다.

<코드>

routing {  
 get(“/”) {  
 call.respondRedirect(“index”) // 302 Found  
 }  
 get(“index”) {  
 call.respondText(“Main page”)  
 }  
}

</코드>

요청 파라미터에 접근하고 싶으면 파라미터의 이름과 값에 대한 Map 역할을 하는 request.queryParameters 객체를 사용하면 된다. 예를 들어 /sum?left=2&right=3라는 형태의 URL로 주어진 두 정수의 합계를 반환하고 싶다고 하자. 이를 위해 left와 right 요청 쿼리 파라미터의 값을 queryParameters 객체에 대한 인덱싱 연산을 통해 얻을 수 있다.

<코드>

routing {  
 // 예: /sum?left=2&right=3 responds with 5  
 get("/sum") {  
 val left = call.request.queryParameters["left"]?.toIntOrNull()  
 val right = call.request.queryParameters["right"]?.toIntOrNull()  
 if (left != null && right != null) {  
 call.respondText("${left + right}")  
 } else {  
 call.respondText("Invalid arguments")  
 }  
 }  
}

</코드>

같은 이름의 파라미터가 한 번 이상 쓰인 경우, get() 함수는 첫 번째로 들어온 파라미터의 값만 반환한다. 반면에 getAll() 함수는 List<String> 타입으로 해당 파라미터의 모든 값으로 이뤄진 리스트를 반환한다.

<코드>

routing {  
 // 예: /sum?arg=1&arg=2&arg=3 responds with 6  
 get("/sum") {  
 val args = call.request.queryParameters.getAll("arg")  
 if (args == null) {  
 call.respondText("No data")  
 return@get  
 }  
 var sum = 0  
 for (arg in args) {  
 val num = arg.toIntOrNull()  
 if (num == null) {  
 call.respondText("Invalid arguments")  
 return@get  
 }  
 sum += num  
 }  
 call.respondText("$sum")  
 }  
}

</코드>

이와 마찬가지로 request.headers.get()와 reqiest.headers.getAll()를 사용하면 요청 헤더 데이터의 값을 얻을 수 있다.

## HTML DSL

HTML DSL 라이브러리와 Ktor HTML 빌더를 함께 사용해 HTML 콘텐트 응답을 생성할 수 있다. 이 방식은 JSP처럼 UI 마크업에 실행 코드를 내장(embedding)시키는 방식을 대신할 수 있는 기술이다. HTML DSL을 사용하면 간결한 문법과 함께 타입 안전성이나 강력한 IDE 코드 인사이트 등 코틀린 코드의 모든 장점을 함께 얻을 수 있다. 이 절에서는 HTML DSL 라이브러리의 세부 사항을 설명하지는 않고, HTML 폼(form)을 만들 때 HTML DSL을 사용하는 방법을 보여주는 예제에 집중할 것이다. 더 많은 정보가 필요한 독자는 HTML DSL 웹사이트인 https://github.com/Kotlin/kotlinx.html를 방문해보라.

난수를 생성하는 간단한 웹 애플리케이션을 만들자. 서버는 그림 16-13처럼 생긴, 사용자가 원하는 난수 범위와 난수의 개수를 입력하는 폼을 제공한다. 서버는 간단한 검증(validation)을 통해 입력한 데이터가 다음을 만족하는지 검사한다.

* 모든 값이 제대로 된 정수여야 한다
* From 값은 To 값보다 작아야 한다
* How many 필드의 값은 양수여야 한다

폼을 제출할 때 이런 요구사항을 모두 만족시키지 못한다면 서버는 조건을 만족시키지 않는 텍스트 필드의 옆에 오류 메시지를 추가해서 클라이언트에게 폼을 돌려준다.

서버 모듈 전체 코드는 상당히 길기 때문에 여기 모든 코드를 표시하는 대신 중요한 부분에 초점을 맞춰서 설명할 것이다(전체 코드는 이 책의 깃허브 리포지터리에서 볼 수 있다).

간단한 리스트와 문단(paragraph)에 대해 HTML DSL을 사용하는 예제를 이미 살펴봤다. 따라서 지금은 HTML 폼을 만드는 방법에 초점을 맞춘다. 다음 코드는 애플리케이션 모듈의 소스코드에서 가져온 것이다.

<코드>

body {  
 h1 { +"난수 생성" }  
 form(action = "/", method = FormMethod.get) {  
 p { +"시작(생성되는 난수에 포함): " }  
 p {  
 numberInput(name = FROM\_KEY) {  
 value = from?.toString() ?: "1"  
 }  
 appendError(FROM\_KEY)  
 }  
 p { +"끝(생성되는 난수에 미포함): " }  
 p {  
 numberInput(name = TO\_KEY) {  
 value = to?.toString() ?: "100"  
 }  
 appendError(TO\_KEY)  
 }  
 p { +"생성할 난수 개수: " }  
 p {  
 numberInput(name = COUNT\_KEY) {  
 value = count?.toString() ?: "10"  
 }  
 appendError(COUNT\_KEY)  
 }  
 p { submitInput { value = "Generate" } }  
 }  
 ...   
 }

</코드>

form() 블록은 body() 안에 있다. form() 호출은 HTML 폼을 정의하면서 텍스트 필드나 버튼 등의 입력 컴포넌트를 추가할 수 있는 영역을 도입한다. action 인자는 폼을 제출할 때 어디로 데이터를 보내야 할지 결정해준다.

HTML DSL은 기본적인 입력 컴포넌트를 모두 지원한다. 그 중 일부를 나열하면 다음과 같다.

* input(): 범용 텍스트 필드
* passwordInput(): 암호를 입력하기 위한 텍스트 필드
* numberInput(): 수치 입력을 위한 텍스트 필드. 이전 값(prior), 다음 값(next) 버튼이 달려있다
* dateInput()/timeInput()/dateTimeInput(): 날짜, 시간을 입력하기 위한 특별한 텍스트 필드
* fileInput(): 브라우저가 실행 중인 로컬 컴퓨터의 파일을 업로드할 때 쓸 수 있는 텍스트 필드. 로컬 파일 목록을 보고 원하는 파일을 선택하기 위한 버튼이 붙어있다

submitInput() 호출은 폼을 제출하는 버튼을 만든다. 이 버튼을 클릭하면 폼 데이터를 HTTP 요청에 집어넣고 action으로 지정한 URL로 전송한다.

이 코드를 브라우저에서 렌더링한 결과를 보면 다음과 같은 HTML 코드를 볼 수 있다.

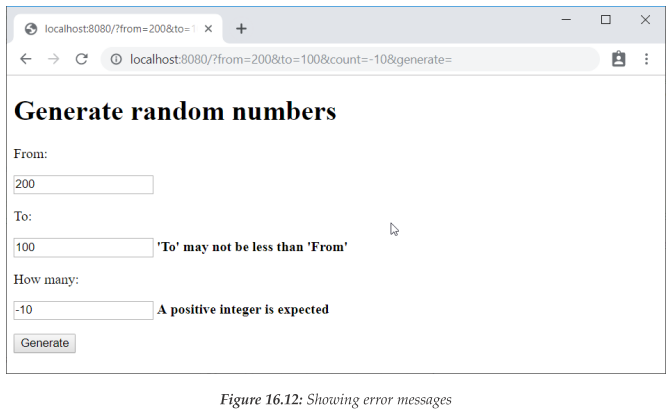
<코드>

<!DOCTYPE html>  
<html>  
 <body>  
 <h1>난수 생성</h1>  
 <form action="/" method="get">  
 <p>시작(생성되는 난수에 포함): </p>  
 <p><input type="number" name="from" value="1111"><strong>시작 값은 끝 값보다 더 작아야 합니다.</strong></p>  
 <p>끝(생성되는 난수에 미포함): </p>  
 <p><input type="number" name="to" value="100"></p>  
 <p>생성할 난수 개수: </p>  
 <p><input type="number" name="count" value="-10"><strong>생성할 난수 개수는 0보다 커야 합니다.</strong></p>  
 <p><input type="submit" value="Generate"></p>  
 </form>  
 </body>  
</html>

</코드>

HTML 태그와 DSL 블록 사이의 직접적인 연관성을 쉽게 알아볼 수 있다. 젯팩 콤포즈의 UI DSL과 마찬가지로 이 UI 코드를 쉽게 리팩터링하고 재사용할 수 있다. 만약 동적인 콘텐트를 제공하기 위해 JSP처럼 HTML 파일 형태를 사용하거나, 벨로시티(Velocity) 등의 템플릿 엔진을 활용한다면 훨씬 더 리팩터링과 재사용이 어려울 것이다. 그림 16-13은 잘못된 데이터를 폼에 전송한 결과다.

그림 16-13: 오류 메시지 표시



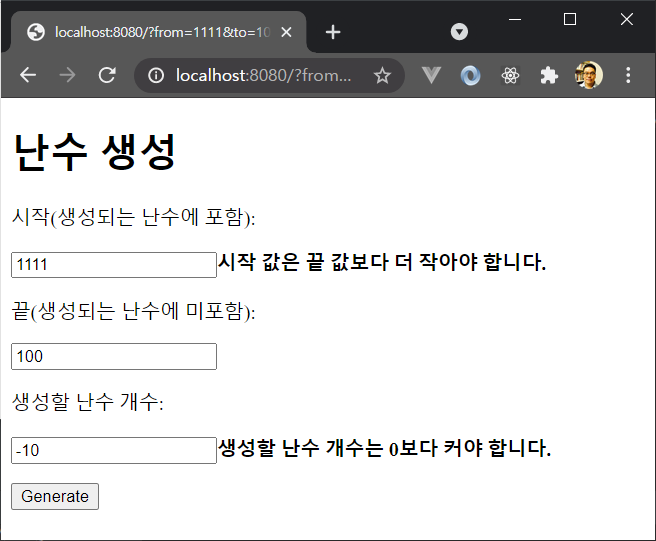
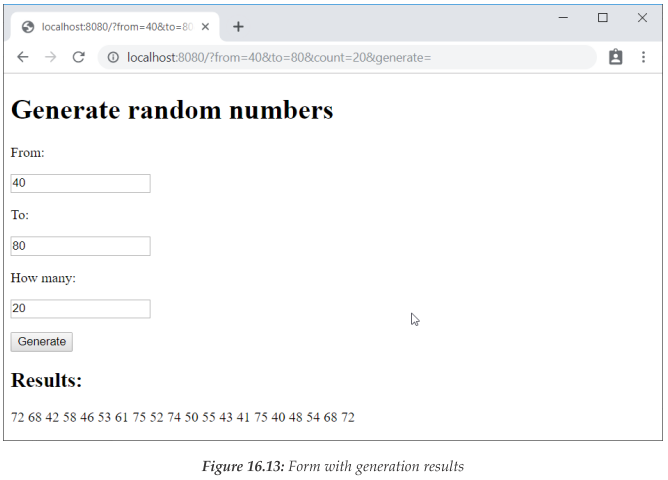
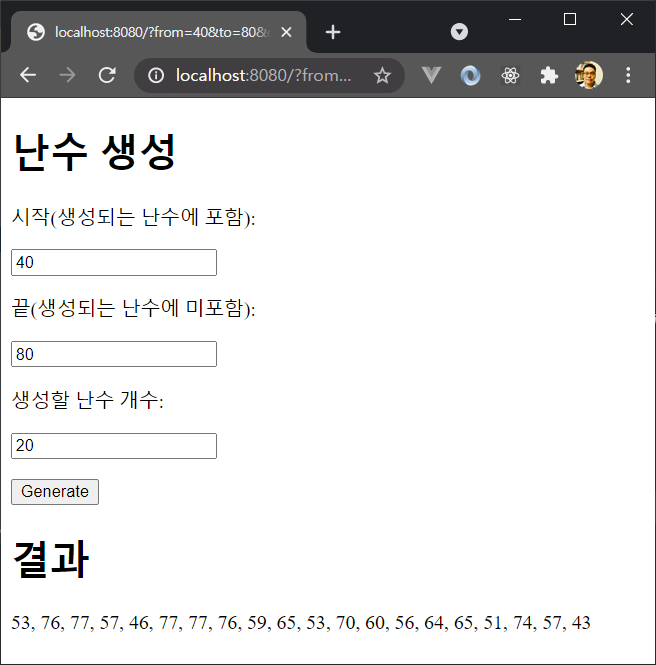


그림 16-14는 페이지 맨 아래 난수를 표시한 모습이다.

그림 16-14: 폼에 의해 생성된 난수





HTML DSL을 Ktor HTML 빌더 라이브러리와 독립적으로 사용할 수도 있다. 이런 경우 DSL 아티팩트 자체에 대한 의존관계를 지정해야만 한다. 예를 들어 그레이들에서는 다음을 설정 파일의 dependencies 블록에 추가해야 한다. 2021년 5월 현재 최신 버전은 0.7.3이다.

<코드>

// 서버 측(자바 버전)  
implementation("org.jetbrains.kotlinx:kotlinx-html-jvm:${kotlinx\_html\_version}")  
  
// 클라이언트 측(js 버전)  
implementation("org.jetbrains.kotlinx:kotlinx-html-js:${kotlinx\_html\_version}")

</코드>

프로젝트가 그레이들이나 메이븐 같은 별도의 빌드 시스템을 사용하지 않는다면, Project Structure 대화창에 새 라이브러리를 추가할 수 있다(이에 대해서는 이미 **13장 동시성**에서 살펴봤다).

Ktor는 HTML 외에도 벨로시티, 타임리프(Thymeleaf), 머스태시(Mustache) 등 여러 가지 유명한 템플릿 엔진을 지원한다. Ktor 웹사이트에서 이런 엔진을 사용하는 방법과 예제를 볼 수 있다.

## 세션 지원

Ktor는 웹 애플리케이션이 서로 다른 HTTP 요청 간에 원하는 데이터를 보존하고 특정 사용자나 클라이언트를 식별할 수 있도록 세션(session)을 지원한다. 세션에 저장해야 하는 정보의 예를 들면 사용자별 설정, 쇼핑 카트 목록, 인증 데이터 등이 있다.

세션을 사용하려면 세션 기능을 install로 설치하면서 데이터를 어떻게 저장할지 지정해야 한다. 예를 들어 클라이언트 쿠키로 세션을 유지하려면 다음과 같이 코드를 작성해야 한다.

<코드>

install(Sessions) {  
 cookie<MyData>("my\_data")  
}

</코드>

여기서 MyData는 세션 데이터를 표현하는 클래스다. ApplicationCall를 사용해 MyData의 인스턴스에 접근할 수 있고, 클라이언트와 통신할 때 자동으로 직렬화/역직렬화가 이뤄진다. 디폴트 직렬화기(serializer)는 Int, String 등의 간단한 타입의 프로퍼티를 처리할 수 있고, SessionSerializer를 구현해서 install 블록에 제공함으로써 디폴트 직렬화기를 오버라이드할 수도 있다. my\_data 값은 서버에 설치된 다른 세션 정보와 MyData 인스턴스를 구분하는 쿠키 키 역할을 한다.

특정 클라이언트가 사이트에 방문한 횟수를 추적하는 간단한 HTML 페이지 예제를 생각해보자. 먼저 방문 횟수를 저장할 클래스를 정의해야 한다.

<코드>

data class Stat(val viewCount: Int)  
private const val STAT\_KEY = "STAT"

</코드>

STAT\_KEY는 다른 세션 데이터와 상태 데이터를 구분하는 키로 쓰인다.

이제 call.sessions 객체의 get()/set()/getOrSet()을 사용해 이 카운터에 접근할 수 있다. 다음 핸들러는 사용자가 루트 페이지에 접근한 횟수를 알려주는 메시지를 표시한다.

<코드>

private suspend fun ApplicationCall.rootPage() {  
 val stat = sessions.getOrSet { Stat(0) }  
 sessions.set(stat.copy(viewCount = stat.viewCount + 1))  
 respondHtml {  
 body {  
 h2 { +"이 페이지에 ${stat.viewCount} 번 방문하셨습니다." }  
 a("/clearStat") { +"방문 횟수 재설정하기" }  
 }  
 }  
}

</코드>

서버가 보통 다중 스레드 환경에서 실행되기 때문에 세션 인스턴스를 불변 객체로 남기는 것이 타당하다. 세션 상태를 가변 객체로 유지하면 (다중 스레드 환경에서) 오류가 생기기 쉽다. 대신 세션을 get()/set() 함수를 사용해 한꺼번에 저장하고 읽어와야 한다. getOrSet() 함수를 사용하면 존재하지 않는 세션 데이터를 초기화할 수 있다.

마지막 단계는 루트와 /clearStat 경로에 대한 라우팅 규칙을 추가하는 것이다. /clearStat은 세션 데이터를 삭제한 다음 사용자를 다시 루트 페이지로 리디렉션시킨다(따라서 루트 페이지에서는 다시 0이 표시된다).

루트 페이지에 대한 라우팅 규칙은 Routing.kt에 위치하며 다음과 같다.

<코드>

fun Application.configureRouting() {  
 routing {  
 get("/") {  
 call.rootPage()  
 }  
 }  
}

</코드>

한편 /clearStat 경로에 대한 라우팅 규칙은 쿠키를 초기화하는 Security.kt에 존재하며, 다음과 같다. Security.kt안에서 쿠키 키 이름인 STATK\_KEY 최상위 변수를 비공개(private)로 정의했기 때문에 같은 파일 안에 쿠키를 재설정하는 코드가 들어가야 한다.

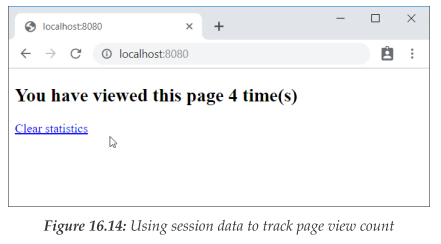
<코드>

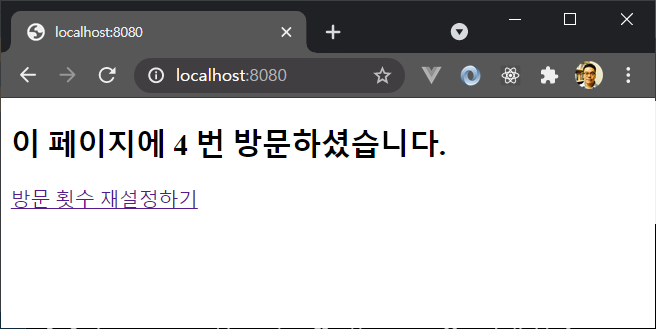
data class Stat(val viewCount: Int) // 앞에서 보여줬던 상태 클래스 정의  
private const val STAT\_KEY = "STAT" // 최상위 변수를 private으로 정의하면 같은 파일 안에서만 사용 가능함  
  
fun Application.configureSecurity() {  
 install(Sessions) {  
 cookie<Stat>(STAT\_KEY)  
 }  
 routing {  
 get("/clearStat") {  
 call.sessions.clear(STAT\_KEY)  
 call.respondRedirect("/")  
 }  
 }  
}

</코드>

애플리케이션을 실행하고 localhost:8080을 브라우저에서 열면 페이지를 재방문할 때마다 카운터가 올라간다. 그림 16-15는 사이트를 여러 번 방문한 결과다.

그림 16-15: 세션 데이터를 사용해 페이지 방문 횟수 추적하기





Clear statistics(방문 횟수 재설정하기) 링크를 클릭하면 서버가 세션 정보를 지우고 카운터를 0으로 재설정한다. /clearStat 핸들러 맨 끝의 responseRedirect()를 보라. 세션 정보를 재설정한 다음, 루트 페이지를 다시 렌더링하기 위해 이 리디렉션이 필요하다.

쿠키 대신 세션을 요청과 응답 헤더에 넣을 수도 있다.

<코드>

install(Sessions) {  
 header<MyData>("my\_data")  
}

</코드>

Ktor 세션은 클라이언트나 서버 아무 쪽에나 저장될 수 있다. 기본적으로 모든 세션 데이터는 클라이언트에 전송되고, 클라이언트는 세션 정보를 기억했다가 다음 요청을 보낼 때 함께 전송한다. Ktor에 내장된 디폴트 직렬화기는 일반 텍스트를 사용하기 때문에 세션 정보가 오가는 것이 보안상 위험할 수 있다. 이 문제를 해결하기 위해 Ktor는 송수신하는 데이터를 추가로 인코딩/디코딩하는 세션 트랜스포머(transformer) 메커니즘을 제공한다.

Ktor에 기본 내장된 트랜스포머 중에 SessionTransportTransformerMessageAuthentication는 세션 데이터와 정해진 알고리즘(디폴트는 SHA256)으로 계산된 세션 데이터에 대한 해시를 함께 전송한다. 가장 단순한 경우, 여러분은 비밀 키만 이 트랜스포머에게 제공하면 된다.

<코드>

install(Sessions) {  
 cookie<Stat>(STAT\_KEY, SessionStorageMemory()) {  
 // 실제로는 SecureRandom 등 암호학적으로 안전한 난수 함수를 써야 한다  
 val key = Random.nextBytes(16)  
 transform(SessionTransportTransformerMessageAuthentication(key))  
 }  
}

</코드>

이 기능을 사용해도 세션 데이터는 바뀌지 않기 때문에, 여전히 제3자가 클라이언트 쪽에서 정보를 볼 수 있지만 세션 정보를 변경할 수는 없다. 서버는 세션 정보에 대한 해시 요약값(digest)을 검사해서 정보가 변조된 경우 비밀키에 기반한 새 세션 정보를 만들어낸다.

더 강력한 보안을 제공하려면 SessionTransportTransformerEncrypt를 사용해야 한다. 이 트랜스포머는 세션 데이터를 암호화해서 제3자가 데이터를 읽지 못하게 막는다. 이 트랜스포머를 설정하려면 암호화 키와 서명 키를 함께 제공해야 한다(서명키는 세션 데이터에 대한 디지털 서명을 생성할 때 쓰인다).

<코드>

install(Sessions) {  
 cookie<Stat>(STAT\_KEY, SessionStorageMemory()) {  
 // 실제로는 SecureRandom 등 암호학적으로 안전한 난수 함수를 써야 한다  
 val encryptionKey = Random.nextBytes(16)   
 val signKey = Random.nextBytes(16)  
 transform(SessionTransportTransformerEncrypt(encryptionKey, signKey))  
 }  
}

</코드>

SessionTransportTransformer를 직접 구현해서 자신만의 트랜스포머를 추가할 수도 있다.

기본적으로 cookie()와 header() 블록은 클라이언트 쪽 세션을 설정한다. 이 경우 모든 세션 데이터가 클라이언트 쪽에 저장되고 매 요청/응답 시마다 세션 데이터도 함께 전송된다. 이에 대한 대안으로, Ktor가 서버 쪽에 세션 정보를 저장할 세션 저장소를 설정하고, 세션 ID만 전송하는 방법이 있다.

<코드>

install(Sessions) {  
 cookie<Stat>(STAT\_KEY, SessionStorageMemory())  
}

</코드>

SessionStorageMemory는 세션 데이터를 서버 메모리에 저장하는 내장 구현이다. 활성화된 클라이언트 수가 늘어남에 따라 메모리 사용량도 늘어나므로 서버 측 세션 정보를 가능한 간결하게 유지해야 한다.

이것으로 Ktor가 제공하는 서버 애플리케이션 관련 기능에 대한 설명을 마친다. 다음 절에서는 Ktor를 사용해 HTTP 클라이언트 프로그램을 작성하는 방법을 살펴본다.

## 클라이언트 기능

Ktor는 서버 애플리케이션뿐 아니라 여러 가지 서버와 통신하는 비동기 클라이언트 개발을 더 쉽게 할 수 있도록 도와준다. 여기서는 HTTP 프로토콜을 사용해 웹 서버와 통신하는 HttpClient 클래스를 중심으로 Ktor가 제공하는 기능 중 일부를 살펴보겠다.

### 요청과 응답

모든 HTTP 클라이언트의 기본 기능은 주로 웹 서버에게 요청을 보내고, 보낸 요청에 따른 응답을 읽어 들이는 것과 관련 있다. Ktor는 HTTP로 요청을 보내고 응답을 받을 때 필요한 다양한 텍스트나 이진 콘텐트 관련 기능을 내장해 제공한다.

Ktor 클라이언트 기능을 사용하려면 우선 다음과 같이 그레이들 설정 파일에 Ktor 클라이언트 코어 기능에 대한 의존관계를 추가해야 한다.

<코드>

implementation("io.ktor:ktor-client-core:$ktor\_version")

</코드>

Ktor 클라이언트에서도 아파치 HttpClient, 자바 HttpClient, 제티, CIO, 안드로이드, OkHttp, 자바스크립트, 네이티브 등 플랫폼에 따라 다양한 엔진을 사용할 수 있다. 우리는 자바 HttpClient를 사용하자. 자바 HttpClient를 사용하려면 다음 의존관계를 더 추가해야 한다.

<코드>

implementation("io.ktor:ktor-client-java:$ktor\_version")

</코드>

이제 실제 클라이언트 코드를 작성할 준비가 됐다. HttpClient로 HTTP 요청을 보내는 가장 단순한 방법은 제네릭 get() 메서드에 타깃 URL을 넘기는 것이다. 제네릭 get() 메서드의 타입 인자는 클라이언트가 서버 응답을 표현하기 위해 반환하는 객체의 타입을 결정한다. 예를 들어 응답 본문을 한 덩어리의 텍스트로 받고 싶다면 get<String>()을 사용할 수 있다.

<코드>

import io.ktor.client.HttpClient  
import io.ktor.client.request.\*  
import kotlinx.coroutines.runBlocking  
import java.time.DayOfWeek  
import java.time.format.TextStyle  
import java.util.\*  
  
fun main() {  
 runBlocking {  
 HttpClient().use {  
 val url = "http://worldtimeapi.org/api/timezone/Australia/Brisbane.txt"  
 val result = it.get<String>(url)  
 val prefix = "day\_of\_week:"  
 val from = result.indexOf(prefix)  
 if (from < 0) return@runBlocking  
 val to = result.indexOf('\n', from + 1)  
 if (to < 0) return@runBlocking  
 val dow = result  
 .substring(from + prefix.length, to)  
 .trim()  
 .toInt()  
 println("브리즈번은 ${DayOfWeek.of(dow).getDisplayName(TextStyle.FULL, Locale.KOREAN)} 입니다!")  
 }  
 }  
}

</코드>

참고로 http://worldtimeapi.org/에 보내지는 요청이 많으면 503 오류가 나올 수도 있다. 콘솔에 503 응답코드가 보이는 예외가 표시되면 잠시 기다렸다 프로그램을 다시 실행해 보라.

문자열 표현 외에 바이트 배열로 응답 본문을 변경해서 바이너리 형태로 데이터를 받을 수도 있다.

<코드>

val bytes = client.get<ByteArray>(url)

</코드>

또는 ByteReadChannel을 활용해 비동기로 데이터를 얻을 수도 있다.

<코드>

val channel = client.get<ByteReadChannel>(url)

</코드>

HttpClient가 제공하는 요청 메서드들은 일시중단 함수이기 때문에 코루틴 문맥 안에서 호출해야 한다. 그래서 이 예제에서는 runBlocking() 함수를 사용했다. 일반적으로 코틀린 코루틴이 제공하는 어떤 비동기 계산을 사용해도 좋다.

HttpClient는 close()를 통해 명시적으로 클라이언트를 종료시켜야 한다는 점에 유의하라. 클라이언트의 범위가 제한된 경우, 다른 Closeable 타입의 인스턴스들과 마찬가지로 use 블록을 사용하면 종료 처리를 감출 수 있다.

짐작했겠지만, get() 메서드는 HTTP GET 동사에 해당한다. Ktor는 post(), put(), delete(), patch(), head(), options()처럼 HTTP 1.x/2.x 표준이 제공하는 모든 요청 방식을 처리하는 비슷한 함수를 지원한다.

각 함수는 HttpRequestBuilder.() -> Unit 타입의 선택적인 람다를 받을 수 있다. 이 람다 안에서 요청에 헤더나 본문 등을 추가하는 작업을 수행할 수 있다. 헤더를 추가하려면 HttpRequestBuilder가 제공하는 header 메서드를 사용하면 된다.

<코드>

client.get<ByteArray>(url) {  
 header(“Cache-Control”, “no-cache”)  
}

</코드>

HttpRequestBuilder의 headers 프로퍼티를 통해 HeadersBuilder 블록을 사용할 수도 있다.

<코드>

HttpClient().use {  
 it.get<ByteArray>(URL) {  
 headers {  
 clear()  
 append("Cache-Control", "no-cache")  
 append("My-Header", "My-Value")  
 }  
 }  
}

</코드>

HttpClient는 서버가 클라이언트 소프트웨어를 식별(웹 브라우저 종류나 버전 등)할 때 도움을 주기 위해 User-Agent 헤더를 제공하는 단순한 방법을 지원한다. 이를 위해서는 UserAgent 기능을 설치하고 agent 프로퍼티에 적절한 헤더 값을 지정하라.

<코드>

val client = HttpClient(){  
 install(UserAgent) {  
 agent = "Test Browser"  
 }  
}

</코드>

이제 client.use{} 등으로 클라이언트를 사용하면 User-Agent 헤더 값이 Test Browser로 설정되어 요청이 전달된다.

미리 정의된 User-Agent 설정이 두 가지 있다.

* BrowserUserAgent(): 크롬, 사파리 등 유명한 브라우저를 지원한다
* CurlUserAgent(): Curl 에 해당하는 에이전트 문자열

이 두 함수는 전체 기능 설치 블록을 대신할 수 있다. 예를 들면 다음과 같다.

<코드>

// io.ktor:ktor-client-java만 가져왔으므로 HttpClient()에 전달할 엔진 종류로 Java만 사용할 수 있다  
val client = HttpClient(Java) {  
 BrowserUserAgent()  
}

</코드>

요청 본문(예: POST 요청)을 제공하고 싶으면 HttpRequestBuilder의 body 프로퍼티를 사용한다. 가장 간단한 방법은 String 표현을 사용하는 것이다.

<코드>

client.get<String>(url) {  
 body = “my\_key1=my\_value1&my\_key2=my\_value2”  
}

</코드>

또는 OutgoingContent를 상속한 TextContent 등의 구현을 제공할 수도 있다. TextContent는 String을 사용하는 것과 비슷하지만 MIME 타입을 지정할 수 있고, ByteArrayContent는 이진 데이터를 넘길 때 유용하고, LocalFileContent는 파일을 전송할 수 있다. 추가로 JsonFeature를 설치하면 자동으로 임의의 객체를 JSON 형태로 직렬화할 수 있다. 직렬화에 대해서는 **17장 마이크로서비스** *구축*의 Ktor 부분을 살펴보라.

submitForm() 함수는 HTML 폼을 모방하여 여러 가지 일반적인 시나리오를 구현한다. 예를 들어 다음 코드는 HTML DSL 절에서 만들었던 서버 애플리케이션에게 폼을 제출한다.

<코드>

HttpClient().submitForm<String>(  
 url = "http://localhost:8080",  
 encodeInQuery = true,  
 formParameters = parametersOf(  
 "from" to listOf("0"),  
 "to" to listOf("100"),  
 "count" to listOf("10")  
 )  
)

</코드>

파라미터를 키-값 쌍 형태로 전달하되 encodeInQuery 인자가 폼 데이터를 어떤 식으로 서버에 전달할지를 결정한다.

* true: 파라미터를 요청 URL의 쿼리 파라미터로 인코딩해서 HTTP GET을 사용해 요청한다
* false: 파라미터를 요청 본문에 인코딩해서 HTTP POST로 요청한다

Ktor 클라이언트는 HTTP 리디렉션을 처리할 수 있다. 이 기능은 디폴트로 설치되어 있기 때문에, 서버가 리디렉션 응답을 돌려주면 클라이언트가 자동으로 새 위치로 이동한다.

## 쿠키

HTTP 쿠키는 클라이언트의 여러 요청 간에 상태를 유지할 때 일반적으로 사용하는 방법으로, HTTP 헤더에 작은 데이터 꾸러미를 전달하는 방식을 사용한다. 클라이언트(웹 브라우저 등)는 처음에 서버로부터 쿠키를 전달받아서 클라이언트 쪽에 저장하며, 이후 다른 요청을 보낼 때마다 쿠키를 함께 전달한다. 쿠키는 서버 세션을 유지할 때 유용하다. 세션에 대해서는 앞에서 Ktor 서버에 대해 설명할 때 이미 다뤘다. 여기서는 클라이언트 쪽에서 쿠키를 다루는 방법을 살펴보자.

HTTP 서버가 쿠키를 사용해 클라이언트 호출 간 데이터를 유지한다면, 클라이언트는 이런 데이터를 제대로 저장했다가 필요할 때마다 HTTP 요청에 추가해줘야 한다. Ktor는 쿠키 기능을 제공하기 때문에 이 작업을 손쉽게 할 수 있다. Ktor 클라이언트의 쿠키 기능 사용법을 알아보기 위해 Ktor 서버에 대해 설명할 때 사용한 뷰 카운터 애플리케이션(그림 16-15)에 대한 간단한 클라이언트를 작성해보자.

<코드>

import io.ktor.client.\*  
import io.ktor.client.request.get  
import io.ktor.client.features.cookies.\*  
import kotlinx.coroutines.delay  
import kotlinx.coroutines.runBlocking  
  
private const val H2 = "<h2>"  
private const val H2\_CLOSE = "<h2>"  
  
fun main() {  
 HttpClient(){  
 install(HttpCookies)  
 }.use { client ->  
 runBlocking {  
 repeat(5) {  
 val htmlText = client.get<String>("http://localhost:8080")  
 val from = htmlText.indexOf(H2)  
 val to = htmlText.indexOf(H2\_CLOSE)  
 if(from < 0 || to < 0) return@runBlocking  
 val message = htmlText.substring(from+H2.length, to)  
 println(message)  
 delay(500)  
 }  
 }  
 }  
}

</코드>

코드를 보면 클라이언트가 루트 경로(/)에 대한 응답을 받아서 <h2> 태그에 들어있는 메시지를 가져온 다음, 표준 출력에 출력한다. 여기서 install(HttpCookies) 호출만 하면 HttpClient가 쿠키를 처리하게 된다. 요청/응답이 다섯 번 일어나기 때문에(그리고 매번 쿠키가 갱신된다) 출력은 다음과 같다.

<코드>

이 페이지에 0 번 방문하셨습니다.  
이 페이지에 1 번 방문하셨습니다.  
이 페이지에 2 번 방문하셨습니다.  
이 페이지에 3 번 방문하셨습니다.  
이 페이지에 4 번 방문하셨습니다.

</코드>

기본적으로 HTTP 클라이언트는 빈 쿠키로 시작하며, 서버가 응답으로 돌려준 쿠키를 다음 요청에 전달한다. 이 과정은 전형적인 브라우저 동작과 일치한다. 때로는 서버에서 쿠키를 받지 않았어도 미리 설정된 쿠키를 요청에 넣어서 전송해야 할 수도 있다. 예를 들어 서버 응답을 확인하기 위한 테스트 케이스에서 쿠키를 사용하는 경우 이런 식으로 미리 설정한 쿠키가 필요할 수 있다. 이런 경우에는 ConstantCookiesStorage를 사용해 쿠키 설정의 storage 프로퍼티를 변경하면서 미리 정의된 쿠키를 설정하면 된다.

<코드>

val client = HttpClient() {  
 install(HttpCookies) {  
 storage = ConstantCookiesStorage(Cookie("STAT", "viewCount=%23i2"))  
 }  
}

</코드>

쿠키의 viewCount 값을 2로 고정한다는 사실을 쉽게 알아볼 수 있다. 이렇게 쿠키 설정을 바꾼 클라이언트 애플리케이션을 실행하면 서버로부터 같은 응답을 다섯 번 돌려받는다.

<코드>

이 페이지에 2 번 방문하셨습니다.  
이 페이지에 2 번 방문하셨습니다.  
이 페이지에 2 번 방문하셨습니다.  
이 페이지에 2 번 방문하셨습니다.  
이 페이지에 2 번 방문하셨습니다.

</코드>

서버에서 받은 쿠키 값을 저장하는 디폴트 동작이 ConstantCookiesStorage로 인해 자동으로 변경된다. 원한다면 CookiesStorage 인터페이스를 구현해서 여러분 자신만의 쿠키 저장 정책을 설정할 수도 있다.

## 결론

이 장에서는 연결된 클라이언트/서버 애플리케이션 개발을 지원하는 Ktor 프레임워크의 기본 기능을 소개했다. 기본 Ktor 프로젝트 구조를 이해하고, 서버와 클라이언트 애플리케이션에 대해 Ktor가 지원하는 응답과 요청 처리, 라우팅 규칙, 세션 등의 기능을 살펴봤다. 여기서는 여러분이 Ktor가 자바/코틀린 개발자에게 제공하는 기능을 살펴볼 수 있도록 몇 가지 기본적인 면에 대해 간략히 다뤘다. Ktor 웹사이트(<https://ktor.io>)를 참고하고 특히 https://ktor.io/samples에 있는 예제에 관심을 기울이기를 바란다.

다음 장에서도 연결과 관련한 주제를 계속 다룬다. 특히 코틀린으로 마이크로서비스를 개발하는 방법에 대해 다룰 것이다. 마이크로서비스 아키텍처의 기초를 설명하고, 특히 Ktor와 스프링 부트(Spring Boot) 플랫폼을 사용해 마이크로서비스를 구축할 때 코틀린이 어떻게 쓰이는지 살펴본다.

## 질문

1. Ktor 프로젝트를 설정하는 기본 단계를 설명하라.
2. Ktor에서 HTML 콘텐트를 어떻게 생성할 수 있는가? HTML DSL의 기본 기능에 대해 설명하라.
3. 클라이언트가 HTTP 요청을 통해 전달한 데이터를 Ktor 서버 애플리케이션에서 어떻게 추출할 수 있는가?
4. Ktor에서 HTTP 응답을 생성하는 기본적인 방법에 대해 설명하라.
5. Ktor의 라우팅 DSL에 대해 설명하라.
6. 웹 애플리케이션에 세션 지원을 어떻게 추가할 수 있는가? 서버와 클라이언트 세션의 차이에 대해 설명하라.
7. Ktor 클라이언트에서 HTTP 요청을 어떻게 만들고 송신할 수 있는지 설명하라.
8. Ktor 클라이언트에서 어떻게 HTTP 응답의 본문과 헤더에 접근할 수 있는가?
9. Ktor의 클라이언트 쪽 쿠키 지원에 대해 설명하라.

1. 옮긴이 - url?parma1=value1&...에서 ? 다음에 오는, &로 구분된 모든 param1=value1 들이 요청 쿼리 파라미터다. [↑](#footnote-ref-1)