JAVACREAM

Training
Consulting
Projectmanagement

Spring Grundlagen

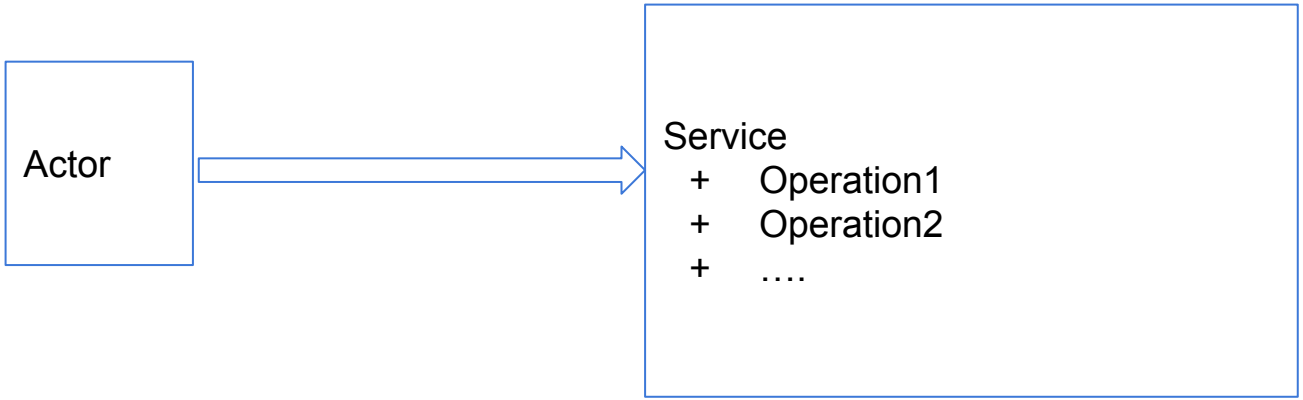
Kurze Vorstellung

JAVACREAM
Training
Consulting
Projectmanagement

- Name
- Rolle im Unternehmen
- Themenbezogene Vorkenntnisse
- Aktuelle Problemstellung
- Konkrete individuelle Zielsetzung

Ausgangssituation

Fachanwendung



Bei Modellierung einer Fachanwendung keinerlei Bezug zu Spring vorhanden

Technisches Modell

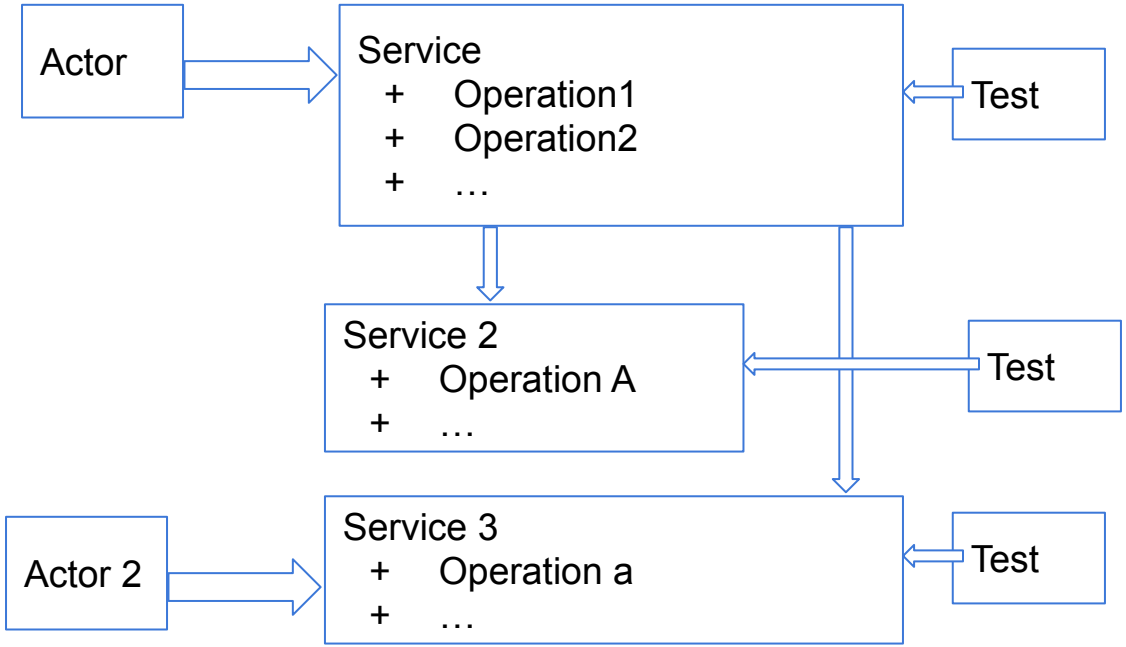
Anforderungen an das Modell

- \+ Wartbarkeit
- \+ Wiederverwendung
- \+ Testbarkeit

Umsetzung durch Modularisierung statt einer monolithischen Applikation

Bezug zu Spring ist indirekt

- \+ Bei Verwendung von Spring ist die Modularisierung eines technisches Modells sehr gut möglich

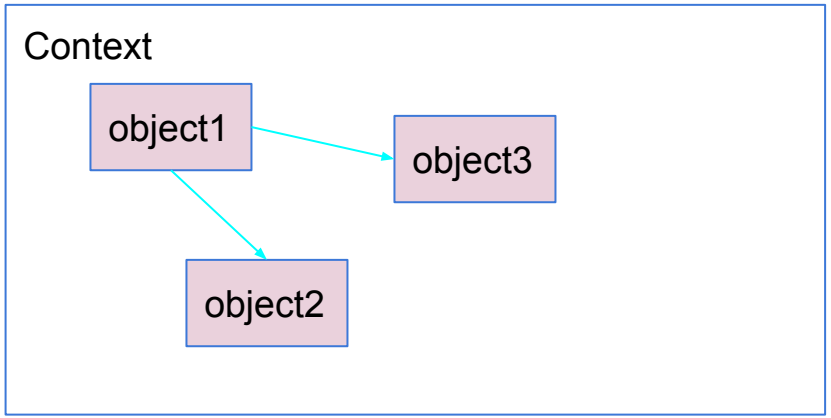


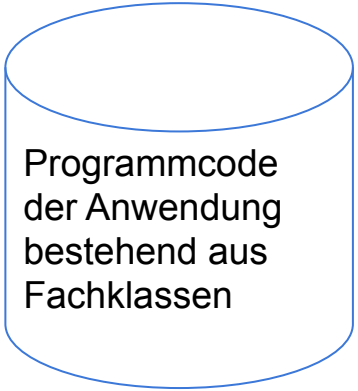
Context & Dependency Injection (CDI)

CDI baut aus den einzelnen Modulen das Objektgeflecht der Anwendung auf

Spring ist eine Umsetzung des Design Patterns Context & Dependency Injection

“Spring ist ein CDI-Framework”



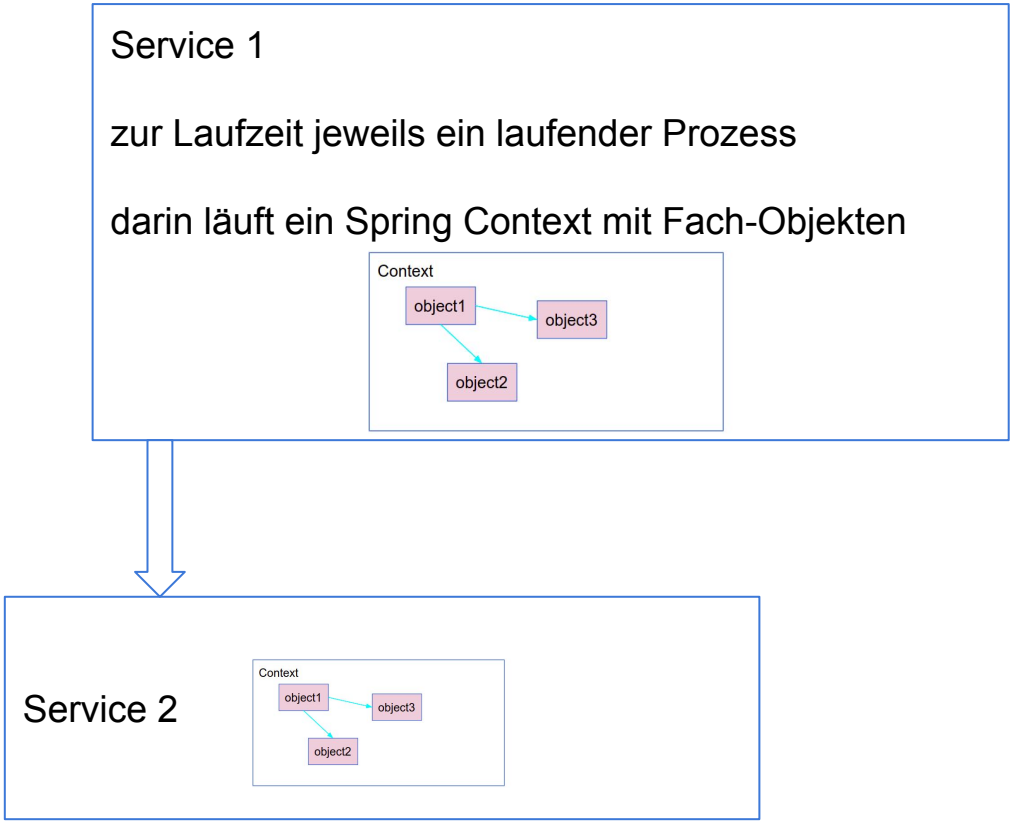


Aufgabe des Contexts

+ Identifikation der relevanten Fachklassen und Instanzierung von Fach-Objekten
+ Identifikation der Abhängigkeiten der Objekte und das Setzen der Abhängigkeit

Exkurs: Module versus Fachklassen

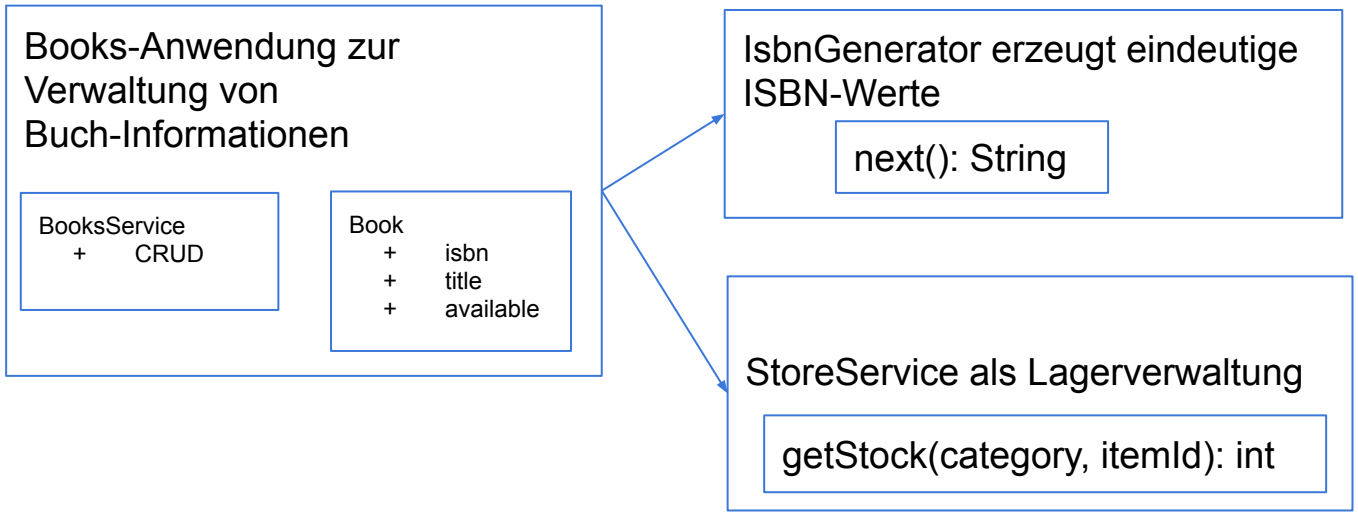
Service-oriented bzw. Microservices



Die Fachanwendung des Trainings

Modell

- Vollständig vorgegeben
- Fachlich einfach



Umsetzung

- Ist auch bereits vorhanden
  - Bisher
    - Die gesamte Datenhaltung In Memory
    - Zugriff ist nur für Actors im selben Prozess möglich
      - Actors = Test-Fälle
- Programmierung ist typisch für eine statisch typisierte Programmiersprache wie Java
  - Operationen sind in Schnittstellen definiert
  - Datenstrukturen sind simple Daten-Container (eigentlich structs oder records)
  - Operationen und Datencontainer definieren das API einer Fachanwendung
  - Zugehörige Implementierung ist eine Klasse, die die Schnittstelle implementiert

Umsetzung

- Bisher
  - Die Anwendung hat keinerlei Bezug zu Spring!
  - Die Anwendung selber ist jedoch CDI-konform!
    - Verifizierung: Relevante Fachklassen (MapBooksService, SimpleStoreService, RandomIsbnGenerator, CounterIsbnGenerator) werden im Rahmen der Anwendung NIEMALS mit new instanziert
    - Dependency ist ein Attribut vom Typ einer API-Schnittstelle + setter-Methode
      - Das ist das GoF-Pattern “Strategy”, das CDI-Pattern ist eine Meta-Pattern aus Strategy und Factory
  - Der Testfall übernimmt die Aufgaben des Contextes
    - new-Operatore
    - Aufruf der setter-Methoden

Spring First Contact

- Exkurs: “Spring” oder “Spring Boot”?
  - Spring = Spring Core ist das CDI-Framework
  - Spring Boot
    - Vereinfachter Build-Prozess
      - Dependency Management mit parent-pom und startern
    - Autoconfigure
      - “Convention over Configuration”
        - Welche Pakete sollen nach Spring-Informationen durchforstet werden?
        - Es wird automatisch eine Konfigurationsdatei namens application.properties eingeladen
  - Spring Core ist prinzipiell unabhängig von Spring Boot, aber es ist fast sinnlos, kein Spring Boot zu benutzen

Programmiermodell von Spring

- So nicht:
  - keine Namenskonventionen
  - benutzt keine Spring-Schnittstellen
    - relevant = “implements ContextAware”
- sondern ausgerichtet auf die Bereitstellung von Meta-Informationen
  - Externe XML-Konfiguration
  - **Java Annotations** (C#: Attributes)
    - @Component
      - oder @Service oder @Repository -> später
    - @Autowired
      - Referenzen auf Spring-relevante Objekte
    - @Value
      - Konfiguration auf einen Key, der in der application.xml eingetragen ist

Programmieren mit Spring

Spring Boot

- @SpringBootApplication
  - @SpringConfiguration
    - damit ist sie Spring-relevant
  - @EnableAutoconfiguration
    - z.B. laden der application.properties|yaml
  - @ComponentScan (“alles in diesem Paket und darunter”)
- @SpringBootTest
  - Scanne das gesamte Projekt nach einer @SpringConfiguration

@Component-Annotations

- Sind “Stereotypen”
  - Keine technische Unterschiede
  - Praktisch für die Dokumentation
- @Component
  - allgemein: “eine Spring-relevante Klasse”
- @Service
  - “Eine Klasse, die Service-Operationen anbietet”
- @Repository
  - “Eine Klasse, die CRUD-Operationen für eine Ressource anbietet”

Autowiring

- Beim Autowiring muss der Context exakt eine geeignete Spring Bean finden
  - Falls das nicht der Fall ist -> Fehler beim Hochfahren des Context
- @Autowired hat als Parameter nur “required = true|false”
  - kann damit auch auf null stehen
- Kandidaten zum Autowiring werden durch das Java Typsystem gefunden
  - Im Detail -> später
-

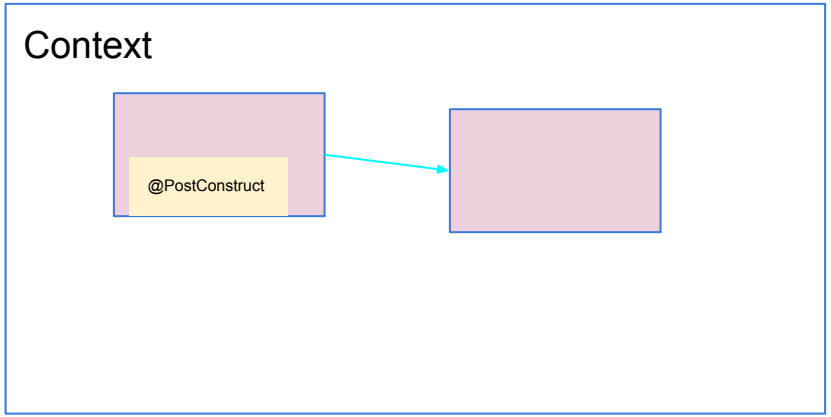
@Value

- Diese ist in der Lage, eine Konfigurationseinstellung zu injected
  - Spring Expression Language
    - ${}
    - Wert
      - Das zu verwendende Objekt
      - **object.property**
      - object.property.property
      - object.method()
- Nicht-vorhandene Konfigurationseinstellungen führen in Standard-Konfiguration zu einem Fehler

Weitere Features

- Erweiterung der Arbeitsweise des Context
  - Relevante Klassen können einen Lifecycle definieren
    - @PostConstruct public void <name>(){}
    - @PreDestroy public void <name>(){}
  - ToDo: Diskussion: Was ist der Unterschied zum Konstruktor?
- Eine Fachklasse kann auch einen parametrisierten Konstruktor aufweisen
  - Parameter werden automatisch als @Autowired aufgefasst
    - ToDo: MapBooksService mit Konstruktor-Parametern
    - ToDo: Diskussion Attribute-Injection versus setter-Injection versus Constructor-Injetion
      - Sind setter-Methoden notwendig?
  - Parameter können auch zur Value-Injection benutzt werden
    - ToDo: RandomIsbnGenerator mit Constructor mit prefix und countryCode

Lifecycle im Context



Aufgabe des Contexts

+ Identifikation der relevanten Fachklassen und Instanzierung von Fach-Objekten
  + Aufruf des Konstruktors
+ Identifikation der Abhängigkeiten der Objekte und das Setzen der Abhängigkeit
+ Aufrufen der @PostConstruct-Methoden

Autowiring im Detail

- Bisher
  - Der Context benutzt das Java Typsystem
    - Berücksichtigt wird hier auch Vererbung / Implementierung von Schnittstellen
    - RandomIsbnGenerator ist ein IsbnGenerator und ein Object
- Neu
  - Der Name des Injection Points (Attribut-Name, Constructor-Parameter-Name, setter-method ohne set + klein) wird ebenfalls benutzt
  - -> nächste Seite, @Resource
  - @Primary
    - bei Mehrdeutigkeiten wird @Primary benutzt
  - Qualifiers
    - Identifizierbar über eine Zeichenkette

ToDo

- Nachvollziehen der Präsentation zum Thema “resolve”
- Überlegen Sie, ob das aktuelle Paket der Strategy-Annotations wirklich gut ist?
  - Bessere Möglichkeit?

Scopes

- Wenn “jemand” eine Dependency benötigt, kann der Context entweder
  - eine bereits vorhanden Instanz benutzen oder
    - @Scope(”singleton”)
  - pro Injection Point eine neue Instanz erzeugen
    - @Scope(”prototype”)
- Standard-Scope ist “singleton”

@Autowiring versus @Resource

- Funktionsweise eines CDI-Frameworks
  - statisch autowiring
  - statisch deterministisch
  - ~~dynamisch autowiring~~
  - ~~dynamisch deterministisch~~
- Autowiring
  - Die Dependency wird durch einen Context-Algorithmus bestimmt
- Deterministisch
  - Der Entwickler legt die Dependency hart fest

Dependency Injection wird beim Hochfahren des Context identifiziert und gesetzt

Spring ist nicht dynamisch!

Verständnisfrage

- 4 Variationen Dependency Injection mit Scopes
  - singleton - singleton
  - prototype - prototype
  - singleton - prototype
  - prototype - singleton
- Gibt es hier irgendwelche potenziellen Einschränkungen
- Vorgriff: Web Anwednungen
- Neuer Scope: “request”
  - singleton - singleton
  - request - request
  - singleton - request
    - Geht nicht bei einem statischen DI-Framework
  - request - singleton

Meta-Informationen für Spring

- Bisher
  - Annotations
    - Wertung
      - Java-Entwickler finden Annotations gut
      - Review und Dokumentation sind aufwändiger
- Nun
  - XML-Dokument
    - Schema, in dem Bean-Definitionen abgelegt werden können
    - Wertung
      - Java-Entwickler finden das aufwändiger
        - insbesondere: Refactoring wird schwer
      - Review und Dokumentation ist einfach
        - Der Objekt-Graph ist abgelegt in einem hierarchischen XML-Dokument
    - Hinweis: Zur Dependency Injection sind bei XML setter-Methoden erforderlich

Exkurs: Spring Tools Suite

- Früher war eine ausgezeichnete Unterstützung für Spring XML
  - Code-Assist unter Berücksichtigung von Java (REFACTORING!)
  - Visualisierung
- Aktuell
  - Keine besondere Unterstützung mehr vorhanden
  - Editieren eines Spring-XML erfolgt über den normalen XML-Editor
- “Mischbetrieb” ist möglich und durchaus gebräuchlich
  - Hinweis: Dieser Mischbetrieb ist vollständig
    - <bean class=...> </bean> class Attribut mit @Autowired, @PostConstruct

ToDo

- Stellen Sie die Anwendung auf einen Mischbetrieb mit einer spring-beans.xml um
  - z.B. Store als XML, Rest mit Annotations

Ein paar Details zum XML

- Spring XML ist eine eigene Programmiersprachen
- Eine Vielzahl weiterer Namespaces erweitert diese Programmiersprache
  - Spring Batch
  - Apache Camel

@Configuration-Klassen

- Jede @SpringBootApplication ist eine @Configuration
  - damit können weitere Annotation wie @ImportResource oder @ComponentScan hinzugefügt werden
  - public-Methoden mit Rückgabewert können dem Context Beans zur Verfügung stellen
    - @Bean
      - @Scope, @Qualifier, @Bean(“name”) werden unterstützt
- Dieses Möglichkeit wird “Java Config” genannt
  - sehr beliebt bei Java-Entwicklern
    - Vorsicht: In der Praxis nicht ganz unkompliziert!
      - @Scope(“singleton”) führt zu einem Method Result Cache
      - -> Spring Advanced
    -

ToDo

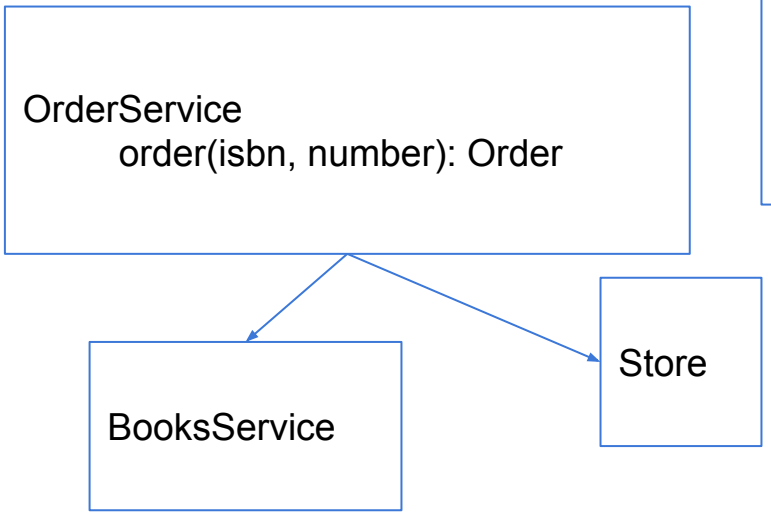
- MapBooksService hat eine interne HashMap
- Stellen Sie diese auf Dependency Injection
  - inklusive ein paar Test-Daten, die Sie über Java anlegen
    - put
- Schreiben Sie nun noch eine weitere Test-Methode im BooksService-Test, die auf Basis der Test-Daten Assertions formuliert

ToDo

- Nachvollziehen der Einführung von Profilen
- Einführen Custom Profile Annotations
- Custom Annotations für z.B.
  - einen “JavacreamSpringTest”
    - automatisch wird das Test-Profile aktiviert
  - eine “ProdConfiguration”

ToDo Step 1

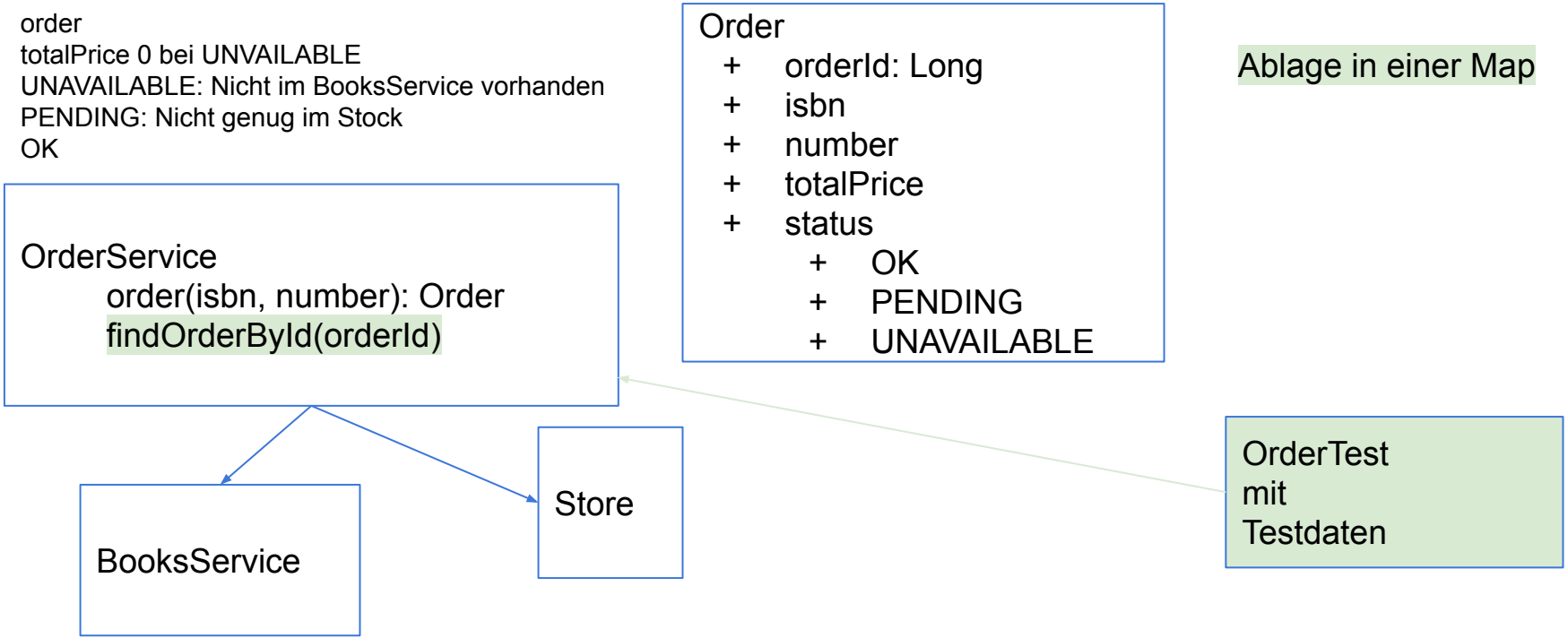
order
totalPrice 0 bei UNVAILABLE
UNAVAILABLE: Nicht im BooksService vorhanden
PENDING: Nicht genug im Stock
OK



Order
+ orderId: Long
+ isbn
+ number
+ totalPrice
+ status
  + OK
  + PENDING
  + UNAVAILABLE

ToDo Step 2

order
totalPrice 0 bei UNVAILABLE
UNAVAILABLE: Nicht im BooksService vorhanden
PENDING: Nicht genug im Stock
OK



Optional: ToDo Step 3

order
totalPrice 0 bei UNVAILABLE
UNAVAILABLE: Nicht im BooksService vorhanden
PENDING: Nicht genug im Stock
OK

OrderService
order(isbn, number): Order
findOrderById(orderId)

BooksService

Store

Order
+ orderId: Long
+ isbn
+ number
+ totalPrice
+ status
+ OK
+ PENDING
+ UNAVAILABLE

Ablage in einer Map

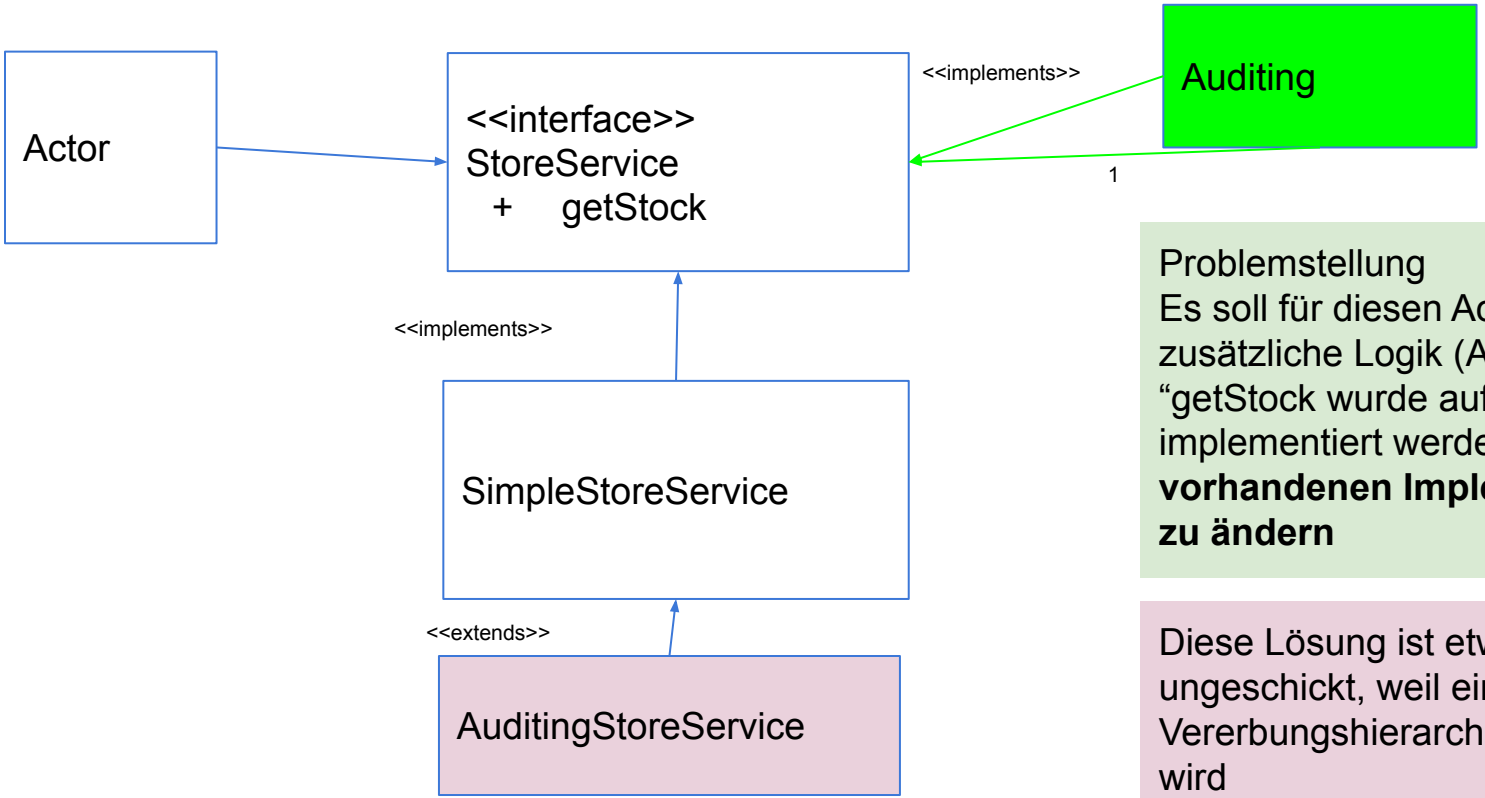
IdGenerator

Einsatz im CounterIsbnGenerator? Scope?

OrderTest mit Testdaten

Spring Core, Part 2: Aspektorientierte Programmierung

Decorator



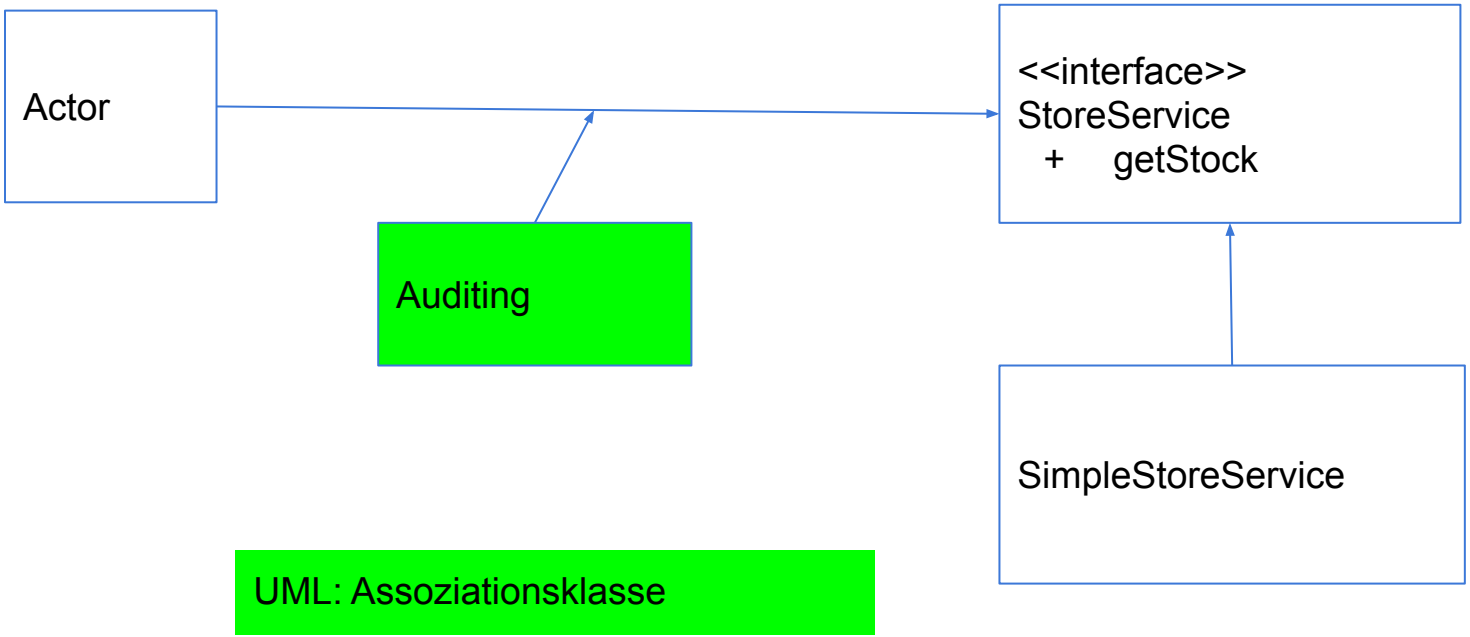
Problemstellung
Es soll für diesen Actor eine zusätzliche Logik (Auditing, “getStock wurde aufgerufen…”) implementiert werden **ohne die vorhandenen Implementierungen zu ändern**

Diese Lösung ist etwas ungeschickt, weil eine statische Vererbungshierarchie aufgebaut wird

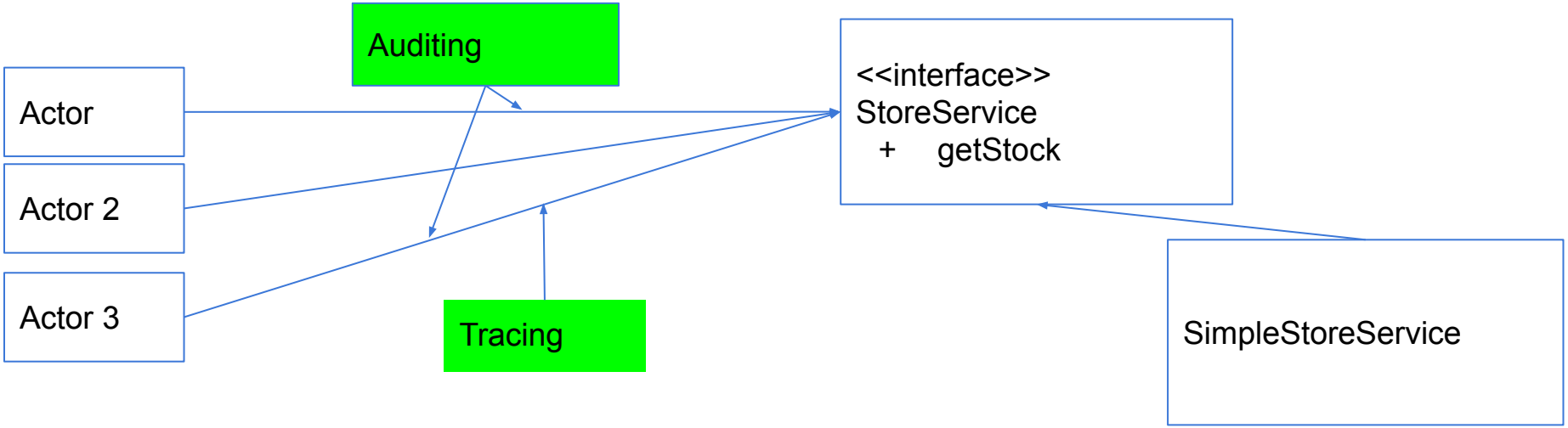
Decorators in Spring

- Nicht ganz unproblematisch im Zusammenhang mit Autowiring

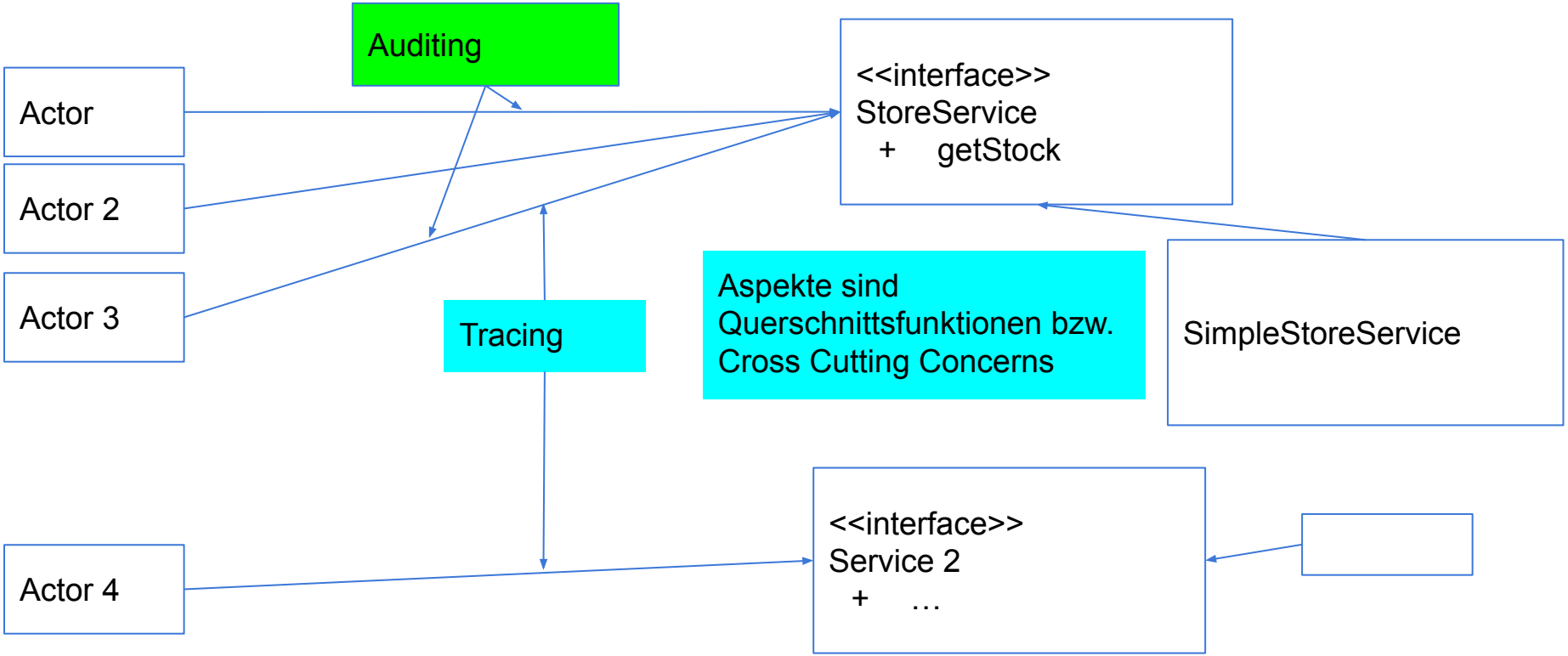
Aspekt: Schritt 1, Änderung in der Darstellung des Modells



Aspekt: Schritt 1, Änderung in der Darstellung des Modells



Aspekt: Schritt 2, Vom Decorator zum Aspect



Spring und AOP

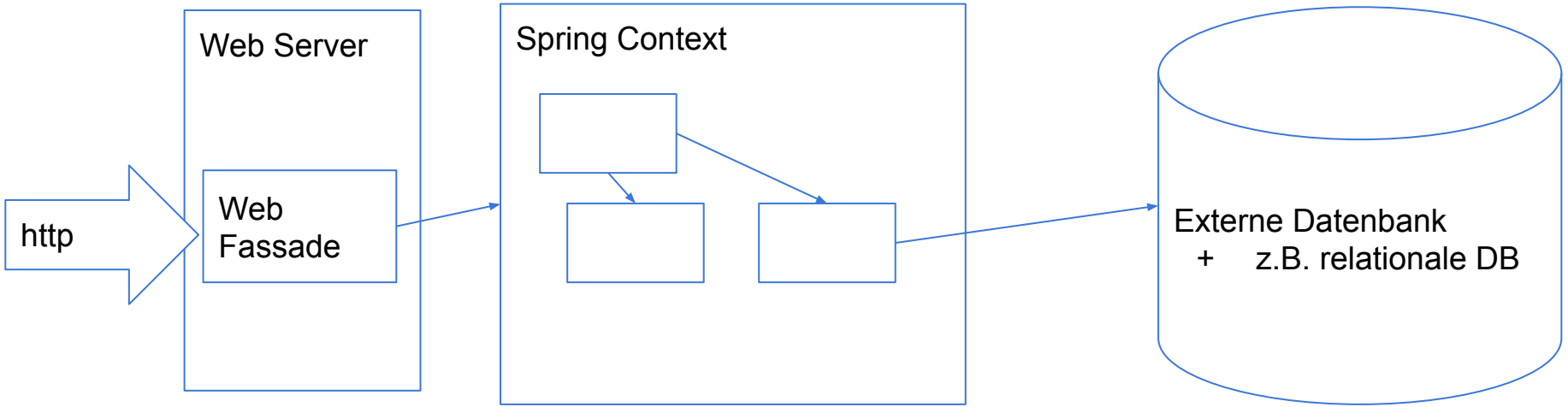
- Spring CDI ist AOP-fähig
  - aber nicht durch eine Spring-Lösung alleine
  - Notwendig ist die Einbindung eines AOP-Frameworks
    - Referenz: AspectJ von der Eclipse-Community
    - Einbindung durch den starter-aop + aspectj-runtime
      - pom.xml

ToDo

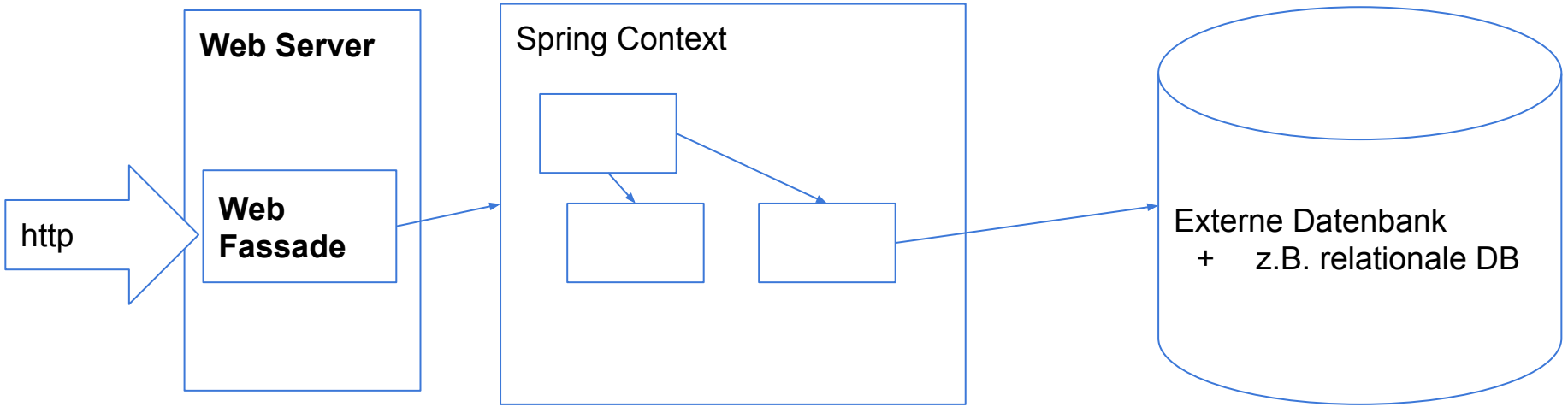
- Nachvollziehen der Hinführung
- Eigenständig
  - Debugging?
  - Eigener Aspekt: TimeMeasure
    - wie lange dauert der Aufruf, System.currentTimemillies()
  - Können Aspekte kombiniert werden
    - Recherche: Wie kann die Reihenfolge der Aspekte kontrolliert werden
      - Erst Tracing dann Measure oder anders herum

Eine reale Anwendung

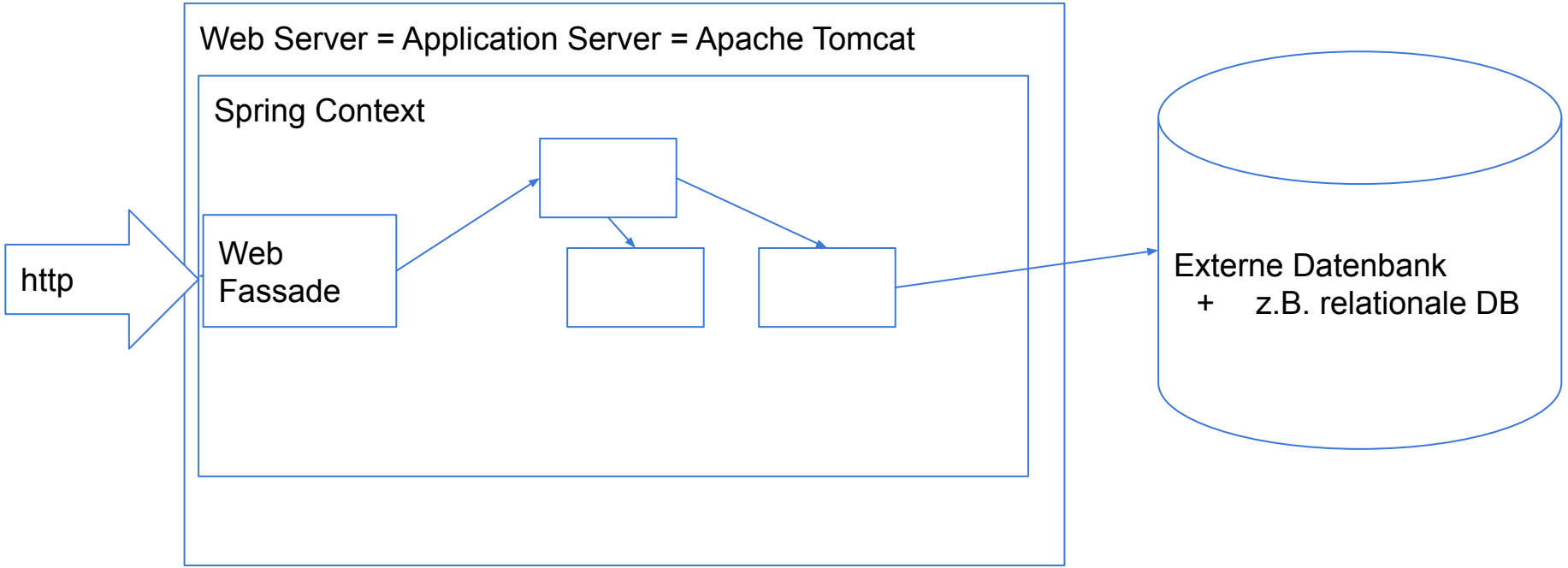
Architektur



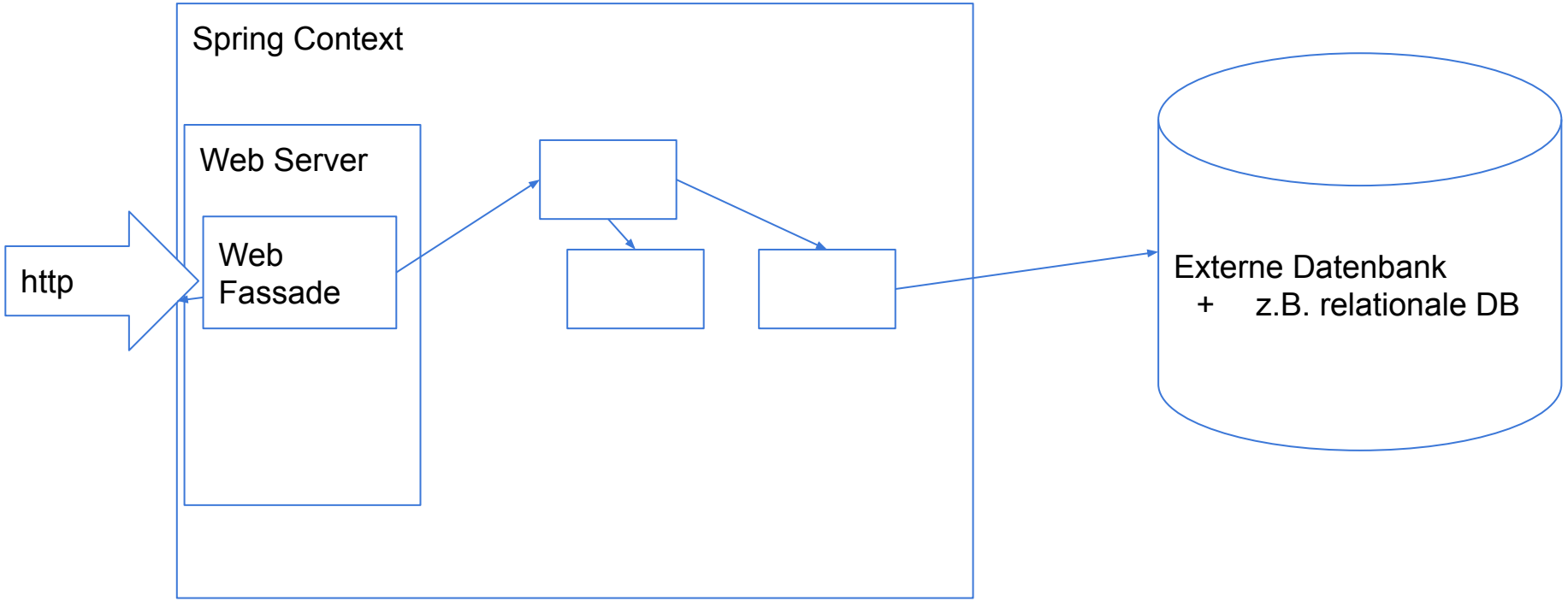
Architektur: Step 1



Umsetzung: Klassisches Deloyment einer Web-Anwendung, .war



Umsetzung: Standalone Java



- Beide Möglichkeiten werden durch den Boot-Build-Prozess unterstützt
- Spring Boot Maven Plugin
  - pom-packaging
    - war
      - Klassische Web Anwendung
    - jar
      - Standalone JAR mit embedded Web Server
        - Hinweis: Kein Spielzeug! Voll produktionstauglich!

- Autoconfigure-Feature
  - unter Berücksichtigung der im Klassenpfad vorhandenen Bibliotheken
    - z.B. “wenn ich im Klassenpfad Spring-Web-Bibliotheken finde, dann auto-konfiguriere ich einen embedded Web Server auf Port 8080”
  - pom.xml
    - z.B. starter-web

Programmiermodell der Web-Fassaden

- Prinzipiell komplett in ihrer Hand
  - service(HttpRequest request, HttpResponse reponse){//low level networking}
- Spezifikation RESTful Web Services
  - Eine Web Fassade ist aufzufassen als ein Service, der CRUD-Operationen für ein Dokument bereitstellt
    - Dokument hat ein Schema
  - Umsetzung mit http
    - Create -> POST
    - Read -> GET
    - Update -> PUT
    - Delete -> DELETE
- Dokumenten-Format text/plain oder application/json

Parametrisierung der Operationen über
+ Http Header
+ Erweiterung der Aufruf-URL
  + Pfad
  + Query-Parameter (?p1=v1&p2=v2

Rückgabe ist das Dokument + ein Status-Code
+ 200 = OK
+ 404 = NOT FOUND

Programmiermodell mit Spring MVC

- Ein allgemeines Programmiermodell für **RESTful WebServices** und klassische HTML-basierte Web-Anwendungen
- Fassaden werden vom Spring Context verwaltet
  - die erzeugten Instanzen müssen jedoch speziell verwaltet, da sie an den Web Server angebunden werden müssen
    - @RestController
- Mapping URL -> Java-Methode erfolgt über Annotations
  - @GetMapping, @PostMapping, …
  - @RequestHeader, @PathVariable, @RequestParameter

ToDo

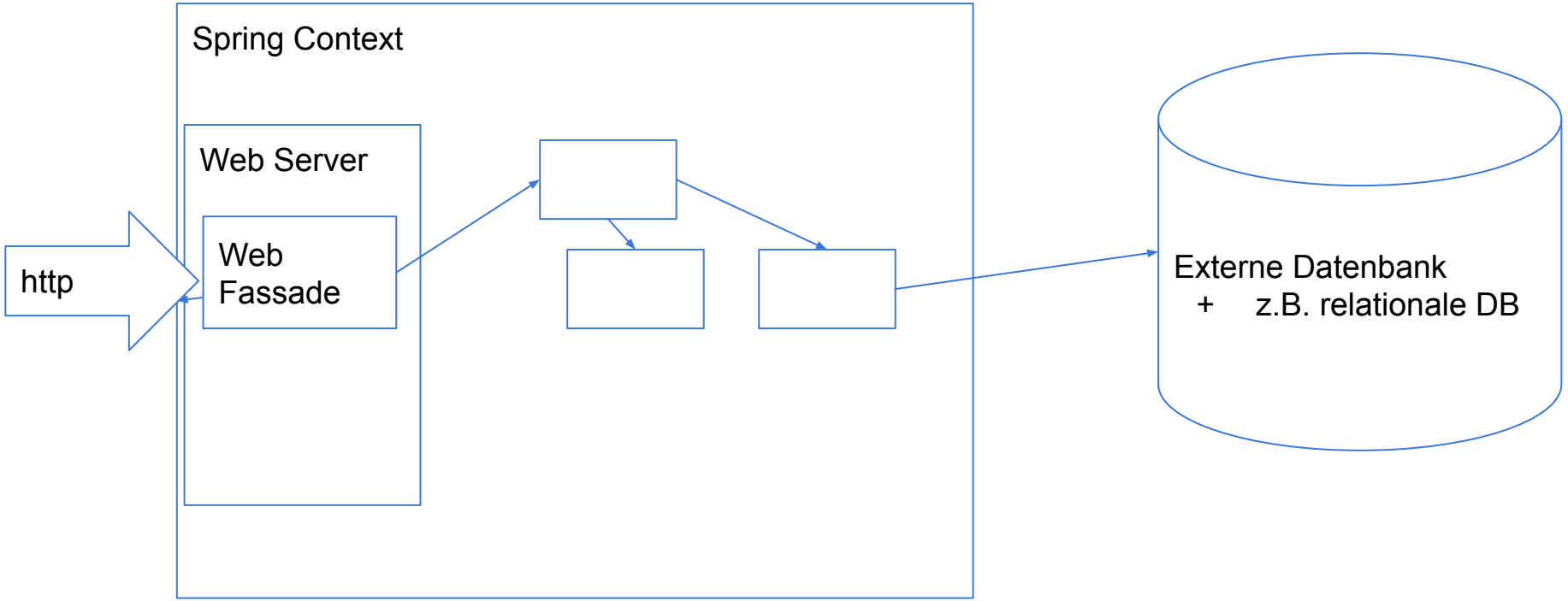
- Schreiben Sie bitte REST-Fassaden für den StoreService und den IsbnGenerator
  - package: …web
  - Namenskonvention
    - StoreWebService
    - WebIsbnGenerator
- Zum WebIsbnGenerator
  - Dieser soll gesteuert über den Header-Parameter “strategy” zwischen random und sequence unterscheiden können

ToDo

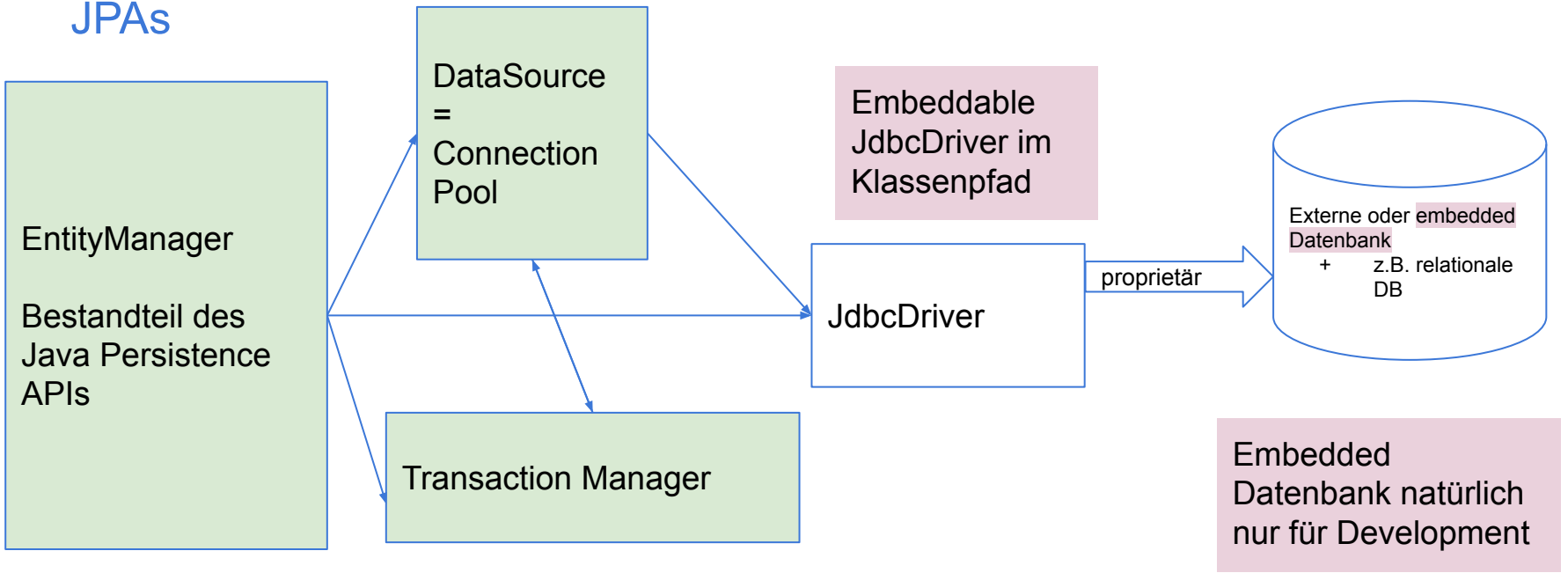
- Implementieren Sie den BooksWebService
  - dafür benötigen Sie alle Http-Methoden
- Überlegen Sie sich geeignete Signaturen für update und findAll
  - Idee: void updatePrice
  - findAllIsbns() oder findAllBookInfos() mit BookInfo (isbn, title)
- Achten Sie auf das Exception-Handling
  - Was passiert denn, wenn die Fassade die BookException wirft?
  - Vernünftige Status sind OK, NotFount, UnprocessableEntity

Datenbank-Zugriffe

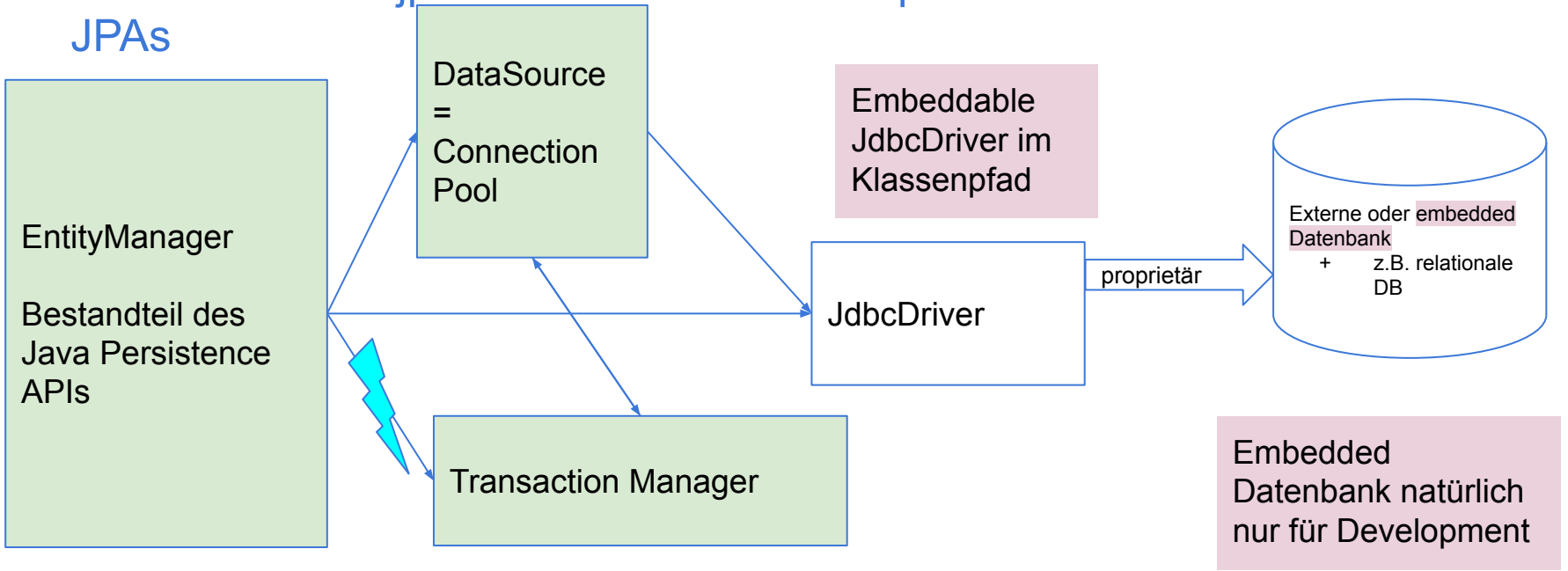
Umsetzung: Standalone Java



- Der starter-data-jpa erweitert den Klassenpfad um die Bibliotheken des JPAs



- Der starter-data-jpa erweitert den Klassenpfad um die Bibliotheken des JPAs



Programmieren mit dem EntityManager

- **Native Queries**
  - **werden mit minimalen Ersetzungen direkt als SQL-Statement gegen die Datenbank gefeuert**
- CRUD-Operationen auf Basis von Entities
  - das gegen die Datenbank gefeuerte SQL wird generiert
    - dazu werden Annotationen auf der Ebene einer Java-Klasse benutzt
      - @Entity
      - @Id
      - …
  - Create -> persist
  - Read -> find, createQuery(query-Ausdruck)
  - Update -> merge
  - Delete -> remove

ToDo

- DatabaseStoreService übernehmen
- SequenceGenerator auf eine Datenbank-Lösung umstellen
  - Ablauf
    - Lese aktuelle id
    - increment
    - zurückschreiben
    - return