

Anwendung der funktionalen Programmierung mit Scala

TH Rosenheim - SoSe 2025

Play Framework

Play Framework

The High Velocity Web Framework for Java and Scala

- Web-Framework für moderne, skalierbare Anwendungen
- Unterstützung für reaktive Anwendungen nach dem **Reactive Manifesto**
- Baut auf **Pekko** (Seit Version 3) bzw. davor auf Akka auf. Durch ActorSystem im Hintergrund hohe Parallelität in der Request-Verarbeitung.

SAMSUNG

LinkedIn

 **UniCredit**



 **zalando**

Play Framework Components

Build tool (sbt or gradle)	
Integrated HTTP Server (Pekko HTTP or Netty)	
HTTP APIs	Routing
Form Binding & Validation	Asynchronous HTTP
Template Engine	I18n support
Build-In Security	Async HTTP Client
Testing Helpers	Data Persistence

Play Framework

Aufbau eines Play Projekts (Standardisiert)

- **app/** Quellcode der Anwendung. Aufteilung z.B. in Schichten *controller/ models/ repositories/ services/*
- **app/Module.scala** Guice Dependency Injection Modul. Hier können dynamisch z.B. je nach Umgebung verschiedene Implementierungen für etwa Repositories registriert werden
- **conf/** Konfigurationsdateien und Ressourcen. Wichtig: *application.conf* und *routes*
- **project/** Enthält sbt-Einstellungen/Plugins für das Projekt. Wichtig: *project/plugins.sbt* (Eintrag für das Play sbt-Plugin) und *project/build.properties* (sbt-Version).
- **test/** Unit- und Integrationstests

Play Framework

Request-Verarbeitung (Architektur Beispiel)

- **Anfrage-Fluss:** Client → **Router** → **Controller** → **Service** → **Repository** → **Datenbank**
(der Router leitet HTTP-Requests an den zuständigen Controller weiter, der ggf. Services und Repositories aufruft)
- **Antwort-Fluss:** Der **Controller** erzeugt eine HTTP-Response (z.B. JSON) und gibt sie via Router an den Client zurück.

Play Framework

Routing

- Zentrale Routing-Datei: `conf/routes`
- Verbindet **HTTP-Requests** mit **Controller-Methoden** („Actions“)
- Syntax: “`HTTP_METHOD URL_PFAD Controller.Action`”
- Parameter werden automatisch typisiert gebunden (Standardmäßig unterstützt: Long, Int, String, UUID)
- Beispiel:
`GET /hello/:name controllers.GreetingController.sayHello(name: String)`

Play Framework

Actions & Results

- **Actions:** Eine Action ist eine Funktion von Request auf Result. Play stellt dafür den Action-Builder bereit. Man schreibt z.B. *Action { ... }* für eine synchrone Action (gibt direkt ein Result zurück) oder *Action.async { ... }* für eine asynchrone Action (gibt ein Future[Result] zurück). Intern kümmert sich Play darum, die Futures auszuführen, bevor die Antwort gesendet wird.
- **Result-Typen:** Play liefert vordefinierte **Result-Objekte** für gängige HTTP-Antworten, z.B. *Ok(...)* (200 OK), *NotFound(...)* (404), usw. Diese Results kann man mit Content versehen: Text (als *Ok("Hello")*), JSON (*Ok(Json.obj(...))*), Datei-Downloads etc. Das Framework setzt passende Content-Types und Status-Codes.

Play Framework

Play JSON

- **Play-JSON Bibliothek:** Teil des Play Frameworks für JSON-Verarbeitung. Basiert auf Jackson, jedoch mit Scala-typischer API (Case Classes, Option etc.)
- **Automatische Mappings:** Mit Makros kann man *Reads*, *Writes* oder *Format* automatisch generieren.
- **Nutzen im Controller:** Via `Json.toJson(person)` kann ein Objekt direkt in JSON konvertiert werden, um z.B. im Response gesendet zu werden. Ebenso lassen sich JSON Requests automatisch ins Modell mappen, mit `request.body.asJson.map(_.as[Person])` etc. (Fehlerbehandlung nötig).

Play Framework

Dependency Injection

Warum DI?

- Erlaubt **lose Kopplung** von Komponenten (Controller ↔ Service ↔ Repository)
- **Testbarkeit**: Mock-Implementierungen im Test einbindbar
- **Austauschbarkeit**: z. B. In-Memory-Repo ↔ DB-Repo ohne Codeänderung

Wie funktioniert DI in Play?

- Standardmäßig verwendet Play **Guice** für DI
- Klassen werden per **Konstruktorinjektion** mit `@Inject` versehen
- Bindings werden automatisch erkannt oder explizit definiert (`@ImplementedBy` oder in `app/Module.scala` mit `bind(...).to(...)`)

Play Framework

Dependency Injection Code

Injection

```
class ItemController @Inject()(
  service: ItemService,
  cc: ControllerComponents
) extends AbstractController(cc)
```

Direktes Binding

```
@ImplementedBy(classOf[DbItemRepository])
trait ItemRepository

class DbItemRepository extends ItemRepository
```

Binding in *app/Module.scala*

```
class Module extends AbstractModule {
  override def configure(): Unit = {
    bind(classOf[ItemRepository]).to(classOf[InMemoryItemRepository])
  }
}
```

Play Framework

Rest-Architektur und CRUD

- **REST (Representational State Transfer):** Architekturstil für Web-APIs. Es werden **HTTP-Methoden** genutzt, um auf **Ressourcen** (z.B. „Todo“) zu operieren.
- **CRUD-Mapping:**
 - **Create:** POST /todos – Erstelle ein neues Todo-Item.
 - **Read:** GET /todos – Liste alle Todos; GET /todos/{id} – lies ein bestimmtes Todo.
 - **Update:** PUT oder PATCH /todos/{id} – Ändere ein bestehendes Todo (z.B. als erledigt markieren).
 - **Delete:** DELETE /todos/{id} – Lösche ein Todo.
- **Beispiel: Todo-Service:**
 - Ressource **Todo** mit Feldern wie id, title (Text) und done (Boolean, ob erledigt).
 - Requests/Responses im **JSON-Format** (z.B. GET /todos liefert JSON-Liste aller Todos)

Play Framework

Service - & Repository-Schicht

Motivation Schichtenbildung: In größeren Anwendungen trennt man Verantwortlichkeiten in Schichten.

- **Service**-Klassen kapseln die **Geschäftslogik** (Business Logic, z.B. Berechnungen, Validierungsregeln, Abläufe).
- **Repository**-Klassen kapseln den **Datenzugriff** (CRUD-Operationen auf DB oder externen APIs).

=> Controller rufen Services auf, Services rufen Repositories auf.

Diese Trennung erhöht die **Wiederverwendbarkeit** und **Testbarkeit**: Services kann man isoliert testen (mit gemockten Repos), Repositories sind auswechselbar (z.B. andere DB).

Play Framework Repository Layer

Definiert typischerweise **Interfaces (Traits)** für Datenquellen. Wichtig: Das restliche System spricht nur das Trait an, so kann die konkrete Datenquelle leicht getauscht werden (Prinzip *Dependency Inversion*).

```
trait TodoRepository {  
  def findById(id: Int): Future[Option[Todo]]  
  def create(todo: Todo): Future[Unit]  
}
```

Play Framework Service Layer

Enthält die Kernlogik der Anwendung. Ein Service kann mehrere Repository-Aufrufe kombinieren, Geschäftsregeln anwenden und das Ergebnis an den Controller weiterreichen. Er stellt sozusagen die „Use Cases“ bereit (z.B. *erstelle neues Item, berechne Statistik, ...*)

```
class TodoService(repo: TodoRepository) {  
  def addTodo(todo: Todo): EitherT[Future, String, Todo] = for {  
    _ ← validateTodo(todo)  
    _ ← validateId(todo)  
    todo ← EitherT(repo.create(todo).map(Right(_)))  
  } yield todo  
  
  private def validateTodo(todo: Todo): EitherT[Future, String, Unit] = ???  
  
  private def validateId(todo: Todo): EitherT[Future, String, Unit] = {  
    EitherT {  
      repo.findById(todo.id).map {  
        case Some(_) ⇒ Left("ID already taken")  
        case None    ⇒ Right()  
      }  
    }  
  }  
}
```

Play Framework

Übung

Selbständiges implementieren einer CRUD Todo-API

- Model
- Repository (trait & class)
- Service
- Controller

Optionale Erweiterung

- Domain spezifische Error statt String in EitherT[Future, String, T]
- Validierung im Service (Einfache Methoden oder z.B. Iron)
- H2 Datenbank Repository

- **Create:** POST /todos – Erstelle ein neues Todo-Item.
- **Read:** GET /todos – Liste alle Todos; GET /todos/{id} – lies ein bestimmtes Todo.
- **Update:** PUT oder PATCH /todos/{id} – Ändere ein bestehendes Todo (z.B. als erledigt markieren).
- **Delete:** DELETE /todos/{id} – Lösche ein Todo.