Utils.kt

```kotlin
package com.testpassword

import com.google.gson.GsonBuilder
import com.google.gson.JsonDeserializer
import com.google.gson.JsonPrimitive
import com.google.gson.JsonSerializer
import com.testpassword.models.Generable
import org.jetbrains.exposed.sql.*
import org.json.JSONObject
import java.time.LocalDate

val PARSER = GsonBuilder()
    .registerTypeAdapter(
        LocalDate::class.java,
        JsonSerializer<LocalDate> { src, _, _ -> if (src == null) null else JsonPrimitive(src.toString
()) })
    .registerTypeAdapter(
        LocalDate::class.java,
        JsonDeserializer<LocalDate> { json, _, _ -> if (json == null) null else LocalDate.parse(json.
asString.split("T").first()) })
    .create()

class ReferenceEntityError(e: Table): Error() {
    override val message = "Reference entity table ${e.tableName} does not contains such records"
}

fun fillDb(sizes: Map<Generable, Int>) = sizes.forEach { it.key.generateAndInsert(it.value) }

fun dropRecordsWithIds(s: String, entityTable: Table) {
    val ids = JSONObject(s).getJSONArray("droppedIds")
    ids.forEachIndexed { i, _ ->
        entityTable.deleteWhere { (entityTable.primaryKey?.columns?.get(0) as Column<Int>) eq ids.getInt
(i) }
    }
}

fun getRecordsWithIds(s: String, entityTable: Table): Query {
    val raw = JSONObject(s).getJSONArray("selectedIds")
    val ids = mutableListOf<Int>()
    raw.forEachIndexed { i, _ -> ids.add(raw.getInt(i)) }
    return if (ids.isEmpty()) entityTable.selectAll() else
        entityTable.select {
            entityTable.primaryKey?.columns?.get(0) as Column<Int> inList ids
        }
}

fun explodeJsonForModel(modelIdField: String, jsonStr: String): Pair<Int, Map<String, Any>> {
    val s = JSONObject(jsonStr)
    val id = s.getInt(modelIdField)
    s.remove(modelIdField)
    val fields = mutableMapOf<String, Any>()
    s.keys().forEach { fields[it] = s.get(it) }
    return id to fields
}
```

Application.kt

```kotlin
package com.testpassword

import com.testpassword.models.*
import com.testpassword.routes.god
import com.zaxxer.hikari.HikariConfig
import com.zaxxer.hikari.HikariDataSource
import io.ktor.application.*
import io.ktor.features.*
import io.ktor.http.*
import io.ktor.routing.*
import org.jetbrains.exposed.sql.*

fun main(args: Array<String>): Unit = io.ktor.server.netty.EngineMain.main(args)

@Suppress("unused") // Referenced in application.conf
fun Application.initDB() =
    Database.connect(HikariDataSource(HikariConfig(environment.config.property("ktor.hikariconfig").getString())))

fun Application.allowCORS() {
    install(CORS) {
        anyHost()
        allowCredentials = true
        allowNonSimpleContentTypes = true
        methods.addAll(HttpMethod.DefaultMethods)
    }
}

//https://stefangaller.at/app-development/kotlin/ktor-rest-api-exposed/
fun Application.module() {
    initDB()
    allowCORS()
    routing {
        route("god") { god() }
        route("base") { base() }
        route("campaign") { campaign() }
        route("employee") { employee() }
        route("equipment") { equipment() }
        route("medicalCard") { medicalCard() }
        route("mission") { mission() }
        route("mre") { mre() }
        route("position") { position() }
        route("transport") { transport() }
        route("weapon") { weapon() }
    }
}
```

MRE.kt

```kotlin
package com.testpassword.models

import com.testpassword.*
import io.ktor.application.*
import io.ktor.http.*
import io.ktor.request.*
import io.ktor.response.*
import io.ktor.routing.*
import org.jetbrains.exposed.sql.*
import org.jetbrains.exposed.sql.transactions.transaction

object MRETable: Table("mre"), Generable {

    val mre_id = integer("mre_id").autoIncrement().primaryKey()
    val breakfast = text("breakfast")
    val lunch = text("lunch")
    val dinner = text("dinner")
    val food_additives = text("food_additives").nullable()
    val kkal = integer("kkal").check { it greaterEq 3000 }
    val proteins = integer("proteins").check { it greater 0 }
    val fats = integer("fats").check { it greater 0 }
    val carbohydrate = integer("carbohydrate").check { it greater 0 }

    override fun generateAndInsert(n: Int) {
        (1..n).forEach {
            MRETable.insert {
                it[breakfast] = F.food().dish()
                it[lunch] = F.food().dish()
                it[dinner] = F.food().dish()
                it[food_additives] = generateSequence { F.food().ingredient() }.take(3).joinToString(separator = " ")
                it[kkal] = F.number().numberBetween(3000, 4500)
                it[proteins] = F.number().numberBetween(1, 400)
                it[fats] = F.number().numberBetween(1, 400)
                it[carbohydrate] = F.number().numberBetween(1, 400)
            }
        }
    }
}

fun ResultRow.toMRE() = MRE(
    this[MRETable.mre_id],
    this[MRETable.breakfast],
    this[MRETable.lunch],
    this[MRETable.dinner],
    this[MRETable.food_additives],
    this[MRETable.kkal],
    this[MRETable.proteins],
    this[MRETable.fats],
    this[MRETable.carbohydrate])

data class MRE(val mreId: Int?,
               val breakfast: String,
               val lunch: String,
               val dinner: String,
               val foodAdditives: String?,
               val kkal: Int,
               val proteins: Int,
               val fats: Int,
               val carbohydrate: Int)

fun Route.mre() {

    get {
        val (t, s) = try {
            call.parameters["ids"]!!.let {
                transaction {
                    PARSER.toJson(getRecordsWithIds(it, MRETable).map { it.toMRE() }) to HttpStatusCode.OK
                }
            }
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

```

MRE.kt

```
    put {
        val (t, s) = try {
            val raw = call.receiveText()
            val (id, f) = explodeJsonForModel("missId", raw)
            transaction {
                MRETable.update({ MRETable.mre_id eq id }) { m ->
                    f["breakfast"]?.let { m[breakfast] = it as String }
                    f["lunch"]?.let { m[lunch] = it as String }
                    f["dinner"]?.let { m[dinner] = it as String }
                    f["foodAdditives"]?.let { m[food_additives] = it as String }
                    f["kkal"]?.let { m[kkal] = it as Int }
                    f["proteins"]?.let { m[proteins] = it as Int }
                    f["fats"]?.let { m[fats] = it as Int }
                    f["carbohydrate"]?.let { m[carbohydrate] = it as Int }
                }
            }
            "$raw updated)" to HttpStatusCode.OK
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    post {
        val (t, s) = try {
            val raw = call.receiveText()
            val m = PARSER.fromJson(raw, MRE::class.java)
            transaction {
                MRETable.insert {
                    it[breakfast] = m.breakfast
                    it[lunch] = m.lunch
                    it[dinner] = m.dinner
                    it[food_additives] = m.foodAdditives
                    it[kkal] = m.kkal
                    it[proteins] = m.proteins
                    it[fats] = m.fats
                    it[carbohydrate] = m.carbohydrate
                }
            }
            "$raw added)" to HttpStatusCode.Created
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    delete {
        val (t, s) = try {
            val droppedIds = call.receiveText()
            transaction { dropRecordsWithIds(droppedIds, MRETable) }
            "MREs with $droppedIds deleted)" to HttpStatusCode.OK
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }
}

```

Base.kt

```kotlin
package com.testpassword.models

import com.testpassword.*
import io.ktor.application.*
import io.ktor.http.*
import io.ktor.request.*
import io.ktor.response.*
import io.ktor.routing.*
import org.jetbrains.exposed.sql.*
import org.jetbrains.exposed.sql.transactions.transaction

object BaseTable: Table("base"), Generable {

    val base_id = integer("base_id").autoIncrement().primaryKey()
    val location = text("location")
    val status = text("status")

    override fun generateAndInsert(n: Int) {
        val statuses = setOf("working", "closed", "destroyed", "abandoned", "captured", "for_sale")
        (1..n).forEach {
            BaseTable.insert {
                it[location] = F.address().city()
                it[status] = statuses.random()
            }
        }
    }
}

fun ResultRow.toBase() = Base(
    this[BaseTable.base_id],
    this[BaseTable.location],
    this[BaseTable.status])

data class Base(val baseId: Int?,
                val location: String,
                val status: String)

fun Route.base() {

    get {
        val (t, s) = try {
            call.parameters["ids"]!!.let {
                transaction {
                    PARSER.toJson(getRecordsWithIds(it, BaseTable).map { it.toBase() }) to
HttpStatusCode.OK
                }
            }
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    put {
        val (t, s) = try {
            val raw = call.receiveText()
            val (id, f) = explodeJsonForModel("baseId", raw)
            transaction {
                BaseTable.update({ BaseTable.base_id eq id }) { b ->
                    f["location"]?.let { b[location] = it as String }
                    f["status"]?.let { b[status] = it as String }
                }
            }
            "$raw updated)" to HttpStatusCode.OK
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    post {
        val (t, s) = try {
            val raw = call.receiveText()
            val b = PARSER.fromJson(raw, Base::class.java)
            transaction {
                BaseTable.insert {
                    it[location] = b.location
                    it[status] = b.status
                }
```

Base.kt

```
            }
            "$raw added)" to HttpStatusCode.Created
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    delete {
        val (t, s) = try {
            val droppedIds = call.receiveText()
            transaction { dropRecordsWithIds(droppedIds, BaseTable) }
            "Bases with $droppedIds deleted)" to HttpStatusCode.OK
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }
}
```

Weapon.kt

```kt
package com.testpassword.models

import com.testpassword.PARSER
import com.testpassword.dropRecordsWithIds
import com.testpassword.explodeJsonForModel
import com.testpassword.getRecordsWithIds
import io.ktor.application.*
import io.ktor.http.*
import io.ktor.request.*
import io.ktor.response.*
import io.ktor.routing.*
import org.jetbrains.exposed.sql.*
import org.jetbrains.exposed.sql.transactions.transaction
import org.json.JSONObject
import java.io.File

object WeaponTable: Table("weapon"), Generable {

    val weapon_id = integer("weapon_id").autoIncrement().primaryKey()
    val name = text("name")
    val type = text("type")
    val caliber = float("caliber").check { it greater 0.0 }.nullable()
    val rate_of_fire = integer("rate_of_fire").check { it greater 0 }.nullable()
    val sighting_range_m = integer("sighting_range_m").check { it greater 0 }.nullable()

    override fun generateAndInsert(n: Int) {
        val rawData = File("resources/static/guns.json").readLines().joinToString(separator = "")
        val jsonBody = JSONObject(rawData).getJSONArray("weapons")
        jsonBody.forEachIndexed { i, _ ->
            val w = jsonBody.getJSONObject(i)
            WeaponTable.insert {
                it[name] = w.getString("name")
                it[type] = w.getString("type")
                it[caliber] = w.getFloat("caliber")
                it[rate_of_fire] = w.getInt("rate_of_fire")
                it[sighting_range_m] = w.getInt("sighting_range_m")
            }
        }
    }
}

fun ResultRow.toWeapon() = Weapon(
    this[WeaponTable.weapon_id],
    this[WeaponTable.name],
    this[WeaponTable.type],
    this[WeaponTable.caliber]?.toDouble(),
    this[WeaponTable.rate_of_fire],
    this[WeaponTable.sighting_range_m])

data class Weapon(val weaponId: Int?,
                  val name: String,
                  val type: String,
                  val caliber: Double?,
                  val rateOfFire: Int?,
                  val sightingRangeM: Int?)

fun Route.weapon() {

    get {
        val (t, s) = try {
            call.parameters["ids"]!!.let {
                transaction {
                    PARSER.toJson(getRecordsWithIds(it, WeaponTable).map { it.toWeapon() }) to
HttpStatusCode.OK
                }
            }
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    put {
        val (t, s) = try {
            val raw = call.receiveText()
            val (id, f) = explodeJsonForModel("weaponId", raw)
            WeaponTable.update({ WeaponTable.weapon_id eq id }) { w ->
```

Weapon.kt

```kotlin
                f["name"]?.let { w[name] = it as String }
                f["type"]?.let { w[type] = it as String }
                f["caliber"]?.let { w[caliber] = it as Float }
                f["rateOfFire"]?.let { w[rate_of_fire] = it as Int }
                f["sighting_rangeM"]?.let { w[sighting_range_m] = it as Int }
            }
            "$raw updated)" to HttpStatusCode.OK
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    post {
        val (t, s) = try {
            val raw = call.receiveText()
            val w = PARSER.fromJson(raw, Weapon::class.java)
            transaction {
                WeaponTable.insert {
                    it[name] = w.name
                    it[type] = w.type
                    it[caliber] = w.caliber?.toFloat()
                    it[rate_of_fire] = w.rateOfFire
                    it[sighting_range_m] = w.sightingRangeM
                }
            }
            "$raw added)" to HttpStatusCode.Created
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    delete {
        val (t, s) = try {
            val droppedIds = call.receiveText()
            transaction { dropRecordsWithIds(droppedIds, WeaponTable) }
            "Weapons with ids $droppedIds deleted)" to HttpStatusCode.OK
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }
}
```

Mission.kt

```kt
package com.testpassword.models

import com.testpassword.*
import io.ktor.application.*
import io.ktor.http.*
import io.ktor.request.*
import io.ktor.response.*
import io.ktor.routing.*
import io.ktor.util.*
import org.jetbrains.exposed.sql.*
import org.jetbrains.exposed.sql.`java-time`.datetime
import org.jetbrains.exposed.sql.transactions.transaction
import java.time.LocalDate
import java.time.LocalDateTime
import java.time.ZoneId
import java.util.*
import kotlin.random.Random

object MissionTable: Table("mission"), Generable {

    val miss_id = integer("miss_id").autoIncrement().primaryKey()
    val camp_id = reference("camp_id", CampaignTable.camp_id, onDelete = ReferenceOption.CASCADE)
    val start_date_and_time = datetime("start_date_and_time").nullable()
    val end_date_and_time = datetime("end_date_and_time").nullable()
    val legal_status = bool("legal_status")
    val departure_location = text("departure_location").nullable()
    val arrival_location = text("arrival_location").nullable()
    val enemies = text("enemies").nullable()

    @InternalAPI
    override fun generateAndInsert(n: Int) {
        val campIds = CampaignTable.selectAll().map { it[CampaignTable.camp_id] }
        (1..n).forEach {
            val st = F.date().between(
                Date.from(LocalDate.of(2014, 3, 18).atStartOfDay(ZoneId.systemDefault()).toInstant()),
                Date()
            )
            val et = F.date().between(st, Date())
            MissionTable.insert {
                it[camp_id] = campIds.random()
                it[start_date_and_time] = st.toLocalDateTime()
                it[end_date_and_time] = et.toLocalDateTime()
                it[legal_status] = Random.nextBoolean()
                it[departure_location] = "${F.address().latitude()} ${F.address().longitude()}"
                it[arrival_location] = "${F.address().latitude()} ${F.address().longitude()}"
                it[enemies] = arrayOf(F.nation().nationality(), F.name().fullName()).random()
            }
        }
    }
}

fun ResultRow.toMission() = Mission(
    this[MissionTable.miss_id],
    this[MissionTable.camp_id],
    this[MissionTable.start_date_and_time],
    this[MissionTable.end_date_and_time],
    this[MissionTable.legal_status],
    this[MissionTable.departure_location],
    this[MissionTable.arrival_location],
    this[MissionTable.enemies])

data class Mission(val missId: Int?,
                   val campId: Int,
                   val startDateAndTime: LocalDateTime?,
                   val endDateAndTime: LocalDateTime?,
                   val legalStatus: Boolean,
                   val departureLocation: String?,
                   val arrivalLocation: String?,
                   val enemies: String?)

fun Route.mission() {

    get {
        val (t, s) = try {
            call.parameters["ids"]!!.let {
```

Mission.kt

```
                transaction {
                    PARSER.toJson(getRecordsWithIds(it, MissionTable).map { it.toMission() }) to HttpStatusCode.OK
                }
            }
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    put {
        val (t, s) = try {
            val raw = call.receiveText()
            val (id, f) = explodeJsonForModel("missId", raw)
            transaction {
                MissionTable.update({ MissionTable.miss_id eq id }) { m ->
                    f["camp_id"]?.let { m[camp_id] = it as Int }
                    f["start_date_and_time"]?.let { m[start_date_and_time] = LocalDateTime.parse(it as String) }
                    f["end_date_and_time"]?.let { m[end_date_and_time] = LocalDateTime.parse(it as String) }
                    f["legal_status"]?.let { m[legal_status] = it as Boolean }
                    f["departure_location"]?.let { m[departure_location] = it as String }
                    f["arrival_location"]?.let { m[arrival_location] = it as String }
                    f["enemies"]?.let { m[enemies] = it as String }
                }
            }
            "$raw updated)" to HttpStatusCode.OK
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    post {
        val (t, s) = try {
            val raw = call.receiveText()
            val m = PARSER.fromJson(raw, Mission::class.java)
            transaction {
                MissionTable.insert {
                    it[camp_id] = m.campId
                    it[start_date_and_time] = m.startDateAndTime
                    it[end_date_and_time] = m.endDateAndTime
                    it[legal_status] = m.legalStatus
                    it[departure_location] = m.departureLocation
                    it[arrival_location] = m.arrivalLocation
                    it[enemies] = m.enemies
                }
            }
            "$raw added)" to HttpStatusCode.Created
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    delete {
        val (t, s) = try {
            val droppedIds = call.receiveText()
            transaction { dropRecordsWithIds(droppedIds, MissionTable) }
            "Missions with $droppedIds deleted)" to HttpStatusCode.OK
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }
}
```

Campaign.kt

```kotlin
package com.testpassword.models

import com.testpassword.*
import io.ktor.application.*
import io.ktor.http.*
import io.ktor.request.*
import io.ktor.response.*
import io.ktor.routing.*
import org.jetbrains.exposed.sql.*
import org.jetbrains.exposed.sql.transactions.transaction
import java.math.BigDecimal
import kotlin.random.Random

object CampaignTable: Table("campaign"), Generable {

    val camp_id = integer("camp_id").autoIncrement().primaryKey()
    val name = text("name")
    val customer = text("customer")
    val earning = decimal("earning", 2, 11).check { it greaterEq 0.0 }
    val spending = decimal("spending", 2, 11).check { it greaterEq 0.0 }
    val execution_status = text("execution_status").nullable()

    override fun generateAndInsert(n: Int) {
        val statuses = setOf("completed", "in the process", "failed", "canceled")
        (1..n).forEach {
            CampaignTable.insert {
                it[name] = F.ancient().titan()
                it[customer] = F.name().fullName()
                it[earning] = F.number().randomDouble(2, 500000, 100000000).toBigDecimal()
                it[spending] = F.number().randomDouble(2, 0, 10000000).toBigDecimal()
                it[execution_status] = if (Random.nextInt(1, 100) >= 70) statuses.random() else statuses
.first()
            }
        }
    }
}

fun ResultRow.toCampaign() = Campaign(
    this[CampaignTable.camp_id],
    this[CampaignTable.name],
    this[CampaignTable.customer],
    this[CampaignTable.earning].toDouble(),
    this[CampaignTable.spending].toDouble(),
    this[CampaignTable.execution_status])

data class Campaign(val campId: Int?,
                    val name: String,
                    val customer: String,
                    val earning: Double,
                    val spending: Double,
                    val executionStatus: String?)

fun Route.campaign() {

    get {
        val (t, s) = try {
            call.parameters["ids"]!!.let {
                transaction {
                    PARSER.toJson(getRecordsWithIds(it, CampaignTable).map { it.toCampaign() }) to
HttpStatusCode.OK
                }
            }
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    put {
        val (t, s) = try {
            val raw = call.receiveText()
            val (id, f) = explodeJsonForModel("campId", raw)
            CampaignTable.update({ CampaignTable.camp_id eq id }) { c ->
                f["name"]?.let { c[name] = it as String }
                f["customer"]?.let { c[customer] = it as String }
                f["earning"]?.let { c[earning] = (it as Double).toBigDecimal() }
                f["spending"]?.let { c[spending] = (it as Double).toBigDecimal() }
```

Campaign.kt

```
                f["executionStatus"]?.let { c[execution_status] = it as String }
            }
            "$raw updated)" to HttpStatusCode.OK
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    post {
        val (t, s) = try {
            val raw = call.receiveText()
            val b = PARSER.fromJson(raw, Campaign::class.java)
            transaction {
                CampaignTable.insert {
                    it[name] = b.name
                    it[customer] = b.customer
                    it[earning] = b.earning.toBigDecimal()
                    it[spending] = b.spending.toBigDecimal()
                    it[execution_status] = b.executionStatus
                }
            }
            "$raw added)" to HttpStatusCode.Created
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    delete {
        val (t, s) = try {
            val droppedIds = call.receiveText()
            transaction { dropRecordsWithIds(droppedIds, CampaignTable) }
            "Campaigns with ids $droppedIds deleted)" to HttpStatusCode.OK
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }
}
```

Employee.kt

```kotlin
package com.testpassword.models

import com.google.gson.*
import com.testpassword.*
import io.ktor.application.*
import io.ktor.http.*
import io.ktor.request.*
import io.ktor.response.*
import io.ktor.routing.*
import io.ktor.util.*
import org.jetbrains.exposed.sql.*
import org.jetbrains.exposed.sql.`java-time`.CurrentDateTime
import org.jetbrains.exposed.sql.`java-time`.date
import org.jetbrains.exposed.sql.`java-time`.year
import org.jetbrains.exposed.sql.transactions.transaction
import java.time.LocalDate
import java.time.ZoneId
import java.util.*
import kotlin.random.Random


object EmployeeTable: Table("employee"), Generable {

    val emp_id = integer("emp_id").autoIncrement().primaryKey()
    val name = text("name")
    val surname = text("surname")
    val date_of_birth = date("date_of_birth").check { it less (CurrentDateTime().year() - 18) }
    val education = text("education").nullable()
    val hiring_date = date("hiring_date").defaultExpression(CurrentDateTime().date())
    val pos_id = reference("pos_id", PositionTable.pos_id, onDelete = ReferenceOption.RESTRICT)
    val is_married = bool("is_married")
    val base_id = reference("base_id", BaseTable.base_id, onDelete = ReferenceOption.SET_NULL).nullable
()

    @InternalAPI
    override fun generateAndInsert(n: Int) {
        val posIds = PositionTable.selectAll().limit(n).map { it[PositionTable.pos_id] }
        val baseIds = BaseTable.selectAll().map { it[BaseTable.base_id] }
        (1..n).forEach {
            val newbieId = EmployeeTable.insert {
                it[name] = F.name().firstName()
                it[surname] = F.name().lastName()
                it[date_of_birth] = F.date().birthday(18, 70).toLocalDateTime().toLocalDate()
                it[education] = F.educator().university()
                it[hiring_date] = F.date().between(
                    // здесь и далее - условной день основания компании.
                    Date.from(LocalDate.of(2014, 3, 18).atStartOfDay(ZoneId.systemDefault()).toInstant
()),
                    Date()
                ).toLocalDateTime().toLocalDate()
                it[pos_id] = posIds.random()
                it[is_married] = Random.nextBoolean()
                it[base_id] = baseIds.random()
            }[emp_id]
            MedicalCardTable.insert {
                it[emp_id] = newbieId
                it[height_cm] = Random.nextInt(150, 200)
                it[weight_kg] = Random.nextInt(40, 120)
                it[diseases] = generateSequence { F.medical().diseaseName() }
                    .take(Random.nextInt(0, 5))
                    .joinToString(separator = "_")
                it[blood] = F.name().bloodGroup()
                it[gender] = Random.nextBoolean()
            }
        }
    }
}

fun ResultRow.toEmployee() = Employee(
    this[EmployeeTable.emp_id],
    this[EmployeeTable.name],
    this[EmployeeTable.surname],
    this[EmployeeTable.date_of_birth],
    this[EmployeeTable.education],
    this[EmployeeTable.hiring_date],
```

Employee.kt

```
    this[EmployeeTable.pos_id],
    this[EmployeeTable.is_married],
    this[EmployeeTable.base_id])

data class Employee(val empId: Int?,
                    val name: String,
                    val surname: String,
                    val dateOfBirth: LocalDate,
                    val education: String?,
                    val hiringDate: LocalDate,
                    val posId: Int,
                    val isMarried: Boolean,
                    val baseId: Int?)

fun Route.employee() {

    get {
        val (t, s) = try {
            call.parameters["ids"]!!.let {
                transaction {
                    PARSER.toJson(getRecordsWithIds(it, EmployeeTable).map { it.toEmployee() }) to HttpStatusCode.OK
                }
            }
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    put {
        val (t, s) = try {
            val raw = call.receiveText()
            val (id, f) = explodeJsonForModel("empId", raw)
            transaction {
                EmployeeTable.update({ EmployeeTable.emp_id eq id }) { e ->
                    f["name"]?.let { e[name] = it as String }
                    f["surname"]?.let { e[surname] = it as String }
                    f["dateOfBirth"]?.let { e[date_of_birth] = LocalDate.parse(it as String) }
                    f["education"]?.let { e[education] = it as String }
                    f["hiringDate"]?.let { e[hiring_date] = LocalDate.parse(it as String) }
                    f["posId"]?.let { e[pos_id] = it as Int }
                    f["isMarried"]?.let { e[is_married] = it as Boolean }
                    f["baseId"]?.let { e[base_id] = it as Int }
                }
            }
            "$raw updated)" to HttpStatusCode.OK
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    post {
        val (t, s) = try {
            val raw = call.receiveText()
            val e = PARSER.fromJson(raw, Employee::class.java)
            transaction {
                EmployeeTable.insert {
                    it[name] = e.name
                    it[surname] = e.surname
                    it[date_of_birth] = e.dateOfBirth
                    it[education] = e.education
                    it[hiring_date] = e.hiringDate
                    it[pos_id] = e.posId
                    it[is_married] = e.isMarried
                    it[base_id] = e.baseId
                }
            }
            "$raw added)" to HttpStatusCode.Created
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    delete {
        val (t, s) = try {
            val droppedIds = call.receiveText()
            transaction { dropRecordsWithIds(droppedIds, EmployeeTable) }
            "Employee with ids $droppedIds deleted)" to HttpStatusCode.OK
```

Employee.kt

```
148         } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
149         call.respondText(text = t, status = s)
150     }
151 }
```

Position.kt

```kt
package com.testpassword.models

import com.testpassword.*
import io.ktor.application.*
import io.ktor.http.*
import io.ktor.request.*
import io.ktor.response.*
import io.ktor.routing.*
import org.jetbrains.exposed.sql.*
import org.jetbrains.exposed.sql.transactions.transaction
import java.math.BigDecimal
import kotlin.random.Random

object PositionTable: Table("Position"), Generable {

    val pos_id = integer("pos_id").autoIncrement().primaryKey()
    val name = text("name")
    val salary = decimal("salary", 2, 11).check { it greater 300.0 }
    val rank = text("rank").nullable()
    val equip_id = reference("equip_id", EquipmentTable.equip_id, onDelete = ReferenceOption.SET_NULL).nullable()
    val forces = postgresEnumeration<FORCES>("forces", "force").nullable()

    override fun generateAndInsert(n: Int) {
        val ranks = setOf("student_officer", "private_second_class", "private_first_class", "junior_sergeant",
            "sergeant", "senior_sergeant", "petty_officer", "ensign", "senior_ensign", "junior_lieutenant", "lieutenant",
            "senior_lieutenant", "captain", "major", "lieutenant_colonel", "colonel") //, "major_general", "lieutenant_general",
        //"colonel_general", "army_general", "marshal") исключим высшие звания из генерации
        val armyPositions = setOf("medic", "miner", "scout", "security", "driver", "torpedo_operator", "pilot",
            "mechanic", "engineer", "navigator", "orderly", "duty", "coach", "artilleryman", "gunner", "sniper", "spy")
        val equipIds = EquipmentTable.selectAll().limit(n).map { it[EquipmentTable.equip_id] }
        (1..n).forEach {
            PositionTable.insert {
                val empPos = if (Random.nextDouble(1.0, 100.0) >= 65) armyPositions.random() else F.job().position()
                it[name] = empPos
                it[salary] = F.number().randomDouble(2, 300, 1000000).toBigDecimal()
                if (empPos in armyPositions) it[rank] = ranks.random()
                it[equip_id] = equipIds.random()
                it[forces] = FORCES.values().random()
            }
        }
    }
}

fun ResultRow.toPosition() = Position(
    this[PositionTable.pos_id],
    this[PositionTable.name],
    this[PositionTable.salary],
    this[PositionTable.rank],
    this[PositionTable.equip_id],
    this[PositionTable.forces])

data class Position(val posId: Int?,
                    val name: String,
                    val salary: BigDecimal,
                    val rank: String?,
                    val equipId: Int?,
                    val forces: FORCES?)

fun Route.position() {

    get {
        val (t, s) = try {
            call.parameters["ids"]!!.let {
                transaction {
                    PARSER.toJson(getRecordsWithIds(it, PositionTable).map { it.toPosition() }) to HttpStatusCode.OK
                }
            }
```

Position.kt

```
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    put {
        val (t, s) = try {
            val raw = call.receiveText()
            val (id, f) = explodeJsonForModel("posId", raw)
            PositionTable.update({ PositionTable.pos_id eq id }) { p ->
                f["name"]?.let { p[name] = it as String }
                f["salary"]?.let { p[salary] = (it as Double).toBigDecimal() }
                f["rank"]?.let { p[rank] = it as String }
                f["equipId"]?.let { p[equip_id] = it as Int }
                f["forces"]?.let { p[forces] = FORCES.valueOf(it as String) }
            }
            "$raw updated)" to HttpStatusCode.OK
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    post {
        val (t, s) = try {
            val raw = call.receiveText()
            val p = PARSER.fromJson(raw, Position::class.java)
            transaction {
                PositionTable.insert {
                    it[name] = p.name
                    it[salary] = p.salary
                    it[rank] = p.rank
                    it[equip_id] = p.equipId
                    it[forces] = p.forces
                }
            }
            "$raw added)" to HttpStatusCode.Created
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    delete {
        val (t, s) = try {
            val droppedIds = call.receiveText()
            transaction { dropRecordsWithIds(droppedIds, PositionTable) }
            "Positions with $droppedIds deleted)" to HttpStatusCode.OK
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }
}
```

Equipment.kt

```kotlin
package com.testpassword.models

import com.testpassword.PARSER
import com.testpassword.dropRecordsWithIds
import com.testpassword.explodeJsonForModel
import com.testpassword.getRecordsWithIds
import io.ktor.application.*
import io.ktor.http.*
import io.ktor.request.*
import io.ktor.response.*
import io.ktor.routing.*
import org.jetbrains.exposed.sql.*
import org.jetbrains.exposed.sql.transactions.transaction

object EquipmentTable: Table("equipment"), Generable {

    val equip_id = integer("equip_id").autoIncrement().primaryKey()
    val camouflage = text("camouflage").nullable()
    val communication = text("communication").nullable()
    val intelligence = text("intelligence").nullable()
    val medical = text("medical").nullable()
    val mre_id = reference("mre_id", MRETable.mre_id, onDelete = ReferenceOption.RESTRICT)
    val extra = text("extra").nullable()

    override fun generateAndInsert(n: Int) {
        val camouflages = setOf("black", "khaki", "olive", "fleckerteppich", "strichtarn", "cce", "vegetata",
            "flora", "forest", "pixel", "woodland", "accupat", "desert", "city")
        val communications = setOf("radio_set", "map", "mobile_satellite", "signal_flares")
        val intelligences = setOf("binoculars", "drone", "radar", "thermal_visor")
        val medicals = setOf("bandage", "harness", "antibiotics", "alcohol", "scissors", "tweezers", "antiseptic",
            "ammonia", "nitroglycerine", "adrenalin")
        val mreIds = MRETable.selectAll().limit(n).map { it[MRETable.mre_id] }
        (1..n).forEach {
            EquipmentTable.insert {
                it[camouflage] = camouflages.random()
                it[communication] = communications.random()
                it[intelligence] = intelligences.random()
                it[medical] = medicals.random()
                it[mre_id] = mreIds.random()
            }
        }
    }
}

fun ResultRow.toEquipment() = Equipment(
    this[EquipmentTable.equip_id],
    this[EquipmentTable.camouflage],
    this[EquipmentTable.communication],
    this[EquipmentTable.intelligence],
    this[EquipmentTable.medical],
    this[EquipmentTable.mre_id],
    this[EquipmentTable.extra])

data class Equipment(val equipId: Int?,
                     val camouflage: String?,
                     val communication: String?,
                     val intelligence: String?,
                     val medical: String?,
                     val mreId: Int,
                     val extra: String?)

fun Route.equipment() {

    get {
        val (t, s) = try {
            call.parameters["ids"]!!.let {
                transaction {
                    PARSER.toJson(getRecordsWithIds(it, EquipmentTable).map { it.toEquipment() }) to HttpStatusCode.OK
                }
            }
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
```

Equipment.kt

```
    }

    put {
        val (t, s) = try {
            val raw = call.receiveText()
            val (id, f) = explodeJsonForModel("equipId", raw)
            transaction {
                EquipmentTable.update({ EquipmentTable.equip_id eq id }) { e ->
                    f["camouflage"]?.let { e[camouflage] = it as String }
                    f["communication"]?.let { e[communication] = it as String }
                    f["intelligence"]?.let { e[intelligence] = it as String }
                    f["medical"]?.let { e[medical] = it as String }
                    f["mreId"]?.let { e[mre_id] = it as Int }
                    f["extra"]?.let { e[extra] = it as String }
                }
            }
            "$raw updated)" to HttpStatusCode.OK
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    post {
        val (t, s) = try {
            val raw = call.receiveText()
            val e = PARSER.fromJson(raw, Equipment::class.java)
            transaction {
                EquipmentTable.insert {
                    it[camouflage] = e.camouflage
                    it[communication] = e.communication
                    it[intelligence] = e.intelligence
                    it[medical] = e.medical
                    it[mre_id] = e.mreId
                    it[extra] = e.extra
                }
            }
            "$raw added)" to HttpStatusCode.Created
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    delete {
        val (t, s) = try {
            val droppedIds = call.receiveText()
            transaction { dropRecordsWithIds(droppedIds, EquipmentTable) }
            "Equipments with $droppedIds deleted)" to HttpStatusCode.OK
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }
}
```

Transport.kt

```kotlin
package com.testpassword.models

import com.testpassword.PARSER
import com.testpassword.dropRecordsWithIds
import com.testpassword.explodeJsonForModel
import com.testpassword.getRecordsWithIds
import io.ktor.application.*
import io.ktor.http.*
import io.ktor.request.*
import io.ktor.response.*
import io.ktor.routing.*
import org.jetbrains.exposed.sql.*
import org.jetbrains.exposed.sql.transactions.transaction
import org.json.JSONObject
import java.io.File

object TransportTable: Table("transport"), Generable {

    val trans_id = integer("trans_id").autoIncrement().primaryKey()
    val name = text("name")
    val type = text("type")
    val status = text("status")

    override fun generateAndInsert(n: Int) {
        val statuses = setOf("available", "under_repair", "destroyed", "broken")
        val rawData = File("resources/static/transports.json").readLines().joinToString(separator = "")
        val jsonBody = JSONObject(rawData).getJSONArray("transport")
        jsonBody.forEachIndexed { i, el ->
            val w = jsonBody.getJSONObject(i)
            TransportTable.insert {
                it[name] = w.getString("name")
                it[type] = w.getString("type")
                it[status] = "available"
            }
        }
    }
}

fun ResultRow.toTransport() = Transport(
    this[TransportTable.trans_id],
    this[TransportTable.name],
    this[TransportTable.type],
    this[TransportTable.status])

data class Transport(val transId: Int?,
                     val name: String,
                     val type: String,
                     val status: String)

fun Route.transport() {

    get {
        val (t, s) = try {
            call.parameters["ids"]!!.let {
                transaction {
                    PARSER.toJson(getRecordsWithIds(it, TransportTable).map { it.toTransport() }) to
HttpStatusCode.OK
                }
            }
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    put {
        val (t, s) = try {
            val raw = call.receiveText()
            val (id, f) = explodeJsonForModel("transId", raw)
            TransportTable.update({ TransportTable.trans_id eq id }) { t ->
                f["name"]?.let { t[name] = it as String }
                f["type"]?.let { t[type] = it as String }
                f["status"]?.let { t[status] = it as String }
            }
            "$raw updated)" to HttpStatusCode.OK
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
```

Transport.kt

```
    }

    post {
        val (t, s) = try {
            val raw = call.receiveText()
            val t = PARSER.fromJson(raw, Transport::class.java)
            transaction {
                TransportTable.insert {
                    it[name] = t.name
                    it[type] = t.type
                    it[status] = t.status
                }
            }
            "$raw added)" to HttpStatusCode.Created
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    delete {
        val (t, s) = try {
            val droppedIds = call.receiveText()
            transaction { dropRecordsWithIds(droppedIds, TransportTable) }
            "Transports with ids $droppedIds deleted)" to HttpStatusCode.OK
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }
}
```

MedicalCard.kt

```kotlin
package com.testpassword.models

import com.testpassword.PARSER
import com.testpassword.dropRecordsWithIds
import com.testpassword.explodeJsonForModel
import com.testpassword.getRecordsWithIds
import io.ktor.application.*
import io.ktor.http.*
import io.ktor.request.*
import io.ktor.response.*
import io.ktor.routing.*
import org.jetbrains.exposed.sql.*
import org.jetbrains.exposed.sql.transactions.transaction

object MedicalCardTable: Table("medical_card"), Generable {

    val med_id = integer("med_id").autoIncrement().primaryKey()
    val emp_id = reference("emp_id", EmployeeTable.emp_id, onDelete = ReferenceOption.CASCADE)
    val height_cm = integer("height_cm")
    val weight_kg = integer("weight_kg")
    val diseases = text("diseases").nullable()
    val blood = text("blood")
    val gender = bool("gender")
}

fun ResultRow.toMedicalCard() = MedicalCard(
    this[MedicalCardTable.med_id],
    this[MedicalCardTable.emp_id],
    this[MedicalCardTable.height_cm],
    this[MedicalCardTable.weight_kg],
    this[MedicalCardTable.diseases],
    this[MedicalCardTable.blood],
    this[MedicalCardTable.gender])

data class MedicalCard(val medId: Int?,
                       val empId: Int,
                       val heightCm: Int,
                       val weightKg: Int,
                       val diseases: String?,
                       val blood: String,
                       val gender: Boolean)

fun Route.medicalCard() {

    get {
        val (t, s) = try {
            call.parameters["ids"]!!.let {
                transaction {
                    PARSER.toJson(getRecordsWithIds(it, MedicalCardTable).map { it.toMedicalCard() }) to
 HttpStatusCode.OK
                }
            }
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    put {
        val (t, s) = try {
            val raw = call.receiveText()
            val (id, f) = explodeJsonForModel("medId", raw)
            transaction {
                MedicalCardTable.update({ MedicalCardTable.med_id eq id }) { m ->
                    f["emp_id"]?.let { m[emp_id] = it as Int }
                    f["heightCm"]?.let { m[height_cm] = it as Int }
                    f["weightKg"]?.let { m[weight_kg] = it as Int }
                    f["diseases"]?.let { m[diseases] = it as String }
                    f["blood"]?.let { m[blood] = it as String }
                    f["gender"]?.let { m[gender] = it as Boolean }
                }
            }
            "$raw updated)" to HttpStatusCode.OK
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

```

MedicalCard.kt

```
    post {
        val (t, s) = try {
            val raw = call.receiveText()
            val m = PARSER.fromJson(raw, MedicalCard::class.java)
            transaction {
                MedicalCardTable.insert {
                    it[emp_id] = m.empId
                    it[height_cm] = m.heightCm
                    it[weight_kg] = m.weightKg
                    it[diseases] = m.diseases
                    it[blood] = m.blood
                    it[gender] = m.gender
                }
            }
            "$raw added)" to HttpStatusCode.Created
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    delete {
        val (t, s) = try {
            val droppedIds = call.receiveText()
            transaction { dropRecordsWithIds(droppedIds, MedicalCardTable) }
            "Cards with $droppedIds deleted)" to HttpStatusCode.OK
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }
}
```

SimpleModels.kt

```kotlin
package com.testpassword.models

import com.github.javafaker.Faker
import io.ktor.util.*
import org.jetbrains.exposed.sql.*
import org.jetbrains.exposed.sql.`java-time`.CurrentDateTime
import org.jetbrains.exposed.sql.`java-time`.date
import org.postgresql.util.PGobject
import java.time.LocalDate
import java.time.ZoneId
import java.util.*
import kotlin.random.Random

/*
https://stackoverflow.com/questions/45723803/how-to-use-postgresql-enum-type-via-kotlin-exposed-orm
https://blog.jdriven.com/2019/07/kotlin-exposed-a-lightweight-sql-library/
 */

val F = Faker() // random data generator

class PGEnum<T : Enum<T>>(enumTypeName: String, enumValue: T?) : PGobject() {
    init {
        value = enumValue?.name
        type = enumTypeName
    }
}

inline fun <reified T : Enum<T>> Table.postgresEnumeration(
    columnName: String,
    postgresEnumName: String
) = customEnumeration(columnName, postgresEnumName,
    { value -> enumValueOf<T>(value as String) }, { PGEnum(postgresEnumName, it) })

interface Generable { fun generateAndInsert(n: Int = 0) = Unit }

enum class FORCES { GF, NAVY, AF }

object WeaponsInEquipment: Table("equip_weapon"), Generable {

    val equip_id = reference("equip_id", EquipmentTable.equip_id)
    val weapon_id = reference("weapon_id", WeaponTable.weapon_id)

    override fun generateAndInsert(n: Int) {
        val weaponIds = WeaponTable.selectAll().map { it[WeaponTable.weapon_id] }
        EquipmentTable.selectAll().map { it[EquipmentTable.equip_id] }.forEach { e ->
            WeaponsInEquipment.insert {
                it[equip_id] = e
                it[weapon_id] = weaponIds.random()
            }
        }
    }
}

object TransportOnMissions: Table("missions_transport"), Generable {

    val miss_id = reference("miss_id", MissionTable.miss_id)
    val trans_id = reference("trans_id", TransportTable.trans_id)

    override fun generateAndInsert(n: Int) {
        val transIds = TransportTable.select { TransportTable.status eq "available" }.map { it[
TransportTable.trans_id] }
        MissionTable.selectAll().map { it[MissionTable.miss_id] }.forEach { m ->
            if (Random.nextBoolean())
                TransportOnMissions.insert {
                    it[miss_id] = m
                    it[trans_id] = transIds.random()
                }
        }
    }
}

object Inspection: Table("inspection"), Generable {

    val emp_id = reference("emp_id", EmployeeTable.emp_id)
    val trans_id = reference("trans_id", TransportTable.trans_id)
```

SimpleModels.kt

```kotlin
    val service_date = date("service_date").defaultExpression(CurrentDateTime().date())

    @InternalAPI override fun generateAndInsert(n: Int) {
        val transIds = TransportTable.selectAll().map { it[TransportTable.trans_id] }
        EmployeeTable.leftJoin(PositionTable)
            .slice(EmployeeTable.emp_id, PositionTable.name)
            .select { PositionTable.name inList listOf("mechanic", "engineer") }
            .map { it[EmployeeTable.emp_id] }
            .forEach { e ->
                if (Random.nextBoolean())
                    Inspection.insert {
                        it[emp_id] = e
                        it[trans_id] = transIds.random()
                        it[service_date] = F.date().between(
                            Date.from(LocalDate.of(2014, 3, 18).atStartOfDay(ZoneId.systemDefault()).toInstant()),
                            Date()
                        ).toLocalDateTime().toLocalDate()
                    }
            }
    }
}

object EmployeeOnMission: Table("missions_emp"), Generable {

    val miss_id = reference("miss_id", MissionTable.miss_id)
    val emp_id = reference("emp_id", EmployeeTable.emp_id)

    override fun generateAndInsert(n: Int) {
        val missIds = MissionTable.selectAll().map { it[MissionTable.miss_id] }
        EmployeeTable
            .leftJoin(PositionTable)
            .select { PositionTable.rank neq null }.map { it[EmployeeTable.emp_id] }
            .forEach { e ->
                EmployeeOnMission.insert {
                    it[miss_id] = missIds.random()
                    it[emp_id] = e
                }
            }
    }
}



/*
https://touk.pl/blog/2019/02/12/how-we-use-kotlin-with-exposed-at-touk/
https://ryanharrison.co.uk/2018/04/14/kotlin-ktor-exposed-starter.html
https://hashrocket.com/blog/posts/faster-json-generation-with-postgresql
https://caelis.medium.com/ktor-send-and-receive-json-6c41c64410af
*/
```

GodRoute.kt

```kotlin
package com.testpassword.routes

import com.testpassword.*
import com.testpassword.models.*
import io.ktor.application.*
import io.ktor.html.*
import io.ktor.http.*
import io.ktor.response.*
import io.ktor.routing.*
import kotlinx.html.*
import org.jetbrains.exposed.sql.SchemaUtils
import org.jetbrains.exposed.sql.transactions.transaction

fun Route.god() {

    val GOD_URL = "localhost:9090/god" //TODO: убрать захардкоженый урл

    get {
        call.respondHtml {
            body {
                h1 { +"GOD-MODE" }
                ul {
                    li {
                        form(action = GOD_URL, method = FormMethod.put) {
                            button(type = ButtonType.submit) {
                                +"PUT to create tables before fill"
                            }
                        }
                    }
                    li {
                        form(action = GOD_URL, method = FormMethod.post) {
                            button(type=ButtonType.submit) {
                                + "POST to fill tables"
                            }
                        }
                    }
                    li {
                        form(action = GOD_URL, method = FormMethod.delete) {
                            button(type = ButtonType.submit) {
                                + "DELETE to drop table and all data"
                            }
                        }
                    }
                }
            }
        }
    }

    put {
        transaction {
            SchemaUtils.create(
                BaseTable, MRETable, EquipmentTable, PositionTable, EmployeeTable, MedicalCardTable,
WeaponTable,
                CampaignTable, MissionTable, TransportTable, WeaponsInEquipment, TransportOnMissions,
Inspection,
                EmployeeOnMission
            )
        }
    }

    post {
        val (t, s) = try {
            transaction {
                fillDb(
                    mapOf(
                        BaseTable to 50,
                        MRETable to 30,
                        EquipmentTable to 75,
                        PositionTable to 50,
                        EmployeeTable to 1000,
                        MedicalCardTable to 0,
                        WeaponTable to 0,
                        CampaignTable to 40,
                        MissionTable to 110,
                        TransportTable to 0,
```

GodRoute.kt

```
                        WeaponsInEquipment to 0,
                        TransportOnMissions to 0,
                        Inspection to 0,
                        EmployeeOnMission to 0
                    )
                )
            }
            "Records generated" to HttpStatusCode.OK
        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
        call.respondText(text = t, status = s)
    }

    delete {
        transaction {
            SchemaUtils.drop(
                BaseTable, MRETable, EquipmentTable, PositionTable, EmployeeTable, MedicalCardTable, WeaponTable,
                CampaignTable, MissionTable, TransportTable, WeaponsInEquipment, TransportOnMissions, Inspection,
                EmployeeOnMission
            )
        }
    }
}
```