

```

1 package com.testpassword
2
3 import com.google.gson.GsonBuilder
4 import com.google.gson.JsonDeserializer
5 import com.google.gson.JsonPrimitive
6 import com.google.gson.JsonSerializer
7 import com.testpassword.models.Generable
8 import org.jetbrains.exposed.sql.*
9 import org.json.JSONObject
10 import java.time.LocalDate
11
12 val PARSER = GsonBuilder()
13     .registerTypeAdapter(
14         LocalDate::class.java,
15         JsonSerializer<LocalDate> { src, _, _ -> if (src == null) null else JsonPrimitive(src.toString())
16             () })
17     .registerTypeAdapter(
18         LocalDate::class.java,
19         JsonDeserializer<LocalDate> { json, _, _ -> if (json == null) null else LocalDate.parse(json.
20             asString().split("T").first()) })
21     .create()
22
23 class ReferenceEntityError(e: Table): Error() {
24     override val message = "Reference entity table ${e.tableName} does not contains such records"
25 }
26
27 fun fillDb(sizes: Map<Generable, Int>) = sizes.forEach { it.key.generateAndInsert(it.value) }
28
29 fun dropRecordsWithIds(s: String, entityTable: Table) {
30     val ids = JSONObject(s).getJSONArray("droppedIds")
31     ids.forEachIndexed { i, _ ->
32         entityTable.deleteWhere { (entityTable.primaryKey?.columns?.get(0) as Column<Int>) eq ids.getInt
33             (i) }
34     }
35 }
36
37 fun getRecordsWithIds(s: String, entityTable: Table): Query {
38     val raw = JSONObject(s).getJSONArray("selectedIds")
39     val ids = mutableListOf<Int>()
40     raw.forEachIndexed { i, _ -> ids.add(raw.getInt(i)) }
41     return if (ids.isEmpty()) entityTable.selectAll() else
42         entityTable.select {
43             entityTable.primaryKey?.columns?.get(0) as Column<Int> inList ids
44         }
45 }
46
47 fun explodeJsonForModel(modelIdField: String, jsonStr: String): Pair<Int, Map<String, Any>> {
48     val s = JSONObject(jsonStr)
49     val id = s.getInt(modelIdField)
50     s.remove(modelIdField)
51     val fields = mutableMapOf<String, Any>()
52     s.keys().forEach { fields[it] = s.get(it) }
53     return id to fields
54 }
```

Application.kt

```
1 package com.testpassword
2
3 import com.testpassword.models.*
4 import com.testpassword.routes.god
5 import com.zaxxer.hikari.HikariConfig
6 import com.zaxxer.hikari.HikariDataSource
7 import io.ktor.application.*
8 import io.ktor.features.*
9 import io.ktor.http.*
10 import io.ktor.routing.*
11 import org.jetbrains.exposed.sql.*
12
13 fun main(args: Array<String>): Unit = io.ktor.server.netty.EngineMain.main(args)
14
15 @Suppress("unused") // Referenced in application.conf
16 fun Application.initDB() =
17     Database.connect(HikariDataSource(HikariConfig(environment.config.property("ktor.hikariconfig").getString())))
18
19 fun Application.allowCORS() {
20     install(CORS) {
21         anyHost()
22         allowCredentials = true
23         allowNonSimpleContentTypes = true
24         methods.addAll(HttpMethod.DefaultMethods)
25     }
26 }
27
28 //https://stefangaller.at/app-development/kotlin/ktor-rest-api-exposed/
29 fun Application.module() {
30     initDB()
31     allowCORS()
32     routing {
33         route("god") { god() }
34         route("base") { base() }
35         route("campaign") { campaign() }
36         route("employee") { employee() }
37         route("equipment") { equipment() }
38         route("medicalCard") { medicalCard() }
39         route("mission") { mission() }
40         route("mre") { mre() }
41         route("position") { position() }
42         route("transport") { transport() }
43         route("weapon") { weapon() }
44     }
45 }
```

```

1 package com.testpassword.models
2
3 import com.testpassword.*
4 import io.ktor.application.*
5 import io.ktor.http.*
6 import io.ktor.request.*
7 import io.ktor.response.*
8 import io.ktor.routing.*
9 import org.jetbrains.exposed.sql.*
10 import org.jetbrains.exposed.sql.transactions.transaction
11
12 object MRETable: Table("mre"), Generable {
13
14     val mre_id = integer("mre_id").autoIncrement().primaryKey()
15     val breakfast = text("breakfast")
16     val lunch = text("lunch")
17     val dinner = text("dinner")
18     val food_additives = text("food_additives").nullable()
19     val kkal = integer("kkal").check { it greaterEq 3000 }
20     val proteins = integer("proteins").check { it greater 0 }
21     val fats = integer("fats").check { it greater 0 }
22     val carbohydrate = integer("carbohydrate").check { it greater 0 }
23
24     override fun generateAndInsert(n: Int) {
25         (1..n).forEach {
26             MRETable.insert {
27                 it[breakfast] = F.food().dish()
28                 it[lunch] = F.food().dish()
29                 it[dinner] = F.food().dish()
30                 it[food_additives] = generateSequence { F.food().ingredient() }.take(3).joinToString(
31                     separator = " ")
32                     it[kkal] = F.number().numberBetween(3000, 4500)
33                     it[proteins] = F.number().numberBetween(1, 400)
34                     it[fats] = F.number().numberBetween(1, 400)
35                     it[carbohydrate] = F.number().numberBetween(1, 400)
36             }
37         }
38     }
39
40 fun ResultRow.toMRE() = MRE(
41     this[MRETable.mre_id],
42     this[MRETable.breakfast],
43     this[MRETable.lunch],
44     this[MRETable.dinner],
45     this[MRETable.food_additives],
46     this[MRETable.kkal],
47     this[MRETable.proteins],
48     this[MRETable.fats],
49     this[MRETable.carbohydrate])
50
51 data class MRE(val mreId: Int?,
52                 val breakfast: String,
53                 val lunch: String,
54                 val dinner: String,
55                 val foodAdditives: String?,
56                 val kkal: Int,
57                 val proteins: Int,
58                 val fats: Int,
59                 val carbohydrate: Int)
50
51
52
53
54
55
56
57
58
59
60
61 fun Route.mre() {
62
63     get {
64         val (t, s) = try {
65             call.parameters["ids"]!!.let {
66                 transaction {
67                     PARSER.toJson(getRecordsWithIds(it, MRETable).map { it.toMRE() }) to HttpStatusCode.OK
68                 }
69             }
70         } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
71         call.respondText(text = t, status = s)
72     }
73 }
```

```

74    put {
75        val (t, s) = try {
76            val raw = call.receiveText()
77            val (id, f) = explodeJsonForModel("missId", raw)
78            transaction {
79                MRETable.update({ MRETable.mre_id eq id }) { m ->
80                    f["breakfast"]?.let { m[breakfast] = it as String }
81                    f["lunch"]?.let { m[lunch] = it as String }
82                    f["dinner"]?.let { m[dinner] = it as String }
83                    f["foodAdditives"]?.let { m[food_additives] = it as String }
84                    f["kkal"]?.let { m[kkal] = it as Int }
85                    f["proteins"]?.let { m[proteins] = it as Int }
86                    f["fats"]?.let { m[fats] = it as Int }
87                    f["carbohydrate"]?.let { m[carbohydrate] = it as Int }
88                }
89            }
90            "$raw updated)" to HttpStatusCode.OK
91        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
92        call.respondText(text = t, status = s)
93    }
94
95    post {
96        val (t, s) = try {
97            val raw = call.receiveText()
98            val m = PARSER.fromJson(raw, MRE::class.java)
99            transaction {
100                MRETable.insert {
101                    it[breakfast] = m.breakfast
102                    it[lunch] = m.lunch
103                    it[dinner] = m.dinner
104                    it[food_additives] = m.foodAdditives
105                    it[kkal] = m.kkal
106                    it[proteins] = m.proteins
107                    it[fats] = m.fats
108                    it[carbohydrate] = m.carbohydrate
109                }
110            }
111            "$raw added)" to HttpStatusCode.Created
112        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
113        call.respondText(text = t, status = s)
114    }
115
116    delete {
117        val (t, s) = try {
118            val droppedIds = call.receiveText()
119            transaction { dropRecordsWithIds(droppedIds, MRETable) }
120            "MREs with $droppedIds deleted)" to HttpStatusCode.OK
121        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
122        call.respondText(text = t, status = s)
123    }
124 }
125

```

```

1 package com.testpassword.models
2
3 import com.testpassword.*
4 import io.ktor.application.*
5 import io.ktor.http.*
6 import io.ktor.request.*
7 import io.ktor.response.*
8 import io.ktor.routing.*
9 import org.jetbrains.exposed.sql.*
10 import org.jetbrains.exposed.sql.transactions.transaction
11
12 object BaseTable: Table("base"), Generable {
13
14     val base_id = integer("base_id").autoIncrement().primaryKey()
15     val location = text("location")
16     val status = text("status")
17
18     override fun generateAndInsert(n: Int) {
19         val statuses = setOf("working", "closed", "destroyed", "abandoned", "captured", "for_sale")
20         (1..n).forEach {
21             BaseTable.insert {
22                 it[location] = F.address().city()
23                 it[status] = statuses.random()
24             }
25         }
26     }
27 }
28
29 fun ResultRow.toBase() = Base(
30     this[BaseTable.base_id],
31     this[BaseTable.location],
32     this[BaseTable.status])
33
34 data class Base(val baseId: Int?,
35                  val location: String,
36                  val status: String)
37
38 fun Route.base() {
39
40     get {
41         val (t, s) = try {
42             call.parameters["ids"]!!.let {
43                 transaction {
44                     PARSER.toJson(getRecordsWithIds(it, BaseTable)).map { it.toBase() } to
45                     HttpStatusCode.OK
46                 }
47             }
48         } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
49         call.respondText(text = t, status = s)
50     }
51
52     put {
53         val (t, s) = try {
54             val raw = call.receiveText()
55             val (id, f) = explodeJsonForModel("baseId", raw)
56             transaction {
57                 BaseTable.update({ BaseTable.base_id eq id }) { b ->
58                     f["location"]?.let { b[location] = it as String }
59                     f["status"]?.let { b[status] = it as String }
60                 }
61             }
62             "$raw updated" to HttpStatusCode.OK
63         } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
64         call.respondText(text = t, status = s)
65     }
66
67     post {
68         val (t, s) = try {
69             val raw = call.receiveText()
70             val b = PARSER.fromJson(raw, Base::class.java)
71             transaction {
72                 BaseTable.insert {
73                     it[location] = b.location
74                     it[status] = b.status
75                 }
76             }
77         } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
78     }
79 }
80
81 
```

```
75         }
76         "$raw added)" to HttpStatusCode.Created
77     } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
78     call.respondText(text = t, status = s)
79 }
80
81 delete {
82     val (t, s) = try {
83         val droppedIds = call.receiveText()
84         transaction { dropRecordsWithIds(droppedIds, BaseTable) }
85         "Bases with $droppedIds deleted)" to HttpStatusCode.OK
86     } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
87     call.respondText(text = t, status = s)
88 }
89 }
```

```

1 package com.testpassword.models
2
3 import com.testpassword.PARSER
4 import com.testpassword.dropRecordsWithIds
5 import com.testpassword.explodeJsonForModel
6 import com.testpassword.getRecordsWithIds
7 import io.ktor.application.*
8 import io.ktor.http.*
9 import io.ktor.request.*
10 import io.ktor.response.*
11 import io.ktor.routing.*
12 import org.jetbrains.exposed.sql.*
13 import org.jetbrains.exposed.sql.transactions.transaction
14 import org.json.JSONObject
15 import java.io.File
16
17 object WeaponTable: Table("weapon"), Generable {
18
19     val weapon_id = integer("weapon_id").autoIncrement().primaryKey()
20     val name = text("name")
21     val type = text("type")
22     val caliber = float("caliber").check { it greater 0.0 }.nullable()
23     val rate_of_fire = integer("rate_of_fire").check { it greater 0 }.nullable()
24     val sighting_range_m = integer("sighting_range_m").check { it greater 0 }.nullable()
25
26     override fun generateAndInsert(n: Int) {
27         val rawData = File("resources/static/guns.json").readLines().joinToString(separator = "")
28         val jsonBody = JSONObject(rawData).getJSONArray("weapons")
29         jsonBody.forEachIndexed { i, _ ->
30             val w = jsonBody.getJSONObject(i)
31             WeaponTable.insert {
32                 it[name] = w.getString("name")
33                 it[type] = w.getString("type")
34                 it[caliber] = w.getFloat("caliber")
35                 it[rate_of_fire] = w.getInt("rate_of_fire")
36                 it[sighting_range_m] = w.getInt("sighting_range_m")
37             }
38         }
39     }
40 }
41
42 fun ResultRow.toWeapon() = Weapon(
43     this[WeaponTable.weapon_id],
44     this[WeaponTable.name],
45     this[WeaponTable.type],
46     this[WeaponTable.caliber]?.toDouble(),
47     this[WeaponTable.rate_of_fire],
48     this[WeaponTable.sighting_range_m])
49
50 data class Weapon(val weaponId: Int?,
51                     val name: String,
52                     val type: String,
53                     val caliber: Double?,
54                     val rateOfFire: Int?,
55                     val sightingRangeM: Int?)
56
57 fun Route.weapon() {
58
59     get {
60         val (t, s) = try {
61             call.parameters["ids"]!!.let {
62                 transaction {
63                     PARSER.toJson(getRecordsWithIds(it, WeaponTable).map { it.toWeapon() })
64                     HttpStatusCode.OK
65                 }
66             } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
67             call.respondText(text = t, status = s)
68         }
69
70         put {
71             val (t, s) = try {
72                 val raw = call.receiveText()
73                 val (id, f) = explodeJsonForModel("weaponId", raw)
74                 WeaponTable.update({ WeaponTable.weapon_id eq id }) { w ->

```

```
75         f["name"]?.let { w[name] = it as String }
76         f["type"]?.let { w[type] = it as String }
77         f["caliber"]?.let { w[caliber] = it as Float }
78         f["rateOfFire"]?.let { w[rate_of_fire] = it as Int }
79         f["sighting_rangeM"]?.let { w[sighting_range_m] = it as Int }
80     }
81     "$raw updated)" to HttpStatusCode.OK
82 } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
83 call.respondText(text = t, status = s)
84 }
85
86 post {
87     val (t, s) = try {
88         val raw = call.receiveText()
89         val w = PARSER.fromJson(raw, Weapon::class.java)
90         transaction {
91             WeaponTable.insert {
92                 it[name] = w.name
93                 it[type] = w.type
94                 it[caliber] = w.caliber?.toFloat()
95                 it[rate_of_fire] = w.rateOfFire
96                 it[sighting_range_m] = w.sightingRangeM
97             }
98         }
99         "$raw added)" to HttpStatusCode.Created
100    } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
101   call.respondText(text = t, status = s)
102 }
103
104 delete {
105     val (t, s) = try {
106         val droppedIds = call.receiveText()
107         transaction { dropRecordsWithIds(droppedIds, WeaponTable) }
108         "Weapons with ids $droppedIds deleted)" to HttpStatusCode.OK
109     } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
110     call.respondText(text = t, status = s)
111 }
112 }
```

```

1 package com.testpassword.models
2
3 import com.testpassword.*
4 import io.ktor.application.*
5 import io.ktor.http.*
6 import io.ktor.request.*
7 import io.ktor.response.*
8 import io.ktor.routing.*
9 import io.ktor.util.*
10 import org.jetbrains.exposed.sql.*
11 import org.jetbrains.exposed.sql.`java-time`.datetime
12 import org.jetbrains.exposed.sql.transactions.transaction
13 import java.time.LocalDate
14 import java.time.LocalDateTime
15 import java.time.ZoneId
16 import java.util.*
17 import kotlin.random.Random
18
19 object MissionTable: Table("mission"), Generable {
20
21     val miss_id = integer("miss_id").autoIncrement().primaryKey()
22     val camp_id = reference("camp_id", CampaignTable.camp_id, onDelete = ReferenceOption.CASCADE)
23     val start_date_and_time = datetime("start_date_and_time").nullable()
24     val end_date_and_time = datetime("end_date_and_time").nullable()
25     val legal_status = bool("legal_status")
26     val departure_location = text("departure_location").nullable()
27     val arrival_location = text("arrival_location").nullable()
28     val enemies = text("enemies").nullable()
29
30     @InternalAPI
31     override fun generateAndInsert(n: Int) {
32         val campIds = CampaignTable.selectAll().map { it[CampaignTable.camp_id] }
33         (1..n).forEach {
34             val st = F.date().between(
35                 Date.from(LocalDate.of(2014, 3, 18).atStartOfDay(ZoneId.systemDefault()).toInstant()),
36                 Date()
37             )
38             val et = F.date().between(st, Date())
39             MissionTable.insert {
40                 it[camp_id] = campIds.random()
41                 it[start_date_and_time] = st.toLocalDateTime()
42                 it[end_date_and_time] = et.toLocalDateTime()
43                 it[legal_status] = Random.nextBoolean()
44                 it[departure_location] = "${F.address().latitude()} ${F.address().longitude()}"
45                 it[arrival_location] = "${F.address().latitude()} ${F.address().longitude()}"
46                 it=enemies = arrayOf(F.nation().nationality(), F.name().fullName()).random()
47             }
48         }
49     }
50 }
51
52 fun ResultRow.toMission() = Mission(
53     this[MissionTable.miss_id],
54     this[MissionTable.camp_id],
55     this[MissionTable.start_date_and_time],
56     this[MissionTable.end_date_and_time],
57     this[MissionTable.legal_status],
58     this[MissionTable.departure_location],
59     this[MissionTable.arrival_location],
60     this[MissionTable.enemies])
61
62 data class Mission(val missId: Int?,
63                     val campId: Int,
64                     val startDateAndTime: LocalDateTime?,
65                     val endDateAndTime: LocalDateTime?,
66                     val legalStatus: Boolean,
67                     val departureLocation: String?,
68                     val arrivalLocation: String?,
69                     val enemies: String?)
70
71 fun Route.mission() {
72
73     get {
74         val (t, s) = try {
75             call.parameters["ids"]!!.let {

```

Mission.kt

```

76             transaction {
77                 PARSER.toJson(getRecordsWithIds(it, MissionTable).map { it.toMission() }) to
78                 HttpStatusCode.OK
79             }
80         } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
81         call.respondText(text = t, status = s)
82     }
83
84     put {
85         val (t, s) = try {
86             val raw = call.receiveText()
87             val (id, f) = explodeJsonForModel("missId", raw)
88             transaction {
89                 MissionTable.update({ MissionTable.miss_id eq id }) { m ->
90                     f["camp_id"]?.let { m[camp_id] = it as Int }
91                     f["start_date_and_time"]?.let { m[start_date_and_time] = LocalDateTime.parse(it as
92 String) }
93                     f["end_date_and_time"]?.let { m[end_date_and_time] = LocalDateTime.parse(it as
94 String) }
95                     f["legal_status"]?.let { m[legal_status] = it as Boolean }
96                     f["departure_location"]?.let { m[departure_location] = it as String }
97                     f["arrival_location"]?.let { m[arrival_location] = it as String }
98                     f["enemies"]?.let { m[enemies] = it as String }
99                 }
100            "$raw updated)" to HttpStatusCode.OK
101        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
102        call.respondText(text = t, status = s)
103    }
104
105    post {
106        val (t, s) = try {
107            val raw = call.receiveText()
108            val m = PARSER.fromJson(raw, Mission::class.java)
109            transaction {
110                MissionTable.insert {
111                    it[camp_id] = m.campId
112                    it[start_date_and_time] = m.startDateAndTime
113                    it[end_date_and_time] = m.endDateAndTime
114                    it[legal_status] = m.legalStatus
115                    it[departure_location] = m.departureLocation
116                    it[arrival_location] = m.arrivalLocation
117                    it[enemies] = m.enemies
118                }
119            }
120            "$raw added)" to HttpStatusCode.Created
121        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
122        call.respondText(text = t, status = s)
123    }
124
125    delete {
126        val (t, s) = try {
127            val droppedIds = call.receiveText()
128            transaction { dropRecordsWithIds(droppedIds, MissionTable) }
129            "Missions with $droppedIds deleted)" to HttpStatusCode.OK
130        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
131        call.respondText(text = t, status = s)
132    }

```

Campaign.kt

```

1 package com.testpassword.models
2
3 import com.testpassword.*
4 import io.ktor.application.*
5 import io.ktor.http.*
6 import io.ktor.request.*
7 import io.ktor.response.*
8 import io.ktor.routing.*
9 import org.jetbrains.exposed.sql.*
10 import org.jetbrains.exposed.sql.transactions.transaction
11 import java.math.BigDecimal
12 import kotlin.random.Random
13
14 object CampaignTable: Table("campaign"), Generable {
15
16     val camp_id = integer("camp_id").autoIncrement().primaryKey()
17     val name = text("name")
18     val customer = text("customer")
19     val earning = decimal("earning", 2, 11).check { it greaterEq 0.0 }
20     val spending = decimal("spending", 2, 11).check { it greaterEq 0.0 }
21     val execution_status = text("execution_status").nullable()
22
23     override fun generateAndInsert(n: Int) {
24         val statuses = setOf("completed", "in the process", "failed", "canceled")
25         (1..n).forEach {
26             CampaignTable.insert {
27                 it[name] = F.ancient().titan()
28                 it[customer] = F.name().fullName()
29                 it[earning] = F.number().randomDouble(2, 500000, 100000000).toBigDecimal()
30                 it[spending] = F.number().randomDouble(2, 0, 10000000).toBigDecimal()
31                 it[execution_status] = if (Random.nextInt(1, 100) >= 70) statuses.random() else statuses
32             .first()
33         }
34     }
35 }
36
37 fun ResultRow.toCampaign() = Campaign(
38     this[CampaignTable.camp_id],
39     this[CampaignTable.name],
40     this[CampaignTable.customer],
41     this[CampaignTable.earning].toDouble(),
42     this[CampaignTable.spending].toDouble(),
43     this[CampaignTable.execution_status])
44
45 data class Campaign(val campId: Int?,
46                      val name: String,
47                      val customer: String,
48                      val earning: Double,
49                      val spending: Double,
50                      val executionStatus: String?)
51
52 fun Route.campaign() {
53
54     get {
55         val (t, s) = try {
56             call.parameters["ids"]!!.let {
57                 transaction {
58                     PARSER.toJson(getRecordsWithIds(it, CampaignTable).map { it.toCampaign() })
59                     to HttpStatusCode.OK
60                 }
61             }
62         } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
63         call.respondText(text = t, status = s)
64     }
65
66     put {
67         val (t, s) = try {
68             val raw = call.receiveText()
69             val (id, f) = explodeJsonForModel("campId", raw)
70             CampaignTable.update({ CampaignTable.camp_id eq id }) { c -
71                 f["name"]?.let { c[name] = it as String }
72                 f["customer"]?.let { c[customer] = it as String }
73                 f["earning"]?.let { c[earning] = (it as Double).toBigDecimal() }
74                 f["spending"]?.let { c[spending] = (it as Double).toBigDecimal() }
75             }
76         } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
77     }
78 }
79
80 
```

```
74         f["executionStatus"]?.let { c[execution_status] = it as String }
75     }
76     "$raw updated)" to HttpStatusCode.OK
77 } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
78 call.respondText(text = t, status = s)
79 }
80
81 post {
82     val (t, s) = try {
83         val raw = call.receiveText()
84         val b = PARSE.fromJson(raw, Campaign::class.java)
85         transaction {
86             CampaignTable.insert {
87                 it[name] = b.name
88                 it[customer] = b.customer
89                 it[earning] = b.earning.toBigDecimal()
90                 it[spending] = b.spending.toBigDecimal()
91                 it[execution_status] = b.executionStatus
92             }
93         }
94         "$raw added)" to HttpStatusCode.Created
95     } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
96     call.respondText(text = t, status = s)
97 }
98
99 delete {
100     val (t, s) = try {
101         val droppedIds = call.receiveText()
102         transaction { dropRecordsWithIds(droppedIds, CampaignTable) }
103         "Campaigns with ids $droppedIds deleted)" to HttpStatusCode.OK
104     } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
105     call.respondText(text = t, status = s)
106 }
107 }
```

Employee.kt

```

1 package com.testpassword.models
2
3 import com.google.gson.*
4 import com.testpassword.*
5 import io.ktor.application.*
6 import io.ktor.http.*
7 import io.ktor.request.*
8 import io.ktor.response.*
9 import io.ktor.routing.*
10 import io.ktor.util.*
11 import org.jetbrains.exposed.sql.*
12 import org.jetbrains.exposed.sql.`java-time`.CurrentDateTime
13 import org.jetbrains.exposed.sql.`java-time`.date
14 import org.jetbrains.exposed.sql.`java-time`.year
15 import org.jetbrains.exposed.sql.transactions.transaction
16 import java.time.LocalDate
17 import java.time.ZoneId
18 import java.util.*
19 import kotlin.random.Random
20
21
22 object EmployeeTable: Table("employee"), Generable {
23
24     val emp_id = integer("emp_id").autoIncrement().primaryKey()
25     val name = text("name")
26     val surname = text("surname")
27     val date_of_birth = date("date_of_birth").check { it less (CurrentDateTime().year() - 18) }
28     val education = text("education").nullable()
29     val hiring_date = date("hiring_date").defaultExpression(CurrentDateTime().date())
30     val pos_id = reference("pos_id", PositionTable.pos_id, onDelete = ReferenceOption.RESTRICT)
31     val is_married = bool("is_married")
32     val base_id = reference("base_id", BaseTable.base_id, onDelete = ReferenceOption.SET_NULL).nullable
33     ()
34
35     @InternalAPI
36     override fun generateAndInsert(n: Int) {
37         val posIds = PositionTable.selectAll().limit(n).map { it[PositionTable.pos_id] }
38         val baseIds = BaseTable.selectAll().map { it[BaseTable.base_id] }
39         (1..n).forEach {
40             val newbieId = EmployeeTable.insert {
41                 it[name] = F.name().firstName()
42                 it[surname] = F.name().lastName()
43                 it[date_of_birth] = F.date().birthday(18, 70).toLocalDateTime().toLocalDate()
44                 it[education] = F.educator().university()
45                 it[hiring_date] = F.date().between(
46                     // здесь и далее – условной день основания компании.
47                     Date.from(LocalDate.of(2014, 3, 18).atStartOfDay(ZoneId.systemDefault()).toInstant
48                     (),
49                     Date()
50                     .toLocalDateTime().toLocalDate()
51                     it[pos_id] = posIds.random()
52                     it[is_married] = Random.nextBoolean()
53                     it[base_id] = baseIds.random()
54                     )[emp_id]
55                     MedicalCardTable.insert {
56                         it[emp_id] = newbieId
57                         it[height_cm] = Random.nextInt(150, 200)
58                         it[weight_kg] = Random.nextInt(40, 120)
59                         it[diseases] = generateSequence { F.medical().diseaseName() }
60                             .take(Random.nextInt(0, 5))
61                             .joinToString(separator = " ")
62                         it[blood] = F.name().bloodGroup()
63                         it[gender] = Random.nextBoolean()
64                     }
65                 }
66             }
67         fun ResultRow.toEmployee() = Employee(
68             this[EmployeeTable.emp_id],
69             this[EmployeeTable.name],
70             this[EmployeeTable.surname],
71             this[EmployeeTable.date_of_birth],
72             this[EmployeeTable.education],
73             this[EmployeeTable.hiring_date],

```

Employee.kt

```

74    this[EmployeeTable.pos_id],
75    this[EmployeeTable.is_married],
76    this[EmployeeTable.base_id])
77
78 data class Employee(val empId: Int?,
79                      val name: String,
80                      val surname: String,
81                      val dateOfBirth: LocalDate,
82                      val education: String?,
83                      val hiringDate: LocalDate,
84                      val posId: Int,
85                      val isMarried: Boolean,
86                      val baseId: Int?)
87
88 fun Route.employee() {
89
90     get {
91         val (t, s) = try {
92             call.parameters["ids"]!!.let {
93                 transaction {
94                     PARSER.toJson(getRecordsWithIds(it, EmployeeTable).map { it.toEmployee() }) to
95                     HttpStatusCode.OK
96                 }
97             } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
98             call.respondText(text = t, status = s)
99         }
100
101        put {
102            val (t, s) = try {
103                val raw = call.receiveText()
104                val (id, f) = explodeJsonForModel("empId", raw)
105                transaction {
106                    EmployeeTable.update({ EmployeeTable.emp_id eq id }) { e ->
107                        f["name"]?.let { e[name] = it as String }
108                        f["surname"]?.let { e[surname] = it as String }
109                        f["dateOfBirth"]?.let { e[date_of_birth] = LocalDate.parse(it as String) }
110                        f["education"]?.let { e[education] = it as String }
111                        f["hiringDate"]?.let { e[hiring_date] = LocalDate.parse(it as String) }
112                        f["posId"]?.let { e[pos_id] = it as Int }
113                        f["isMarried"]?.let { e[is_married] = it as Boolean }
114                        f["baseId"]?.let { e[base_id] = it as Int }
115                    }
116                }
117                "$raw updated)" to HttpStatusCode.OK
118            } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
119            call.respondText(text = t, status = s)
120        }
121
122        post {
123            val (t, s) = try {
124                val raw = call.receiveText()
125                val e = PARSER.fromJson(raw, Employee::class.java)
126                transaction {
127                    EmployeeTable.insert {
128                        it[name] = e.name
129                        it[surname] = e.surname
130                        it[date_of_birth] = e.dateOfBirth
131                        it[education] = e.education
132                        it[hiring_date] = e.hiringDate
133                        it[pos_id] = e.posId
134                        it[is_married] = e.isMarried
135                        it[base_id] = e.baseId
136                    }
137                }
138                "$raw added)" to HttpStatusCode.Created
139            } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
140            call.respondText(text = t, status = s)
141        }
142
143        delete {
144            val (t, s) = try {
145                val droppedIds = call.receiveText()
146                transaction { dropRecordsWithIds(droppedIds, EmployeeTable) }
147                "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 }
```

```

1 package com.testpassword.models
2
3 import com.testpassword.*
4 import io.ktor.application.*
5 import io.ktor.http.*
6 import io.ktor.request.*
7 import io.ktor.response.*
8 import io.ktor.routing.*
9 import org.jetbrains.exposed.sql.*
10 import org.jetbrains.exposed.sql.transactions.transaction
11 import java.math.BigDecimal
12 import kotlin.random.Random
13
14 object PositionTable: Table("Position"), Generable {
15
16     val pos_id = integer("pos_id").autoIncrement().primaryKey()
17     val name = text("name")
18     val salary = decimal("salary", 2, 11).check { it greater 300.0 }
19     val rank = text("rank").nullable()
20     val equip_id = reference("equip_id", EquipmentTable.equip_id, onDelete = ReferenceOption.SET_NULL).nullable()
21     val forces = postgresEnumeration<FORCES>("forces", "force").nullable()
22
23     override fun generateAndInsert(n: Int) {
24         val ranks = setOf("student_officer", "private_second_class", "private_first_class", "junior_sergeant",
25             "sergeant", "senior_sergeant", "petty_officer", "ensign", "senior_ensign", "junior_lieutenant",
26             "lieutenant", "senior_lieutenant", "captain", "major", "lieutenant_colonel", "colonel") //, "major_general",
27             "lieutenant_general",
28             // "colonel_general", "army_general", "marshal" ) исключим высшие звания из генерации
29         val armyPositions = setOf("medic", "miner", "scout", "security", "driver", "torpedo_operator", "pilot",
30             "mechanic", "engineer", "navigator", "orderly", "duty", "coach", "artilleryman", "gunner", "sniper",
31             "spy")
32         val equipIds = EquipmentTable.selectAll().limit(n).map { it[EquipmentTable.equip_id] }
33         (1..n).forEach {
34             PositionTable.insert {
35                 val empPos = if (Random.nextDouble(1.0, 100.0) >= 65) armyPositions.random() else F.job
36                 .position()
37                 it[name] = empPos
38                 it[salary] = F.number().randomDouble(2, 300, 1000000).toBigDecimal()
39                 if (empPos in armyPositions) it[rank] = ranks.random()
40                 it[equip_id] = equipIds.random()
41                 it[forces] = FORCES.values().random()
42             }
43         }
44     }
45     fun ResultRow.toPosition() = Position(
46         this[PositionTable.pos_id],
47         this[PositionTable.name],
48         this[PositionTable.salary],
49         this[PositionTable.rank],
50         this[PositionTable.equip_id],
51         this[PositionTable.forces])
52
53     data class Position(val posId: Int?,
54                         val name: String,
55                         val salary: BigDecimal,
56                         val rank: String?,
57                         val equipId: Int?,
58                         val forces: FORCES?)
59
60     fun Route.position() {
61         get {
62             val (t, s) = try {
63                 call.parameters["ids"]!!.let {
64                     transaction {
65                         PARSER.toJson(getRecordsWithIds(it, PositionTable).map { it.toPosition() }) to
66                         HttpStatusCode.OK
67                     }
68                 }
69             }
70         }
71     }
72 }
```

```

68         } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
69         call.respondText(text = t, status = s)
70     }
71
72     put {
73         val (t, s) = try {
74             val raw = call.receiveText()
75             val (id, f) = explodeJsonForModel("posId", raw)
76             PositionTable.update({ PositionTable.pos_id eq id }) { p ->
77                 f["name"]?.let { p[name] = it as String }
78                 f["salary"]?.let { p[salary] = (it as Double).toBigDecimal() }
79                 f["rank"]?.let { p[rank] = it as String }
80                 f["equipId"]?.let { p[equip_id] = it as Int }
81                 f["forces"]?.let { p[forces] = FORCES.valueOf(it as String) }
82             }
83             "$raw updated)" to HttpStatusCode.OK
84         } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
85         call.respondText(text = t, status = s)
86     }
87
88     post {
89         val (t, s) = try {
90             val raw = call.receiveText()
91             val p = PARSER.fromJson(raw, Position::class.java)
92             transaction {
93                 PositionTable.insert {
94                     it[name] = p.name
95                     it[salary] = p.salary
96                     it[rank] = p.rank
97                     it[equip_id] = p.equipId
98                     it[forces] = p.forces
99                 }
100            }
101            "$raw added)" to HttpStatusCode.Created
102        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
103        call.respondText(text = t, status = s)
104    }
105
106    delete {
107        val (t, s) = try {
108            val droppedIds = call.receiveText()
109            transaction { dropRecordsWithIds(droppedIds, PositionTable) }
110            "Positions with $droppedIds deleted)" to HttpStatusCode.OK
111        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
112        call.respondText(text = t, status = s)
113    }
114 }
```

Equipment.kt

```

1 package com.testpassword.models
2
3 import com.testpassword.PARSER
4 import com.testpassword.dropRecordsWithIds
5 import com.testpassword.explodeJsonForModel
6 import com.testpassword.getRecordsWithIds
7 import io.ktor.application.*
8 import io.ktor.http.*
9 import io.ktor.request.*
10 import io.ktor.response.*
11 import io.ktor.routing.*
12 import org.jetbrains.exposed.sql.*
13 import org.jetbrains.exposed.sql.transactions.transaction
14
15 object EquipmentTable: Table("equipment"), Generable {
16
17     val equip_id = integer("equip_id").autoIncrement().primaryKey()
18     val camouflage = text("camouflage").nullable()
19     val communication = text("communication").nullable()
20     val intelligence = text("intelligence").nullable()
21     val medical = text("medical").nullable()
22     val mre_id = reference("mre_id", MRETable.mre_id, onDelete = ReferenceOption.RESTRICT)
23     val extra = text("extra").nullable()
24
25     override fun generateAndInsert(n: Int) {
26         val camouflages = setOf("black", "khaki", "olive", "fleckerteppich", "strichtarn", "cce", "vegetata",
27             "flora", "forest", "pixel", "woodland", "accupat", "desert", "city")
28         val communications = setOf("radio_set", "map", "mobile_satellite", "signal_flares")
29         val intelligences = setOf("binoculars", "drone", "radar", "thermal_visor")
30         val medicals = setOf("bandage", "harness", "antibiotics", "alcohol", "scissors", "tweezers", "antiseptic",
31             "ammonia", "nitroglycerine", "adrenalin")
32         val mreIds = MRETable.selectAll().limit(n).map { it[MRETable.mre_id] }
33         (1..n).forEach {
34             EquipmentTable.insert {
35                 it[camouflage] = camouflages.random()
36                 it[communication] = communications.random()
37                 it[intelligence] = intelligences.random()
38                 it[medical] = medicals.random()
39                 it[mre_id] = mreIds.random()
40             }
41         }
42     }
43 }
44
45 fun ResultRow.toEquipment() = Equipment(
46     this[EquipmentTable.equip_id],
47     this[EquipmentTable.camouflage],
48     this[EquipmentTable.communication],
49     this[EquipmentTable.intelligence],
50     this[EquipmentTable.medical],
51     this[EquipmentTable.mre_id],
52     this[EquipmentTable.extra])
53
54 data class Equipment(val equipId: Int?,
55                     val camouflage: String?,
56                     val communication: String?,
57                     val intelligence: String?,
58                     val medical: String?,
59                     val mreId: Int,
60                     val extra: String?)
61
62 fun Route.equipment() {
63
64     get {
65         val (t, s) = try {
66             call.parameters["ids"]!!.let {
67                 transaction {
68                     PARSER.toJson(getRecordsWithIds(it, EquipmentTable).map { it.toEquipment() }) to
69                     HttpStatusCode.OK
70                 }
71             } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
72             call.respondText(text = t, status = s)
73     }
74 }

```

Equipment.kt

```

73    }
74
75    put {
76        val (t, s) = try {
77            val raw = call.receiveText()
78            val (id, f) = explodeJsonForModel("equipId", raw)
79            transaction {
80                EquipmentTable.update({ EquipmentTable.equip_id eq id }) { e ->
81                    f["camouflage"]?.let { e[camouflage] = it as String }
82                    f["communication"]?.let { e[communication] = it as String }
83                    f["intelligence"]?.let { e[intelligence] = it as String }
84                    f["medical"]?.let { e[medical] = it as String }
85                    f["mreId"]?.let { e[mre_id] = it as Int }
86                    f["extra"]?.let { e[extra] = it as String }
87                }
88            }
89            "$raw updated)" to HttpStatusCode.OK
90        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
91        call.respondText(text = t, status = s)
92    }
93
94    post {
95        val (t, s) = try {
96            val raw = call.receiveText()
97            val e = PARSE.fromJson(raw, Equipment::class.java)
98            transaction {
99                EquipmentTable.insert {
100                    it[camouflage] = e.camouflage
101                    it[communication] = e.communication
102                    it[intelligence] = e.intelligence
103                    it[medical] = e.medical
104                    it[mre_id] = e.mreId
105                    it[extra] = e.extra
106                }
107            }
108            "$raw added)" to HttpStatusCode.Created
109        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
110        call.respondText(text = t, status = s)
111    }
112
113    delete {
114        val (t, s) = try {
115            val droppedIds = call.receiveText()
116            transaction { dropRecordsWithIds(droppedIds, EquipmentTable) }
117            "Equipments with $droppedIds deleted)" to HttpStatusCode.OK
118        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
119        call.respondText(text = t, status = s)
120    }
121 }

```

```

1 package com.testpassword.models
2
3 import com.testpassword.PARSER
4 import com.testpassword.dropRecordsWithIds
5 import com.testpassword.explodeJsonForModel
6 import com.testpassword.getRecordsWithIds
7 import io.ktor.application.*
8 import io.ktor.http.*
9 import io.ktor.request.*
10 import io.ktor.response.*
11 import io.ktor.routing.*
12 import org.jetbrains.exposed.sql.*
13 import org.jetbrains.exposed.sql.transactions.transaction
14 import org.json.JSONObject
15 import java.io.File
16
17 object TransportTable: Table("transport"), Generable {
18
19     val trans_id = integer("trans_id").autoIncrement().primaryKey()
20     val name = text("name")
21     val type = text("type")
22     val status = text("status")
23
24     override fun generateAndInsert(n: Int) {
25         val statuses = setOf("available", "under_repair", "destroyed", "broken")
26         val rawData = File("resources/static/transports.json").readLines().joinToString(separator = "")
27         val jsonBody = JSONObject(rawData).getJSONArray("transport")
28         jsonBody.forEachIndexed { i, el ->
29             val w = jsonBody.getJSONObject(i)
30             TransportTable.insert {
31                 it[name] = w.getString("name")
32                 it[type] = w.getString("type")
33                 it[status] = "available"
34             }
35         }
36     }
37 }
38
39 fun ResultRow.toTransport() = Transport(
40     this[TransportTable.trans_id],
41     this[TransportTable.name],
42     this[TransportTable.type],
43     this[TransportTable.status])
44
45 data class Transport(val transId: Int?,
46                      val name: String,
47                      val type: String,
48                      val status: String)
49
50 fun Route.transport() {
51
52     get {
53         val (t, s) = try {
54             call.parameters["ids"]!!.let {
55                 transaction {
56                     PARSER.toJson(getRecordsWithIds(it, TransportTable).map { it.toTransport() }) to
57                     HttpStatusCode.OK
58                 }
59             } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
60             call.respondText(text = t, status = s)
61         }
62
63         put {
64             val (t, s) = try {
65                 val raw = call.receiveText()
66                 val (id, f) = explodeJsonForModel("transId", raw)
67                 TransportTable.update({ TransportTable.trans_id eq id }) { t ->
68                     f["name"]?.let { t[name] = it as String }
69                     f["type"]?.let { t[type] = it as String }
70                     f["status"]?.let { t[status] = it as String }
71                 }
72                 "$raw updated)" to HttpStatusCode.OK
73             } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
74             call.respondText(text = t, status = s)
75         }
76     }
77 }

```

Transport.kt

```
75    }
76
77    post {
78        val (t, s) = try {
79            val raw = call.receiveText()
80            val t = PARSER.fromJson(raw, Transport::class.java)
81            transaction {
82                TransportTable.insert {
83                    it[name] = t.name
84                    it[type] = t.type
85                    it[status] = t.status
86                }
87            }
88            "$raw added)" to HttpStatusCode.Created
89        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
90        call.respondText(text = t, status = s)
91    }
92
93    delete {
94        val (t, s) = try {
95            val droppedIds = call.receiveText()
96            transaction { dropRecordsWithIds(droppedIds, TransportTable) }
97            "Transports with ids $droppedIds deleted)" to HttpStatusCode.OK
98        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
99        call.respondText(text = t, status = s)
100    }
101 }
```

```

1 package com.testpassword.models
2
3 import com.testpassword.PARSER
4 import com.testpassword.dropRecordsWithIds
5 import com.testpassword.explodeJsonForModel
6 import com.testpassword.getRecordsWithIds
7 import io.ktor.application.*
8 import io.ktor.http.*
9 import io.ktor.request.*
10 import io.ktor.response.*
11 import io.ktor.routing.*
12 import org.jetbrains.exposed.sql.*
13 import org.jetbrains.exposed.sql.transactions.transaction
14
15 object MedicalCardTable: Table("medical_card"), Generable {
16
17     val med_id = integer("med_id").autoIncrement().primaryKey()
18     val emp_id = reference("emp_id", EmployeeTable.emp_id, onDelete = ReferenceOption.CASCADE)
19     val height_cm = integer("height_cm")
20     val weight_kg = integer("weight_kg")
21     val diseases = text("diseases").nullable()
22     val blood = text("blood")
23     val gender = bool("gender")
24 }
25
26 fun ResultRow.toMedicalCard() = MedicalCard(
27     this[MedicalCardTable.med_id],
28     this[MedicalCardTable.emp_id],
29     this[MedicalCardTable.height_cm],
30     this[MedicalCardTable.weight_kg],
31     this[MedicalCardTable.diseases],
32     this[MedicalCardTable.blood],
33     this[MedicalCardTable.gender])
34
35 data class MedicalCard(val medId: Int?,
36                         val empId: Int,
37                         val heightCm: Int,
38                         val weightKg: Int,
39                         val diseases: String?,
40                         val blood: String,
41                         val gender: Boolean)
42
43 fun Route.medicalCard() {
44
45     get {
46         val (t, s) = try {
47             call.parameters["ids"]!!.let {
48                 transaction {
49                     PARSER.toJson(getRecordsWithIds(it, MedicalCardTable).map { it.toMedicalCard() }) to
50                     HttpStatusCode.OK
51                 }
52             } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
53             call.respondText(text = t, status = s)
54         }
55
56         put {
57             val (t, s) = try {
58                 val raw = call.receiveText()
59                 val (id, f) = explodeJsonForModel("medId", raw)
60                 transaction {
61                     MedicalCardTable.update({ MedicalCardTable.med_id eq id }) { m ->
62                         f["emp_id"]?.let { m[emp_id] = it as Int }
63                         f["heightCm"]?.let { m[height_cm] = it as Int }
64                         f["weightKg"]?.let { m[weight_kg] = it as Int }
65                         f["diseases"]?.let { m[diseases] = it as String }
66                         f["blood"]?.let { m[blood] = it as String }
67                         f["gender"]?.let { m[gender] = it as Boolean }
68                     }
69                 }
70                 "$raw updated)" to HttpStatusCode.OK
71             } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
72             call.respondText(text = t, status = s)
73         }
74     }

```

MedicalCard.kt

```
75    post {
76        val (t, s) = try {
77            val raw = call.receiveText()
78            val m = PARSER.fromJson(raw, MedicalCard::class.java)
79            transaction {
80                MedicalCardTable.insert {
81                    it[emp_id] = m.empId
82                    it[height_cm] = m.heightCm
83                    it[weight_kg] = m.weightKg
84                    it[diseases] = m.diseases
85                    it[blood] = m.blood
86                    it[gender] = m.gender
87                }
88            }
89            "$raw added)" to HttpStatusCode.Created
90        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
91        call.respondText(text = t, status = s)
92    }
93
94    delete {
95        val (t, s) = try {
96            val droppedIds = call.receiveText()
97            transaction { dropRecordsWithIds(droppedIds, MedicalCardTable) }
98            "Cards with $droppedIds deleted)" to HttpStatusCode.OK
99        } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
100       call.respondText(text = t, status = s)
101    }
102 }
```

```

1 package com.testpassword.models
2
3 import com.github.javafaker.Faker
4 import io.ktor.util.*
5 import org.jetbrains.exposed.sql.*
6 import org.jetbrains.exposed.sql.`java-time`.CurrentDateTime
7 import org.jetbrains.exposed.sql.`java-time`.date
8 import org.postgresql.util.PGobject
9 import java.time.LocalDate
10 import java.time.ZoneId
11 import java.util.*
12 import kotlin.random.Random
13
14 /*
15 https://stackoverflow.com/questions/45723803/how-to-use-postgresql-enum-type-via-kotlin-exposed-orm
16 https://blog.jdriven.com/2019/07/kotlin-exposed-a-lightweight-sql-library/
17 */
18
19 val F = Faker() // random data generator
20
21 class PGEnum<T : Enum<T>>(enumTypeName: String, enumValue: T?) : PGobject() {
22     init {
23         value = enumValue?.name
24         type = enumTypeName
25     }
26 }
27
28 inline fun <reified T : Enum<T>> Table.postgresEnumeration(
29     columnName: String,
30     postgresEnumName: String
31 ) = customEnumeration(columnName, postgresEnumName,
32     { value -> enumValueOf<T>(value as String) }, { PGEnum(postgresEnumName, it) })
33
34 interface Generable { fun generateAndInsert(n: Int = 0) = Unit }
35
36 enum class FORCES { GF, NAVY, AF }
37
38 object WeaponsInEquipment: Table("equip_weapon"), Generable {
39
40     val equip_id = reference("equip_id", EquipmentTable.equip_id)
41     val weapon_id = reference("weapon_id", WeaponTable.weapon_id)
42
43     override fun generateAndInsert(n: Int) {
44         val weaponIds = WeaponTable.selectAll().map { it[WeaponTable.weapon_id] }
45         EquipmentTable.selectAll().map { it[EquipmentTable.equip_id] }.forEach { e ->
46             WeaponsInEquipment.insert {
47                 it[equip_id] = e
48                 it[weapon_id] = weaponIds.random()
49             }
50         }
51     }
52 }
53
54 object TransportOnMissions: Table("missions_transport"), Generable {
55
56     val miss_id = reference("miss_id", MissionTable.miss_id)
57     val trans_id = reference("trans_id", TransportTable.trans_id)
58
59     override fun generateAndInsert(n: Int) {
60         val transIds = TransportTable.select { TransportTable.status eq "available" }.map { it[
61             TransportTable.trans_id] }
62         MissionTable.selectAll().map { it[MissionTable.miss_id] }.forEach { m ->
63             if (Random.nextBoolean())
64                 TransportOnMissions.insert {
65                     it[miss_id] = m
66                     it[trans_id] = transIds.random()
67                 }
68         }
69     }
70
71 object Inspection: Table("inspection"), Generable {
72
73     val emp_id = reference("emp_id", EmployeeTable.emp_id)
74     val trans_id = reference("trans_id", TransportTable.trans_id)

```

```

75     val service_date = date("service_date").defaultExpression(.currentTimeMillis().date())
76
77     @InternalAPI override fun generateAndInsert(n: Int) {
78         val transIds = TransportTable.selectAll().map { it[TransportTable.trans_id] }
79         EmployeeTable.leftJoin(PositionTable)
80             .slice(EmployeeTable.emp_id, PositionTable.name)
81             .select { PositionTable.name inList listOf("mechanic", "engineer") }
82             .map { it[EmployeeTable.emp_id] }
83             .forEach { e ->
84                 if (Random.nextBoolean())
85                     Inspection.insert {
86                         it.emp_id = e
87                         it[trans_id] = transIds.random()
88                         it[service_date] = F.date().between(
89                             Date.from(LocalDate.of(2014, 3, 18).atStartOfDay(ZoneId.systemDefault())),
90                             Date(),
91                             ).toLocalDateTime().toLocalDate()
92                     }
93             }
94     }
95 }
96
97 object EmployeeOnMission: Table("missions_emp"), Generable {
98
99     val miss_id = reference("miss_id", MissionTable.miss_id)
100    val emp_id = reference("emp_id", EmployeeTable.emp_id)
101
102    override fun generateAndInsert(n: Int) {
103        val missIds = MissionTable.selectAll().map { it[MissionTable.miss_id] }
104        EmployeeTable
105            .leftJoin(PositionTable)
106            .select { PositionTable.rank neq null }.map { it[EmployeeTable.emp_id] }
107            .forEach { e ->
108                EmployeeOnMission.insert {
109                    it[miss_id] = missIds.random()
110                    it[emp_id] = e
111                }
112            }
113    }
114 }
115
116
117
118 /*
119 https://touk.pl/blog/2019/02/12/how-we-use-kotlin-with-exposed-at-touk/
120 https://ryanharrison.co.uk/2018/04/14/kotlin-ktor-exposed-starter.html
121 https://hashrocket.com/blog/posts/faster-json-generation-with-postgresql
122 https://caelis.medium.com/ktor-send-and-receive-json-6c41c64410af
123 */

```

GodRoute.kt

```

1 package com.testpassword.routes
2
3 import com.testpassword.*
4 import com.testpassword.models.*
5 import io.ktor.application.*
6 import io.ktor.html.*
7 import io.ktor.http.*
8 import io.ktor.response.*
9 import io.ktor.routing.*
10 import kotlincs.html.*
11 import org.jetbrains.exposed.sql.SchemaUtils
12 import org.jetbrains.exposed.sql.transactions.transaction
13
14 fun Route.god() {
15
16     val GOD_URL = "localhost:9090/god" //TODO: убрать захардкоженый урл
17
18     get {
19         call.respondHtml {
20             body {
21                 h1 { +"GOD-MODE" }
22                 ul {
23                     li {
24                         form(action = GOD_URL, method = FormMethod.put) {
25                             button(type = ButtonType.submit) {
26                                 +"PUT to create tables before fill"
27                             }
28                         }
29                     }
30                     li {
31                         form(action = GOD_URL, method = FormMethod.post) {
32                             button(type=ButtonType.submit) {
33                                 + "POST to fill tables"
34                             }
35                         }
36                     }
37                     li {
38                         form(action = GOD_URL, method = FormMethod.delete) {
39                             button(type = ButtonType.submit) {
40                                 + "DELETE to drop table and all data"
41                             }
42                         }
43                     }
44                 }
45             }
46         }
47     }
48
49     put {
50         transaction {
51             SchemaUtils.create(
52                 BaseTable, MRETable, EquipmentTable, PositionTable, EmployeeTable, MedicalCardTable,
53                 WeaponTable,
54                 CampaignTable, MissionTable, TransportTable, WeaponsInEquipment, TransportOnMissions,
55                 Inspection,
56                 EmployeeOnMission
57             )
58         }
59     }
60     post {
61         val (t, s) = try {
62             transaction {
63                 fillDb(
64                     mapOf(
65                         BaseTable to 50,
66                         MRETable to 30,
67                         EquipmentTable to 75,
68                         PositionTable to 50,
69                         EmployeeTable to 1000,
70                         MedicalCardTable to 0,
71                         WeaponTable to 0,
72                         CampaignTable to 40,
73                         MissionTable to 110,
74                         TransportTable to 0,
75                     )
76                 )
77             }
78         } catch (e: Exception) {
79             e.printStackTrace()
80         }
81     }
82 }

```

GodRoute.kt

```
74             WeaponsInEquipment to 0,
75             TransportOnMissions to 0,
76             Inspection to 0,
77             EmployeeOnMission to 0
78         )
79     )
80 }
81 "Records generated" to HttpStatusCode.OK
82 } catch (e: Exception) { e.toString() to HttpStatusCode.BadRequest }
83 call.respondText(text = t, status = s)
84 }
85
86 delete {
87     transaction {
88         SchemaUtils.drop(
89             BaseTable, MRETable, EquipmentTable, PositionTable, EmployeeTable, MedicalCardTable,
90             WeaponTable,
91             CampaignTable, MissionTable, TransportTable, WeaponsInEquipment, TransportOnMissions,
92             Inspection,
93             EmployeeOnMission
94     )
95 }
```