|  |
| --- |
| 버전확인  gradle –-version |

**초기화**

그래들이 초기화될 때 settings.gradle과 build.gradle 파일의 상호작용을 통

해서 build.gradle에 있는 프로젝트의 정보를 수집합니다. settings.gradle 파

일의 정보로 해당 프로젝트를 싱글 프로젝트로 구성할 것인지 멀티 프로젝트로 구

성할 것인지 결정하고, 프로젝트의 인스턴스를 생성합니다. 현재 작업 중인 루트

디렉터리에 settings.gradle 파일이 없으면 상위 디렉터리에 settings.gradle

파일이 있는지 확인합니다. 상위 디렉터리에도 settings.gradle 파일이 없다면

그 빌드는 싱글 프로젝트로 인식되어 실행됩니다.

**설정**

프로젝트 인스턴스화가 끝나면 설정 정보가 프로젝트에 반영되는 단계입니다. 그

래들은 그루비로 만들어진 빌드 도구라 런타임 시에도 설정을 반영할 수 있는데,

settings.gradle 파일의 정보로 프로젝트를 설정합니다.

**실행**

그래들은 이전 단계에서 구성된 Task의 부분집합을 이름으로 구분하고, 그 이름

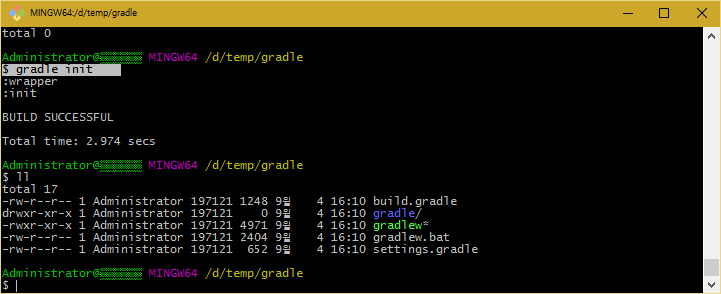
이 Gradle 명령에 전달되어 실행됩니다

|  |
| --- |
| help 명령을 입력하면 확인할 수 있습니다.  gradle help --task :init |

**기본 gradle 프로젝트 생성**

gradle init --type basic

타입 기본값은 ‘basic’으로, type 옵션을 추가하지 않으면 프로젝트가 Basic 타입으로 구성됩니다. init 명령이 성공하면 ‘BUILD SUCCESSFUL’이 출력됩니다.‘wrapper’는 init 명령을 수행하면 자동으로 표시됩니다(그래들은 설치 없이 사용할수 있도록 래퍼 실행 파일을 제공합니다).



init 실행 시 자동으로 생성한 gradlew, gradlew.bat 파일을 GitHub 같은 저장소에 공유하면 같은 프로젝트를 진행하고 있는 다른 사용자는 그래들을 설 치하지 않고도 사용할 수 있습니다. 즉, 그래들을 설치하지 않은 다른 사용자가 ‘gradlew’를 입력하면 그래들을 자동으로 내려받아 gradle 명령을 실행합니다.

이 설정은 ‘gradle/wrapper/gradle-wrapper.properties’에 정의되어 있습니다. settings.gradle은 멀티 프로젝트를 구성하는 데 사용되는 파일이고,

build.gradle은 그래들을 사용할 때 가장 중요하고 기본이 되는 파일입니다.

build.gradle 파일의 내용을 좀 더 자세히 확인해 보겠습니다.

**task추가**

|  |
| --- |
| task hello {  println 'Gradle Hello'  } |

**task 확인하기**

|  |
| --- |
| gradle tasks |

**task 실행**

|  |
| --- |
| gradle hello |

**Task 실행 순서 제어하기**

실행 순서를 제어하는 가장간단한 방법은 <<(레프트 시프트Left Shift) 연산자를 사용하는 것입니다(일반적으로 C

언어에서는 비트 연산자로 사용하지만, 그래들에서는 구문을 삽입한다는 의미의 클로저Closure02로사용합니다).

|  |
| --- |
| task hello {  println 'Gradle Hello'  }  task run << {  println 'running now'  }  task start {  println 'ready'  }  gradle run |

run Task에 추가한 레프트 시프트 연산자 때문에 run Task가 가장 마지막에 실행됩니다. 이는 run Task가 빌드의 맨 마지막 단계까지 유효범위가 유지되기 때문입니다.

First와 Last를 이용하여 연관성 있는 Task들을 좀 더 구조적으로 순서를 표현할 수 있습니다.(지정된 단어)

|  |
| --- |
| task cellphone {  description = 'Display calling message'  doLast {  println 'doLast 1통화하기'  }  doFirst {  println 'doFirst 전화걸기'  }  doLast {  println 'doLast 2전화끊기'  }  } |

**디폴트 Task 지정하기**

gradle만 처도 기본 디폴트 task가 실행된다

|  |
| --- |
| defaultTasks 'myBasicTask'  task myBasicTask << {  println "default task"  }  task other << {  println "other run"  } |

**Task에 설명 추가하기**

|  |
| --- |
| defaultTasks 'myBasicTask'  task myBasicTask(description:'디폴트 Task입니다.') << {  println "default task"  }  gradle tasks |

**다른 Task와 연관 지어 실행하기**

|  |
| --- |
| tasks.myBasicTask.dependsOn other |
| task myBasicTask(description:'디폴트 Task입니다.', dependsOn:'other') << {  println "default task"  } |

**로깅 설정하기**

|  |
| --- |
| Logging.level = LogLevel.레벨 |

표 2-1 그래들의 로그 레벨

|  |  |
| --- | --- |
| 레벨 | 용도 |
| LIFECYCLE | 에러 메시지 출력 |
| QUIET | 간결한 출력 (시스템 로그는 표시하지 않음) |
| INFO | QUIET보다 상세한 메시지 출력 |
| DEBUG | 모든 메시지 출력 |
|  |  |

LIFECYCLE은 기본값으로, 아무런 로그 레벨 파라미터를 설정하지 않을 때 동작합니다. 앞에서 만들었던 Task에 로그 레벨 파라미터를 주고 실행해 보면 출력 결과가 다른 것을 확인할 수 있습니다.

QUIET로 설정하면 다음 그림처럼 ‘BUILD SUCCESSFUL’이 출력되지 않습니다.

이와는 반대로 DEBUG 레벨로 실행하면 Task에 관련된 모든 정보가 출력됩니다.

|  |
| --- |
| task myBasicTask(description:'디폴트 Task입니다.', dependsOn:'other') << {  logging.level = LogLevel.DEBUG  println "default task"  } |

**Task 그룹화하기**

Task들을 그룹으로 분류하여 좀 더 명확하게 관리할 수 있습니다. Group 변수를

선언하고 각 Task에 할당하면 해당 그룹의 Task로 인식합니다.

|  |
| --- |
| def myGroup = 'first'  task mymemtask1(group: myGroup) << {  println "my mem1"  }  task mymemtask2(group: myGroup) << {  println "my mem2"  } |

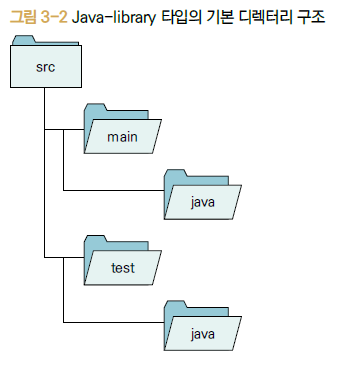
**그래들로 자바 프로젝트 만들기**

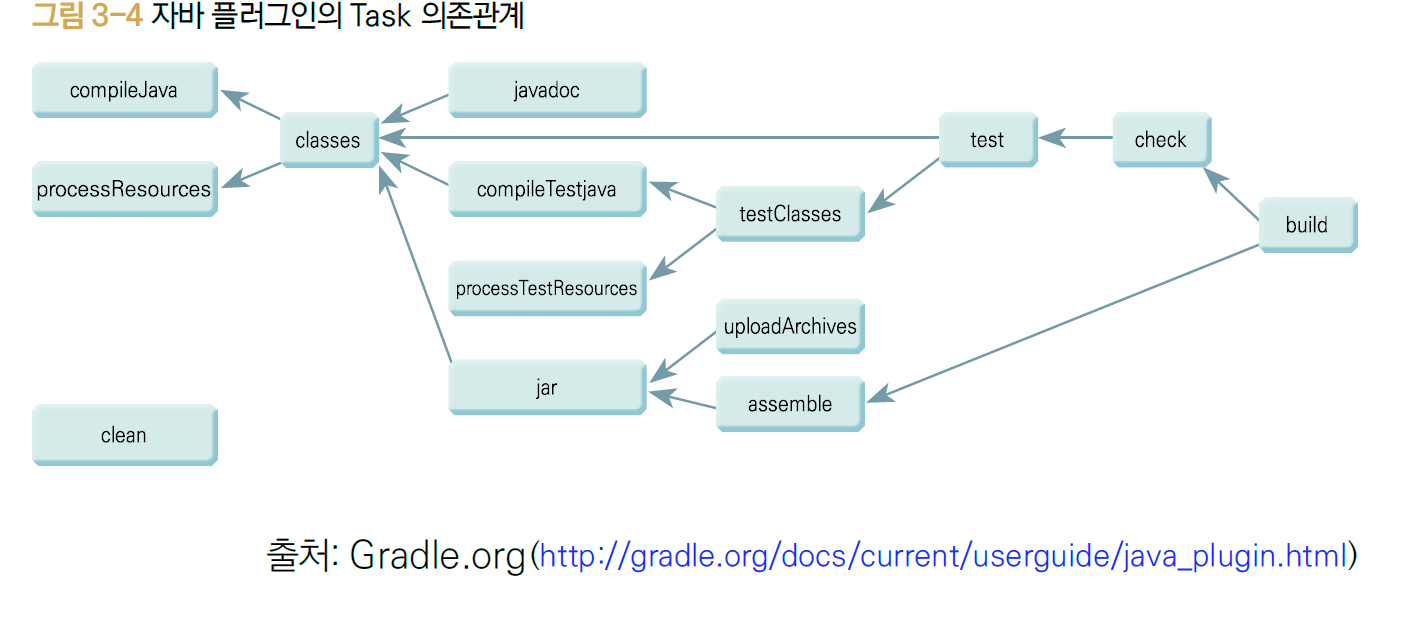
이미 build.gradle 파일이 있는 경우에는 다음과 같이 플러그인 설정만 해도 됩니다.

|  |
| --- |
| apply plugin : 'java' |

init 명령에 java-library 타입으로 새로운 프로젝트를 생성합니다.

|  |
| --- |
| gradle init –-type java-library |





**레이아웃 구성하기**

클래스는 아니지만 로그 설정이나 기타 다른 라이브러리 설정 파일이 클래스패스에 로드되어야 하는 경우에 다음과 같이 Task로 별도의 resources 디렉터리를

만들 수 있습니다.다음 내용을 build.gradle 파일에 추가하고 gradle makeResourceFolder를실행합니다.

|  |
| --- |
| task makeResourceFolder << {  sourceSets\*.resources.srcDirs\*.each {it.mkdirs()}  } |

gradle init 명령을 사용하지 않고 build.gradle 파일만 공유하여 사용하는경우도 있을 것입니다. 이럴 때 한번에 소스 디렉터리와 리소스 디렉터리가 생성

되도록 makeResourceFolder Task를 수정하겠습니다.makeResourceFolder Task에 java를 추가하여 새로운 Task를 만듭니다. 여기서

sourceSets는 디렉터리 전체를 의미하며, 실제로 sourceSets는 그래들 내부에서 인터페이스로 정의되어 있습니다. Java 플러그인을 사용할 때 java의 소스 디렉터리와 테스트 디렉터리를 논리적으로 표현하는 단위가 sourceSets입니다.

|  |
| --- |
| task initJavaFolder << {  sourceSets\*.java.srcDirs\*.each { it.mkdirs() }  sourceSets\*.resources.srcDirs\*.each {it.mkdirs() }  } |

그런데 기존 프로젝트의 디렉터리 위치가 그래들과는 다른 경우도 있어서 소스디렉터리가 두 개가 되거나 별도의 디렉터리도 함께 지정해야 하는 경우가 있습

니다. 이런 경우에는 sourceSets 속성 하위에 디렉터리를 추가합니다. 예를 들어, 자바 프로젝트의 소스 디렉터리 역할을 하는 디렉터리명이 ‘mysrc’라면 다음

과 같이 추가하면 됩니다. build.gradle 파일에 이 내용을 추가한 후 initJavaFolder Task를 실행합니다.

|  |
| --- |
| sourceSets {  main{  java{  srcDirs = ['src', 'mysrc']  }  }  } |
| sourceSets{  main{  java{  srcDirs = ['src', 'mysrc']  }  }  test{  java{  srcDirs = ['test', 'mytest']  }  }  } |

추가한 디렉터리를 그래들에서 잘 인식하는지 확인해 보겠습니다. 다음 내용의

printJavafolder Task를 build.gradle 파일에 추가하고 실행합니다.

|  |
| --- |
| task printJavafolder << {  sourceSets {  main{  println "java.srcDirs = ${java.srcDirs}"  println "resources.srcDirs = ${resources.srcDirs}"  }  }  } |
| srcDirs 디렉터리에 추가한 mysrc 디렉터리까지 함께 출력됩니다.  :printJavafolder  java.srcDirs = [K:\99.work\gradlesource\gradlejava\src, K:\99.work\gradlesource  gradlejava\mysrc]  resources.srcDirs = [K:\99.work\gradlesource\gradlejava\src\main\resources]  BUILD SUCCESSFUL  Total time: 3.951 secs |

최종본

|  |
| --- |
| **apply** plugin: 'java'  **sourceCompatibility** = 1.8  targetCompatibility = 1.8  buildDir = 'build'  **repositories** {  jcenter()  }  **task** makeResourceFolder << {  **sourceSets**\*.resources.**srcDirs**\*.each {it.mkdirs()}  }  **task** initJavaFolder << {  **sourceSets**\*.java.**srcDirs**\*.each { it.mkdirs() }  **sourceSets**\*.resources.**srcDirs**\*.each {it.mkdirs() }  }  **sourceSets** {  **main** {  java {  **srcDirs** = ['src', 'mysrc']  }  }  **test** {  java {  **srcDirs** = ['test', 'mytest']  }  }  }  **task** printJavafolder << {  **sourceSets** {  **main** {  println "java.srcDirs = ${java.srcDirs}"  println "resources.srcDirs = ${resources.srcDirs}"  }  }  }  **dependencies** {  compile 'org.slf4j:slf4j-api:1.7.5'  testCompile 'junit:junit:4.11'  } |

엔코딩

|  |
| --- |
| compileJava.options.encoding = 'UTF-8'  또는 gradle.properties 파일에서 그래들의 jvmargs 환경변수로 값을 추가할  수 있습니다. org.gradle.jvmargs=-Dfile.encoding=UTF-8 |

컴파일

|  |
| --- |
| gradle compileJava |

Test쪽 제외하고 컴파일 하기

|  |
| --- |
|  |
| 여기서 compileJava를 실행하는 것은 자바 파일을 실행하려는 목적이므로 test를 무시하고 실행해 보겠습니다.  test를 무시하는 방법으로는 test 디렉터리를 삭제하는 방법도 있지만, 경우에 따라서는 test 코드가 작성된 기존 프로젝트에 코드를 추가하여 실행해야 할 수도있습니다.  예를 들어, 이미 사용하고 있는 라이브러리로 개발을 진행하거나 오픈소스 프로젝트를 받아서 개발할 때 test를 먼저 실행하는 경우가 많습니다.  그런데 지금처럼 변경 사항이 많지 않을 때 test를 무시하고 실행할 수 있다면, 기존프로젝트의 설정을 보존하고 시간을 단축할 수 있습니다.  앞에서 프로젝트 레이아웃을 설정할 때 sourceSets 속성을 이용해서 디렉터리를 추가한 것처럼 test를 건너뛰고자 할 때도 마찬가지로  sourceSets에 설정을추가하여 test를 무시하고 실행할 수 있습니다.  앞에서 오류가 발생한 것은 compileJava에 test 디렉터리가 포함되어 있기 때문이므로 test 실행 시 test 디렉터리에 대해서 제외하게 하는 설정이 필요합니다.  그래서 exclude를 추가합니다. |
| sourceSets{  main{  java{  srcDirs = ['src', 'mysrc']  exclude 'test/\*'  }  }  } |
|  |

**삭제후 재컴파일**

|  |
| --- |
| clean task |
| build 디렉터리는 반복해서 파일이 생성되는 디렉터리이므로 이전에 생성된 파일과 중복되지 않도록 파일을 삭제한 후에 컴파일해야 하는 경우가 있습니다. 이  럴 때는 clean Task를 사용합니다. clean Task를 사용하면 빌드한 결과물이 생성되는 build 디렉터리와 디렉터리 내용 전체가 삭제됩니다. 또한, clean Task는 다른 Task와 조합해서 사용할 수 있습니다. |

**의존성 관리하기**

**자바 프로젝트의 라이브러리 스코프**

자바 프로젝트에서 지원하는 스코프 확인

|  |
| --- |
| gradle dependencies    스코프별로 포함된 라이브러리가 출력되는데, 주로 사용하는 스코프는 compile,testCompile, runtime입니다. |

스코프 종류

|  |  |  |
| --- | --- | --- |
| 스코프 설정 | 관련 Task | 설명 |
| compile | compileJava | 컴파일 시 포함해야 할 때 |
| runtime | - | 실행시점에 포함해야 할 때 |
| testCompile | compileTestJava | 테스트를 위해 컴파일할 때 포함해야 할 때 |
| testRuntime | test | 테스트를 실행시킬 때 |

서블릿으로 웹 페이지를 작성할 때 실행할 수 있는 클래스를 작성하기 위해서는 HttpServlet 클래스를 상속받아서 처리해야 합니다. 그리고 HttpServlet 클래스를 상속하기 위해서는servlet-api.jar 파일이 컴파일 시 포함되어야 합니다. 이렇게 해서 컴파일을 하면 HTTP 요청을 처리할 수 있는 클래스가 준비됩니다

**라이브러리 추가**

<http://mvnrepository.com/> 에서 lib 검색한다. 여러 repository등이있다.

|  |
| --- |
| iText 라이브러리를 추가하여 Document 클래스와 Paragraph 클래스를 사용할 수 있도록 하겠습니다. 다음과 같이 iText 라이브러리를 compile 스코프로dependencies에 추가합니다. |
| dependencies {  compile 'com.itextpdf:itextpdf:5.5.5'  compile 'org.slf4j:slf4j-api:1.7.5'  testCompile 'junit:junit:4.11'  } |

**패키징하기**

|  |
| --- |
| Java 플러그인을 사용한 상태에서 build 명령을 실행하면 기본으로 JAR 형태로패키징됩니다. build는 메이븐에서 package를 실행할 때처럼 가장 마지막에 수행되는 명령으로, 이미 수행된 명령인 경우에는 [그림 3-15]와 같이 UP-TO-DATE 가 Task 옆에 표시됩니다. 그래들의 특성 중에서 점진적인 빌드라는 표현이 있는데, 이는 이렇게 이미 수행이 완료된 Task가 다시 실행되는 경우에는 해당 Task를 실행하지 않고 다음 Task를 실행하는 것을 말합니다. |
|  |
|  |
| 그래들에서 JAR 파일은 libs 디렉터리에 생성되며 기본값으로 설정되어 있습니다. 실행하기 위한 형태로 배포하는 JAR 파일과 iText 라이브러리를 추가해서  사용하는 것처럼 다른 프로젝트에 라이브러리로 사용하는 경우가 있어서 JAR 파일은 두 가지 형태로 만듭니다.  JAR 파일이 위치한 경로로 이동해서 JAR를 실행하면 ‘Manifest 속성이 없다’는오류와 함께 실행이 되지 않습니다. |
|  |
| WAR 파일을 만들 때 WEB-INF 디렉터리와 메타 정보를 추가하는 것처럼 JAR파일을 만들 때도 JAR 파일에 규격에 맞도록 Manifest 정보를 작성하고 여러 클  래스 파일 중에 실제로 실행될 main 메서드를 가진 클래스를 명시해야 합니다.그래들은 데스크탑에서 실행될 프로그램을 만드는 경우에 Application 플러그  인02을 이용해서 메타 정보를 작성하지 않고 단순하게 실행할 수 있습니다.앞의 자바 파일에 Application 플러그인을 추가하고 mainClass를 지정해 보겠습니다. |
| apply plugin: 'application'  mainClassName = "pdfs.Generic.ITextHello" |

**실행시 아규먼트 파라미터 넣기**

|  |
| --- |
| run {  args  } |
| gradle run -Pargs=”ggggg” |

**Runnable JAR 만들기**

|  |
| --- |
| jar {  from files(sourceSets.main.output.classesDir)  from {configurations.compile.collect {zipTree(it)}} {  exclude "META-INF/\*.SF"  exclude "META-INF/\*.DSA"  exclude "META-INF/\*.RSA"  }  manifest {  attributes "Main-Class": "pdfs.ParamPDF"  }  } |
|  |

**JAR Task 실행**

|  |
| --- |
|  |

**배포용 파일묶음 만들기**

|  |
| --- |
| 여러 오픈소스를 보면 사용자의 편의를 위해 실행 파일 외에도 프로젝트 내용을ZIP이나 Tar로 묶어서 제공합니다. 프로젝트의 소스 디렉터리를 사용자에게 제공하려면 Application 플러그인의 distZip Task를 사용해서 파일을 묶을 수있습니다. |
| Application 플러그인의 distZip Task 실행    distZip Task가 성공적으로 실행되면 build 디렉터리 하위에 ZIP 파일과 스크립트가 생성됩니다. |

**ZIP 파일로 소스 파일 묶기**

|  |
| --- |
| task srcZip(type:Zip) {  classifier = 'src'  from sourceSets\*.allSource  destinationDir = file("$buildDir/myzips")  archiveName = "$project.name-sources.zip"  } |

**Tar 파일로 소스 파일 묶기**

|  |
| --- |
| task srcTar(type:Tar) {  classifier = 'src'  from sourceSets\*.allSource  destinationDir = file("$buildDir/myzips")  archiveName = "$project.name-sources.tar.gz"  compression = Compression.GZIP  } |

**그래들 마이그레이션**

**스프링 MVC 웹 프로젝트 구성**

|  |
| --- |
| WAR는 자바 기반의 웹 프로젝트의 최종 결과 포맷입니다. 이것은 기본적인 내용이지만 웹 프로젝트에서 배포를 위한 최소한의 단위가 되므로 매우 중요합니다 |
| apply plugin:'war' |

|  |
| --- |
| WAS(Web Application Server)마다 배포 방식과 설정이 조금씩 다르지만 기본적으로WAR 파일에 대한 규격은 동일합니다. WAR 파일과 JAR 파일 모두 파일 시스템측면에서는 ZIP 파일과 유사하며 JVM 위에서 실행할 수 있도록 메타 정보가 추가되어 있습니다. JAR의 경우에는 실행될 main 클래스를 명시하며, WAR 파일의경우에는 단독으로 실행할 수 없고 서버 컨테이너에 의해서 실행되므로 배포에 대한 메타 정보가 담겨 있습니다. 그래서 web.xml을 배포 서술자DD, Deploy Description라고 부르며, 이 파일에는 웹 프로젝트에 대한 설정 정보가 담겨 있습니다.서버에서는 배포 서술자 정보를 읽어 컨텍스트를 생성합니다. 컨텍스트는 기본적으로 WAR 파일명과 동일한 이름으로 생성되고, 사용자가 웹 디렉터리에 있는 자원에 접근할 수 있게 합니다. 다음과 같이 JSP, HTML 같은 웹 리소스들이 위치할 디렉터리를 생성합니다. |
| def webappDir = "$rootDir/src/main/webapp" |
| 최근에는 보안상의 이유로 사용자가 JSP에 직접적인 접근을 할 수 없게 WEBINF  하위에 디렉터리를 만들어 사용합니다. 이 예제에서도 WEB-INF 하위에 디렉토리에 지정합니다 |
| file(webappDir + "/WEB-INF/views").mkdirs()  file(webappDir + "/META-INF").mkdir() |
|  |
| 그래들로 기존에 생성한 웹 디렉터리 하위에 디렉터리들을 생성할 수 있습니다.기존에 자바 소스 디렉터리를 생성하는 구문과 결합하여 프로젝트 디렉터리 생성을 하나의 Task로 구성하고 이클립스나 인텔리제이와 같은 IDE의 메타 정보를생성할 때 의존관계를 맺도록 설정한다면 좀 더 간편하게 시작할 수 있습니다 |
| task initProject << {  sourceSets\*.java.srcDirs\*.each { it.mkdirs() }  sourceSets\*.resources.srcDirs\*.each { it.mkdirs() }  def webappDir = "$rootDir/src/main/webapp"  file(webappDir + "/WEB-INF/views").mkdirs()  file(webappDir + "/META-INF").mkdir()  } |

**라이브러리 구성**

|  |
| --- |
| 자바 기반의 웹 프로젝트를 구성할 때 가장 기본이 되는 서블릿부터 추가해 보겠습니다. 서블릿 3.0 이후 버전부터는 web.xml을 사용하지 않고 어노테이션 기반으로 설정을 간소화해서 사용할 수 있습니다. 여기에서는 서블릿 3.0 이후의 버전을 사용합니다.  서블릿 API는 각 서버에서 제공하기 때문에 소스에 포함하지 않고 서버의 자원을사용하도록 설정하는 것이 효율적입니다. 이렇게 서버에서 제공하는 라이브러리를 사용하는 경우에는 provided를 사용하는데, 다음과 같이 설정하면 서버에서제공하는 서블릿을 사용하게 됩니다. |
| providedCompile group: 'javax.servlet', name: 'javax.servlet-api',  version:'3.0.1' |
| [스프링 라이브러리 설정 ]  compile group: 'org.springframework', name: 'spring-webmvc',version:'3.2.9.RELEASE'  compile group: 'org.springframework', name: 'spring-tx', version:'3.2.9.RELEASE'  compile group: 'org.springframework', name: 'spring-jdbc',version:'3.2.9.RELEASE'  compile group: 'org.springframework', name: 'spring-orm',version:'3.2.9.RELEASE' |
|  |
| 스프링에서 common-logging을 기본 로그로 사용하므로 이를배제하고 SL4J를 사용하도록 하는 설정이 필요합니다. 이렇게 특정 라이브러리를배제하려면 exclude로 해당 라이브러리를 참조하지 않도록 설정할 수 있습니다. |
| configuration.exclude group: 'commons-logging', module: 'commons-logging' |

그 외 라이브러리 설정은 Java 플러그인을 사용하면서 라이브러리를 설정할 때와 유사합니다. 라이브러리 전체 설정 방법은 https://github.com/sjyun/gradlespring/blob/master/build.gradle에서 확인할 수 있습니다.

**스프링 JavaConfig 설정**

**컨텍스트 초기화 설정**

|  |
| --- |
| web.xml을 사용하지 않으므로 web.xml처럼 서블릿 정보를 등록하고 config클래스들을 서블릿 컨텍스트에 등록하도록 설정하는 클래스가 필요합니다. 이 클래스는  AbstractAnnotationConfigDispatcherServletInitializer 클래스를 상속하여 작성하거나 WebApplicationInitializer 인터페이스를 상속받아 작성할 수 있습니다. 이 예제에서는 AbstractAnnotationConfigDispatch  erServletInitializer로 클래스를 작성합니다.서블릿 매핑 설정에서 URL을 기본 root 경로로 지정하고 스프링 MVC의 설정을관리할 WebConfig 클래스를 getServletConfigClasses 메서드에 등록합니다. |
| package info.thecodinglive.config;  import org.springframework.web.servlet.support.AbstractAnnotationConfigDispat  cherServletInitializer;  public class WebInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {  @Override  protected Class<?>[] getServletConfigClasses() {  return new Class[]{WebConfig.class};  }  @Override  protected String[] getServletMappings() {  return new String[]{"/"};  }  } |

**WebConfig 클래스 작성**

|  |
| --- |
| 클래스가 설정을 위한 클래스임을 나타내기 위해서 configuration 어노테이션을 사용합니다. 그리고 WebConfig 설정을 기반으로 스프링 MVC 설정이 가능하게 하려면 enablewebmvc 설정을 추가합니다 |
| @EnableWebMvc  @Configuration  @ComponentScan(basePackages = {"info.thecodinglive.controller"} )  public class WebConfig extends WebMvcConfigurerAdapter{} |

**JSP 파일 액세스를 위한 WebCofig 클래스 설정**

|  |
| --- |
| WEB-INF 하위에 JSP 파일을 두고 접근할 수 있게 하려면 별도의 설정이 필요합니다. 이는 viewResolver를 사용하면 되는데, 이 예제에서는 UrlBasedViewResolver를 사용하겠습니다. |
| @Bean  public UrlBasedViewResolver setupViewResolver() {  UrlBasedViewResolver resolver = new UrlBasedViewResolver();  resolver.setPrefix("/WEB-INF/views/");  resolver.setSuffix(".jsp");  resolver.setViewClass(JstlView.class);  return resolver;  } |

**인덱스 페이지 컨트롤러 설정**

|  |
| --- |
| 사용자가 http://localhost:8080을 입력했을 때 home.jsp 파일을 보여주기위해 별도의 컨트롤러를 생성할 수도 있지만, 인덱스 페이지처럼 별도의 로직이필요없는 페이지를 매핑하는 경우에는 ViewControllerRegistry에 view 이름을 매핑합니다. |
| @Override  public void addViewControllers( org.springframework.web.servlet.config.  annotation.ViewControllerRegistry registry) {  registry.addViewController("/").setViewName("home");  } |
| 이제 사용자가 http://localhost:8080/을 입력하면 WEB-INF\views 하위에  있는 home.jsp 파일을 호출하게 됩니다. |

**DB 설정**

|  |
| --- |
| 이 예제에서는 별도의 설치 없이 실습할 수 있게 HSQLDB03를 사용합니다. DataSource로 JDBC 접속 정보를 설정하는데, Datasource의 setter 메서드를 이용해서 구성할 수 있습니다. |
| @Bean  public DataSource dataSource() {  DriverManagerDataSource dataSource = new DriverManagerDataSource();  dataSource.setDriverClassName(env.getRequiredProperty(PROPERTY\_NAME\_DATABASE\_DRIVER));  dataSource.setUrl(env.getRequiredProperty(PROPERTY\_NAME\_DATABASE\_URL));  dataSource.setUsername(env.getRequiredProperty(PROPERTY\_NAME\_DATABASE\_USERNAME));  dataSource.setPassword(env.getRequiredProperty(PROPERTY\_NAME\_DATABASE\_PASSWORD));  return dataSource;  } |
| HSQLDB은 메모리상에서 사용하므로 URL을 다음과 같이 설정합니다.  jdbc:hsqldb:mem:spring |

**하이버네이트 설정**

|  |
| --- |
| 자바 기반의 기술로 DB를 제어하려면 소스 레벨에서 DB에 접속할 수 있어야 하는데, RDBMS 벤더가 다양하므로 자바의 기본 API에서는 이를 제공하지 않고JDBC 스펙을 제공하여 각 벤더에서 스펙에 부합하는 드라이버를 제공합니다. 따라서 각 RDBMS에 맞는 JDBC 드라이버를 로드해야 합니다. 이 부분에 대한 처리가 바로 DataSource 설정입니다.SQL이 아니라 하이버네이트로 DB를 제어하려면 DB에 접속하기 위해서 JDBC  드라이버를 사용하는 것처럼 객체와 DB 매핑에 대한 설정이 필요합니다. 하이네이트에서는 다양한 유형의 RDBMS를 지원하기 위해 dialect를 사용합니다. 이 예제에서는  HSQLDB를 사용하기 위해 ‘org.hibernate.dialect.HSQLDialect’로 설정합니다. |
| hibernate.dialect=org.hibernate.dialect.HSQLDialect  hibernate.show\_sql=true  hibernate.format\_sql=false  hibernate.hbm2ddl.auto=create-drop  entitymanager.packages.to.scan=info.thecodinglive.model |

|  |
| --- |
| 나중에 MySQL 등의 다른 DB로 변경할 경우에도 dialect를 ‘org.hibernate.dialect.MySQL5InnoDBDialect’로 설정하고 DataSource 메서드에서MySQL에 맞는 JDBC 드라이버 정보를 입력하면 DB를 제어하는 로직 부분을  수정하지 않고 사용할 수 있습니다.DB 관련 프로그래밍을 할 때 부담되는 것 중의 하나는 테스트 데이터입니다. 이런 부담을 줄이기 위해서 hbm2ddl.auto 설정을 사용할 수 있습니다. 특히 예제처럼 실제 운영되는 서비스에서 사용되는 데이터가 아니라 새로운 기술 도입을 위한변경이나 테스트가 필요한 경우에 유용합니다. 예제에서는 create-drop을 사용했는데 create-drop을 적용하면 웹 애플리케이션이 시작될 때 엔티티Entity 매핑설정을 기반으로 하이버네이트의 세션 팩토리Session Factory가 DB 스키마를 생성한후에 세션 팩토리가 종료되면 스키마도 함께 초기화됩니다.DataSource와 dialect 설정이 끝나면 다음으로 SessionFactory를 설정합니다. SessionFactory는 세션을 관리하고 생성하는 객체로, JPA의 EntityManagerFactory와 대응되는 객체입니다. 세션을 설정하는 이유는 하이버네이트를 통해 쿼리를 실행하기 위해서입니다. 이는 JDBC를 사용할 때 Connection객체를 얻은 후에 쿼리를 실행한 것처럼 하이버네이트 안에서 쿼리를 실행할 수있는 쿼리 실행자라고 생각하면 됩니다.Table 객체로 매핑할 모델 클래스들이 있는 패키지를 setPackagesToScan 메서드로 설정합니다. |
| @Bean  public LocalSessionFactoryBean sessionFactory() {  LocalSessionFactoryBean sessionFactoryBean = new  LocalSessionFactoryBean();  sessionFactoryBean.setDataSource(dataSource());  sessionFactoryBean.setPackagesToScan(env.  getRequiredProperty(PROPERTY\_NAME\_ENTITYMANAGER\_PACKAGES\_TO\_SCAN));  sessionFactoryBean.setHibernateProperties(hibProperties());  return sessionFactoryBean;  } |

|  |
| --- |
| 세션 설정을 끝나면 관리 대상이 될 엔티티를 설정합니다. 엔티티를 설정할 때 클래스명과 테이블명이 다를 경우에는 Table 어노테이션으로 설정할 수 있습니다 |
| @Entity  @Table(name="teams")  public class Team {  @Id  @GeneratedValue  private Integer id;  private String teamName;  private Integer rating; |
| 엔티티 설정까지는 하이버네이트를 사용하기 위한 준비 단계라고 할 수 있습니다. 이제 하이버네이트를 사용해서 DB를 제어하도록 DAO 인터페이스와 Service클래스를 작성하겠습니다. |

**Repository 레이어 구성**

|  |
| --- |
| TeamDAO 인터페이스에서는 Team 인스턴스를 전달받아 추가하거나 업데이트하고 목록을 반환하기 위한 메서드를 정의합니다. |
| package info.thecodinglive.repository;  import info.thecodinglive.model.Team;  import java.util.List;  public interface TeamDao {  public void addTeam(Team team);  public void updateTeam(Team team);  public Team getTeam(int id);  public void deleteTeam(int id);  public List<Team> getTeams();  } |
| DAO 인터페이스를 기반으로 앞에서 설정한 sessionFactory로 실제 DB에 제어를 가하는 DAO 구현체를 만듭니다. TeamDao를 상속받은 후에 Repository 어노테이션으로 이 클래스가 DB 접근에 사용되는 클래스임을 명시하고 하이버네이트세션을 사용할 수 있도록 sessionFactory의 인스턴스를 스프링의 DI를 이용해서 주입합니다. |
| package info.thecodinglive.repository;  import info.thecodinglive.model.Team;  import org.hibernate.Session;  import org.hibernate.SessionFactory;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Repository;  import java.util.List;  @Repository  public class TeamDaoImpl implements TeamDao{  } |
|  |

|  |
| --- |
| sessionFactory에서 세션을 얻을 때 openSession으로 session을 얻을 수 있  는데, 이 경우에는 openSession 메서드를 호출할 때마다 새로운 Session 객체  를 얻게 됩니다. 그리고 DB 조작이 끝난 후에는 반드시 세션을 닫아주어야 합니  다. 예제의 경우에는 웹 프로젝트로 스프링에 Transaction 설정을 위임하여 사  용하므로 getCurrentSession 메서드로 현재 context에 바인딩된 Session 객  체를 얻습니다. |
| @Autowired  private SessionFactory sessionFactory;  private Session getCurrentSession() {  return sessionFactory.getCurrentSession();  } |

**팀 추가**

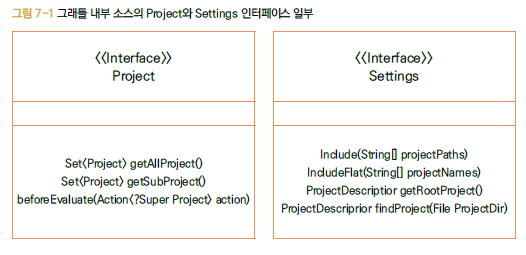
|  |
| --- |
| 하이버네이트의 세션을 얻은 후에는 세션을 통해서 데이터를 조작할 수 있습니다. 데이터 추가는 save 메서드로 할 수 있습니다. |
| public void addTeam(Team team) {  getCurrentSession().save(team);  } |
| 입력한 데이터는 get 메서드로 조회합니다. TeamId 필드를 DB에서 시퀀스값으  로 자동 생성하게 설정했으므로 TeamId 값과 같은 행에 속하는 데이터를 얻을 수  있습니다. |
| public Team getTeam(int id) {  Team team = (Team) getCurrentSession().get(Team.class, id);  return team;  } |

**전체 목록 반환하기**

|  |
| --- |
| 한 행의 데이터를 가져오는 메서드를 확장해서 전체 목록에 대한 데이터를 얻을수 있습니다. 쿼리로 데이터를 조회할 때 ‘SELECT \* FROM table명‘ 과 같이 테이블에 해당하는 전체 데이터를 조회하려는 경우에는 컬럼을 명시하지 않고 생략해서 사용할 수 있는 것처럼 하이버네이트의 HQL04에서는 쿼리를 축약해서 메모리에 로드된 DB 데이터를 from 절로 가져올 수 있습니다. 그래서 ‘from Team’을‘Select \* From Team’과 동일한 의미로 사용하여 Team 타입의 데이터 목록을리스트 형태로 반환합니다. |
| @SuppressWarnings("unchecked")  public List<Team> getTeams() {  return getCurrentSession().createQuery("from Team").list();  } |

**멀티 프로젝트 구성**

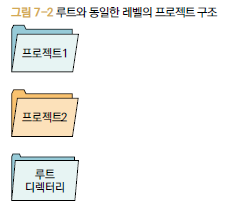
그래들에서는 settings.gradle 파일을 이용하여 간편하게 멀티 프로젝트를 구성할 수 있습니다



settings.gradle 파일과 매칭되는 Settings 인터페이스는 멀티 프로젝트 구성에 필요한 include, includeFlat과 같은 메서드를 가지고 있습니다. 멀티 프로젝트를 구성하려면 settings.gradle 파일에 include로 프로젝트를 추가하면 됩니다. 프로젝트 인터페이스에서는 getAllProject와 getSubProject로 전체프로젝트에 적용할 내용과 부분 프로젝트에 적용할 내용을 구분하여 적용할 수 있습니다. 그리고 멀티 프로젝트를 구성하여 평가하는 부분에 대해 추가 작업을 할수 있습니다.

**동일 레벨의 멀티 프로젝트**

동일레벨의 프로젝트를 구성해 보겠습니다. 여기서 동일레벨이란 프로젝트의 뎁스Depth가 같은 것을 의미합니다.



[그림 7-2]와 같이 프로젝트1과 프로젝트2, 루트 프로젝트가 최상위 디렉터리 안(같은 뎁스)에 있는 경우가 있습니다. 프로젝트를 새롭게 구성할 경우에는 계층 구조를 만들지만, 기존에 사용하던 프로젝트들을 참조하여 전체 프로젝트를 재구성하거나 설정을 공유할 목적으로 프로젝트를 새롭게 만드는 경우에는 이 구조를 사용합니다.

[그림 7-2]와 같은 형태일 때 settings.gradle 파일에서 includeflat을 설정합니다.

|  |
| --- |
| includeFlat 'proj1', 'proj2' |
|  |
| 한 디렉터리 안에 myroot와 proj1, proj2가 함께 있습니다. 이때 proj1과proj2의 settings.gradle을 제거하고 myroot 프로젝트의 settings.gradle에서includeFlat으로 프로젝트를 추가합니다. |
|  |

**동일 레벨의 멀티 프로젝트 Task**

단일 프로젝트에서는 단순히 Task를 만들어서 실행했지만, 멀티 프로젝트로 구성하면 프로젝트마다 개별적으로 설정을 지정하거나 Task를 실행할 수 있습니다.

|  |
| --- |
| 간단하게 프로젝트 이름을 출력하는 Task를 만들어서 각 프로젝트의 이름을출력해 보겠습니다. 전체 프로젝트에 대한 설정은 allProject로 하고 하위 프로젝트에 대한 설정은 subProject로 합니다. allProject의 DSL안에 Task를 작  성하면 모든 프로젝트에 적용됩니다. |
| [ 멀티 프로젝트에서 프로젝트 이름 출력 Task ]  allprojects {  task printProjName << {  println "${project.name}"  }  } |
|  |

**프로젝트 평가하기**

많은 수의 서브 프로젝트를 사용하여 개발을 진행하다 보면 모듈마다 각각 필요한설정이 달라서 각 모듈의 빌드 파일을 작성해서 관리하기가 어려운 경우가 많습니다. 반대로 루트 프로젝트의 빌드 파일에 설정을 몰아서 작성해도 양이 너무 많아져서 관리가 힘듭니다. 이런 경우에는 afterEvaluate로 프로젝트를 평가할 수있습니다. 프로젝트를 평가한다는 것은 검증한다는 것과 같습니다. 플러그인의 적용 여부에 따라 Task 적용이 달라지는 경우가 많으므로 예제에서는 proj1에 그래들의 Java 플러그인을 적용하고 이를 확인해 보겠습니다.

proj1의 build.gradle 파일에 Java 플러그인을 적용하는 내용을 추가합니다.

|  |
| --- |
| apply plugin: 'java' |
| 루트 프로젝트에서 afterEvalute로 플러그인 적용 여부를 확인하는 스크립트를  작성합니다. afterEvalute는 그래들에서 프로젝트의 인스턴스가 초기화될 때  자동으로 실행되고, 하위 프로젝트 중에서 Java 플러그인이 적용된 프로젝트를  출력합니다. |
| subprojects{  subproject -> afterEvaluate {  if(subproject.plugins.hasPlugin("java")) {  println "java 적용된 project name: $subproject.name"  }  }  } |
| Proj1에 Java 플러그인이 적용되어 있는 것을 확인할 수 있습니다 |
| 이처럼 afterEvalute를 이용하여 검증 로직을 추가해 두면 다수의 하위 프로젝트를 사용할 때 유용합니다. |

**계층 레벨의 멀티 프로젝트**

이번에는 상하 관계가 있는 멀티 프로젝트를 설정하겠습니다. 기존에는 모두 동일한 레벨이었고, 여기에서는 최상위 프로젝트에서 의존관계를 지정하는 멀티 프로젝트를 설정합니다.

|  |  |
| --- | --- |
|  | 계층 관계이므로 settings.gradle 파일을 루트 디렉터리에서 작성하는데, 이디렉터리가 멀티 프로젝트의 루트 디렉터리가 됩니다. 일반적으로 디렉터리명과 프로젝트명은 동일하지만 프로젝트명과 디렉터리명을 다르게 하고 싶으면rootProject.name을 변경하면 됩니다. |
| include 'myroot', 'proj1', 'proj2'  rootProject.name = 'gradle-hirmulti' |
|  | |

|  |
| --- |
| myroot 상위 디렉터리인 gradle-hirmulti 디렉터리에 settings.gradle 파일을 놓고 설정합니다. 이전에 생성한 myroot 디렉터리 안의 settings.gradle 파일을 제거하거나 이동시킨 후에 projects 명령으로 프로젝트 관계 설정을 확인합니다. |
| 최상위 디렉터리인 gradle-hirmulti가 루트 프로젝트로 인식되고 나머지 프로젝트들이 등록되었습니다. |

**스프링 MVC 프로젝트를 멀티 프로젝트로 변경하기**

**루트 프로젝트의 공통 설정**

루트 프로젝트에서는 서브 프로젝트에 공통으로 적용할 사항을 설정하고 소스 디렉터리는 포함하지 않습니다. 이렇게 소스 디렉터리를 포함하지 않는 경우에는Base 플러그인을 사용하여 설정할 수 있습니다.

루트 프로젝트에 적용할 내용을 다음과 같습니다.

● 공통으로 사용할 플러그인

● 자바 버전 설정

● 공통 라이브러리 설정

루트 프로젝트에서 서브 프로젝트들에 적용할 내용들은 subprojects DSL을 사용합니다. subProjects 하위에 작성된 내용들은 settings.gradle 파일에 명시된 서브 프로젝트들에 적용됩니다.

루트 프로젝트의 전체 build.gradle 파일 내용은 다음과 같습니다.

|  |
| --- |
| apply plugin: 'base'  subprojects {  apply plugin: 'java'  apply plugin: 'maven'  apply plugin: 'idea'  apply plugin: 'eclipse'  apply plugin: 'eclipse-wtp'  apply plugin: 'groovy'  sourceCompatibility = 1.8  targetCompatibility = 1.8  /\*  encoding  \*/  [compileJava, compileTestJava]\*.options\*.encoding = 'UTF-8'  ext {  javaVersion = '1.8'  springVersion = '3.2.9.RELEASE'  slf4jVersion = '1.7.5'  logbackVersion = '1.0.13'  }  repositories {  mavenLocal()  jcenter()  }  List loggers = [  "org.slf4j:slf4j-api:${slf4jVersion}",  "org.slf4j:jcl-over-slf4j:${slf4jVersion}",  "org.slf4j:log4j-over-slf4j:${slf4jVersion}",  "org.slf4j:jul-to-slf4j:${slf4jVersion}",  "ch.qos.logback:logback-core:${logbackVersion}",  "ch.qos.logback:logback-classic:${logbackVersion}"  ]  configurations {  all.collect {  configuration -> configuration.exclude group: 'commons-logging',  module: 'commons-logging'  configuration.exclude group: 'log4j', module: 'log4j'  configuration.exclude group: 'org.slf4j', module: 'slf4j-log4j12'  configuration.exclude group: 'org.slf4j', module: 'slf4j-jcl'  configuration.exclude group: 'org.slf4j', module: 'slf4j-jdk14'  }  }  dependencies {  compile localGroovy()  compile loggers  compile group: 'mysql', name: 'mysql-connector-java', version:'5.1.21'  compile group: 'org.hsqldb', name: 'hsqldb', version:'2.3.2'  compile group: 'commons-dbcp', name: 'commons-dbcp', version:'1.4'  compile group: 'javassist', name: 'javassist', version:'3.12.1.GA'  compile group: 'org.springframework', name: 'spring-tx',  version:'3.2.9.RELEASE'  compile group: 'org.springframework', name: 'spring-jdbc',  version:'3.2.9.RELEASE'  compile "org.springframework:spring-context:${springVersion}"  compile "org.springframework:spring-aspects:${springVersion}"  compile "org.springframework:spring-beans:${springVersion}"  compile group: 'cglib', name: 'cglib', version:'2.2.2'  testCompile group: 'junit', name: 'junit', version:'4.7'  testCompile group: 'org.springframework', name: 'spring-test',  version:'3.2.9.RELEASE'  testCompile group: 'org.springframework', name: 'spring-test-mvc',  version:'1.0.0.M1'  }  } |