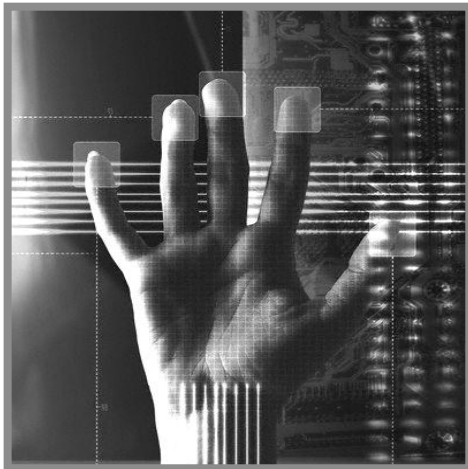


Software Engineering und Projektmanagement



Ticketline 3.1

2018S - 12. April 2018

Wolfgang Gruber

Email: sepm@inso.tuwien.ac.at

Web: <http://www.inso.tuwien.ac.at/lectures/>



INSO - Industrial Software

Institut für Rechnergestützte Automation | Fakultät für Informatik | Technische Universität Wien

- 1 Einführung
- 2 Dependency Injection & Spring Framework
- 3 OR-Mapping & Java Persistence API
- 4 REST-Services
- 5 REST-Client

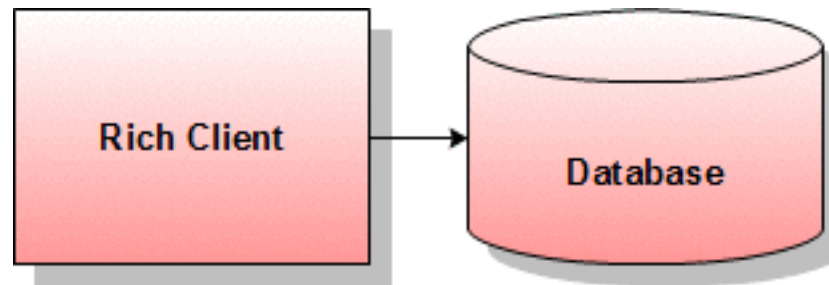
System für Ticket-Verkauf für Veranstaltungen (Kino, Theater, Konzerte)

Funktionalität:

- Ticket-Verkauf
- Prämien-System
- Verkauf von Merchandising-Artikeln
- Auswertungen

Einzelbeispiel

- **Client-Server-Architektur**

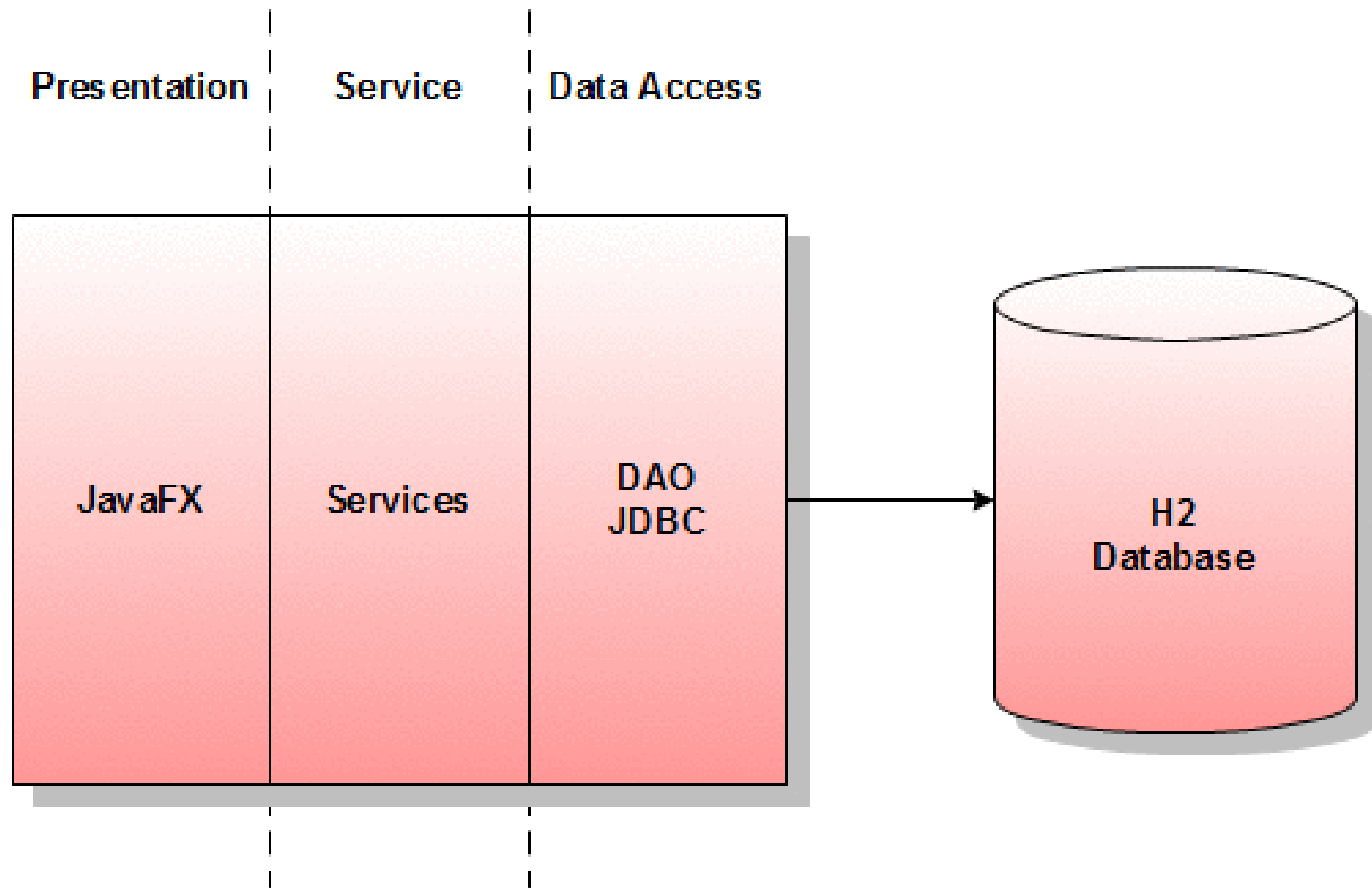


Ticketline

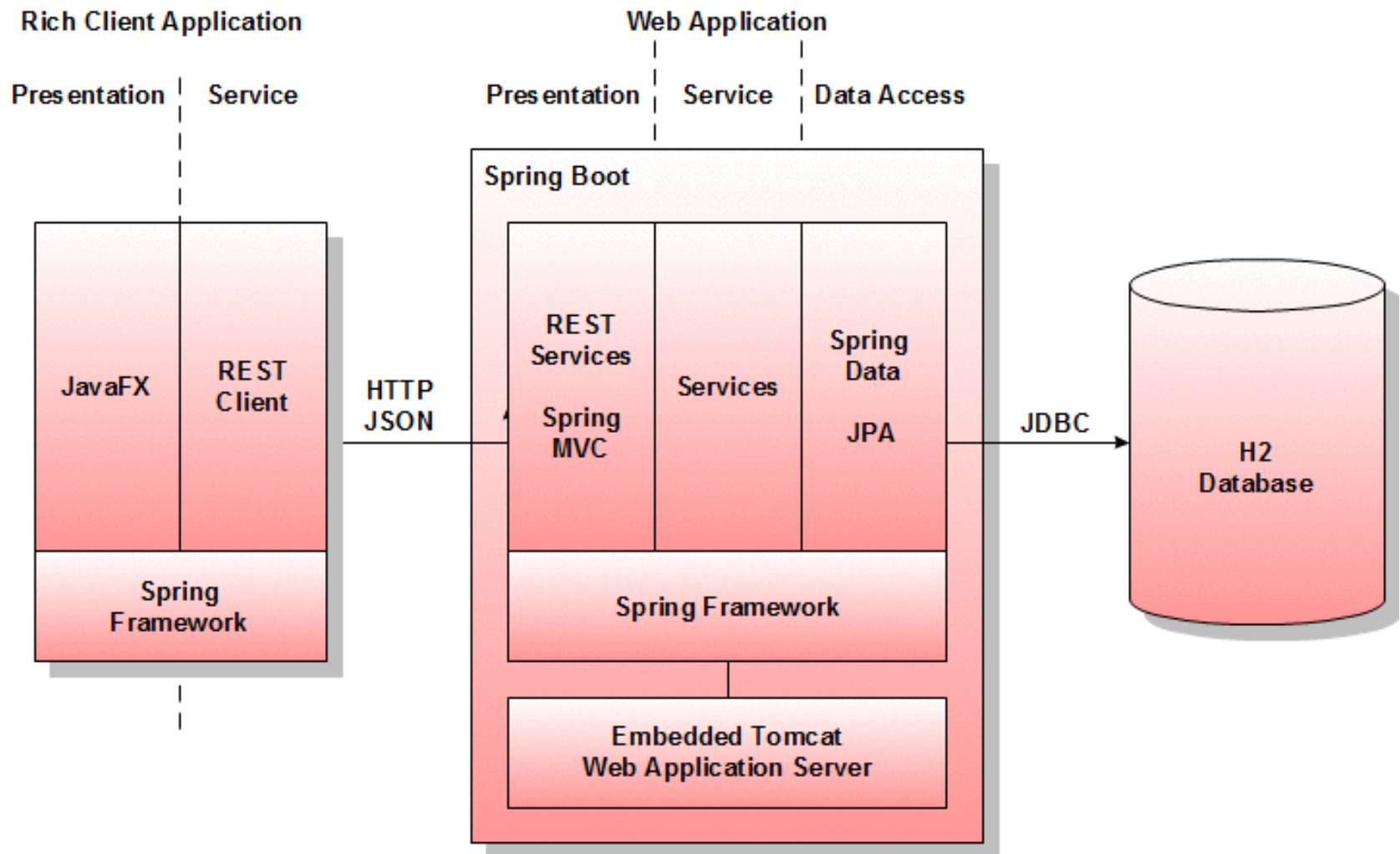
- **REST-basierte Architektur mit Rich Client**



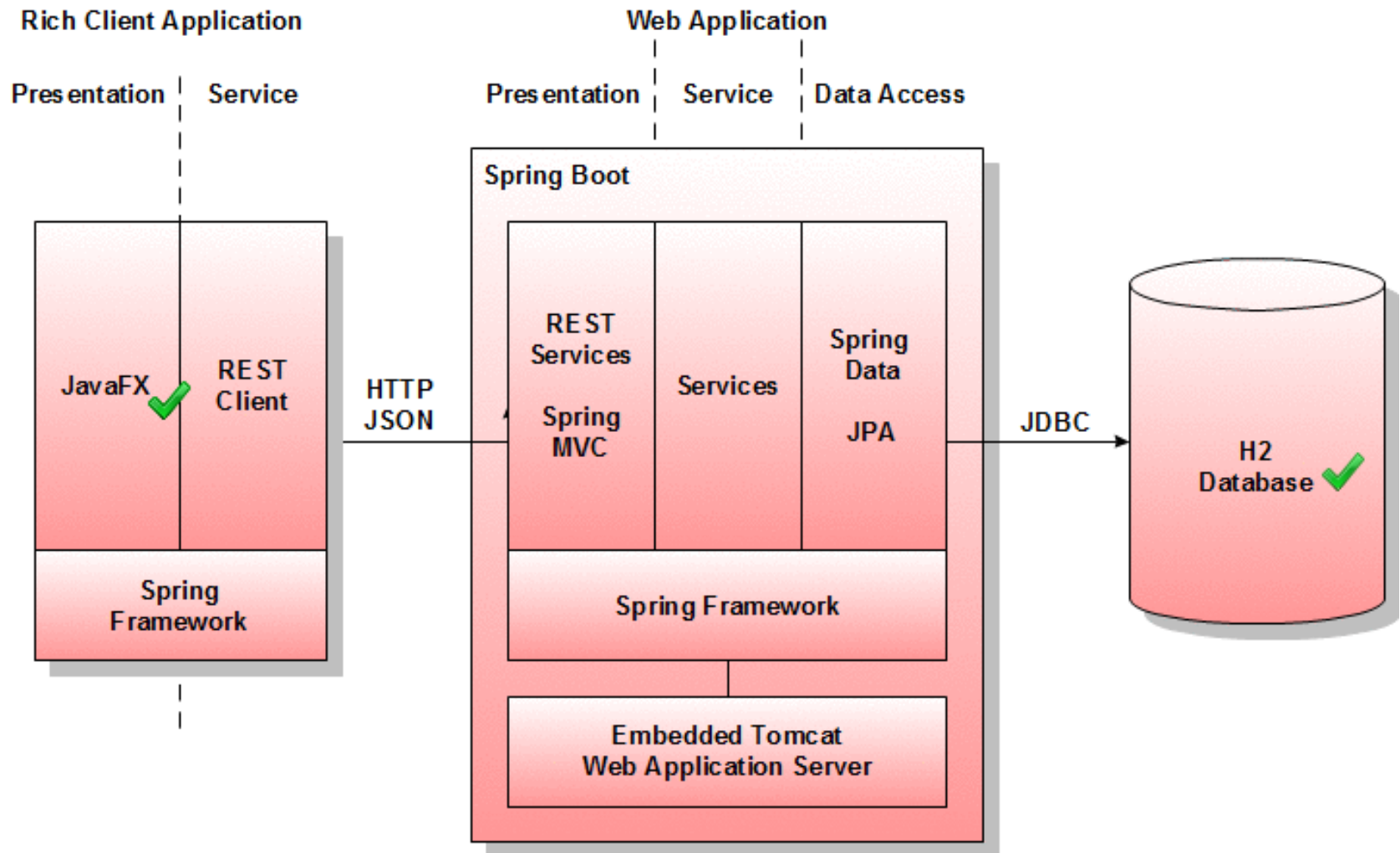
Architektur des Einzelbeispiels



Architektur von Ticketline



Ticketline Architektur



Java Enterprise Edition (Java EE)

- **Sammlung von Standards (Java Specification Request; JSR) für die Entwicklung von Enterprise-Applikationen**
- **Entwicklung im Rahmen des Java Community Process (JCP)**
- **Aktuelle Version: Java EE 8 (seit Herbst 2017)**
- **Ticketline: Verwendung von Java EE 7-Standards**
- **Zukünftig: Weiterentwicklung als Jakarta EE durch die Eclipse Foundation**

Java SE- und Java EE-Plattform

Java EE Platform Specification

Servlet

JPA

Bean
Validation

JSF

CDI

EJB

JAX-RS

...

Java SE Platform Specification

JDBC

JAXP

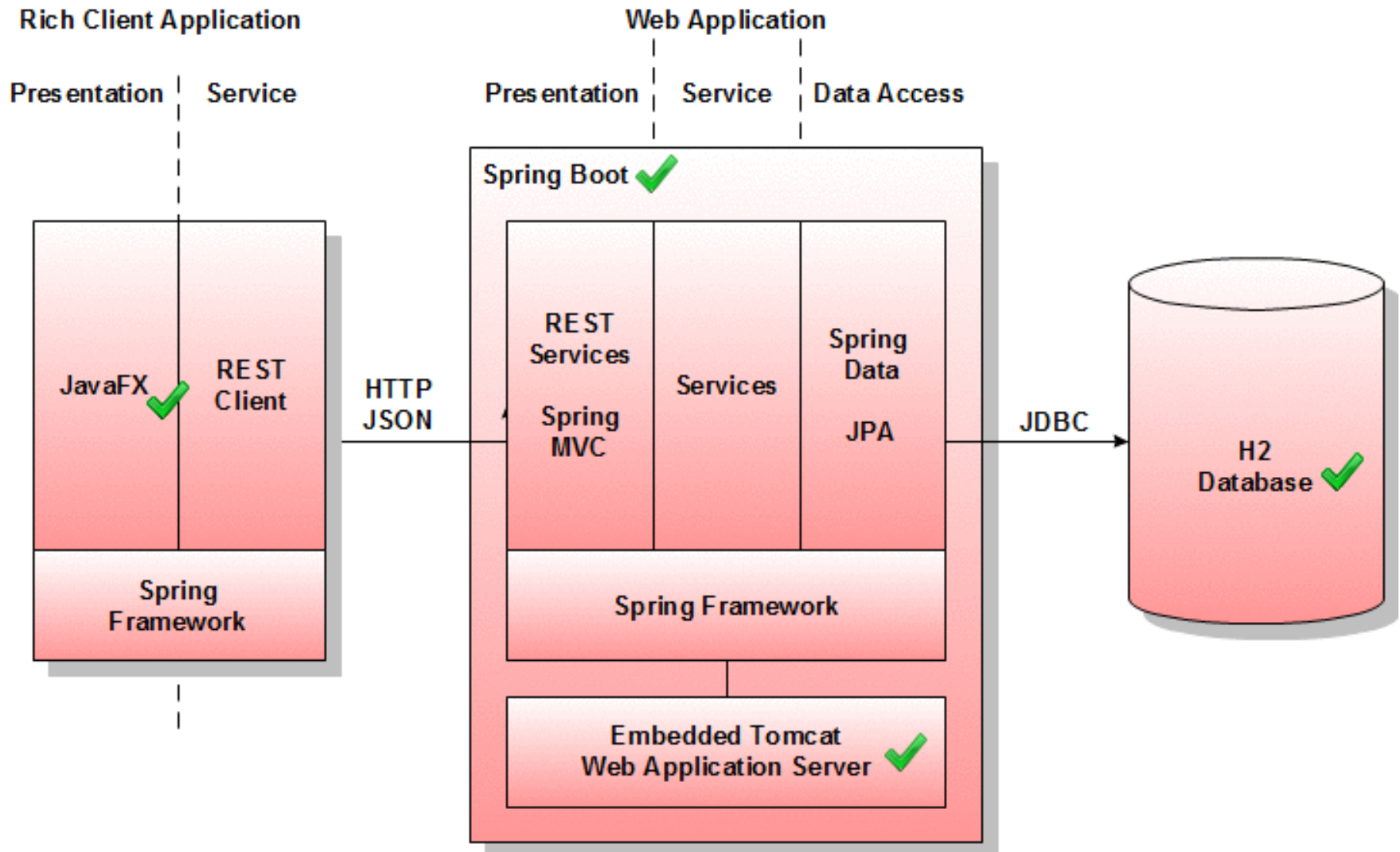
JPMS

..

- **Applikationen, die mittels HTTP aufgerufen werden**
 - GUI: HTML, JavaScript, CSS, Bilder
 - Services: REST-Services, SOAP
- **Packaging in Web Archives (*.war), die auf einem Server deployt werden**
- **Mittels Servlet API (JSR 340)**
 - Definiert die Struktur der WAR-Datei
 - API für die Verarbeitung von HTTP-Requests
 - Servlet: Endpunkt am Server

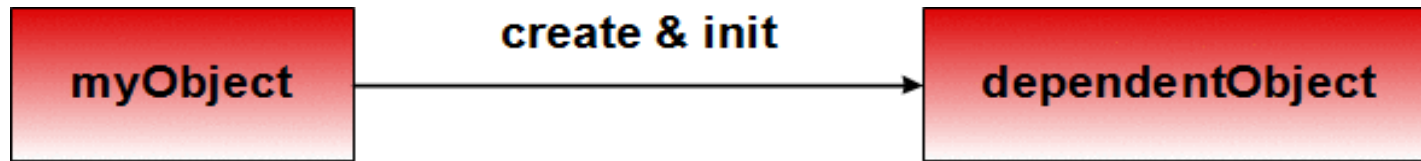
- **Applikationsframework für das einfache Setup von Spring-Applikationen („opinionated defaults configuration“)**
- **Einfache Integration von vordefinierten Bibliotheken**
- **Einheitliche Applikationskonfiguration (Properties, YAML)**
- **Embedded Web Application Server (Apache Tomcat)**
- **Monitoring-Features: Healthchecks, Metriken**
- **JAR-Datei, die Web Application Server und Web Application enthält**

Ticketline Architektur



- **Design Pattern, bei dem Abhängigkeiten zu anderen Objekten durch einen Container aufgelöst werden**
- **Ältere Namen:**
 - Inversion of Control
 - Hollywood-Principle („Don't call us, we call you“)
- **Erreichte durch das Spring Framework eine weite Verbreitung**
- **In den letzten Jahren verstärkte Nutzung von Annotationen**

Direkte Instanziierung



Instanziierung:

```
SqlConnection con = new SqlConnection(); // Beispiel-Code
```

Initialisierung:

```
con.setUsername("user");  
con.setPassword("streng-geheim");  
con.setUrl("jdbc:hsqldb:hsqldb://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

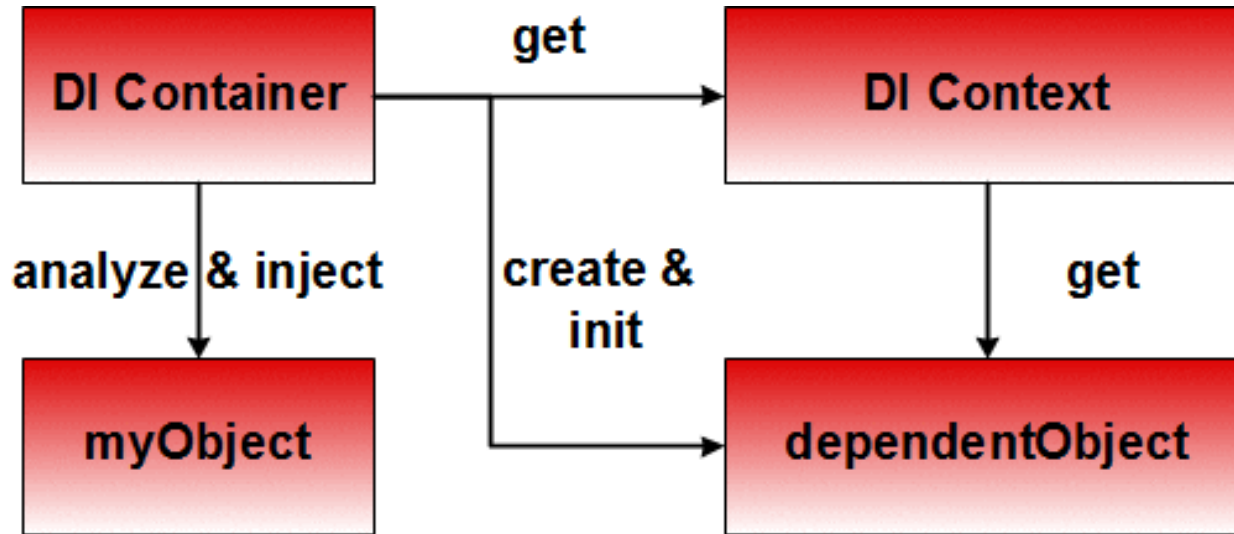
Verwendung:

```
Connection con = DbConnection.getConnection();
```

Probleme:

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

Dependency Injection (DI)



DI Container:

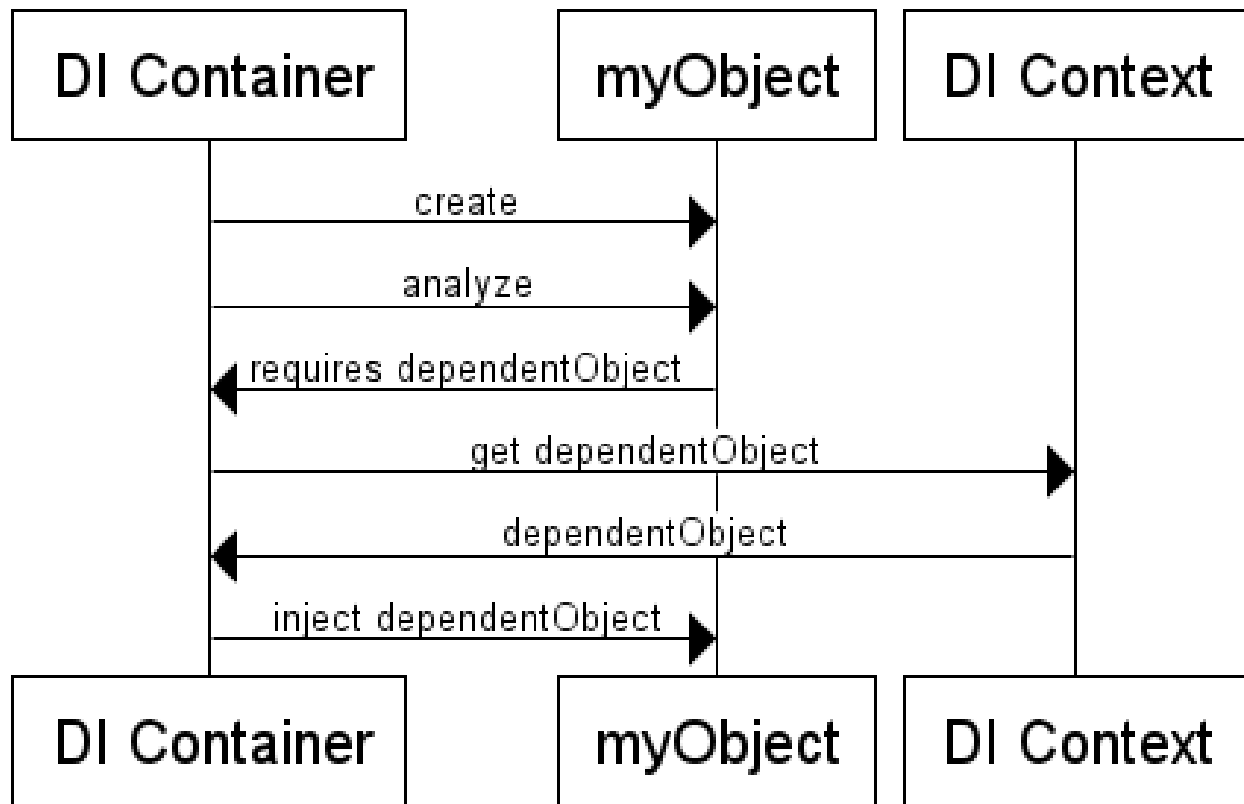
- Zentrale Ablaufumgebung der Applikation, die das Modell der Abhängigkeiten aufbaut
- Analysiert myObject mittels Reflection auf vorhandene Meta-Daten (Annotationen)
- Lookup der Abhängigkeiten im DI Context
- Initialisiert Objekte, die nicht im DI Context vorhanden sind, und speichert sie im DI Context

DI Context:

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

Dependency Injection – Ablauf I

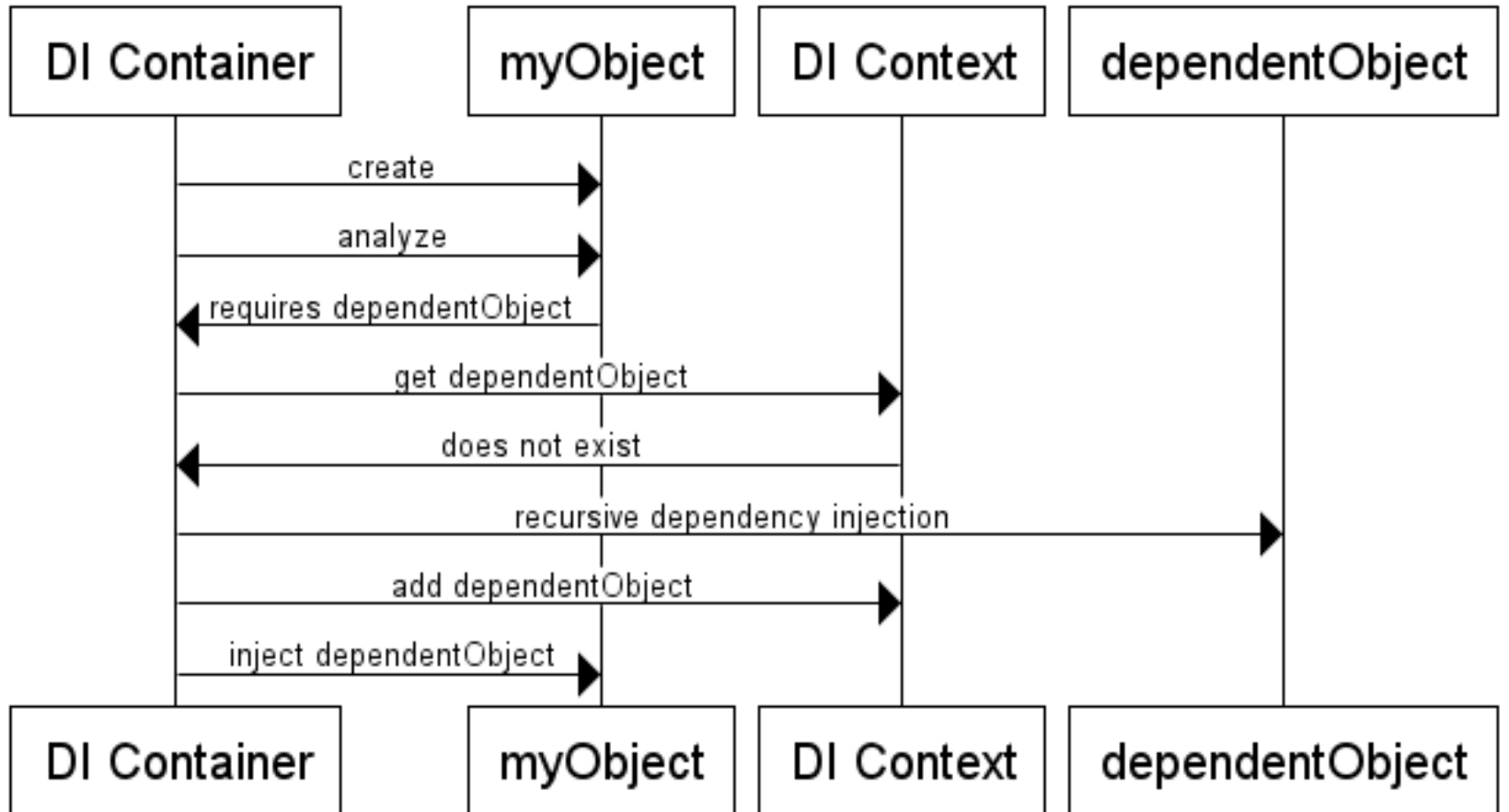
Das Objekt dependentObject existiert und wird vom DI Context verwaltet



www.websequencediagrams.com

Dependency Injection – Ablauf II

Das Objekt dependentObject existiert noch nicht im DI Context.



www.websequencediagrams.com

Dependency Injection

- **Arten der Dependency Injection**
 - Constructor Injection
 - Field Injection
 - Setter Injection
- **Scope: Gibt den Lebenszeitraum eines Objekts an**
 - Application bzw Singleton: Für die Lebensdauer der Applikation
 - Session: Für eine Benutzersitzung
 - Request: Für einzelne Requests
 - Spezielle Scopes: View, Process

Dependency Injection - Beispiel

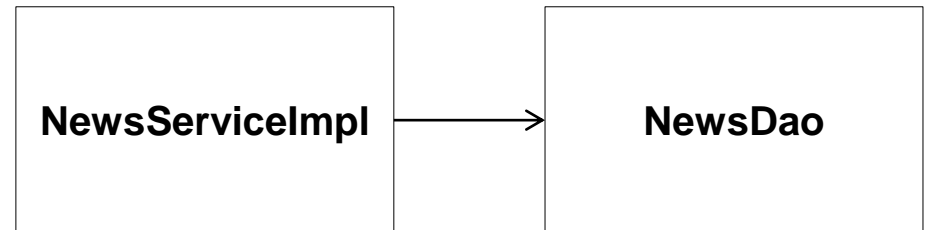
```
public interface NewsDao {  
}
```

```
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;  
}  
}
```



- **Entwickelt von Rod Johnson im Rahmen des Buchs "Expert One-on-One J2EE Design and Development"**
- **Erste Version 2003 veröffentlicht**
- **Dependency Injection bildet den Kern**
- **Anbindung an andere Frameworks & Bibliotheken**
- **Weitere Module für spezielle Domänen**
 - Spring Batch
 - Spring Security
 - Spring Integration
 - Spring Cloud

- **Application Context: Verwaltet die Bean Definitions & Beans**
- **Bean: Ein vom Spring Framework verwaltetes Objekt**
- **Bean Definition: Metadaten zu einer Bean; Dienen zur Konstruktion der Bean und Auflösen der Dependencies**
- **Konfiguration des Application Contexts:**
 - Annotationen
 - Java-Konfigurationsklassen (JavaConfig)
 - XML

Spring Framework - JavaConfig

@Configuration

```
public class NewsServiceConfig {
```

```
    @Inject
```

```
    private NewsDaoConfig newsDaoConfig;
```

@Bean

```
public NewsService createNewsService() {
```

```
    NewsService newsService = new NewsServiceImpl();
```

```
    newsService.setNewsDao(newsDaoConfig.createNewsDaoConfig());
```

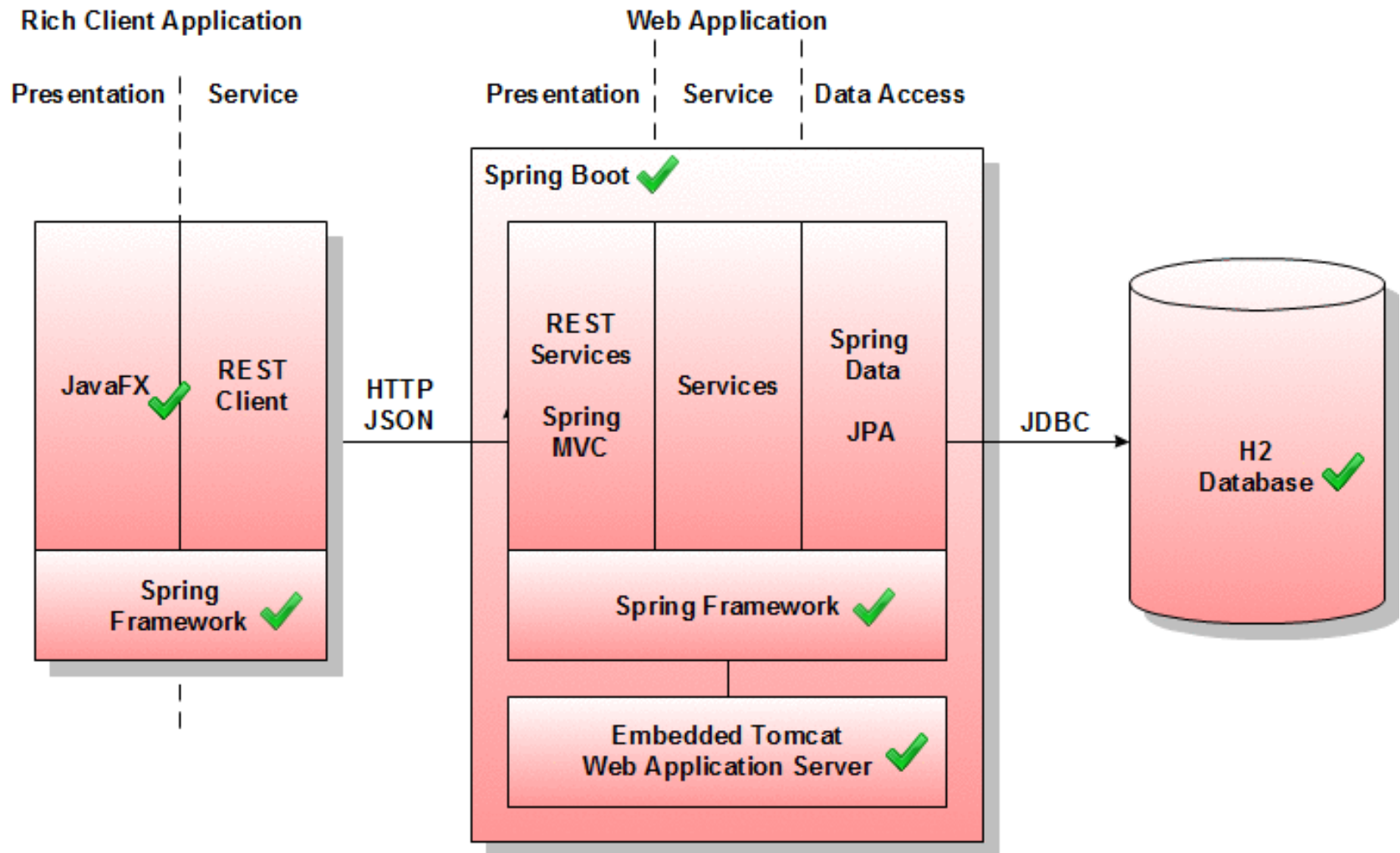
```
    return newsService;
```

```
}
```

Spring Framework Injection Annotationen

- **@Autowired** – Injiziert eine Bean
- **@Component** – Definiert ein Objekt als Bean
- **@Repository** – Spezialisierung von **@Component** für Data Access-Layer
- **@Service** – Spezialisierung von **@Component** für Service-Layer
- **@Controller** bzw. **RestController** – Spezialisierung von **@Component** für Presentation-Layer
- **@PostConstruct** - Lifecycle-Methode zum Initialisieren einer Bean
- **@PreDestroy** – Lifecycle-Methode zum Aufräumen einer Bean

Ticketline Architektur



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

- **Offizieller Standard für ORM in Java (JSR-338)**
- **Bestandteil von Java EE, kann aber auch in Java SE-Applikationen verwendet werden**
- **Von Hibernate und TopLink inspiriert**
- **Nutzt Convention-over-Configuration**
- **Verschiedene Implementierungen:**

Hibernate, EclipseLink, Apache OpenJPA

- **Bestandteile:**
 - Entity- und Relationen-Modell
 - EntityManager
 - Query-Mechanismen: JPQL, Criteria API

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:

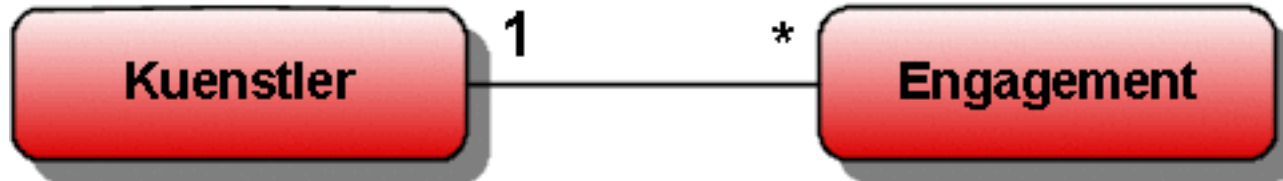
- Automatische Generierung durch Datenbank
- Eigene Tabelle, aus der Id generiert wird
- Datenbank Sequenzen

@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")
```

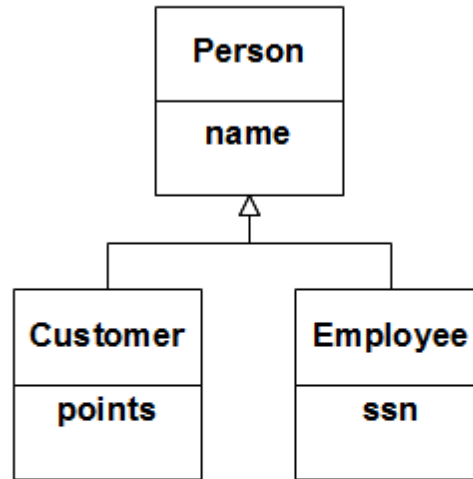
```
private Kuenstler kuenstler;
```

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

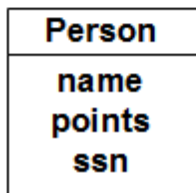
- **Single Table per Class Hierarchy Strategy:**
Eine Tabelle mit den Attributen aller Klassen
- **Joined Subclass Strategy:**
Normalisiertes Datenbankschema, vererbte Klassen werden gejoint
- **Table Per Concrete Class Strategy:**
Eine Tabelle je Klasse mit allen Attributen der Klasse

Vererbung - Beispiel

Java Class Hierarchy



Single Table per Class Hierarchy



Joined Subclass

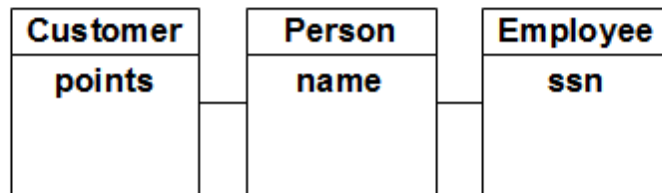
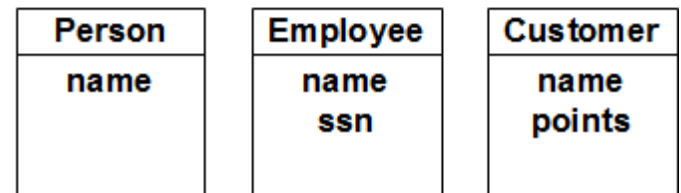


Table per Concrete Class



- **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)**

Transaction Management – Standard JPA

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

@Transactional

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

Varianten:

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

Best Practice: Annotation auf Interface-Ebene

```
Kuenstler k = new Kuenstler();
```

```
k.setNachname("Dent");
```

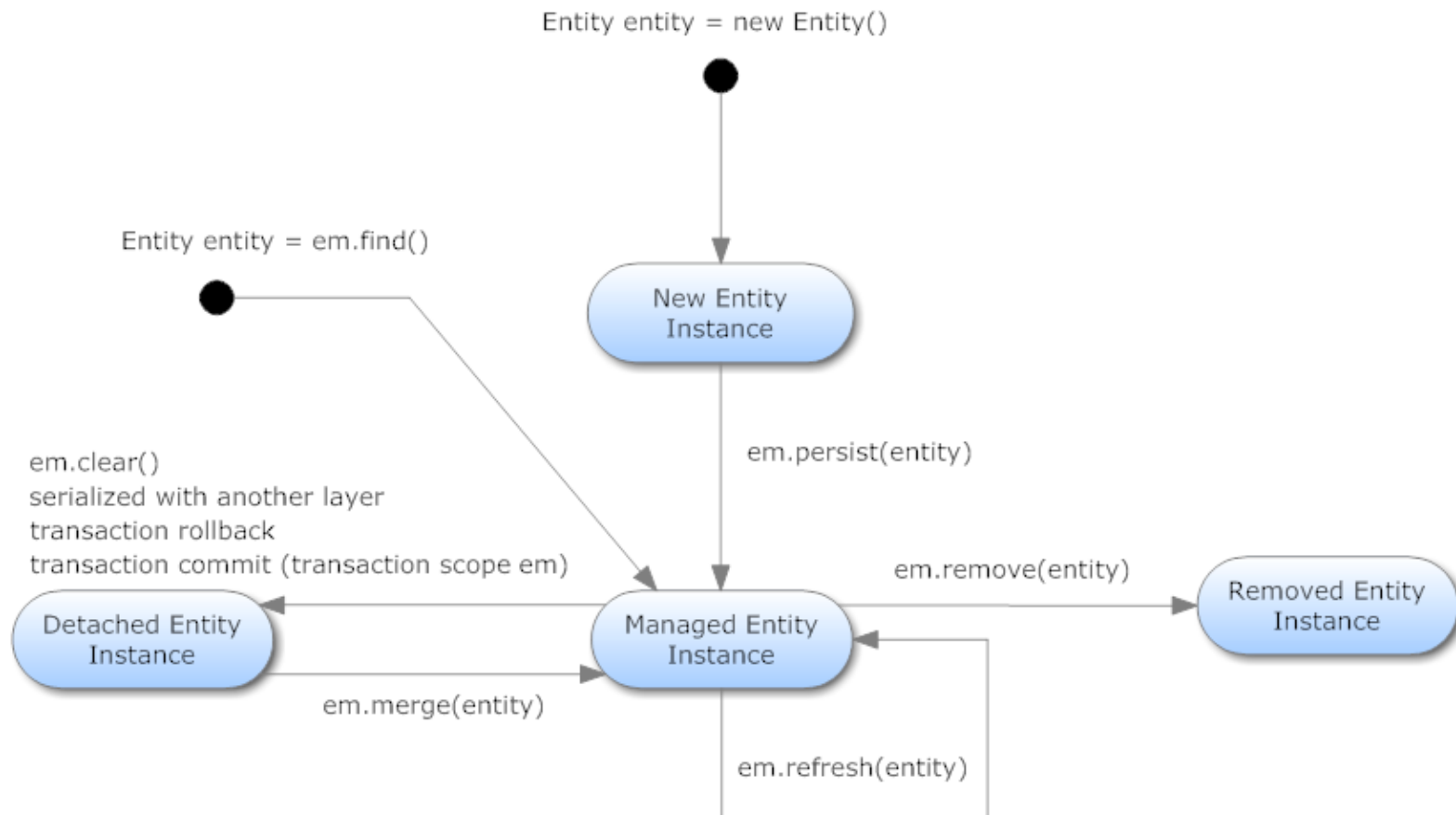
```
entityManager.persist(k);
```

```
k.setNachname("Beeblebrox");
```

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

```
entityManager.remove(k);
```

Entity-Lifecycle in JPA



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

- Transparentes Nachladen von Relationen, die den `FetchType.LAZY` besitzen:
- Adresse `a = kuenstler.getAdresse();`
- 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().getBezirk()`

- **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 a FROM Artist a WHERE a.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**

```
String queryString =  
„SELECT a FROM Artist a WHERE a.lastname = :name“;
```

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

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

```
q.setMaxResults(10);
```

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

- **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);
```

- **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;
```

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

- **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**

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")

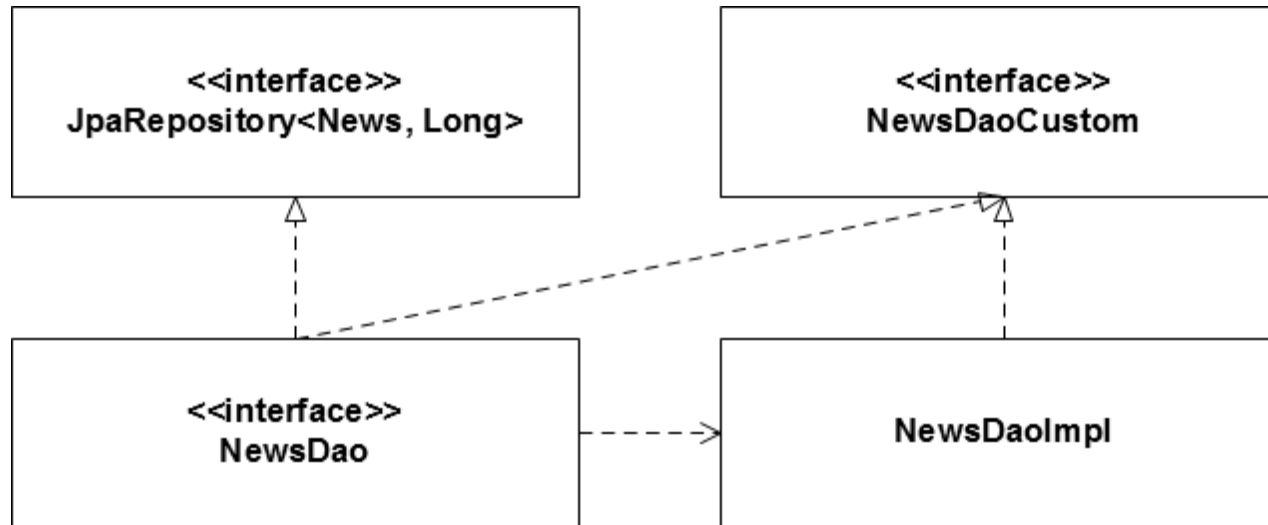
```
public List<News> findNews(@Param("title") String titleQuery);  
// Achtung: Bei LIKE %:title verwenden
```

```
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

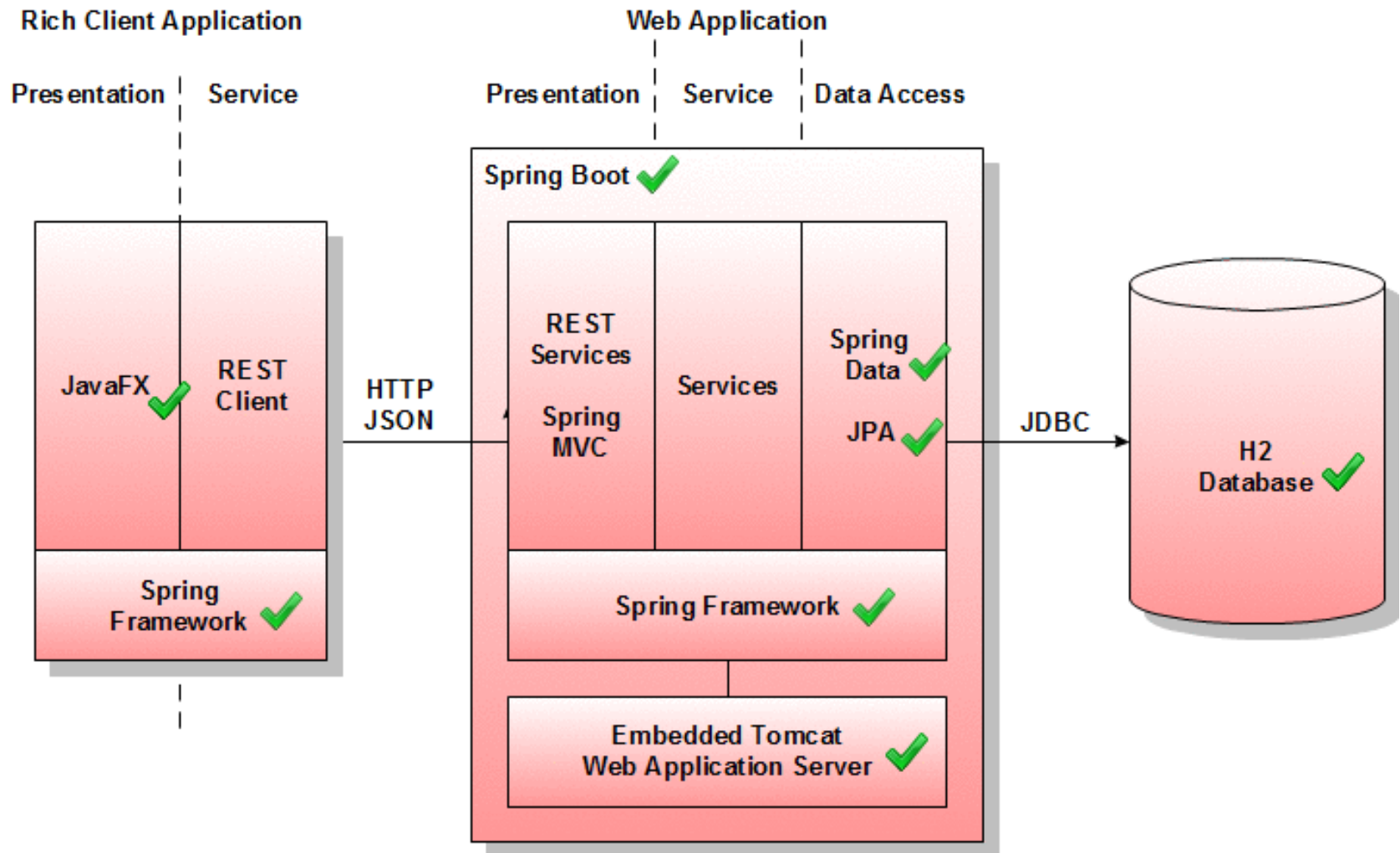


- **JpaRepository**: Von Spring Data definiertes Interface
- **NewsDaoCustom**: Definiert zusätzliche Methoden
- **NewsDaoImpl**: Implementiert zusätzliche Methoden; Name muss auf Impl enden
- **NewsDao**: Implizite Verwendung der Implementierung der NewsDaoImpl

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;  
}
```

Ticketline Architektur



@Service

```
public class NewsServiceImpl implements NewsService {
```

@Autowired

```
private NewsDao newsDao;
```

@Autowired

```
private PersonDao personDao;
```

@Transactional

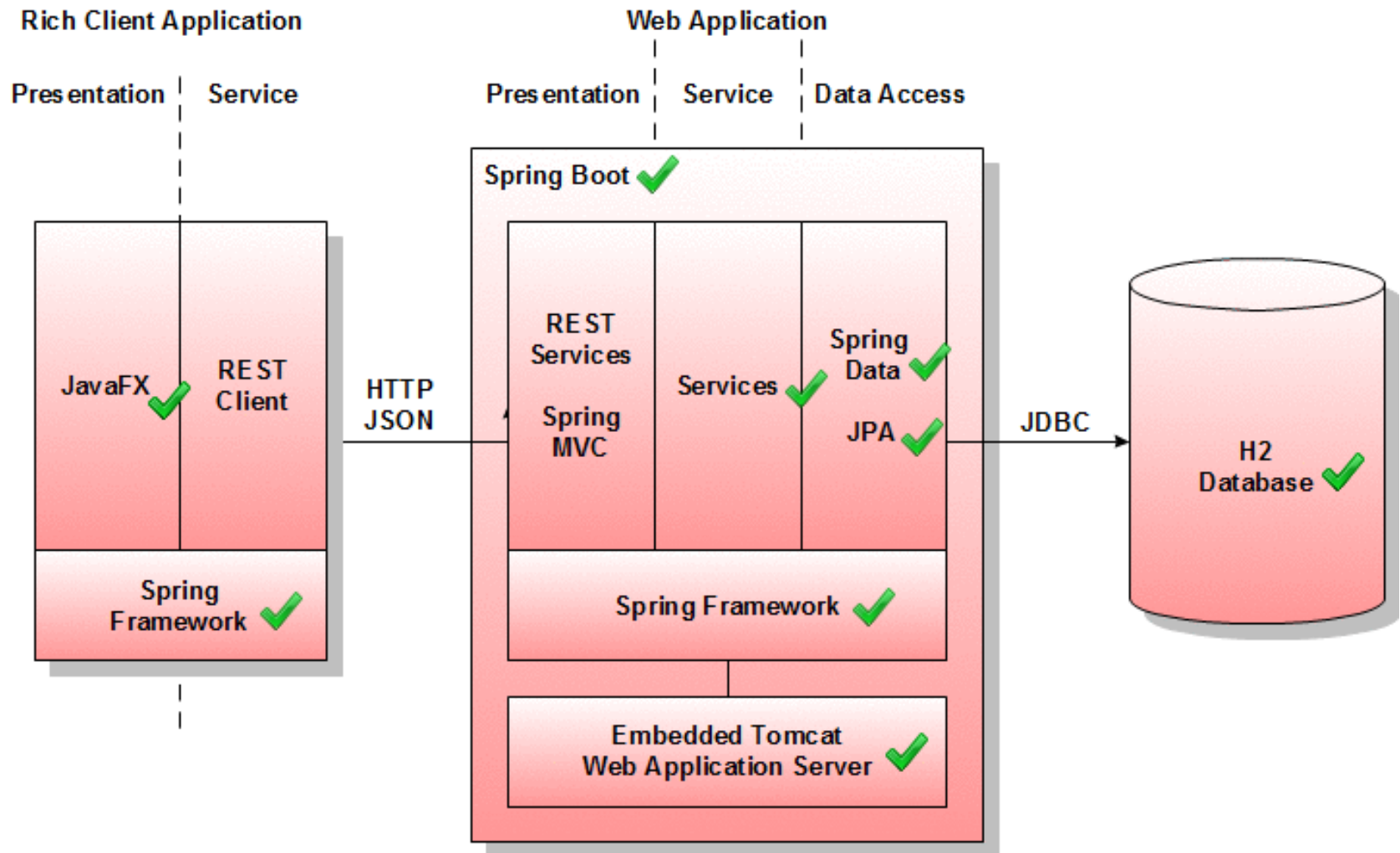
```
public News getNews(Integer personId) throws ServiceException {
```

```
    Person p = personDao.findOne(personId);
```

```
    p.getAddress(); // lazy-loading
```

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

Ticketline Architektur




- **Architektur-Stil für serverseitige Schnittstellen**
- **Definiert in der Dissertation von Roy Fielding**
- **Nutzung von HTTP als Transport-Protokoll**
- **Eigenschaften**
 - Ressourcen als zentrales Element
 - Identifizierbarkeit & Adressierbarkeit der Ressourcen
 - Unterschiedliche Repräsentationen
 - Standardisierte Operationen
 - Zustandslosigkeit der Services
- **Oftmals keine standard-getreue Umsetzung**

Uniform Resource Locator (URL)

- Spezialisierung einer URI (definiert in RFC 3986)
- Zur Identifikation & Lokalisierung von Ressourcen

http://www.ticketline.at: 8080/ticket/get?id=4242#owner


Scheme Host Port Path Query Fragment

Request-Parameter in Query:

name_1=value_1&name_2=value_2

Url-Encoding:

ticket%2Fget%3Fid%3D4242%23owner

Hypertext Transfer Protocol (HTTP)

- **Version 1.1 in IETF RFC 2616 definiert**
- **Text-basiertes & zustandsloses Protokoll**
- **Nutzung von TCP/IP als Transport-Protokoll**
- **Adressierung über URLs**
- **Messages bestehen aus Header & Body**
- **Verschiedene Operationen: GET, POST, PUT, DELETE**
- **Standardisierte Header & Response-Codes**
- **Zustand per Session-Management (Cookies)**
- **HTTP 2.0 seit Anfang 2015 offizieller Standard**

HTTP-Request & -Response Beispiel

GET /index.html HTTP/1.1

Host: www.tuwien.ac.at

User-Agent: Mozilla/5.0

Accept: text/html,application/xhtml+xml,application/xml

Accept-Language: de,en

Cookie: session_id=AF2452292342

HTTP/1.1 200 OK

Date: Tue, 29 Oct 2013 22:08:47 GMT

Set-Cookie: session_id=AF2452292342; path=/

Content-Type: text/html; charset=iso-8859-1

Server: Apache/2.1.14

X-Powered-By: PHP/5.4.2

<html>

<head>

.....

- **Aufruf: $\{\text{operation}\} \{\text{path}\} \text{HTTP}/\{\text{version}\}$**
- **Aufbau der Header: $\{\text{name}\}: \{\text{value}\}$**
- **Eigene Header über X- $\{\text{name}\}$**
- **Response liefert einen Statuscode**
- **Gruppierung der HTTP-Statuscodes**
 - 1XX: Information
 - 2XX: Erfolg
 - 3XX: Weiterleitung
 - 4XX: Vom Client verursachte Fehler
 - 5XX: Serverseitige Fehler

HTTP- bzw REST-Operationen

Operation	Beschreibung
GET	Fordert eine Ressource vom Server an (read-only)
POST	Legt eine neue Ressource an, zu der es noch keine URL gibt; Parameter werden im Body des Requests übergeben
PUT	Legt eine neue Ressource an oder ändert eine bestehende Ressource
DELETE	Entfernt eine Ressource vom Server

REST-Services in Ticketline

- **Verwendung von Spring MVC**
- **Datenformat: JSON**

```
@RestController
```

```
@RequestMapping(value = "/news")
```

```
public class NewsController
```

```
    @RequestMapping(value = "/list", method = RequestMethod.GET,  
        produces = "application/json")
```

```
    public List<News> listNews() { .. }
```

```
    @RequestMapping(value =("/{id}")
```

```
    public News getNews(@PathVariable("id") Integer id) { .. }
```

```
    @RequestMapping(value = "/search")
```

```
    public News findNews(@RequestParam("q", required = true) String query)
```

```
    @RequestMapping(value = "/add", method = RequestMethod.POST,  
        consumes = "application/json")
```

```
    public Integer create(@RequestBody News news) { .. }
```

JavaScript Object Notation (JSON)

- Auf JavaScript basierendes Datenformat
- JSON ist ein reduzierter Sprachumfang von JavaScript
- Datentypen: Numerisch, String, Boolean, Array, Objekt

```
{  
  "id": 5,  
  "created": true,  
  "title": "The Hitchhiker's Guide to the Galaxy",  
  "tags": [ 2, true, "Hello World", null ],  
  "author": {  
    "firstname": "Douglas",  
    "lastname": "Adams"  
  }  
}
```

1. Validierung von DTOs:

Nutzung von JSR 303 – Bean Validation

```
@RequestMapping(  
value = "/", method = RequestMethod.POST, consumes = "application/json")  
public Message publishNews(@Valid @RequestBody NewsDto news) {
```

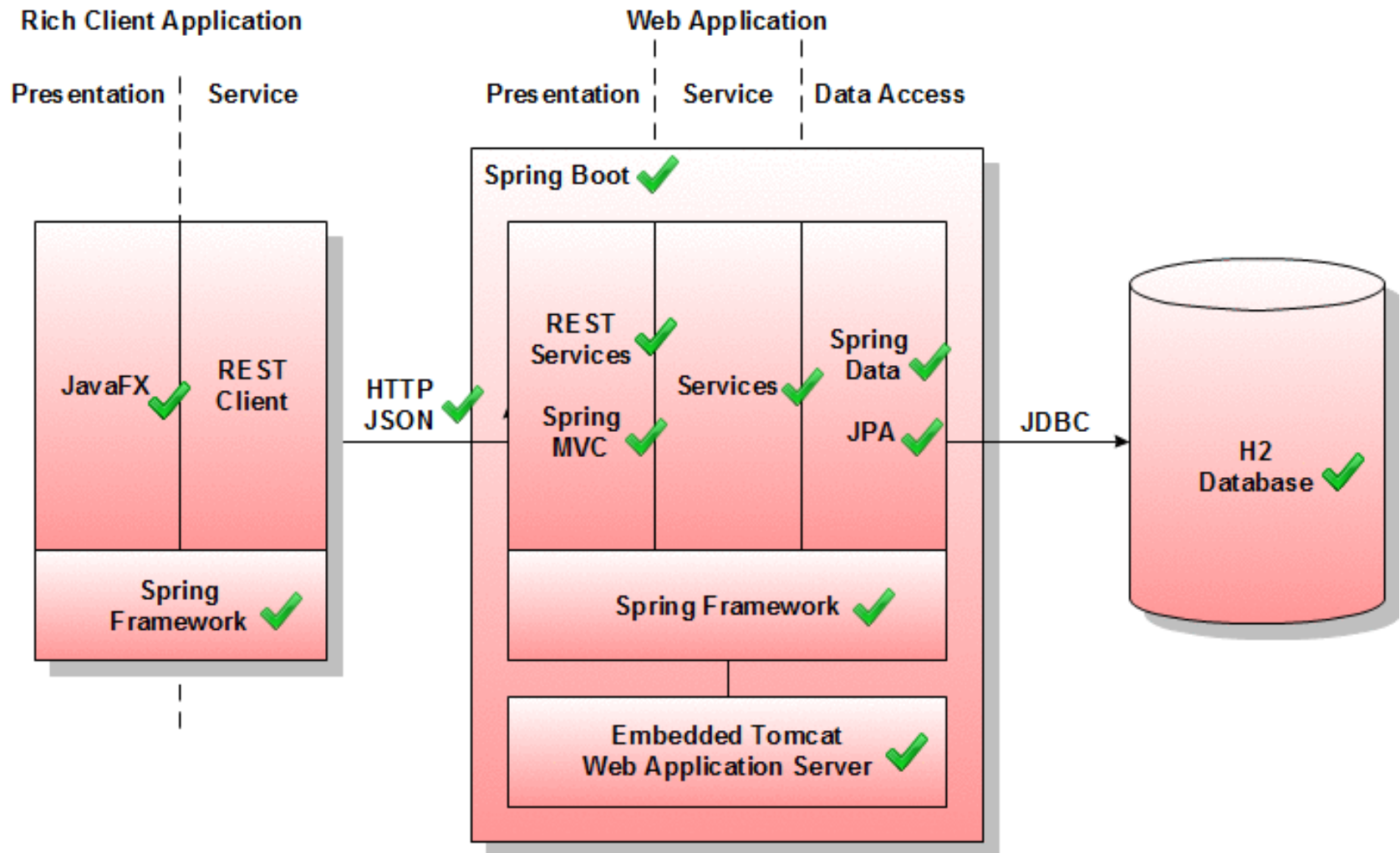
2. Validierung einzelner Parameter:

Gilt für @PathVariable und @RequestParam

```
@RequestMapping(value =("/{id}", method = RequestMethod.GET)  
public NewsDto getNewsById(@PathVariable("id") Integer id)
```

Verarbeitung erfolgt in RestErrorHandler basierend auf Exception

Ticketline Architektur



- **Verwendung des RestTemplate & UriComponentsBuilder von Spring**

Beispiel für UriComponentsBuilder:

```
UriComponents uriComponents = UriComponentsBuilder.newInstance()  
    .scheme("http").host("ticketline.at").path("/movie/{movie}").build()  
    .expand("star_wars")  
    .encode();
```

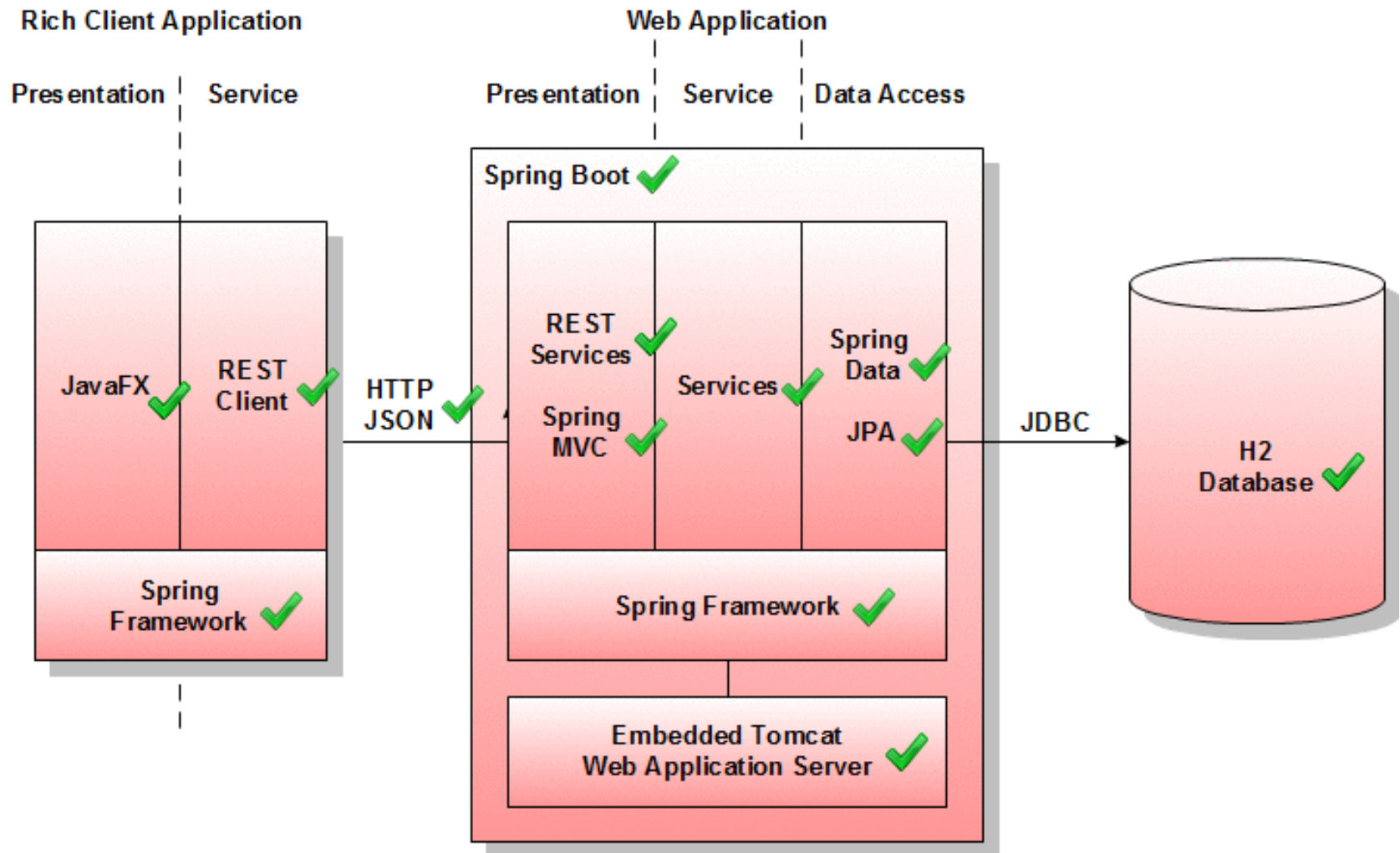
```
URI uri = uriComponents.toUri();
```

- **Generische Methoden: execute, exchange**
- **Spezifische Methoden: getForObject, postForEntity**
- **HttpEntity für Request**
- **ResponseEntity als Response**

Beispiel für RestTemplate:

```
RestTemplate restTemplate = restClient.getRestTemplate();  
HttpEntity<NewsDto> request =  
    new HttpEntity<NewsDto>(news, headers);  
ResponseEntity<MessageDto> response =  
    restTemplate.postForEntity(url, request, MessageDto.class);  
MessageDto msg = response.getBody();
```


Ticketline Architektur



- **Dokumentation der Frameworks & Bibliotheken**
- **Websites z.B. stackoverflow.com**
- **Ticketline – Getting Started Guide**
- **Tuwel-Forum:**
- **Software Engineering und Projektmanagement (PR 4,0)**

Fehlermeldungen:

- Genaue Fehlerbeschreibung
- Kompletter Stack Trace
- Source Code der betroffenen Stelle
- SSCCE – Short Self Contained Correct (Compilable) Example

Gutes Gelingen mit Ticketline!