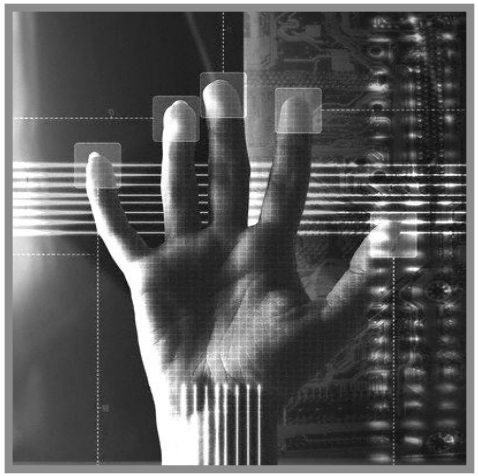
Software Engineering und Projektmanagement



SE&PM – Tutorial zur Gruppenphase

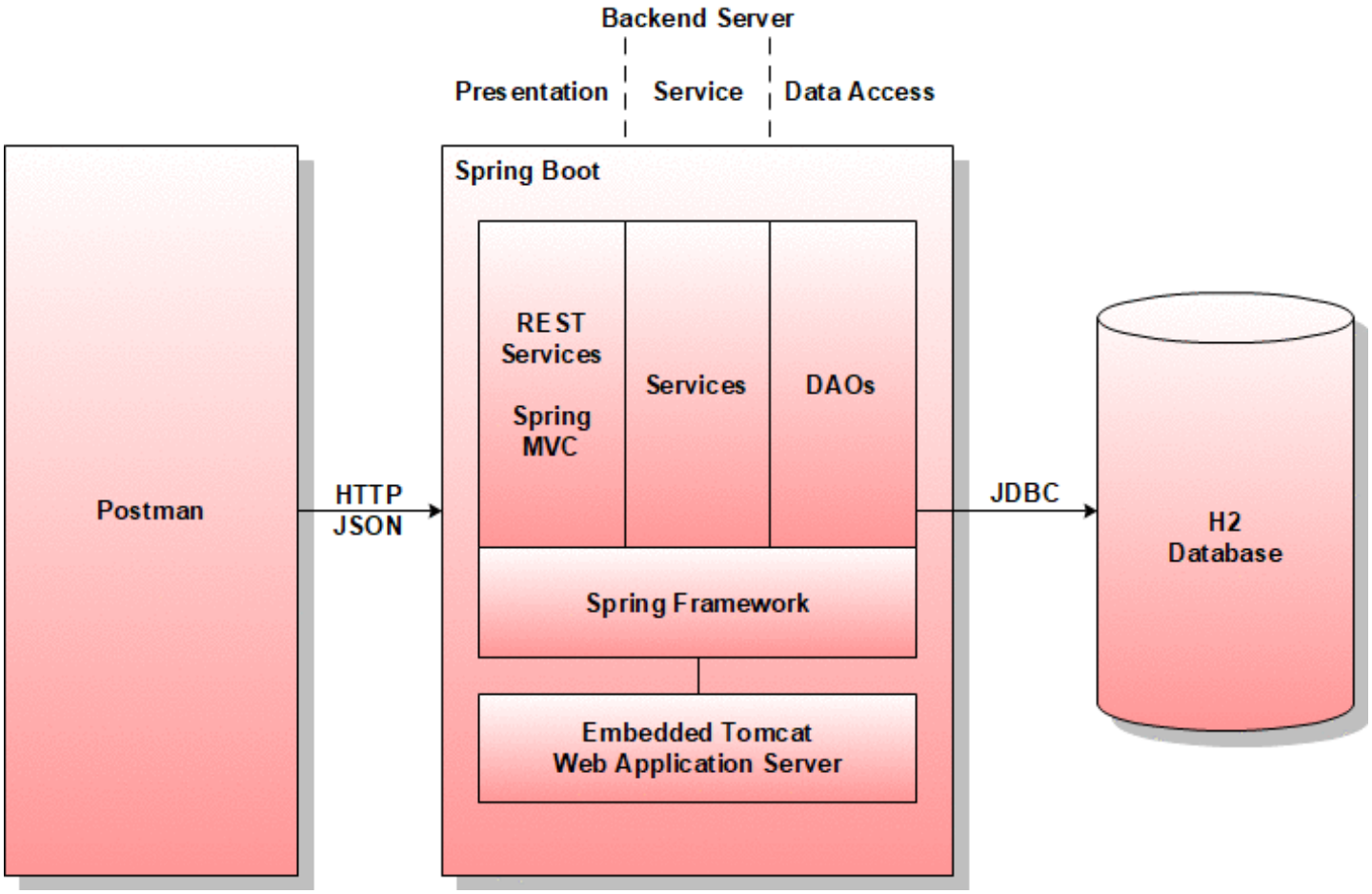
2019S - 4. April 2019

Wolfgang Gruber

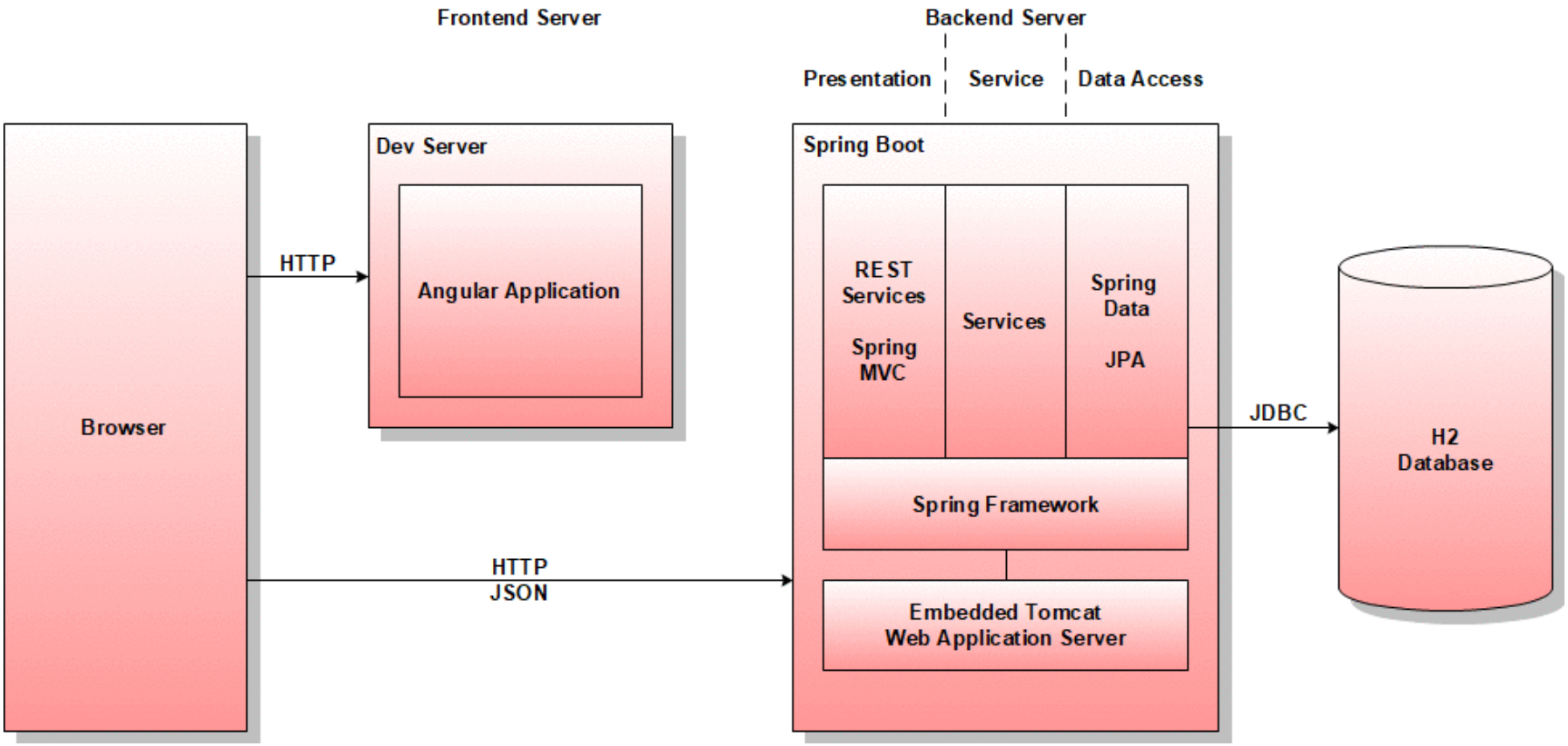
Web: https://tuwel.tuwien.ac.at/course/view.php?id=153
INSO: sepm@inso.tuwien.ac.at
QSE: sepm@qse.ifs.tuwien.ac.at

Inhalt

Architektur Einzelbeispiel



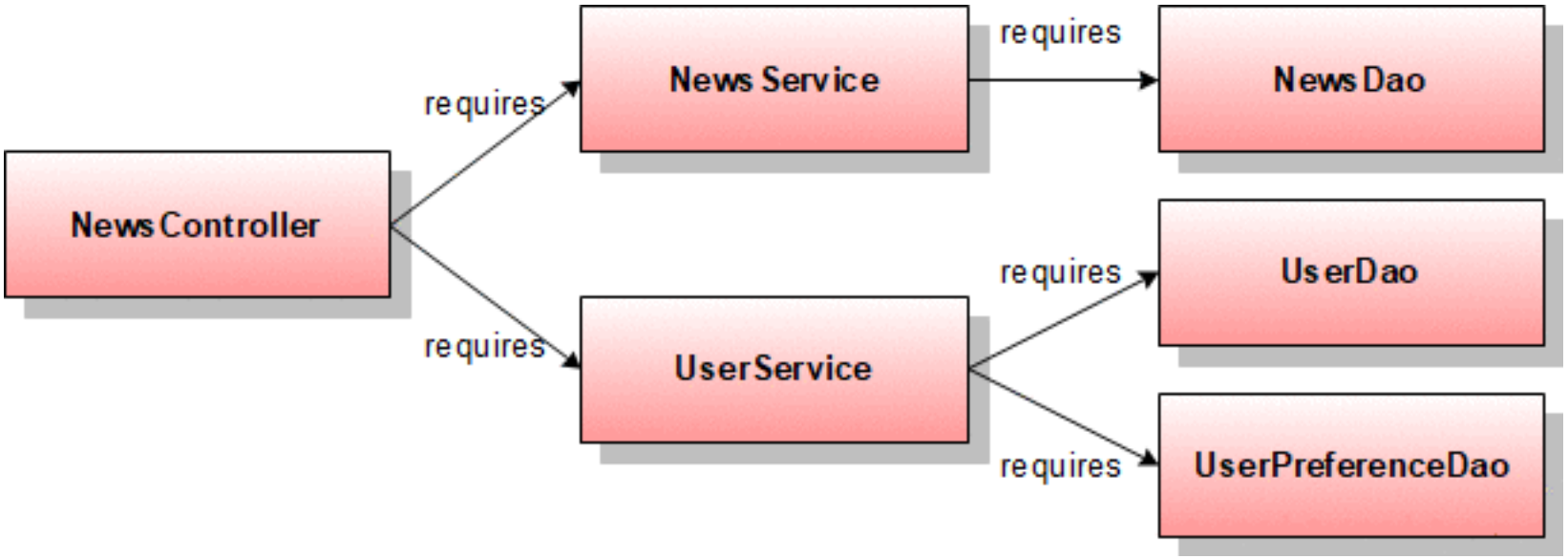
Architektur Gruppenphase



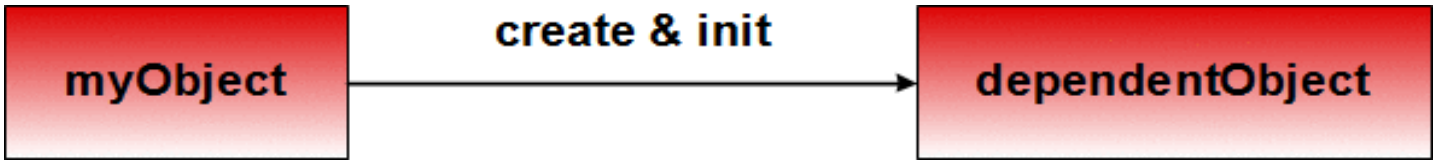
Dependency Injection - Einleitung

- **Design Pattern, bei dem Abhängigkeiten zu anderen Objekten durch einen Container aufgelöst werden**
- **Veraltete Namen:**
  Inversion of Control, Hollywood-Principle
- **Erreichte durch das Spring Framework eine weite Verbreitung**

Objektgraph



Direkte Instanzierung



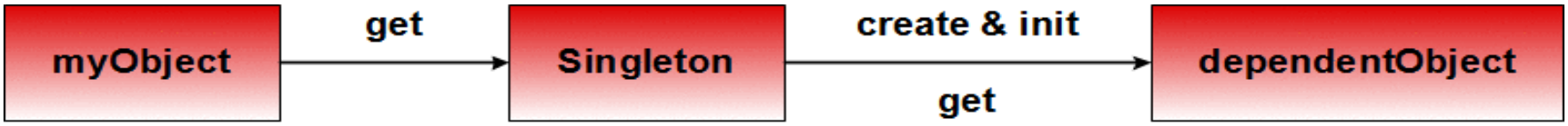
Instanzierung:
SqlConnection con = new SqlConnection(); // Beispiel-Code

Initialisierung:
con.setUsername("user");
con.setPassword("streng-geheim");
con.setUrl("jdbc:hsqldb:hsql://localhost/ticketline");

Verwendung:
con.execute("SELECT * FROM something");

Probleme:
- Aufwändige Erzeugung
- Codeduplizierung
- Object-Cluttering
- Enge Kopplung
- Schwierig zu testen

Dependency Lookup

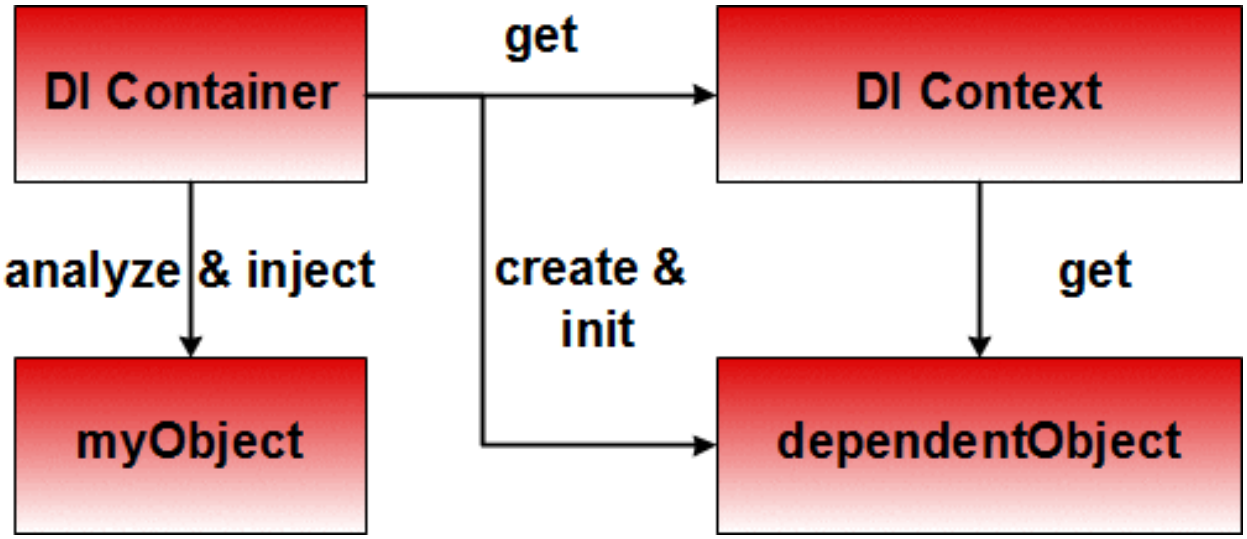


Globaler Zugriffspunkt auf Objekte (zB Services)
Singleton ist oftmals gleichzeitig Factory
Instanzierung & Initialisierung in Singleton
Design Pattern, das heutzutage oftmals als Anti-Pattern betrachtet wird

Verwendung:
Connection con = DbConnection.getConnection();

Probleme:
- Global für die komplette Applikation
- Implementierung kann nur schwer ausgetauscht werden
- Schlechte Testbarkeit

Dependency Injection (DI)



DI Container:

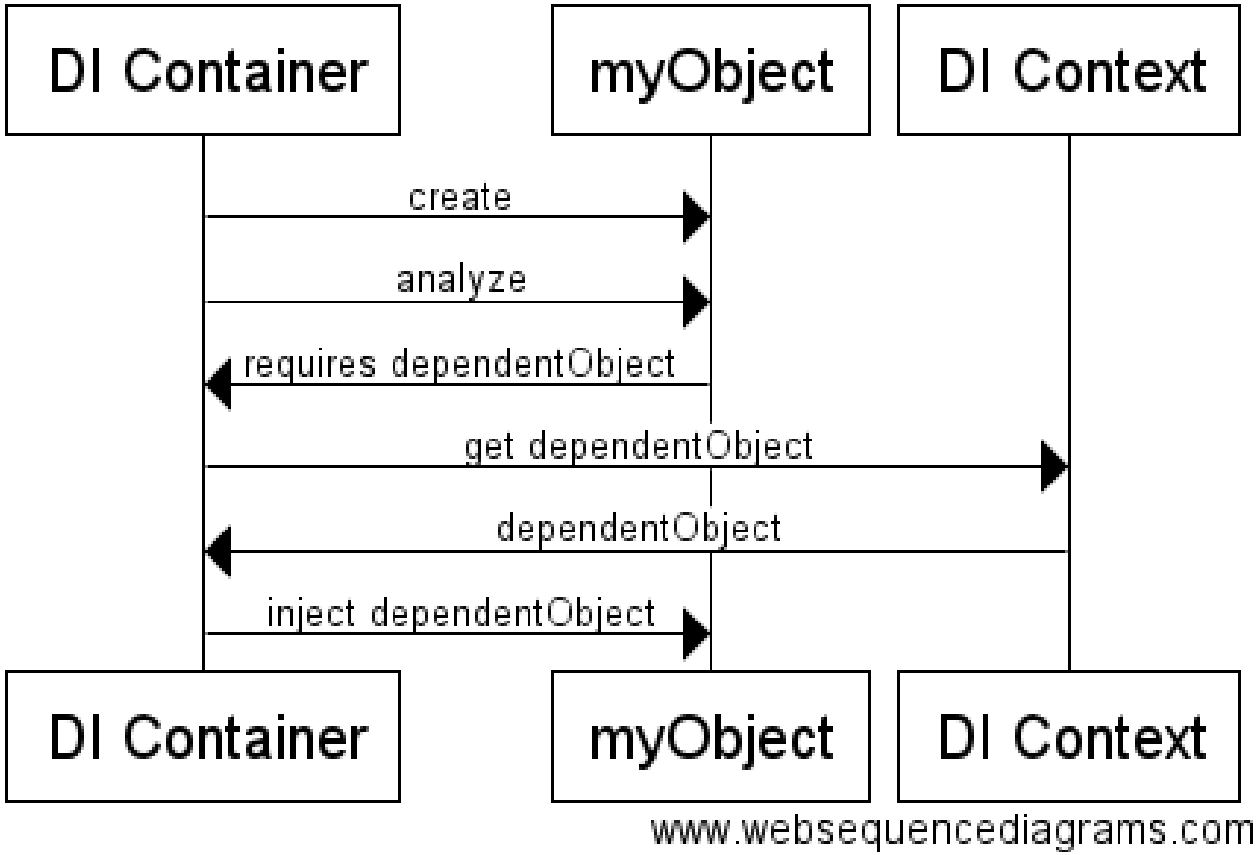
- Zentrale Ablaufumgebung der Applikation, die das Modell der Abhängigkeiten aufbaut
- Analysiert myObject mittels Reflection auf vorhandene Meta-Daten (Annotationen)
- Lookup der Abhängigkeiten im DI Context
- Initialisiert Objekte, die nicht im DI Context vorhanden sind, und speichert sie im DI Context

DI Context:

- Speichert die Objekte, die injiziert werden können
- Vereinfacht gesehen eine Map<ID, Object>

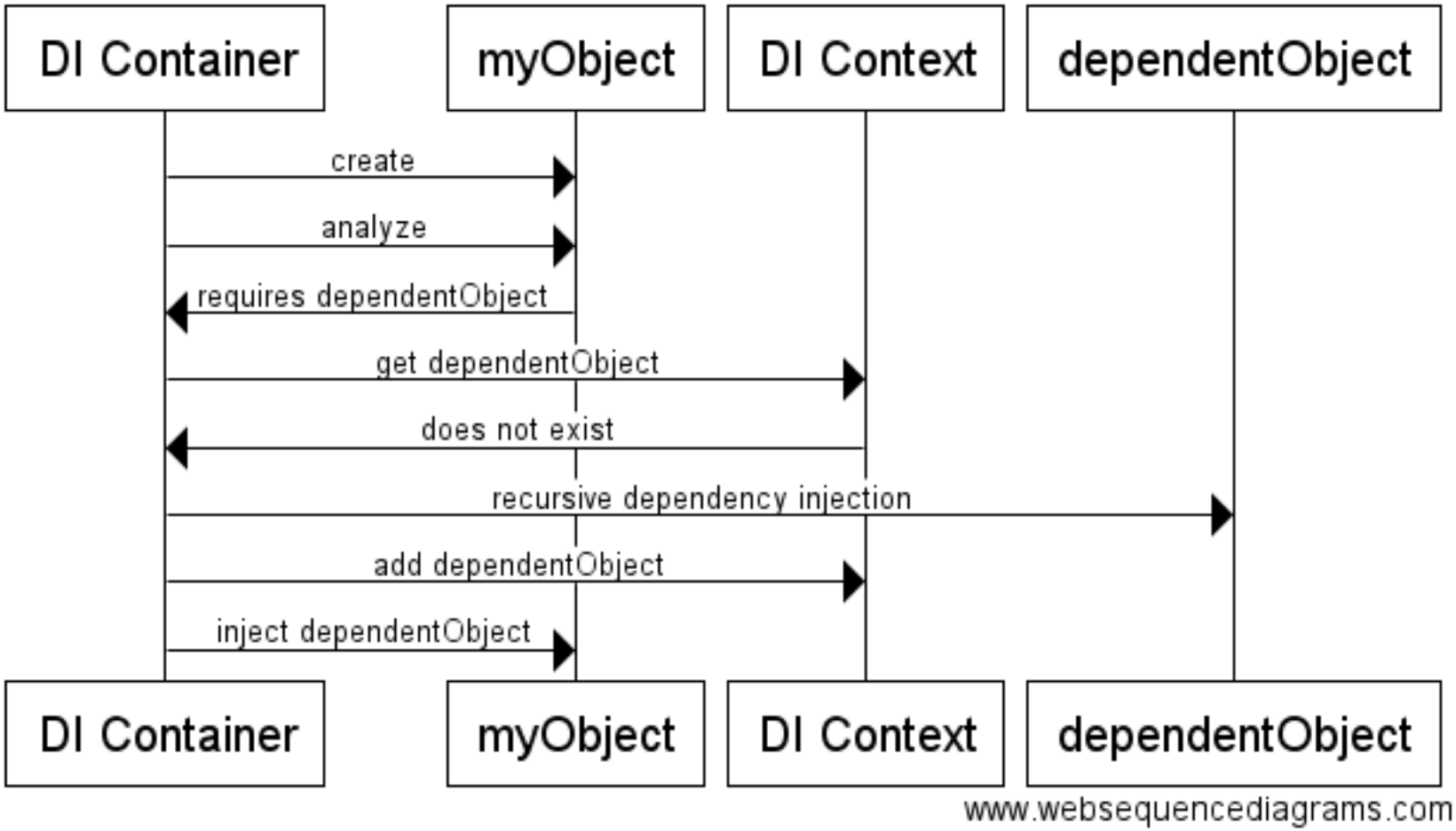
Dependency Injection – Ablauf I

Das Objekt dependentObject existiert und wird vom DI Context verwaltet



Dependency Injection – Ablauf II

Das Objekt dependentObject existiert noch nicht im DI Context.

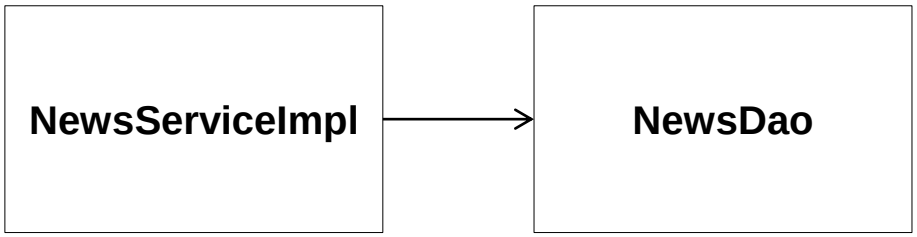


Dependency Injection

- **Arten der Dependency Injection**
  - Constructor Injection
  - Field Injection
  - Setter Injection

Dependency Injection - Beispiel

```
public interface NewsDao {
}

public class NewsServiceImpl {

  // Field Injection – Spring Framework
  @Autowired
  private NewsDao newsDao;

  // Constructor Injection
  public NewsServiceImpl(NewsDao newsDao) {
       this.newsDao = newsDao;
  }

  // Setter Injection
  public void setNewsDao(NewsDao newsDao) {
       this.newsDao = newsDao;
  }
}
```



Spring Framework

- **Entwickelt von Rod Johnson im Rahmen des Buchs "Expert One-on-One J2EE Design and Development"**
- **Erste Version 2004 veröffentlicht**
- **Dependency Injection bildet den Kern**
- **Bindet an andere Frameworks & Bibliotheken ein**
- **Besteht aus Modulen:**
  - Spring Context: DI-Container
  - Spring MVC: Web-Framework
  - Spring Data Access: Unterstützung für Datenbankzugriffe
  - Spring Test: Unterstützung für Unit-Tests
  - Spring Caching: Cache-Abstraktion

Spring Framework

- **Application Context: DI Container, der die Beans verwaltet**
- **Spring Managed Bean:**
  - Ein vom Spring Framework verwaltetes Objekt
  - Besitzen einen eindeutigen Namen
  - Haben einen bestimmten Scope
- **Konfiguration des Application Contexts:**
  - Annotationen
  - Java-Konfigurationsklassen (JavaConfig)
  - XML

Spring Framework - Annotationen

Das Spring Framework scannt beim Start definierte Packages auf Typen, die annotiert sind:

- **@Component: Allgemeine Annotation für eine Spring Managed Bean**
- **@Repository: Für DAO-Klassen**
- **@Service: Für Service-Klassen**
- **@RestController: Für REST-Endpoints**

Lifecycle-Annotationen:

- **@PostConstruct: Methode wird nach DI ausgeführt**
- **@PreDestroy: Methode wird vor dem Zerstören der Bean ausgeführt**

Spring Framework - JavaConfig

Für die Erstellung komplexerer Beans

```
@Configuration
public class NewsServiceConfig {
    @Autowired
    private NewsDaoConfig newsDaoConfig;

    @Bean
    public NewsService createNewsService() {
      NewsService newsService = new NewsServiceImpl();
      newsService.setNewsDao(newsDaoConfig.createNewsDaoConfig());
      return newsService;
}
```

Spring Managed Bean - Beispiel

```
@Service
public class NewsServiceImpl {
  private NewsDao newsDao;

  @Autowired
  public NewsServiceImpl(NewsDao newsDao) {
    this.newsDao = newsDao;
  }

  @PostConstruct
  public void init() {
    // do something
 }
}
```

Scope

Gibt den Lebenszeitraum einer Bean an

- **Application bzw Singleton: Für die Lebensdauer des Application Contexts (Default-Scope)**
- **Session: Für eine Benutzersitzung**
- **Request: Für einzelne Requests**
- **Spezielle Scopes: View, Process**

Standardmässig werden Beans mit längerer Lebenszeit in Beans mit gleicher oder kürzerer Lebenszeit injectet

Objekt-relationales Mapping (ORM)

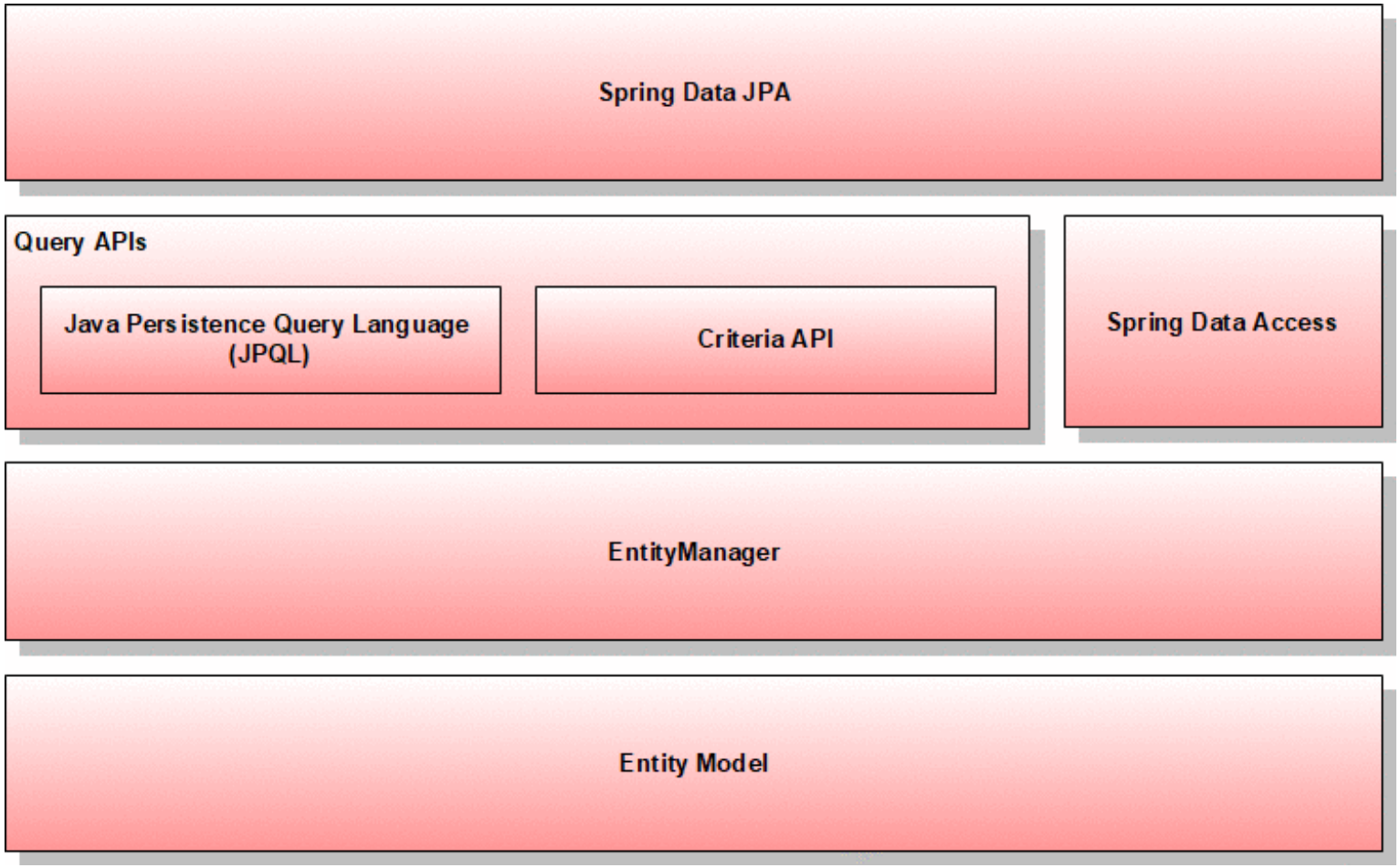
- **Verfahren zur Speicherung von Objekten in Datenbanken**
- **Mapping von Klassen und Objekten auf Tabellen und Zeilen**
- **O/R-Impedance Mismatch:**
  - Objektidentität
  - Datentypen
  - Relationen
  - Vererbung

Java Persistence API 2.2 (JPA 2.2)

- Offizieller Standard für ORM in Java (JSR-338)
- Von Hibernate und TopLink inspiriert
- Nutzt Convention-over-Configuration
- Verschiedene Implementierungen:

  Hibernate, EclipseLink, Apache OpenJPA

Aufbau von JPA / Spring Data JPA



Entities

Annotierte Java Beans, die in Datenbank persistiert werden

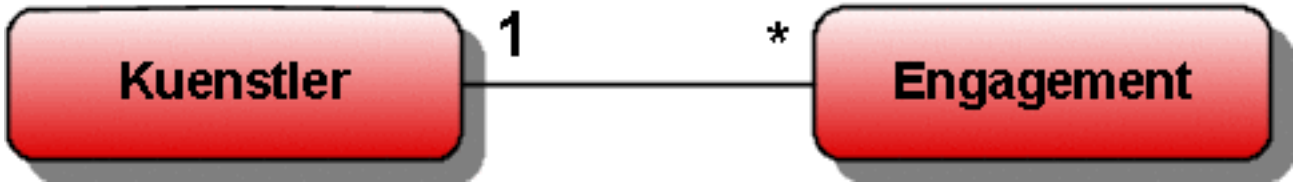
```
@Entity
@Table( name="artist" ) // optional
public class Artist implements Serializable {

private String firstname;

// Field Access
@Column(name = "name", nullable = false, length = 50)
private String lastname;

// Property Access
@Column(name = "title_long")
public String getTitle() {
        return this.title;
}

// Keine Persistierung
@Transient
Private Integer sum;
```

Primary Key Mapping

- **Mittels @Id**
- **Strategien:**
  - Automatische Generierung durch Datenbank
  - Eigene Tabelle, aus der Id generiert wird
  - Datenbank Sequenzen

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Integer id;
```

Relationen (1 / 2) - Beispiel



- **Klasse Kuenstler**

```
@OneToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL,
    mappedBy = "kuenstler")

private Set<Engagement> engagements = new
    HashSet<Engagement>();
```

- **Klasse Engagement**

```
@ManyToOne(fetch=FetchType.EAGER)

@JoinColumn(name="KUENSTLER_ID")

private Kuenstler kuenstler;
```

Relationen (2 / 2)

- **Typen:**
  - OneToOne
  - OneToMany / ManyToOne
  - ManyToMany
- **Richtung:**
  - unidirektional
  - bidirektional
- **Loading Strategie:**
  - Eager: Bei Abfrage
  - Lazy: Bei erstem Zugriff
- **Cascading:**
  - All, Persist, Merge, Refresh, Remove, Detach

Vererbung

- **Single Table per Class Hierarchy Strategy:**

  Eine Tabelle mit den Attributen aller Klassen

- **Joined Subclass Strategy:**

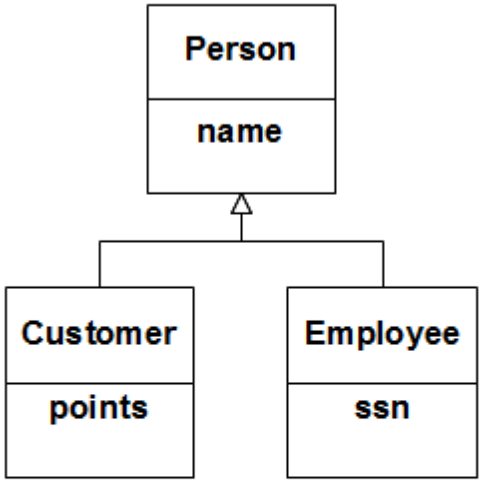
  Normalisiertes Datenbankschema, vererbte Klassen werden gejoint

- **Table Per Concrete Class Strategy:**

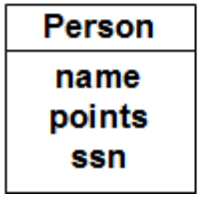
  Eine Tabelle je Klasse mit allen Attributen der Klasse

Vererbung - Beispiel

Java Class Hierarchy



Single Table per Class Hierarchy



Joined Subclass

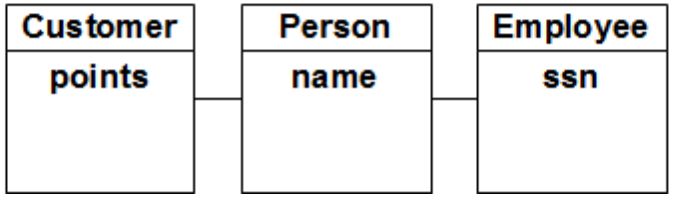
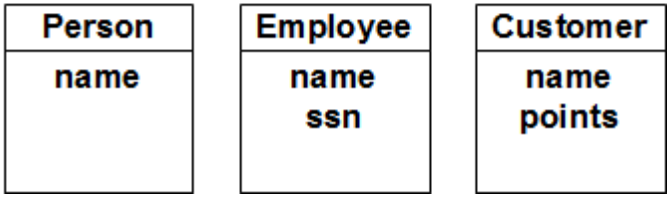


Table per Concrete Class



EntityManager

- **Zentrales Interface für die Verwendung von JPA**
- **Verwaltet den Persistence Context mit allen geladenen Entities**
- **Managt die Transaktionen**
- **Verantwortlich für das Cachen von Entities (First und Second Level Cache)**

Entities

```
Kuenstler k = new Kuenstler();
k.setNachname("Dent");
entityManager.persist(k);

k.setNachname("Beeblebrox");

Kuenstler k = entityManager.findById(Kuenstler.class,1);
entityManager.remove(k);
```

Transaction Management – Standard JPA

```
EntityTransaction tx = entityManager.getTransaction();
try {
  tx.begin();
  // do something
  tx.commit();
} catch (Exception e) {
  try {
    tx.rollback();
  } catch (Exception e) { /* ignore */ }
}
```

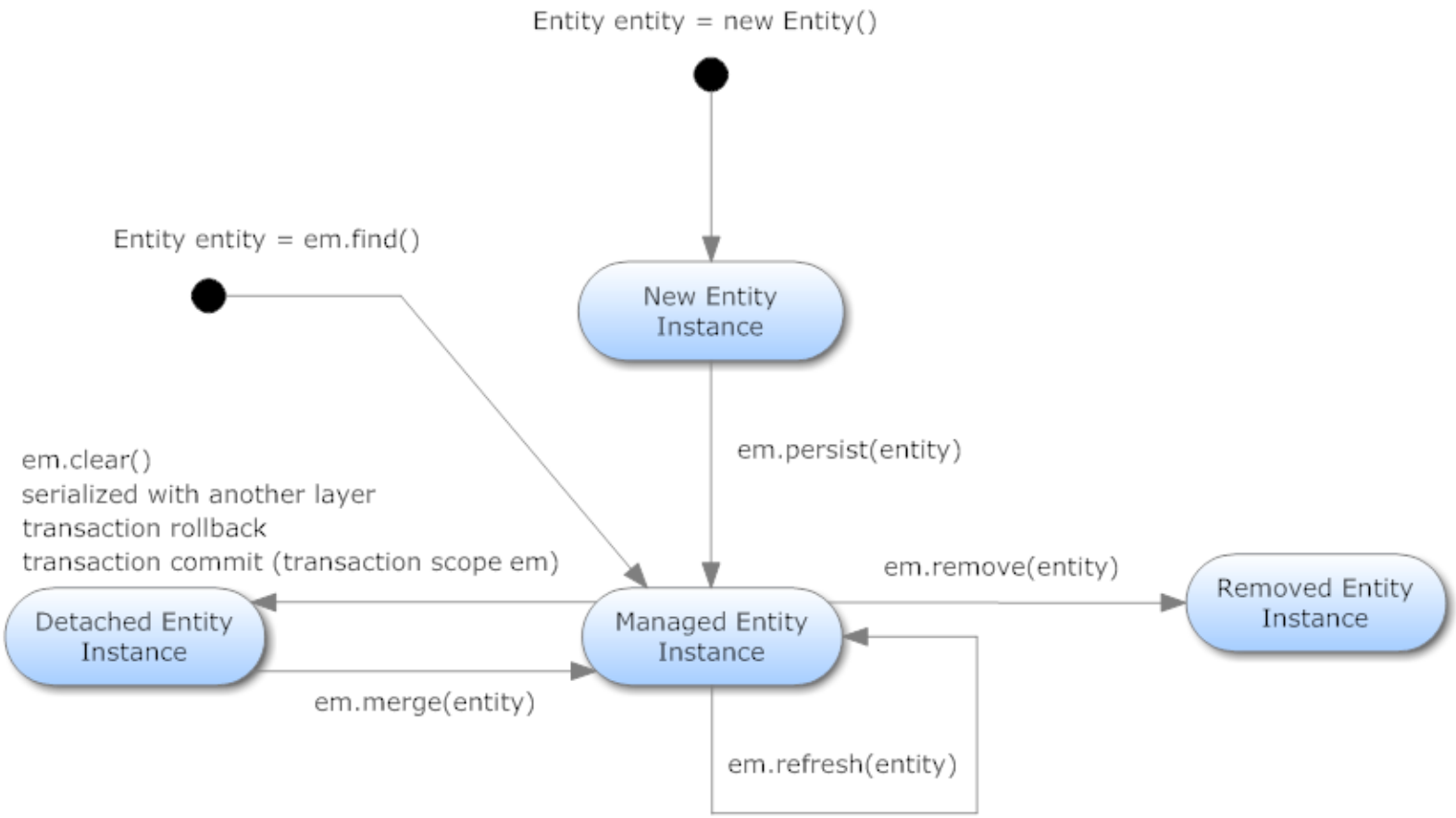
Transaction Management – Spring @Transactional

```
@Transactional
public void doSomething() {
  // do something
}
```

Varianten:

- **Position: Typ (Interface, Klasse), Methode**
- **Ebene: Interface, Klasse**
- **Typ: Schreibend, Lesend**

Entity-Lifecycle in JPA



Quelle: https://kptek.wordpress.com/2012/06/20/entity-lifecycle/

Lazy Loading

- **Transparentes Nachladen von Relationen, die den FetchType.LAZY besitzen:**

```
Adresse a = kuenstler.getAddresse();
```

- **Lazy Loading erfolgt in eigenen SELECT-Queries**
- **Voraussetzung: Entities müssen im Status „Managed“ sein (aktive Transaktion)**
- **Achtung: Hauptursache für schlechte Performance (n+1-Problem)!**
- **Anti-Pattern:**

```
kuenstler.getAdresse().getLand().getHauptstadt().getBezirke()
```

Queries

- **Standard Queries**

  TypedQuery<Entity> q = entityManager.createQuery(JPQL String oder Criteria Query, Entity.class);

- **Native Queries**

  Query q = entityManager.createNativeQuery(SQL String);

- **Named Queries**

  TypedQuery<Entity> q = entityManager.createNamedQuery(Query Name, Entity.class);

- **Query-Interface:**
  - setParameter(name,obj)
  - setParameter(position,obj)
  - getResultList();
  - getSingleResult();

Java Persistence Query Language (JPQL)

- **Query Language von JPA**
- **basiert auf SQL, arbeitet auf Objektebene**
- **SELECT p FROM Person p WHERE p.lastname = 'Dent';**
- **Named Parameter als :name**
- **Positional Parameter als ?1**
- **Zeichen-Substitution in LIKE:**
  - Einzelner Buchstabe: _
  - Mehrere Buchstaben: %
- **Unterstützung von UPDATE und DELETE für Batch-Operationen**

JPQL-Beispiel

```
String queryString =
„SELECT p FROM Person p WHERE p.lastname = :name“;

TypedQuery<Person> q = entityManager.createQuery(
  queryString, Person.class
);

q.setParameter("name", "Dent");

q.setMaxResults(10);

List<Person> personen = q.getResultList();
```

Criteria API

- **Programmatische Erstellung von Queries über ein API**
- **Für dynamische Abfragen**
- **Erstellen über entityManager.getCriteriaBuilder()**
- **Zugriff auf Attribute über**
  - Statisches Metamodel, zB Person_.nachname
  - Dynamisch, zB personRoot.get("nachname");

```
CriteriaQuery<Customer> q = cb.createQuery(Customer.class);
Root<Customer> customer = q.from(Customer.class);
q.select(customer);
```

Validierung in JPA

- **JSR 303 – Bean Validation**
- **Automatische Validierung von Entities vor der Persistierung**
- **Validierungsinformationen als Annotationen in den Entities**

```
public class Person {
@NotNull @Size(min = 5, max = 50)
private String name;

@Past
private Date birthday;

@Min(1) @Max(500)
private Integer points;
```

Spring Data Access

- **Deklaratives Transaktionsmanagement mittels @Transactional**
- **Eigene Exception-Hierarchie mit DataAccessException als Root-Exception (Achtung: RuntimeExceptions!)**
- **Automatische Exception-Übersetzung**
- **@Repository für DAOs**

Spring Data JPA

- **Zusatzmodul für das Spring Framework**
- **Vereinfachte Entwicklung von DAOs**
- **Beinhaltet CRUD-Funktionalität**
- **Zentrale Interfaces**
  - JpaRepository
  - PagingAndSortingRepository
- **Queries per**
  - Annotation
  - Methodenname
  - XML-basierte Named Queries
- **Eigener Code nur bei speziellen Abfragen notwendig**

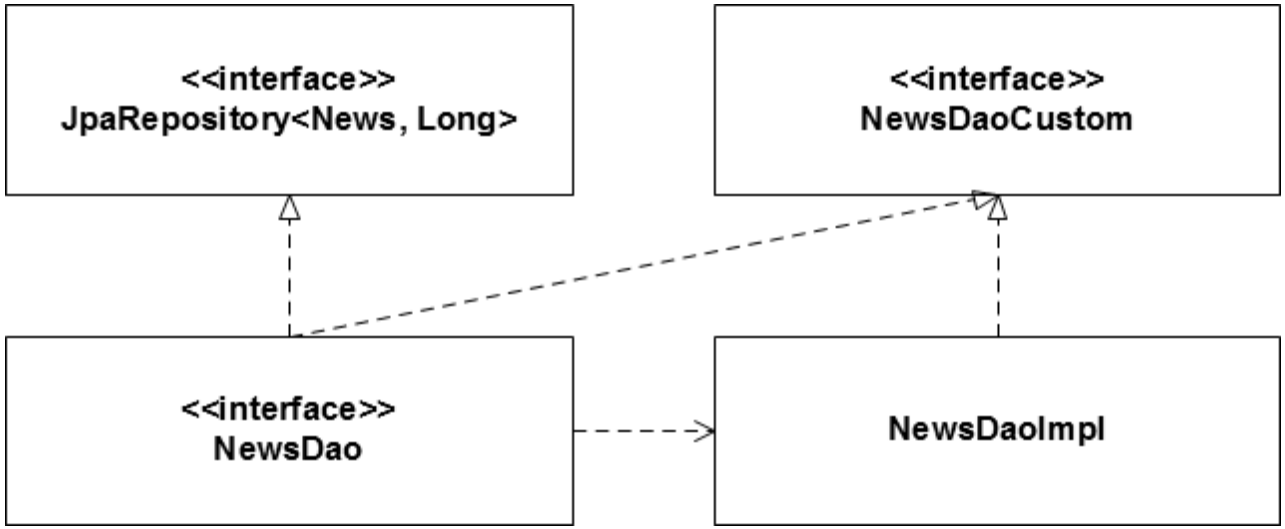
Spring Data JPA Beispiel

Methoden werden nur im Interface definiert → Keine Implementierung notwendig

```
@Repository
public interface NewsDao extends JpaRepository<News, Integer> {
// Vererbte Methoden von CrudRepository
public News save(News n);
public News findOne(Integer id);
public List<News> findAll();
public News delete(Integer id);

@Query("SELECT n FROM News n WHERE n.title = :title")
public List<News> findNews(@Param(„title“) String titleQuery);
// Achtung: Bei LIKE %:title verwenden

public List<News> findByTitleAndAuthor(String title, String author);
```

Methodenname: findBy${property}${keyword}${property}
Keywords: And, Or, Between, LessThan, GreaterThan, IsNull, IsNotNull, usw

Spring Data JPA – Custom Query-Methods – 1 / 2



- **JpaRepository: Von Spring Data definiertes Interface**
- **NewsDaoCustom: Definiert zusätzliche Methoden**
- **NewsDaoImpl: Implementiert zusätzliche Methoden; Name muss auf Impl enden**
- **NewsDao: Implizite Verwendung der Implementierung der NewsDaoImpl**

Spring Data JPA – Custom Query-Methods – 2 / 2

```
public class NewsDaoImpl implements NewsDaoCustom {

  @PersistenceContext
  private EntityManager em;

  public News findNews() {
      return (News) this.em
            .createQuery("SELECT n FROM News n WHERE n.id = 1")
            .getSingleResult();
 }
}

public class NewsServiceImpl implements NewsService {
  @Autowired
  private NewsDao newsDao;
}
```

Services

```
@Service
public class NewsServiceImpl implements NewsService {

@Autowired
private NewsDao newsDao;

@Autowired
private PersonDao personDao;

@Transactional
public News getNews(Integer personId) throws ServiceException {
    Person p = personDao.findOne(personId);
    p.getAddress(); // lazy-loading

    List<News> news = newsDao.findNewsByPerson(p);
```

Architektur Gruppenphase