

Tablegen

Batix Tablegen

Tablegen ist ein leistungsfähiges Tool zur Code-basierten Definition von Datenbank-Containern im CMS-System. Es ermöglicht eine reproduzierbare und wartbare Erstellung von Datenbankstrukturen direkt aus dem Quellcode heraus.

Warum Tablegen?

Die manuelle Erstellung von Containern über die CMS-Oberfläche hat einige Nachteile:

- Der Prozess ist zeitaufwändig und fehleranfällig
- Die Konfiguration lässt sich schwer für Live-Deployments reproduzieren
- Änderungen sind schlecht nachvollziehbar

Tablegen löst diese Probleme durch einen Code-first Ansatz:

- Container werden durch Kotlin-Klassen definiert
- Die Struktur ist im Code versioniert
- Deployment-Prozesse sind idempotent und automatisierbar

Installation

Maven Repository einbinden

Zunächst das zusätzliche Batix Maven Repository (wie [hier](#) erwähnt) in die `build.gradle.kts` einfügen:

plugins/[plugin-name]/build.gradle.kts

```
repositories {  
    // ...  
  
    maven {
```

```

// https://git.batix.gmbh/maven-packages/registry
url = uri("https://git.batix.gmbh/api/v4/projects/322/packages/maven")

authentication {
    create<HttpHeaderAuthentication>("header")
}

if (!System.getenv("CI_JOB_TOKEN").isNullOrEmpty()) {
    credentials(HttpHeaderCredentials::class) {
        name = "Job-Token"
        value = System.getenv("CI_JOB_TOKEN")
    }
} else {
    credentials(HttpHeaderCredentials::class) {
        name = "Private-Token"
        value = property("batix.gitlab.pat").toString()
    }
}
}
}

```

Abhängigkeit hinzufügen

Im Anschluss die Dependency ergänzen (die neueste Version ist [hier](#) zu sehen): Der vorangestellte Kommentar vereinfacht Versionsupdates.

plugins/[plugin-name]/build.gradle.kts

```

dependencies {
    // ...

    // https://git.batix.gmbh/maven-packages/registry/-/packages/909
    implementation("com.batix.table:tablegen:2.1.1")
}

```

Verwendung

Die Containerdefinition erfolgt über annotierte Kotlin-Klassen. Hier ein Beispiel für einen Personen-Container:

plugins/Personen/src/main/kotlin/com.batix.personen/Person.kt { .code-title }

```

package com.batix.personen

// import...

// nur die `@BatixContainer` Annotation ist notwendig,
// der Rest ist optional.
@BatixContainer("bxc_person", "Person")
@ContainerCategory("Personen")
@ContainerDesignation("Andere Anzeigename")
@ContainerDescription("In diesem Container werden Personen gespeichert")
@ContainerPattern("%name%")
class Person: Entity {
    constructor() : super()

    @SingleLineStringField(name = Fields.TITLE, length = 10)
    var titel: String? = null

    @SingleLineStringField(name = Fields.NAME, length = 100, encrypt = false, multiline = true,
unique = true)
    var name: String? = null

    @IntField(Fields.AGE, IntFieldType.TINY, true)
    var age: Int? = null

    companion object {
        object Fields {
            const val TITLE = "title"
            const val NAME = "name"
            const val AGE = "age"
        }
    }
}

```

Wichtige Annotations

Tablegen stellt verschiedene Annotations zur Verfügung, um Container und Felder zu definieren. Die wichtigsten sind:

- `@BatixContainer` : Definiert einen Container mit Namen und Titel
- `@ContainerCategory` : Kategorie des Containers
- `@SingleLineStringField` / `@MultiLineStringField` : Ein-/Mehrzeiliges Textfeld

- `@IntField` : Ganzzahlfeld
- `@SingleLinkField` / `MultipleLinkField` : Verknüpfungen zu anderen Containern

Jedes Feld kann durch zusätzliche Annotations wie `@FieldDesignation` oder `@FieldDescription` näher beschrieben werden.

Die vollständige API kann in der [Readme](#) nachvollzogen werden.

Container initialisieren

Die Container-Initialisierung erfolgt typischerweise in der `load()`-Methode des Plugins:

plugins/Personen/src/main/kotlin/com.batix.personen/Plugin.kt `{.code-title}`

```
package com.batix.personen

// import...

class ImportPlugin: Plugin() {
    override fun load() {
        // init Logik...

        val ti = TableInitializer().registerContainerClass<Person>()
        ConnectionPool.withSystemConnection { conn ->
            ti.initializeTables(conn)
        }
    }
}
```

“ Info

Die `TableInitializer`- Klasse unterstützt chaining, um die Registrierung vieler Container auf einmal zu vereinfachen.

```
// Registrierung der Entity-Klassen
val i = TableInitializer()
    .registerContainerClass<Person>()
    .registerContainerClass<Street>()
    .registerContainerClass<Mitarbeiter>()
```

Laden von Objekten aus der Datenbank

Tablegen bietet verschiedene Möglichkeiten, um Objekte aus aus `com.batix.table.ContainerRecords` oder `java.sql.ResultSet` zu laden. Die Basisklasse `Entity` stellt dafür mehrere Konstruktoren bereit.

“ Hinweis

Zur Vereinfachung wird in Folgenden Beispielen der Kontext

```
class ImportPlugin: Plugin() {
    override fun load() {
        □ ConnectionPool.withSystemConnection { conn ->
        □     // Codebeispiel...
        }
    }
}
```

bei **Plugin.kt** weggelassen.

Mit Entity-Klasse

Eine typische Entity-Klasse implementiert folgende Konstruktoren. Es sollte aber immer mindestens der parameterlose Konstruktor `constructor() : super()` definiert sein, damit die automatische Felderzuweisung korrekt funktioniert.

Person.kt

```
class Person : Entity {
    constructor() : super()

    // Konstruktor für ContainerRecord
    constructor(record: ContainerRecord, conn: Connection) : super(record, conn)

    // Konstruktor für ResultSet
    constructor(rs: ResultSet, suffix: String, conn: Connection) : super(rs, suffix, conn)

    // Felddefinitionen...
}
```

Der `ContainerRecord`-Konstruktor weist dabei automatisch alle Felder aus dem Record dem Objekt zu. Eager-geladene `SingleLinkObjects` werden sofort initialisiert.

Initialisierung in `load()`:

Plugin.kt

```
// val id = ...
conn.prepareNamedParameterStatement("SELECT p.* FROM bxc_person p WHERE p.id = :id").use {
    stmt ->
        stmt.setInt("id", id)
        stmt.executeQuery().use { rs ->
            if (rs.next()) {
                val person = Person(rs, "p", conn)
                logI("Person: $person ${person.title} ${person.name}")
            }
        }
    }
}
```

Komplexe Abfragen mit JOINS

Für komplexere Abfragen mit verknüpften Objekten kann ein SQL-Statement mit Aliassen verwendet werden (hier mit unterobjekt `Adress`):

Plugin.kt

```
// val id = ...

// Lädt Person mit verknüpfter Adresse in einem Query
val sql = """
    SELECT p.*, a.*
    FROM bxc_person p
    LEFT JOIN bxc_address a ON p.address = a.id
    WHERE p.id = :id
"""

conn.prepareNamedParameterStatement(sql).use { stmt ->
    stmt.setInt("id", id)
    stmt.executeQuery().use { rs ->
        if (rs.next()) {
            // Lädt Person und Adresse aus einem ResultSet
        }
    }
}
```

```

        val person = Person(rs, "p", "a", conn)
    }
}

```

Laden einzelner Objekte

Die Klasse `BatixRepository` bietet Methoden um einzelne Objekte oder Records aus der Datenbank zu laden:

Plugin.kt

```

// val id = ...
val person = BatixRepository.getById<Person>(id, conn)

```

Speichern von Objekten

Zum Speichern von Änderungen bietet Tablegen die `setRecordFields()`-Methode.

Alle Felder speichern

Ein neuer Konstruktor zur Erstellung von Person Objekten:

Person.kt

```

constructor( id: String, titel: String, name: String, age: Int, active: Boolean ) : super() {
    this.id = id
    this.titel = titel
    this.name = name
    this.age = age
    this.active = active
}

```

Plugin.kt

```

// unique ID aus Datum/Uhrzeit zur korrekten Anzeige
// des Erstellungsdatums des Datenbankeintrags
val id = UniqueID.createHexId() // com.batix.util.UniqueID
val person = Person(
    id = id,

```

```

    titel = "Dr.",
    name = "Beatrix Batixson",
    age = 42,
    active = true,
)

val table = BatixRepository.getTable<Person>(conn)
var record = table.readRecord(person.id, conn)

// Auf vorhandenen Datensatz zugreifen oder neuen erstellen
if (record == null) {
    record = table.createRecord(conn, person.id, person.active)
    person.setRecordFields(record)
    record.createRecordInDatabase(conn)
} else {
    person.setRecordFields(record)
    record.updateRecordInDatabase(conn)
}

```

Selektives Speichern

Es können auch nur ausgewählte Felder aktualisiert werden:

```

// Nur Name und Alter aktualisieren
person.setRecordFields(record, listOf(
    Person.Fields.NAME,
    Person.Fields.AGE
))
record.updateRecordInDatabase()

```

Codegenerierung für bestehende Container

Für die Migration bestehender Container zu Tabellen steht ein Groovy-Script zur Verfügung. Dieses generiert die Entity-Klassen aus der Datenbankstruktur.

Script ausführen

1. Das [Codegenerator-Script](#) in die Groovy-Konsole des CMS (Entwicklertools → Groovy) kopieren
2. Tabellen für die Generierung eintragen:

CMS Groovy-Konsole

```
def tables = [  
    "bxc_person",  
    "bxc_address"  
    // weitere Tabellen...  
]
```

3. Script ausführen - die generierten Klassen können als Basis für die Tablegen-Implementierung verwendet werden

⚠ Achtung

Die generierten Klassen sollten manuell überprüft und ggf. angepasst werden. Insbesondere sollten:

- Feldtypen und -eigenschaften validiert werden
- Verknüpfungen korrekt konfiguriert werden
- Zusätzliche Validierungen ergänzt werden

Revision #1

Created 9 May 2025 06:19:10 by Batix

Updated 9 May 2025 06:19:10 by Batix