

Anwendung der funktionalen Programmierung mit Scala

TH Rosenheim - SoSe 2025



Play Framework - Part 2 Datenbankzugriff



Play Framework Bausteine der Datenbankintegration

- Schema-Verwaltung: Das Datenbankschema wird versioniert gepflegt, damit Änderungen nachvollziehbar und automatisiert ausrollbar sind => Flyway
- Verbindung: Die Anwendung nutzt einen performanten Connection Pool, um effizient und nebenläufig auf die Datenbank zuzugreifen => HikariCP (via Slick)
- **SQL-Zugriff**: Für typsicheren und funktionalen Zugriff auf die Datenbank verwenden wir eine Scala-native API mit Unterstützung für Mapping, Queries und Streaming => **Slick**



Flyway



Flyway Datenbankschema-Migrationen

- Warum Migrationen? Bei der Entwicklung ändern sich DB-Schemata. Ein Migrationstool wie Flyway ermöglicht es, SQL-Skripte versioniert abzulegen und bei App-Start automatisch auszuführen.
- Migrationstypen: Es gibt versionierte Migrationen (Prefix V) und Repeatable-Migrationen (Prefix R). Versionierte Skripte (z.B. V1__init.sql) werden in aufsteigender Reihenfolge einmalig ausgeführt. Repeatable-Skripte (R__<name>.sql) haben keine feste Versionsnummer und werden bei jedem Flyway-Migrate erneut ausgeführt, sofern sich ihr Inhalt geändert hat
- **Beispiel**: V1__init.sql legt das initiale Schema an Tabellen, V2__add_index.sql fügt einen Index hinzu. Ein R__refresh_views.sql könnte z.B. definieren, wie Materialized Views neu erstellt werden, dieses läuft bei jeder Migration



Flyway Beispiel Migrationen

```
-- V001__init.sql

CREATE TABLE todo
(

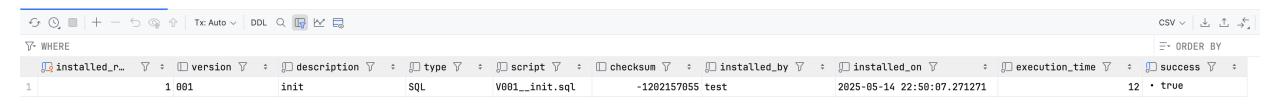
id SERIAL PRIMARY KEY,
title TEXT NOT NULL,
description TEXT NOT NULL,
done BOOLEAN NOT NULL
)
```

```
-- R__refresh_completed.sql
CREATE OR REPLACE VIEW completed_todos AS
SELECT * FROM todo WHERE done = TRUE;
```

Wichtig: Doppelter Unterstrich nach V# / R



Flyway Migrationshistorie



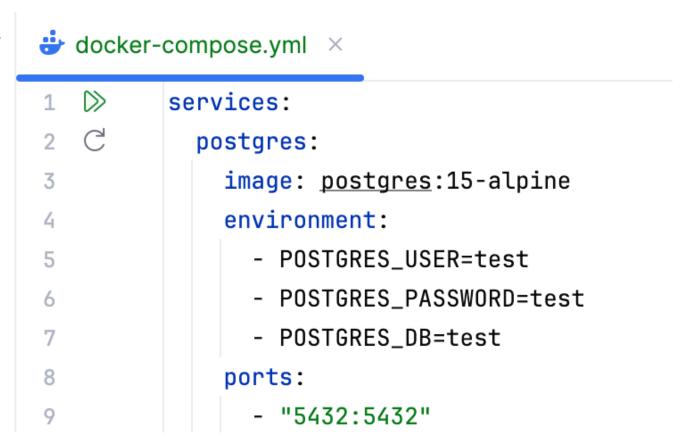
- Zentrale Steuerung: Flyway legt in jeder Datenbank eine eigene Tabelle (flyway_schema_history) an, um den Zustand aller durchgeführten Migrationen zu verwalten
- Schutz vor Doppel-Ausführung: Jede Migration wird mit Version, Dateiname, Prüfsumme und Ausführungszeitpunkt erfasst, so verhindert Flyway versehentliche Wiederholungen
- Validierung & Sicherheit: Beim Start prüft Flyway, ob lokale Skripte mit der History übereinstimmen (Checksum-Validation), verhindert unbemerkte manuelle Änderungen am Schema



Flyway Lokale Datenbank über Docker Compose

Für die lokale Entwicklung verwenden wir Docker Compose

=> https://docs.docker.com/compose/





Flyway Dependencies

build.sbt

```
val flywayVersion = "11.8.2"
lazy val root = (project in file("."))
  .enablePlugins(PlayScala)
  .settings(
   name := """play-framework""",
   version := "1.0-SNAPSHOT",
   scalaVersion := "3.7.0",
   // Notwendig da sonst ein Versionskonflikt besteht
   dependencyOverrides += "com.fasterxml.jackson.core" % "jackson-databind" % "2.14.3",
   libraryDependencies ++= Seq(
     guice,
     "io.github.iltotore" %% "iron" % "3.0.1",
     // Postgres JDBC (Java Database Connectivity) Driver
     "org.postgresql" % "postgresql" % "42.7.5",
     // Flyway
     "org.flywaydb" % "flyway-core"
                                  % flywayVersion,
     "org.flywaydb" % "flyway-database-postgresql" % flywayVersion
```



Flyway Konfiguration der Datenbankverbindung

• HOCON erlaubt einen Fallback auf Umgebungsvariablen zu definieren durch Syntax: \${?NAME}

conf/db.conf

```
db {
    default {
        driver = "org.postgresql.Driver"
        slickDriver = "slick.jdbc.PostgresProfile"
        host = "localhost"
        host = ${?DB_HOST}
        port = "5432"
        port = ${?DB_PORT}
        name = "test"
        name = ${?DB_NAME}
        url = "jdbc:postgresql://"${db.default.host}":"${db.default.port}"/"${db.default.name}""
        username = "test"
        username = ${?DB_USERNAME}
        password = "test"
        password = "test"
        password = ${?DB_PASSWORD}
    }
}
```

conf/application.conf

```
# ...
include "db.conf"
# ...
```



Flyway FlywayMigrator

app/FlywayMigrator.scala

```
import com.google.inject.{Inject, Singleton}
import org.flywaydb.core.Flyway
import play.api.Configuration
aSingleton
class FlywayMigrator @Inject() (config: Configuration) {
  // DB-Verbindungsdaten aus der Configuration lesen
  private val url: String = config.get[String]("db.default.url")
  private val user: String = config.get[String]("db.default.username")
  private val pass: String = config.get[String]("db.default.password")
  // Flyway konfigurieren und Migrationen ausführen
  Flvwav
    .configure()
    .dataSource(url, user, pass)
    .locations("classpath:db/migration")
    .load()
    .migrate()
```



Flyway IntegrationFlywayMigrator beim Start ausführen

Module.scala

```
class Module extends AbstractModule {
  override def configure() = {
    bind(classOf[Clock]).toInstance(Clock.systemDefaultZone)

    // FlywayMigrator direkt bei Start der Application instanziieren → Migrationen ausführen bind(classOf[FlywayMigrator]).asEagerSingleton()
  }
}
```



Slick



Slick Was ist Slick?

- Typsicherer SQL-Zugriff: Slick ("Scala Language-Integrated Connection Kit") erlaubt es, SQL-artige Abfragen direkt in Scala zu formulieren (mit statischer Typprüfung zur Compilezeit)
- Functional Relational Mapping (FRM): Statt klassischem ORM verfolgt Slick einen funktionalen Ansatz, Tabellen werden als Collections behandelt, auf denen man mit map, filter, flatMap arbeitet
- Asynchron & erweiterbar: Alle DB-Operationen sind nicht-blockierend (über Future), lassen sich transaktional kombinieren (DBIO) und bei Bedarf auch streamen (mit Pekko)



Slick Ablauf eines Datenbankzugriffs

- **TableQuery**: Repräsentiert eine Tabelle als typsichere Scala-Collection. Abfragen erfolgen z. B. über filter, map, sortBy. (Vergleiche Seq)
- **DBIOAction**: Auf einer Query kann man z.B. *.result*, *.insert*, *.update* etc. aufrufen. Das ergibt eine Aktion, aber noch keine Ausführung.
- **Database**: Stellt die eigentliche Verbindung zur Datenbank dar, wird z. B. über Guice in der Module.scala erzeugt und bereitgestellt
- **db.run(...)**: Führt die DBIOAction wirklich aus und liefert das Ergebnis als Future[...] zurück => der Zugriff ist also immer asynchron



Slick Beispiel

```
case class Todo(id: Long, title: String, done: Boolean)
class TodoTable(tag: Tag) extends Table[Todo](tag, "todos") {
  def id = column[Long]("id", O.PrimaryKey, O.AutoInc)
  def title = column[String]("title")
  def done = column[Boolean]("done")
  def * = (id, title, done) <> (Todo.tupled, Todo.unapply)
val todos = TableQuery[TodoTable]
// Lade ein Todo mit bestimmter Id
def getById(id: Long): Future[Option[Todo]] =
  db.run(todos.filter(_.id == id).result.headOption)
```



Slick Definiton der TableQueries

```
class TodoTable(tag: Tag) extends Table[Todo](tag, "todos") {
    def id = column[Long]("id", O.PrimaryKey, O.AutoInc)
    def title = column[String]("title")
    def done = column[Boolean]("done")
    def * = (id, title, done) 	> (Todo.tupled, Todo.unapply)
}
val todos = TableQuery[TodoTable]
```

- TableQuery-Klassen: In Slick definieren wir Tabellen als Klassen mit explizitem Mapping für jede Spalte. Das ist präzise, aber bei großen Schemata sehr aufwändig und fehleranfällig
- Wartungsaufwand: Jede Schemaänderung (z. B. neue Spalte, geänderte Typen) muss manuell im Scala-Code nachgezogen werden => inkonsistenter Code kann zu Laufzeitfehlern führen
- Lösung: Slick bietet ein Codegenerierungs-Tool, das aus einer bestehenden Datenbank automatisch typisierte TableQuery-Klassen erzeugt => Slick Codegen



Slick Dependencies

build.sbt

```
val flywayVersion = "11.8.2"
val slickVersion = "3.6.0"
lazy val root = (project in file("."))
  .enablePlugins(PlayScala)
  .settings(
   name := """play-framework""",
   version := "1.0-SNAPSHOT".
   scalaVersion := "3.7.0",
   // Notwendig da sonst ein Versionskonflikt besteht
   dependencyOverrides += "com.fasterxml.jackson.core" % "jackson-databind" % "2.14.3",
   Compile / sourceGenerators += slick.taskValue,
   libraryDependencies ++= Seq(
     // Bisherige Dependencies
     // Slick (Functional Relational Mapping)
     "com.typesafe.slick" " slick-codegen" % slickVersion.
     "com.typesafe.slick" " "slick-hikaricp" % slickVersion
```



Slick Database

Module.scala

```
import slick.jdbc.JdbcBackend.Database

class Module extends AbstractModule {
  override def configure() = {
    bind(classOf[Clock]).toInstance(Clock.systemDefaultZone)

    // FlywayMigrator direkt bei Start der Application instanziieren → Migrationen ausführen bind(classOf[FlywayMigrator]).asEagerSingleton()

    // Slick Database Instanz aus der Configuration erzeugen bind(classOf[Database]).toInstance(Database.forConfig("db.default"))
  }
}
```



Übung



Übung SlickTodoRepository

- Ziel der Übung: InMemoryTodoRepository aus der letzten VL/Übung durch eine Slick-Version ersetzen.
- Starter-Code eingecheckt unter https://github.com/innFactory-Classrooms/afps/tree/main/vl07/play
 - 1. Projekt in Intellij öffnen
 - 2. Postgres DB starten (Über Intellij doppelter grüner Pfeil oder "docker-compose up -d")
 - 3. Play App starten und einen Request abschicken => Löst Flyway migrate aus
 - 4. Sicherstellen das Slick Codegen ausgeführt wurde (SBT-Shell "reload" und dann "compile")=> Tables.scala in target/scala-3.7.0/src_managed/main/slick/db/Tables.scala
 - 5. Übungen siehe Kommentare in app/repositories/SlickTodoRepository.scala