

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

Tx: Auto DDL												CSV	↓	↑	↶
WHERE												ORDER BY			
	installed_r...	version	description	type	script	checksum	installed_by	installed_on	execution_time	success					
1		1 001	init	SQL	V001__init.sql	-1202157055	test	2025-05-14 22:50:07.271271	12	• true					

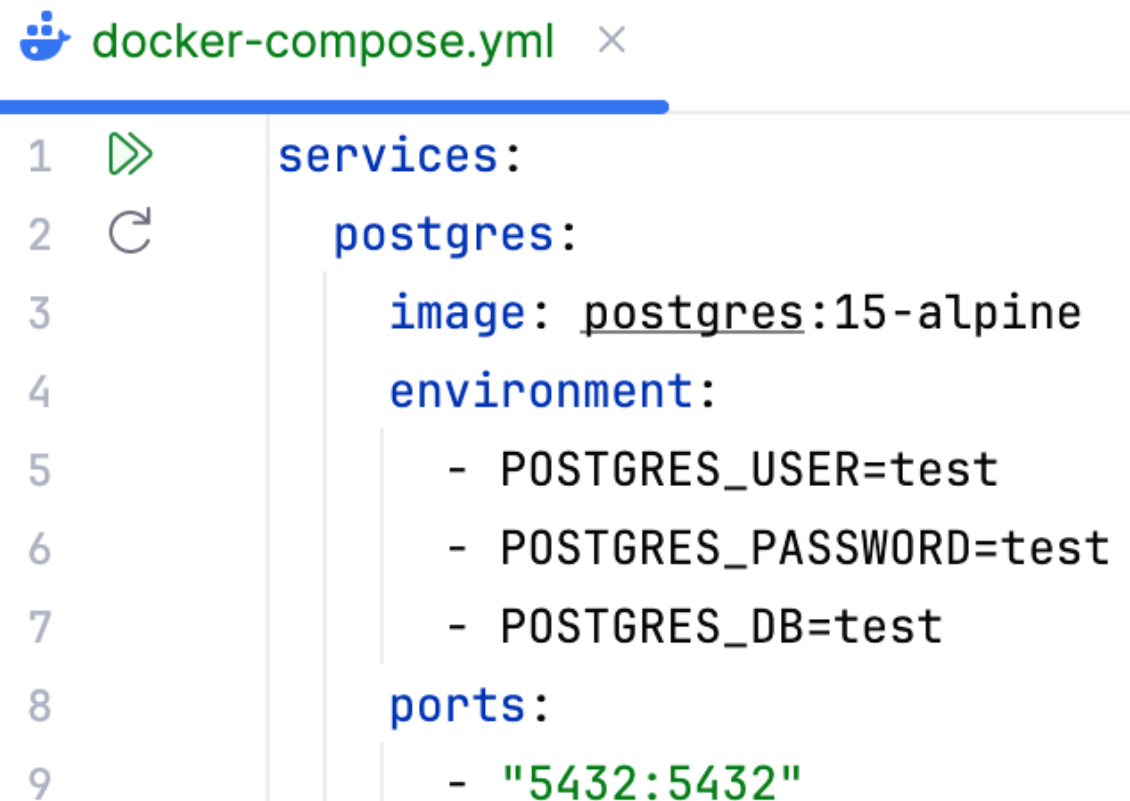
- **Zentrale Steuerung:** Flyway legt in jeder Datenbank eine eigene Tabelle (**flyway_schema_history**) an, um den Zustand aller durchgeführten Migrationen zu verwalten
- **Schutz vor Doppel-Ausführung:** Jede Migration wird mit Version, Dateiname, Prüfsumme und Ausführungszeitpunkt erfasst, so verhindert 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/>

A screenshot of a code editor window titled 'docker-compose.yml'. The editor shows a YAML configuration for a Docker service named 'postgres'. The configuration includes the image 'postgres:15-alpine' and an environment with three variables: 'POSTGRES_USER=test', 'POSTGRES_PASSWORD=test', and 'POSTGRES_DB=test'. The ports section is set to '5432:5432'. Line numbers 1 through 9 are visible on the left side of the editor.

```
services:
  postgres:
    image: postgres:15-alpine
    environment:
      - POSTGRES_USER=test
      - POSTGRES_PASSWORD=test
      - POSTGRES_DB=test
    ports:
      - "5432:5432"
```


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,
      "org.typelevel" %% "cats-core" % "2.13.0",
      "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 = ${?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

@Singleton
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
  Flyway
    .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 instanziiieren → 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", 0.PrimaryKey, 0.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

Joins & Weiterverarbeitung

- **Join-Methoden:** Slick bietet verschiedene Methoden zur Verknüpfung von Tabellen: `join`, `joinLeft`, `joinRight`, `joinFull`, jeweils mit `.on(...)` zur Join-Bedingung.
- **Ergebnis als Tupel:** Der Rückgabewert eines Joins ist ein Tupel (z. B. `(User, Order)` oder `(User, Option[Order])` bei `joinLeft`), das in `.map`-Ausdrücken weiterverarbeitet wird
- **Weiterverarbeitung:** Du kannst auf die Elemente des Tupels per Mustererkennung zugreifen, z. B.
 - `.map { case (user, orderOpt) => ... }`
 - `.map(q => q._1.filter(_id === 1))`

Slick

Gängige Methoden

- **filter(...)** Filtert Zeilen – entspricht WHERE.
Beispiel: `todos.filter(_.done === false)`
- **map(...)** Projiziert einzelne Spalten oder berechnet Werte.
Beispiel: `todos.map(_.title)`
- **sortBy(...)** Sortiert das Ergebnis – entspricht ORDER BY.
Beispiel: `todos.sortBy(_.id.desc)`
- **take(n) / drop(n)** Begrenzen die Anzahl oder setzen Offset, in SQL LIMIT / OFFSET.
Beispiel: `.drop(10).take(5)`
- **length / size / count** Zählt Zeilen – entspricht SELECT COUNT(*).
Beispiel: `todos.length.result`
- **min, max, avg, sum** Aggregatfunktionen auf numerischen Spalten.
Beispiel: `todos.map(_.id).max.result`
- **distinct** Entfernt doppelte Zeilen.
Beispiel: `todos.map(_.title).distinct`

Slick

Monadic Joins (for-Comprehensions)

- Tabellen können wie Collections in einer for-Comprehension kombiniert werden.
- **Einschränkung:** Nur **inner joins** sind erlaubt, keine joinLeft, joinRight, joinFull. Für solche Fälle muss .joinLeft.on(...) außerhalb der for-Comprehension verwendet werden.

```
val query = for {  
  user ← users  
  order ← orders if order.userId === user.id  
} yield (user, order)
```

Slick

Weiterführende Dokumentation

- <https://scala-slick.org/doc/stable/>
- <https://books.underscore.io/essential-slick/essential-slick-3.pdf>

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"           % slickVersion,
      "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 instanziiieren → Migrationen ausführen
    bind(classOf[FlywayMigrator]).asEagerSingleton()

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

Übung

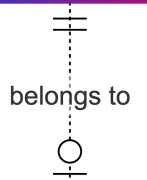
Übung

SlickTodoRepository



- 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 FlywayMigrator 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
- Übung Part 1: Siehe Kommentare in app/repositories/SlickTodoRepository.scala
- Übung Part 2: Todos um eine Kategorie erweitern. Nötige Bausteine:
 - SQL-Migration, z.B. V002__category.sql (Tip: CREATE TABLE & ALTER TABLE notwendig)
 - Todo case class um Option[Category] erweitern
 - SlickTodoRepository in getByld und getAll einen join einbauen (Welcher join?)
 - Für Testing: Kategorie und Todo Verknüpfung über IntelliJ Database Tool anlegen

TODO		
Long	id	PK
String	title	
String	description	
Boolean	done	
Long	categoryId	FK



CATEGORY		
Long	id	PK
String	name	