

스프링 R2DBC를 활용한 작은 코틀린 SQL DSL 개발기

모빌리티42

오현석

발표자 소개

모빌리티 42 CTO

20여년 개발자

기술 번역가/저자

Programming in Scala

코틀린 인 액션

아토믹 코틀린

코틀린 함수형 프로그래밍

고성능 파이썬

순수 함수형 데이터 구조

배워서 바로 쓰는 스프링 프레임워크

한권으로 읽는 컴퓨터 구조와 프로그래밍

리액트 혹은 인 액션

등등등



DSL

- 특정 목적에 맞춰 개발된 언어
 - 간단한 문법
 - 제한적인 단어
 - 명확한 의미
- 내부 DSL
 - 어떤 프로그래밍 언어 안에서 DSL을 제공하는 것
 - 호스트 언어의 기능을 100% 활용
 - DSL의 장점도 100% 제공
- 어떤 의미에서는 모든 코딩은 DSL 개발이라 할 수 있음



내부 DSL 예 - Exposed 쿼리 DSL(코틀린)

```
object StarWarsFilms : Table() {
    val id: Column<Int> = integer("id").autoIncrement()
    val sequelId: Column<Int> = integer("sequel_id").uniqueIndex()
    val name: Column<String> = varchar("name", 50)
    val director: Column<String> = varchar("director", 50)
    override val primaryKey = PrimaryKey(id, name = "PK_StarWarsFilms_Id")
}

fun select() {
    val directors = StarWarsFilms
        .select(StarWarsFilms.director)
        .where { StarWarsFilms.sequelId less 5 }
        .withDistinct().map {
            it[StarWarsFilms.director]
        }
}
```

내부 DSL 예 - AssertJ 테스트 매처(자바)

- 테스트 결과 검증을 위한 단언문 라이브러리도 내부 DSL

```
class SimpleAssertionsExample {  
    @Test  
    fun a_few_simple_assertions() {  
        assertThat("The Lord of the Rings").isNotNull()  
            .startsWith("The")  
            .contains("Lord")  
            .endsWith("Rings");  
    }  
}
```

모든 시스템은 프로그래머가 그
시스템을 기술하기 위해 만든
DSL에 의해 구축된다

- 밥 마틴

쿼리 DSL 키워드 나가기

언어 기능 중 일부는 언어를 자라나게 하는 작업을 돕기 위해 설계되어야 한다

- 가이 스틸(Guy Steele Jr.)

쿼리 언어의 최종 목표

```
class UserTable: Entity() {  
    var id by long  
    var name by string  
    var email by string  
}  
  
fun main() {  
    val sql = from(UserTable()) { table ->  
        select {table ->  
            table{::id}  
        }  
        where { table ->  
            (table{::id} eq 10) and  
            (20.0 eq (table{::name} - count(table{::name})))  
        }  
    }  
  
    println(sql.toString())  
}
```


쿼리 DSL 언어의 최종 목표

```
class UserTable: Entity() {  
    var id by long  
    var name by string  
    var email by string  
}
```

타입을 정적으로 지정할 수 있는 엔티티 정의

```
fun main() {  
    val sql = from(UserTable()) { table ->  
        select { table ->  
            table{::id}  
        }  
        where { table ->  
            (table{::id} eq 10) and  
            (20.0 eq (table{::name} - count(table{::name})))  
        }  
    }  
  
    println(sql.toString())  
}
```

쿼리 언어의 최종 목표

```
class UserTable: Entity() {  
    var id by long  
    var name by string  
    var email by string  
}
```

from / select / where 사용 위치 제한

```
fun main() {  
    val sql = from(UserTable()) { table ->  
        select { table ->  
            table{::id}  
        }  
        where { table ->  
            (table{::id} eq 10) and  
            (20.0 eq (table{::name} - count(table{::name})))  
        }  
    }  
  
    println(sql.toString())  
}
```

쿼리 언어의 최종 목표

```
class UserTable: Entity() {  
    var id by long  
    var name by string  
    var email by string  
}
```

```
fun main() {  
    val sql = from(UserTable()) { table ->  
        select {table ->  
            table{::id}  
        }  
        where { table ->  
            (table{::id} eq 10) and  
            (20.0 eq (table{::name} - count(table{::name})))  
        }  
    }  
  
    println(sql.toString())  
}
```


테이블의 필드를 실수없이 지정할 수 있게 함

쿼리 언어의 최종 목표

```
class UserTable: Entity() {
    var id by long
    var name by string
    var email by string
}

fun main() {
    val sql = from(UserTable()) { table ->
        select {table ->
            table{::id}
        }
        where { table ->
            (table{::id} eq 10) and
            (20.0 eq (table{::name} - count(table{::name})))
        }
    }

    println(sql.toString())
}
```



SQL을 위한 표현식 제공

최종 목적에 이르는 여정과 코틀린 기능

- 타입을 정적으로 지원할 수 있는 엔티티 정의
 - 프로퍼티 위임 - `getValue`, `setValue`
 - 프로퍼티 위임 프로바이더 - `provideDelegate`
- `from`, `select`, `where` 사용 위치 제한
 - 수신객체 지정 람다, `DSLMarker`
- 테이블 필드를 실수 없이 지정할 수 있게 하기
 - 프로퍼티 참조, 값 클래스를 통한 타입
- SQL을 위한 표현식 제공
 - 연산자 오버로딩, 봉인된 클래스

현재 상황

- 타입을 정적으로 지원할 수 있는 엔티티 정의
 - 프로퍼티 위임 - `getValue`, `setValue`
 - 프로퍼티 위임 프로바이더 - `provideDelegate`
- `from`, `select`, `where` 사용 위치 제한
 - 수신객체 지정 람다, `DSLMarker`
- 테이블 필드를 실수 없이 지정할 수 있게 하기
 - 프로퍼티 참조, 값 클래스를 통한 타입
- SQL을 위한 표현식 제공
 - 연산자 오버로딩, 봉인된 클래스

엔티티 정의

- 엔티티 정의에 필요한 요소
 - 필드 이름을 알 수 있어야 함
 - 필드 타입을 간편하게 지정할 수 있어야 함
 - 런타임에 객체에서 값을 설정하고 읽을 수 있어야 함
- 이를 위해 필요한 기능
 - 클래스에 정의된 프로퍼티의 이름과 타입을 알 수 있어야 함
 - 게터와 세터를 가로챌 수 있어야 함
- 일반적인 구현 방법:
 - 리플렉션
 - 바이트코드 생성 라이브러리 사용
- 코틀린의 해법: 프로퍼티 위임과 프로퍼티 위임 프로바이더

프로퍼티 위임

- 위임객체:
 - getValue와 setValue 연산자 함수가 정의된 함수
 - 프로퍼티 뒤에 "by 위임객체" 형태로 사용함

```
object LongDelegate {  
    operator fun getValue(thisRef: Entity, property: KProperty<*>): Long =  
        thisRef.values[property.name] as Long  
  
    operator fun setValue(thisRef: Entity, property: KProperty<*>, value: Long) {  
        thisRef.values[property.name] = value  
    }  
}  
  
class Entity {  
    val values = mutableMapOf<String, Any>()  
  
    var id by LongDelegate  
}
```

장점과 한계

- 장점

- 런타임에 리플렉션 사용 없음
- 정적으로 타입이 계산됨
- 관습에 의한 코딩 (coding by convention)
 - `getValue`, `setValue` 연산자만 제공하면 어떤 객체든 위임객체가 될 수 있음

- 한계

- 프로퍼티 값을 설정하거나 읽어야만 프로퍼티 정체를 파악할 수 있음
 - 위임 객체의 `getValue`, `setValue`는 각각의 프로퍼티 값을 읽거나 써야만 호출됨
 - 하지만 우리는...
 - 프로퍼티가 직접 사용되기 전에 그 이름과 타입을 알고 싶음

- 코틀린에서

- 처음에는 프로퍼티 위임만 제공함
- 1.4부터 프로퍼티 위임 프로바이더로 문제점 해결

프로퍼티 위임 프로바이더

- 프로퍼티 위임을 담당할 객체를 생성하는 객체
 - 관습에 의한 코딩
 - 적절한 타입의 연산자 함수로 **provideDelegate**만 제공하면 됨
 - 이때 **provideDelegate**가 반환하는 객체는 **프로퍼티 위임 객체**여야 함
- 객체 생성시 **by**문에 의해 프로퍼티에 위임이 설정될 때마다 호출됨
 - **getValue**와 **setValue**보다 더 앞선 시점
 - 선언되는 프로퍼티의 이름과 속성을 객체 생성 시점에 알 수 있음

엔티티 구현의 몇가지 트릭

- 엔티티가 프로퍼티 역할을 하게 함
 - 자신의 모든 필드를 `getValue`, `setValue`에서 사용할 수 있음
 - 필드 자신의 값 뿐 아니라 엔티티 객체의 상태를 참조해 더 적합한 동작 제공
- 엔티티들이 상속할 기반 클래스를 정의해 공통 필드 제공
- 프로퍼티 위임 프로바이더를 타입별 싱글톤 객체로 제공
 - 필드의 타입을 싱글톤을 통해 알 수 있음
 - 이들의 공통 구현을 상위 타입으로 정의할 수 있음

엔티티 구현: 프로퍼티 프로바이더

```
typealias PropProvider<T>
    = PropertyDelegateProvider<Entity,ReadWriteProperty<Entity,T>>

abstract class Prop<T> {
    val delegator = PropProvider<T> { entity, prop ->
        println("provideDelegate: making ${prop.name} type ${prop.returnType}")
        ((entity?._props) ?:
            mutableMapOf<String,KProperty<*>>()).also{
            entity._props=it}[prop.name] = prop
        }
        entity as ReadWriteProperty<Entity,T>
    }
}

object LongProp:Prop<Long>()
object StringProp:Prop<String>()
```

프로퍼티 프로바이더 설명(1 of 3)

- 제네릭 타입 별명 PropProvider
 - **PropertyDelegateProvider**를 짧게 부름
 - PropertyDelegateProvider는 함수 인터페이스(자바의 SAM)
 - Entity는 우리가 만들 엔티티 클래스
 - 모든 엔티티 클래스는 Entity 클래스의 자손
 - **ReadWriteProperty**는 getValue와 setValue를 제공하는 타입을 표현하기 위한 코틀린 인터페이스

```
typealias PropProvider<T>  
    = PropertyDelegateProvider<Entity, ReadWriteProperty<Entity, T>>
```

프로퍼티 프로바이더 설명(2 of 3)

- Entity안에 `_props`라는 가변 맵 프로퍼티가 있음
 - 해당 프로퍼티를 필요하면 초기화하고, 프로퍼티 이름에 대해 전달받은 `prop`을 설정
- `delegator`는 `PropProvider<T>` 타입 객체임
 - 전달받은 **entity**를 그대로 반환함

```
abstract class Prop<T> {  
    val delegator = PropProvider<T> { entity, prop ->  
        println("provideDelegate: making ${prop.name} type ${prop.returnType}")  
        ((entity?._props) ?:  
            mutableMapOf<String, KProperty<*>>()).also{  
                entity._props=it}[prop.name] = prop  
            }  
        entity as ReadWriteProperty<Entity, T>  
    }  
}
```

프로퍼티 프로바이더 설명(3 of 3)

- 프로퍼티 딜리게이트는 별도의 객체일 필요가 없음
 - 타입별로 다른 싱글톤 객체를 돌려줌
 - 런타임에 리플렉션을 쓰지 않고 Prop<T> 객체와 동등성 비교를 통해 필드 타입을 알아낼 수 있음

```
object LongProp: Prop<Long>()  
object StringProp: Prop<String>()
```


엔티티 구현: 엔티티 기반 클래스(1 of 3)

- 추상 클래스로 정의
 - ReadWriteProperty<Entity,Any>를 상속
- 프로퍼티 관련 데이터 저장에 필요한 맵 정의
 - _values : 실제 값을 저장할 맵
 - _props : KProperty 타입의 값을 저장할 맵
(_field만 사용한다면 필요 없음)
 - _fields : Prop 타입의 값을 저장할 맵

```
abstract class Entity: ReadWriteProperty<Entity,Any> {  
    var _values: MutableMap<String,Any>? = mutableMapOf()  
    var _props: MutableMap<String,KProperty<*>>? = mutableMapOf()  
    var _fields: MutableMap<String,Prop<*>>? = mutableMapOf()
```

엔티티 구현 엔티티 기반 클래스(2 of 3)

```
override operator fun getValue(thisRef: Entity, property: KProperty<*>): Any {  
    println("getValue: ${property.name}")  
    return _values?.get(property.name)  
        ?: throw IllegalArgumentException("property does not exist: ${property.name}")  
}  
  
override operator fun setValue(thisRef: Entity, property: KProperty<*>, value: Any) {  
    println("setValue: ${property.name} to $value")  
    _values?.let {  
        it[property.name] = value  
    } ?: throw IllegalArgumentException("property does not exist: ${property.name}")  
}
```

엔티티 구현 엔티티 기반 클래스(3 of 3)

```
companion object {  
    val long get() = long()  
    val string get() = string()  
  
    inline fun long():PropProvider<Long> = LongProp.delegator  
    inline fun string():PropProvider<String> = StringProp.delegator  
  
    fun Entity.toJson():String = _values?.entries?  
        .joinToString(",",",","{","}") {(key,value) ->  
            if(value is String) "'$key':'$value'" else "'$key':$value" } ?: "{}"  
}
```

엔티티 사용

```
class MyEntity: Entity() {  
    var id by long  
    var name by string  
}  
  
fun main() {  
    val x = MyEntity(); x.id = 10L; x.name = "Hyunsok Oh"  
    println(x.toJson())  
}
```

- jetbrains://idea/navigate/reference?project=DslBasics&path=com/enshahar/KotlinDslSupport/propertyDelegateProvider/PropertyDelegateProvider.kt

현재 상황

- ~~타입을 정적으로 지원할 수 있는 엔티티 정의~~
 - ~~프로퍼티 위임 - `getValue`, `setValue`~~
 - ~~프로퍼티 위임 프로바이더 - `provideDelegate`~~
- **from, select, where 사용 위치 제한**
 - **수신객체 지정 람다, DSLMarker**
- ~~테이블 필드를 실수 없이 지정할 수 있게 하기~~
 - ~~프로퍼티 참조, 값 클래스를 통한 타입~~
- ~~SQL을 위한 표현식 제공~~
 - ~~연산자 오버로딩, 봉인된 클래스~~

SQL 질의의 시작 - From

- 2가지 방식
 - `from(entity) { ... }`
 - Entity를 파라미터로 받는 함수
 - `Entity.from { ... }`
 - Entity에 대한 확장 함수
- `{ ... }` 안에서는:
 - from에서 지정한 엔티티들에 대해서 `select`, `where` 등을 수행할 수 있어야 함
 - `{ ... }` 안에서 사용할 수 있는 함수와 프로퍼티를 우리(DSL 라이브러리 개발자)가 원하는 대로 제어할 수 있어야 함

어떤 문맥 안에서 사용할 수 있는 단어 공급,제한

- OOP의 클래스 내부 문맥에서는 이미 우리도 잘 써먹고 있음
 - `this`를 생략
 - 클래스 멤버 프로퍼티와 멤버 함수를 자유롭게 쓸 수 있
 - 단어 공급자와 사용자가 **같은 클래스 정의 안에** 위치함...
- DSL의 경우
 - 단어 공급자(DSL라이브러리)와 사용자가 **다름**
 - 사용자에게 특정 문맥을 **주입해줄 방법이 필요**
- **수신객체 지정 람다**(lambda with a receiver)
 - 람다가 암시적으로 다른 객체를 전달받음 - 객체지향 **수신 객체 전달 방식** 처럼 보임
 - 수신객체 지정 람다 안에서는 **`this`를 생략**
 - 람다를 작성하는 사람은 `this` 안의 단어(함수와 프로퍼티)를 간편하게 쓸 수 있음 - 문맥 / 단어 공급

쿼리 시작: from 함수

- **from(Entity) { ... }** 스타일의 쿼리 DSL 정의하기로 결정
 - 반환 값: 쿼리를 만들 수 있는 어떤 객체. **Query<Entity>**
 - 람다 { ... } 는?
 - **DSL을 위한 문맥**을 제공
 - QueryDsl을 수신객체로 지정한 람다를 사용
 - QueryDsl은 엔티티에 접근할 수 있어야 함
 - 쿼리 정의시 필드 정보를 사용해야 함
 - 2가지 방식
 - QueryDsl 이 엔티티를 **프로퍼티**로 저장
 - 수신객체 지정 람다가 엔티티를 **파라미터**로 받음

쿼리 DSL의 뼈대

```
fun <Table:Entity> from(e:Table,  
    block:QueryDslContext<Table>.(Table)->Unit):Query<Table>  
  
class QueryDslContext<Table:Entity>(val data:Query<Table>) {  
    // 여기서 DSL에 사용할 다양한 단어를 제공  
}  
  
class Query<Table:Entity>(val table:Table) {  
    // 이 안에 쿼리를 만드는데 필요한 자세한 데이터를 저장  
}
```

구현하기 : from 함수

- Query객체를 만들고
- QueryDslContext에게 그 객체를 넘기면서 block을 실행하고
- 1에서 만든 Query 객체를 반환

```
fun <Table:Entity> from(table:Table, block:QueryDslContext<Table>.(Table)->Unit) =  
    Query<Table>(table).also {  
        QueryDslContext<Table>(it).block(table)  
    }
```


프로젝션 DSL 추가

- QueryDslContext 안에 정의
- 수신객체 지정 람다 block은 프로젝션을 위한 문맥 안에서 실행됨

```
class QueryDslContext<Table:Entity>(val data:Query<Table>) {  
    fun select(block: ProjectionContext<Table>.(Table)->Unit) {  
        ProjectionContext<Table>(data).block(data.table)  
    }  
}  
  
class ProjectionContext<Table:Entity>(val data:Query<Table>) {  
}
```

필터링 DSL 추가

- select와 비슷

```
class QueryDslContext<Table:Entity>(val data:Query<Table>) {  
    fun where(block: FilteringContext<Table>.(Table)->Unit) {  
        FilteringContext<Table>(data).block(data.table)  
    }  
}  
  
class FilteringContext<Table:Entity>(val data:Query<Table>) {  
}
```

쿼리 기본 골격 완성

- 원하대로 select/where 사용할 수 있음

```
val sql = from(UserTable()) { table ->
  select {
  }
  where {
  }
}
```

문제점...

- select나 where 안에서 select나 where 사용 가능
 - @DslMarker를 사용해 해결
- 같은 수준에서 select, where를 여러 번 호출 가능
- 두번째 문제는 코틀린 내부 DSL에서는 해결방법 없음 😞

```
val sql = from(UserTable()) { table ->
    select {
        select { }
        where { }
    }
    select { }
    where {}
    where {}
}
```

DSL에서 this의 영역 제한 - @DslMarker

- 애너테이션 클래스를 만들면서 @DslMarker를 지정

```
@DslMarker  
annotation class QueryDslMarker
```

- DSL 문맥 클래스 앞에 우리가 정의한 마커 클래스 애너테이션을 추가

```
@QueryDslMarker  
class QueryDslContext<Table:Entity>(val data:Query<Table>)  
  
@QueryDslMarker  
class ProjectionContext<Table:Entity>(val data:Query<Table>)  
  
@QueryDslMarker  
class FilteringContext<Table:Entity>(val data:Query<Table>)
```

DSL 마커의 효과: 암시적 this 제한

- 같은 마커가 붙은 클래스들 안에서는 **바깥쪽** 영역의 `this`를 암시적으로 사용할 수 **없음**
- **`this@`**형태를 사용하면 **우회 가능**

```
val sql : Query<UserTable> = from(UserTable()) { this: QueryDslContext<UserTable> table : UserTable ->
    select { this: ProjectionContext<UserTable> it: UserTable
        | where {}
        | this@from.select{}
    }
    where { this: FilteringContext<UserTable> it: UserTable
        | select {}
    }
}
```


현재 상황

- ~~타입을 정적으로 지원할 수 있는 엔티티 정의~~
 - ~~프로퍼티 위임 - `getValue, setValue`~~
 - ~~프로퍼티 위임 프로바이더 - `provideDelegate`~~
- ~~`from, select, where` 사용 위치 제한~~
 - ~~수신객체 지정 람다, `DSLMarker`~~
- **테이블 필드를 실수 없이 지정할 수 있게 하기**
 - **프로퍼티 참조, 값 클래스를 통한 타입**
- SQL을 위한 표현식 제공
 - 연산자 오버로딩, 봉인된 클래스

컬럼 이름을 안전하게 적을 수 있게 하기

- 대안 3가지
 - 문자열 : "User"
 - 손쉬운 방식
 - 타이핑 오류 나기 쉬움
 - 클래스 프로퍼티 참조
 - 타이핑 번거로움(UserTable::를 항상 쳐야 함)
 - IDE가 :: 다음에 필드 이름을 표시해줌
 - 다른 방법? 나중에

```
from(UserTable()) { table ->
  select {
    "id"
    UserTable::id
  }
  ...
}
```

컬럼을 프로젝트션에 넣기

- 이름을 적기만 해서는 아무 효과가 없음: 연산이 필요함
 - 단항 연산자로 넣기 - 코틀린 연산자 함수 사용
 - 멤버 호출 연산(.)을 사용 - 멤버 함수나 확장 함수로 정의
 - 중위함수 표기 사용하기 - infix 가 붙은 멤버 함수나 확장 함수

```
from(UserTable()) { table ->
  select {
    +UserTable::id
    UserTable::id.put()
    UserTable::id `as` "alias"
  }
  ...
}
```

컬럼 이름을 안전하게 적고 사용하기 해주는 다른 방법

- 엔티티의 **invoke** 연산자 함수를 정의
 - 이때 invoke의 파라미터 타입을 **KProperty**를 생성하는 함수로 제한
- 쓰는 쪽에서는 {::id} 처럼 프로퍼티를 생성하는 람다를 통해 필드 사용
- 타이핑도 비교적 단순하고, 잘못된 이름을 입력할 수 없음

```
@QueryDslMarker
class ProjectionContext<Table:Entity>(val data:Query<Table>) {
    ...
    inline operator fun Table.invoke(block:Table.()->KProperty<*>) {
        data.addProjection(this.block())
    }
    ...
}
```

```
val sql = from(UserTable()) { t ->
    select { table ->
        table{::id}
    }
}
```

값 클래스를 사용한 타입 한정

```
// 블록의 시그니처를 Alias<Table>을 받게 변경
fun <Table:Entity> from(table:Table,
                        block:QueryDslContext<Table>.(Alias<Table>)->Unit) =

@QueryDslMarker
class ProjectionContext<Table:Entity>(val data:Query<Table>) {
    ...
    // Alias<Table>에 대한 invoke함수 정의
    inline operator fun Alias<Table>.invoke(block:Table.()->KProperty<*>) {
        data.addProjection(this.v.block())
    }
}

// 값 클래스로 부가비용을 최소화
@JvmInline
value class Alias<V:Entity>(val v:V)
```

```
// 사용방법은 같음
val sql = from(UserTable()) { t ->
    select { table->
        table{::id}
    }
}
```

데이터 저장

- 오버로딩을 통해 여러 타입 지원 가능 - 같은 이름의 함수로 가독성 향상
- 디폴트 파라미터 지정을 통해 필요한 디폴트 값 쉽게 제공
- 함수 **파라미터 계산 순서**는 앞에서 뒤로 이뤄짐
 - 디폴트 값을 계산할 때 자기보다 왼쪽 파라미터들을 쓸 수 있음

```
class Query<Table:Entity>(val table:Alias<Table>) {  
    var projections: MutableList<Projection>? = null  
    inline fun <reified T> addProjection(prop: KProperty1<Table,T>,  
                                        alias: String=prop.name) {  
        ...  
    }  
    inline fun <reified T> addProjection(prop: KProperty<T>, alias: String=prop.name) {  
        (projections ?: (mutableListOf<Projection>())).add(Projection(prop.name,alias))  
    }  
}
```

현재 상황

- ~~타입을 정적으로 지원할 수 있는 엔티티 정의~~
 - ~~프로퍼티 위임 - `getValue, setValue`~~
 - ~~프로퍼티 위임 프로바이더 - `provideDelegate`~~
- ~~`from, select, where` 사용 위치 제한~~
 - ~~수신객체 지정 람다, `DSLMarker`~~
- ~~테이블 필드를 실수 없이 지정할 수 있게 하기~~
 - ~~프로퍼티 참조, 값 클래스를 통한 타입~~
- SQL을 위한 표현식 제공
 - 연산자 오버로딩, 봉인된 클래스

Where 조건식 설계

- 조건식을 어떻게 표현하게 할 것인가?
 - **완전한 SQL 식 구현**
 - **봉인된 클래스**로 SQL 식을 저장할 클래스 계층 설계
 - **코틀린 연산자** 등을 활용해 SQL 식 구현
- 봉인된 클래스 계층을 얼마나 자세히 구분할 것인가?
 - 타입을 자세히 구분
 - 사용자의 실수를 **정적으로** 줄일 수 있음
 - 구현해야 할 함수 가짓수가 늘어남
 - DSL 작성자는 편해지지만 DSL 라이브러리 개발자는 힘들
- 한두가지 클래스로 통치기
 - 실수를 **동적(실행 시점)으로** 잡아내야 함

예: 최대한 간단한 타입 사용 - 타입정의

```
sealed interface SqlExpression
data class Const<T:Number>(val v:T): SqlExpression
data class Field(val name:String): SqlExpression
data class Bop(val op: SqlBop, val e1: SqlExpression,
               val e2: SqlExpression): SqlExpression
data class Fun(val fop: SqlFun,
               val e1: List<SqlExpression>): SqlExpression
```

```
enum class SqlBop {
    AND, OR, PLUS, MINUS, EQ
}
enum class SqlFun(val arity:Int) {
    COUNT(1),
    AVERAGE(1)
}
```

예: 최대한 간단한 타입 사용 - 연산구현

@QueryDslMarker

```
class FilteringContext<Table:Entity>(val data:Query<Table>) {
```

```
...
```

```
operator fun SqlExpression.plus(other: SqlExpression) =  
    Bop(SqlBop.PLUS, this, other)
```

```
operator fun SqlExpression.minus(other: SqlExpression) =  
    Bop(SqlBop.MINUS, this, other)
```

```
infix fun SqlExpression.and(other: SqlExpression) = Bop(SqlBop.AND, this, other)
```

```
infix fun SqlExpression.or(other: SqlExpression) = Bop(SqlBop.OR, this, other)
```

```
infix fun SqlExpression.eq(other: SqlExpression) = Bop(SqlBop.EQ, this, other)
```

```
fun count(v: SqlExpression) = Fun(SqlFun.COUNT, listOf(v))
```

```
fun <T:Number> T.sql(): SqlExpression = Const(this)
```

```
fun AliasField.sql(): SqlExpression = Field(this.name)
```

```
}
```

필터링 코드 - 상수와 필드 사용 불편

```
val sql = from(UserTable()) { table ->
  select {table ->
    table{::id}
  }
  where { table ->
    (table{::id}.sql() eq 10.sql()) and
    (20.0.sql() eq (table{::name}.sql()
      - count(table{::name}.sql())))
  }
}
```


논리, 산술을 구분한 클래스 계층구조

```
sealed interface SqlExpression
sealed interface SqlLogicExpression: SqlExpression
sealed interface SqlArithExpression: SqlExpression
```

```
enum class LConst: SqlLogicExpression { TRUE, FALSE }
data class LField(val name:String): SqlLogicExpression
data class LBop(val op: SqlLogicBop, val e1: SqlLogicExpression,
               val e2: SqlLogicExpression): SqlLogicExpression
data class LFun(val fop: SqlFun, val e1: List<SqlExpression>): SqlLogicExpression
```

```
data class Eq(val e1: SqlExpression, val e2: SqlExpression): SqlLogicExpression
```

```
data class AConst<T:Number>(val v:T): SqlArithExpression
data class AField(val name:String): SqlArithExpression
data class ABop(val op: SqlArithBop, val e1: SqlArithExpression,
               val e2: SqlArithExpression): SqlArithExpression
data class AFun(val fop: SqlFun, val e1: List<SqlExpression>): SqlArithExpression
```

```
enum class SqlLogicBop { AND, OR }
enum class SqlArithBop { PLUS, MINUS }
enum class SqlFun(val arity:Int) { COUNT(arity: 1), AVERAGE(arity: 1) }
```

더 세분화한 경우의 연산자 정의

- 여러 타입 사이의 다양한 연산자를 **오버로딩**으로 작성해야 함
- 모든 경우를 따지기 힘들
 - 순서를 잘 정해서
 - 빠뜨리지 말고 ...
- DSL 라이브러리 작성자가 고생하면 사용자는 덜 고생을...


```

operator fun <T:Number> AliasField.plus(other: T) : ABop = ABop(SqlArithBop.PLUS, AField(this.name), AConst(other))
operator fun <T:Number> T.plus(other: SqlArithExpression) : ABop = ABop(SqlArithBop.PLUS, AConst(v: this), other)
operator fun AliasField.plus(other: SqlArithExpression) : ABop = ABop(SqlArithBop.PLUS, AField(this.name), other)
operator fun AliasField.minus(other: SqlArithExpression) : ABop = ABop(SqlArithBop.MINUS, AField(this.name), other)
operator fun SqlArithExpression.plus(other: AliasField) : ABop = ABop(SqlArithBop.PLUS, e1: this, AField(other.name))
operator fun SqlArithExpression.minus(other: AliasField) : ABop = ABop(SqlArithBop.MINUS, e1: this, AField(other.name))
operator fun SqlArithExpression.plus(other: SqlArithExpression) : ABop = ABop(SqlArithBop.PLUS, e1: this, other)
operator fun SqlArithExpression.minus(other: SqlArithExpression) : ABop = ABop(SqlArithBop.MINUS, e1: this, other)

infix fun AliasField.and(other: SqlLogicExpression) : LBop = LBop(SqlLogicBop.AND, LField(this.name), other)
infix fun AliasField.or(other: SqlLogicExpression) : LBop = LBop(SqlLogicBop.OR, LField(this.name), other)
infix fun SqlLogicExpression.and(other: AliasField) : LBop = LBop(SqlLogicBop.AND, e1: this, LField(other.name))
infix fun SqlLogicExpression.or(other: AliasField) : LBop = LBop(SqlLogicBop.OR, e1: this, LField(other.name))
infix fun SqlLogicExpression.and(other: SqlLogicExpression) : LBop = LBop(SqlLogicBop.AND, e1: this, other)
infix fun SqlLogicExpression.or(other: SqlLogicExpression) : LBop = LBop(SqlLogicBop.OR, e1: this, other)

infix fun <T:Number> SqlExpression.eq(other: T) : Eq = Eq(e1: this, AConst(other))
infix fun <T:Number> AliasField.eq(other: T) : Eq = Eq(AField(this.name), AConst(other))
infix fun <T:Number> T.eq(other: SqlArithExpression) : Eq = Eq(AConst(v: this), other)
infix fun AliasField.eq(other: SqlLogicExpression) : Eq = Eq(LField(this.name), other)
infix fun AliasField.eq(other: SqlArithExpression) : Eq = Eq(AField(this.name), other)
infix fun SqlLogicExpression.eq(other: AliasField) : Eq = Eq(e1: this, LField(other.name))
infix fun SqlArithExpression.eq(other: AliasField) : Eq = Eq(e1: this, AField(other.name))
infix fun SqlExpression.eq(other: SqlExpression) : Eq = Eq(e1: this, other)

fun count(v: SqlArithExpression) : AFun = AFun(SqlFun.COUNT, listOf(v))
fun count(v: AliasField) : AFun = AFun(SqlFun.COUNT, listOf(AField(v.name)))

```


더 세분화한 경우의 코드

```
val sql = from(UserTable()) { table ->
  select {table ->
    table{::id}
  }
  where { table ->
    (table{::id} eq 10) and
    (20.0 eq (table{::name} - count(table{::name})))
  }
}
```

sql 생성

- Data로부터 sql 생성 테이블 이름 필요함
- from 함수 변경

```
inline fun <reified Table:Entity> from(table:Table,  
    block:QueryDslContext<Table>.(Alias<Table>)->Unit) =  
    Query<Table>(<Table::class.simpleName  
        ?: throw IllegalStateException("class no name"), Alias(table)).also {  
        QueryDslContext<Table>(it).block(Alias(table))  
    }
```

SQL 생성

- 각 케이스 클래스의 toString을 오버라이드하거나 별도의 멤버 함수로 정의

```
override fun toString():String {  
    val pString = projections?.let {  
        it.joinToString(",", prefix = "SELECT ", postfix = "\n") {  
            "${it.fieldName} ${it.alias}"  
        }  
    } ?: "SELECT *\n"  
  
    val from = "FROM $tableName\n"  
  
    val wString = filterExpr?.let {  
        "WHERE $it"  
    } ?: ""  
  
    return pString + from + wString  
}
```

우리가 다룬 DSL 작성 기술들

- 수신 객체 지정 람다를 받는 고차 함수를 통해 DSL을 시작
 - DSL 작성을 위한 문맥 객체를 수신객체 지정 람다의 수신 객체로 사용함
- `@DSLMarker`가 지정된 애너테이션을 통해 DSL 영역 겹침 해결
- 연산자 구현
 - 연산자 함수를 통해 적절한 연산자 의미 구현
 - 중위 함수(`inline function`)를 사용해 괄호 사용 줄이기
 - 함수 오버로딩과 디폴트 파라미터를 활용해 편의성 높이기
- 봉인된 클래스를 사용해 표현식의 표현력 높이기
- (결과 출력) `toString`과 `joinToString`을 사용한 문자열 생성

실전: 스프링 데이터 R2DBC와 쿼리 DSL 합치기

Spring Data R2DBC(Reactive Relational Database Connectivity)



스프링 데이터 R2DBC

- 반응형 데이터베이스 드라이버를 사용한 스프링 데이터 프로젝트
 - 함수형 API 제공
 - `R2dbcEntityTemplate` 제공
 - `ReactiveCrudRepository`와 `CoroutineCrudRepository` 제공
 - 다양한 데이터베이스 지원
 - H2, MariaDB, SQL Server, MySQL, Postgres 등
 - `DatabaseClient`를 통해 데이터베이스 연동
 - 트랜잭션 처리: `@Transactional`, `TransactionalOperator`
 - 현재 버전 3.2.5
 - 참고:
<https://docs.spring.io/spring-data/relational/reference/>

데이터베이스 설정 추가 - 주의점

- URL에서 풀을 활성화시켰다면 yml이나 properties 파일에서 pool로 풀을 활성화시키지 말아야 함(둘을 동시에 쓰면 안됨)
- 서버 타임존 설정이 안 먹을 수 있음
 - 예외: java.time.DateTimeException: **Invalid ID for region-based ZoneId**, invalid format ...
 - 해결: **@Configuration** 이 붙은 클래스에 다음 빈 추가

```
@Bean
fun mysqlCustomizer(): ConnectionFactoryOptionsBuilderCustomizer =
    ConnectionFactoryOptionsBuilderCustomizer { builder ->
        builder.option(MySqlConnectionFactoryProvider.SERVER_ZONE_ID,
ZoneId.of("UTC"))
    }
```


TransactionalOperator로 트랜잭션 처리

- **주의점:**
 - TransactionalOperator가 지켜보는 Flow의 흐름을 끊으면 안됨

바람직한 예:

```
Tx.transaction {  
    flow {  
        Flow 사용한 디비 연산들  
        emit 결과  
    }  
}
```

잘못된 예:

```
Tx.transaction {  
    // (1)  
    Flow 사용한 디비연산+collect  
    // (2)  
    flow {  
        emit 결과  
    }  
}
```

- **이유: (1)과 (2) 사이에서 예외 발생시 플로우를 통해 예외가 전달되지 않음**

실전 DSL로 가는 멀고 험한 길

악마는 디테일에 숨겨져 있다

DatabaseClient와 연동하기 위한 추가 개발

- 파라미터 바인딩 지원
 - Alias와 비슷한 Param 값 클래스 추가
 - from, select, where의 블록이 Param타입의 값을 파라미터로 받게 람다 타입 변경
 - 여러 파라미터를 받도록 쿼리 데이터 타입을 제네릭화
- DatabaseClient연동
 - 쿼리 객체 생성 후 DatabaseClient와 연동해 결과 Flow를 만들어내는 추가 DSL제공
- 이 과정에서 정적 타입을 전달하고, 인라이닝을 활용

결과는... (아직 개발중)

- SQL DSL 관련 클래스
 - 51개 코틀린 파일, 3800라인
- 엔티티 관련 클래스
 - 46개 코틀린 파일, 1200라인
- R2DBC 트랜잭션 처리 DSL 관련 파일
 - 7개 코틀린 파일, 500라인

쿼리와 R2DBC DatabaseClient 연결 예

```
val query = Server.selectTo(Rs) { server, rs ->
    val f1 = _Region { ::_region_rowid } join server { ::_region_rowid }
    val f2 = _Ec2 { ::_ec2_rowid } join server { ::_ec2_rowid }
    val f3 = Member { ::member_rowid } join server { ::member_rowid }
    select {
        server { ::server_rowid } put rs { ::serverRowid }
        server { ::title } put rs { ::title }
        f1 { ::_region_rowid } put rs { ::regionRowid }
        server { ::url } put rs { ::url }
        server { ::version } put rs { ::version }
        server { ::_ec2_rowid } put rs { ::ec2Rowid }
        server { ::deletable } put rs { ::deletable }
        f3 { ::name } put rs { ::register }
        server { ::deletedate } put rs { ::deletedate }
        server { ::regdate } put rs { ::regdate }
    }
}
```

쿼리와 R2DBC DatabaseClient 연결 예

```
query.r2dbc(client).toList().let {  
    it.map { rs ->  
        MdlServer().also {  
            it.serverRowid = "${rs.serverRowid}"  
            it.title = rs.title  
            it.url = rs.url  
            it.regionCat = rs.regionCat  
            it.version = rs.version  
            it.ec2Cat = rs.ec2Cat  
            it.deletable = rs.deletable.toInt() == 1  
            it.register = rs.register  
            it.deletedate = eUtc.of(rs.deletedate.toString())!!  
            it.regdate = eUtc.of(rs.regdate.toString()) ?: eUtc.now()  
        }  
    }  
}
```

**사용자들이 자신의 언어를 키울 수 있는 도구를
언어에 넣을 필요가 있다.**

나는 여러분이, 나처럼, 언어를 키워봤으면 한다.

- 가이 스틸(Guy Steele Jr.)

감사합니다