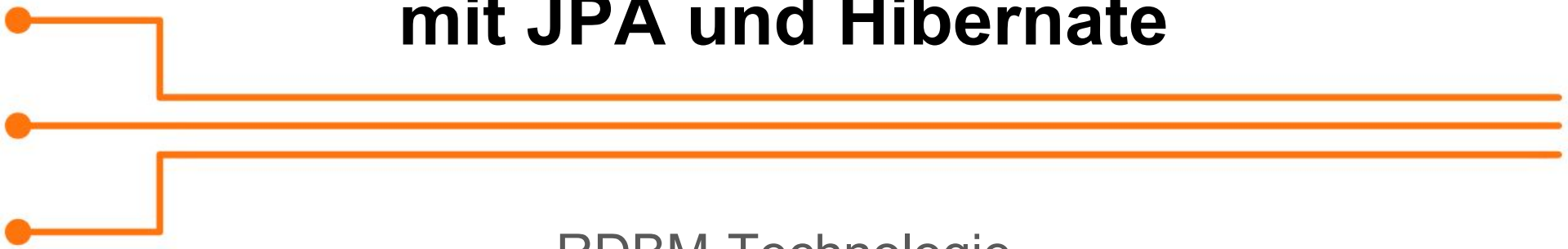
Datenbankanwendungen
mit JPA und Hibernate



RDBM-Technologie

Datenmanagement

JPA - Java Persistence API

- Java-Schnittstellenspezifikation für objektrelationales Mapping
- javax.persistence
- Referenzimplementierung Eclipse-Link
- Hibernate ist eine Implementierung von JPA

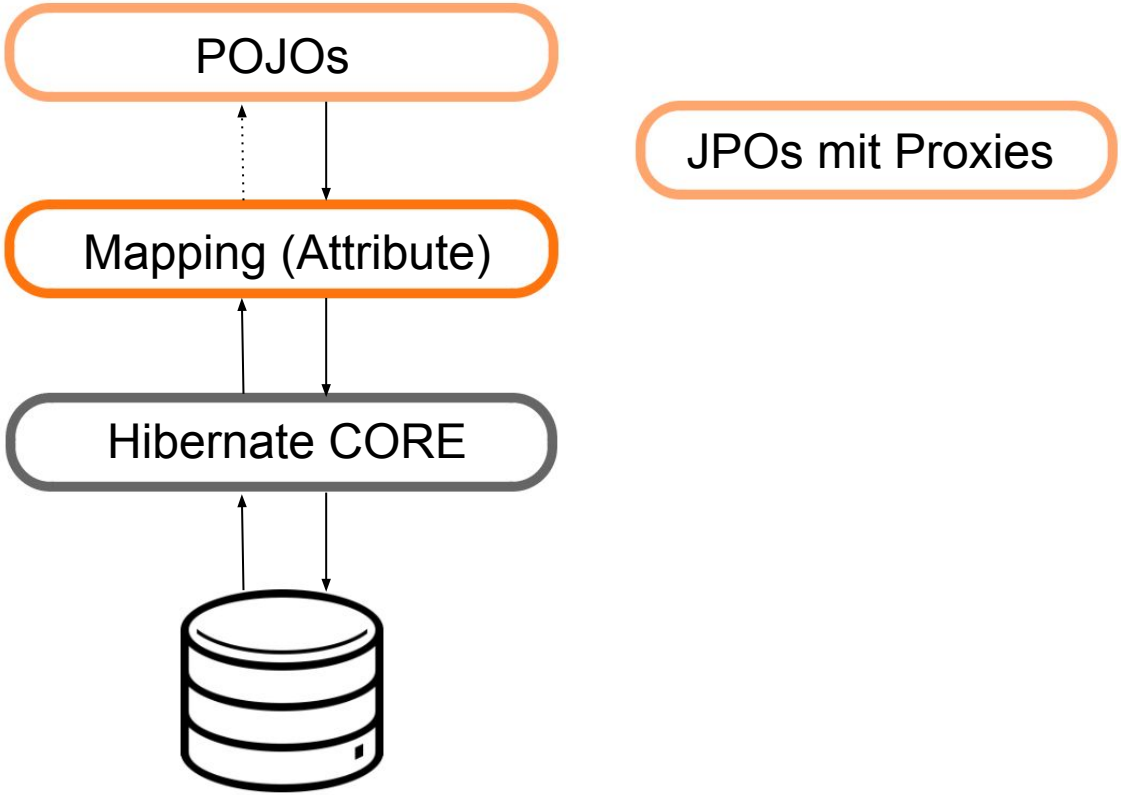
Was kann JPA/Hibernate?

- Objektrelationales Mapping auf Config-Ebene
- Attributierung von Klassen und Properties (Metadaten)
- Es wird dynamisch generierter Code zum (de)materialisieren von Objekten verwendet
- JPA wird von verschiedenen Frameworks implementiert (u.A. Hibernate)
- Keine spezifische Implementierung für eine Datenbank - Verwendung von JDBC-Treibern

Was kann JPA/Hibernate?

- Verbindungs- und Transaktionsmanagement
- Connection Pooling
- Caching und Locking
- Lazy Loading
- Abfragesprachen auf Domain Ebene (High Level Domain SQL)

Persistenzschicht mit Hibernate



Mapping

Einfache Techniken

- Class
  Top Level Mapping Entität
- ID
  Primärschlüssel
- Properties
  Getter/Setter für private Felder werden automatisch erkannt
- Value Klassen
  Komponenten Mapping für integrierte einfache Typen wie DateTime
  Beinhalten innere Properties werden aber nicht über neue Klassen gemappt

Mapping

Beziehungen

- One-To-One
- One-To-Many
- Many-To-One
- Many-To-Many
- Cascading

  Was geschieht eigentlich mit Objektbäumen bei Save/Update/Delete?

Mapping
mit XML-Files

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="domain">
      <class name="Person">
            <id name="persno">
                  <generator class="native"/>
            </id>
            <property name="fname“/>
            <property name="lname"/>
            <property name="salary"/>
            <property name="fdate"/>
            <property name="ldate"/>
      </class>
</hibernate-mapping>
```

Mapping
mit Annotations

- Annotations seit Java 1.5

```
@Entity //aus javax.persistence
@Table(name=„PERSON“) //optional
public class Person {
      @Column(name=„name“) //Der Name der Spalte in der Datenbank
      private String name;
      public Person() { }
      public String getName() { return name; }
      public void setName(String value) { name = value; }
}
```

- Meta-Informationen werden damit direkt zum Code hinzugefügt
- Hibernate muss für die Verwendung von Annotations entsprechend konfiguriert werden

SessionFactory

- Bei Verwendung von Annotations muss Hibernate entsprechend konfiguriert werden
- Annotations werden zur Laufzeit beim Erstellen der sog. SessionFactory verarbeitet

```
Configuration configuration = new Configuration();
configuration.configure("hibernate.cfg.xml");
StandardServiceRegistry serviceRegistry =
new StandardServiceRegistryBuilder().applySettings(configuration.getProperties()).build();
factory = configuration.buildSessionFactory(serviceRegistry);
```

Sessions und Transactions

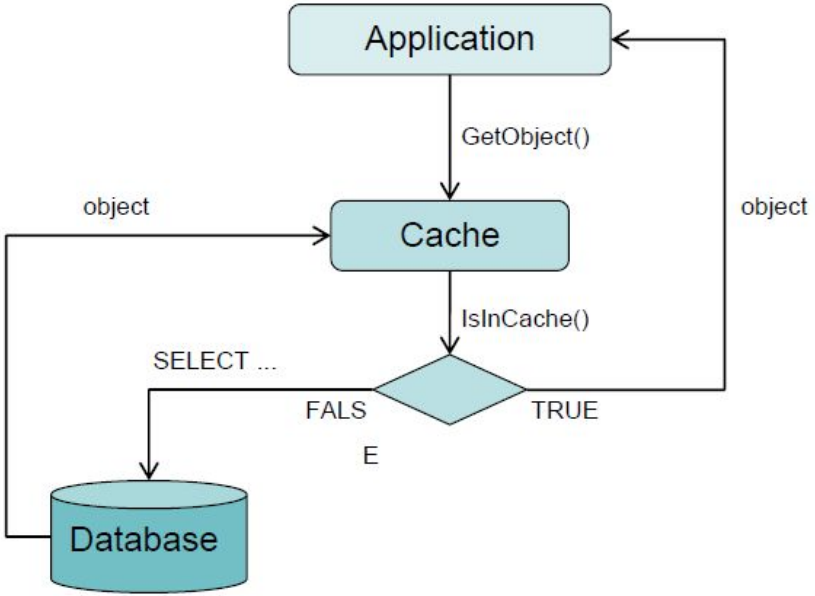
- Sessions werden von der SessionFactory erzeugt
- Session.save() speichert die Änderungen (falls keine Transaktion verwendet wird)
- Werden Transaktionen verwendet müssen diese committed werden!

```
Session session= null;
Transaction tx= null;
try{
      session = sessionFactory.openSession();
      tx = session.beginTransaction();
      session.save(myObject);
      tx.commit();
} catch(Exception ex) {
      if(tx != null) tx.rollback();
} finally {
      if(session != null) session.close();
}
```

Caching

Zwischenspeichern von Objekten

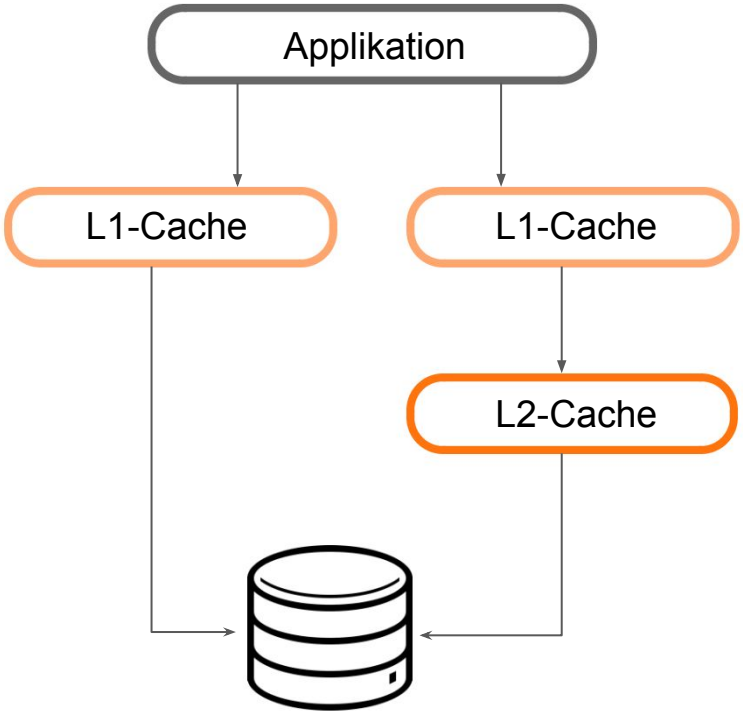
- Häufig verwendete Datensätze werden in schnelleren Speichern vorgehalten
- Zugriff auf gecachteObjekte ist transparent
- Cache-Hit / Cache-Miss
- Objekte die geladen werden kommen automatisch in den Cache
- Reduktion der Datenbankzugriffe
- Problem: Change-Behind-The-Cache
- Verschiedene Cache-Implementierungen



Caching

auf zwei Ebenen

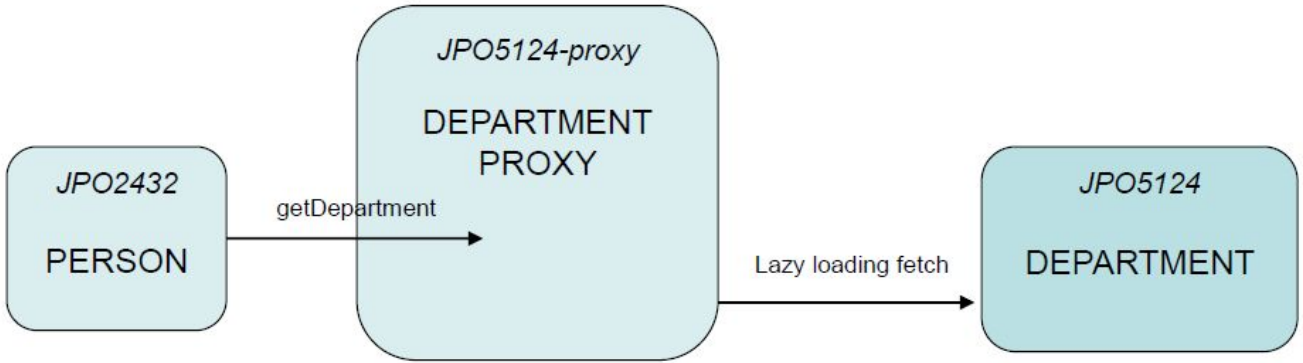
- First Level Cache
  - Flüchtig (transient)
  - Session basiert
  - Immer aktiv (ist in Session implementiert)
- Second Level Cache
  - Optional
  - Pro Anwendung nur eine Instanz
  - SessionFactory.CacheProviderInterface
  - Implementierung variiert von JVM lokal bis Cluster
- Java Persisted Object(JPO)
- LazyLoadingund Proxy Objekte



Proxies und Lazy Loading

Java Persisted Objects zur Laufzeit

	2432	John	Doe	2014-01-01
5124	Sales and Marketing	…	…	
…				



Locking

Java Persisted Objects zur Laufzeit

- First Level Cache
  - Flüchtig (transient)
  - Session basiert
  - Immer aktiv (ist in Session implementiert)
- Second Level Cache
  - Optional
  - Pro Anwendung nur eine Instanz
  - SessionFactory.CacheProviderInterface
  - Implementierung variiert von JVM lokal bis Cluster
- Java Persisted Object(JPO)
- LazyLoading und Proxy Objekte

Caching

Query Caching

- Queries werden mit ihrem Ergebnis abgespeichert
- IDs werden aus dem L2 Cache geladen
- Gültigkeit der Objekte wird mitttels Zeitstempel im L2 Cache überprüft
- Query Cache muss manuell aktiviert werden
  `hibernate.cache.use_query_cache = true`
- Queries müssen auch cachable gesetzt werden
  `Query.setCacheable()`

Hibernate Query Language

Abfragen auf Domain-Ebene

- SQL-artige Abfragesprache für objekt-basierte Queries

```
Query query = session.createQuery(„from Person“);
```

- Ausführen der Query mit Result-Set-Selection

`query.list()`	ergibt java.util.List
`query.iterator()`	ergibt java.util.iterator
`query.scroll()`	ergibt org.hibernate.ScrollableResult zum „Browsen“ durch die Ergbnisse
`query.uniqueResult()`	wenn das Ergebnis ein einzelner Wert oder NULL ist

- Alle Standard SQL Keywords und Features werden unterstützt
- Platzhalter Entities können bei komplexen Abfragen verwendet werden

Hibernate Query Language

Filterkriterien

- Type-safeQueries

```
Criteria crit = session.createCriteria(Person.class);
crit.add(Restrictions.eq(„name“, „Franz“));
List<Person> franzes = crit.list();
```

+ Klassennamen im Code führt zu Typsicherheit (allerdings nur bei Klassen, nicht bei Properties)
+ Änderungen auf Klassenebene sind zur Compilezeit sichtbar
+ Keine HQL oder SQL Kenntnisse benötigt
- Criterias unterstützen keine Joins
- Properties sind nicht type-safe weil sie als Strings angegeben werden

Weitere Hibernate-Funktionen

Validierung

- Validierungsfunktionen auf dem Domain-Level
- Java Annotationsfür die Regelerstellung
- Es gibt vordefinierte Regeln wie

```
@NotNull
@Length(max=20)
@Range(min=1, max=5, message=„Out ofRange“)
@Email
```

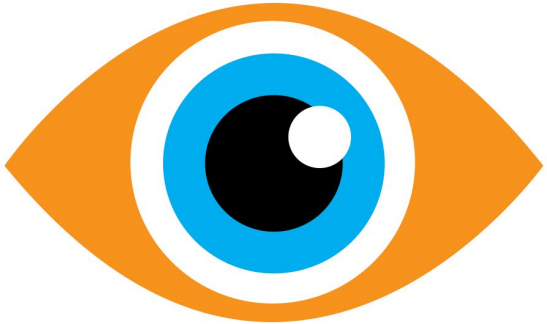
- Eigene Regeln können erstellt werden
- Die Regeln aus dem Klassenmodell werden auf die Datenbank übertragen

Weitere Hibernate-Funktionen

Tools

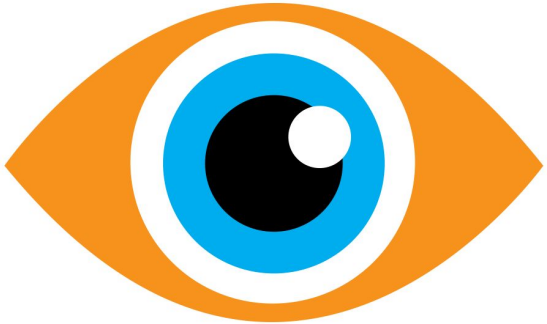
- Eclipse-Plugins
- Mapping-Editor mit Auto-completion (Klassen, Tabellen, Metatags)
- Syntax-Highlighting
- Query Console - konfigurieren von Datenbankverbindungen, Queries, Results
- Reverse Engineering - Mappings, Datenbanken und Klassen aus DB-Schema erstellen

Hibernate - auf einen Blick



- Natürliches objektorientiertes Programmiermodell
  Klassen
  Vererbung
  Polymorphismus
- Variables Mapping für Kompositionen und abhängige Objekte
- Keine Post-Build Aktionen
- Keine (spürbare) Code-Generierung oder Injection

Hibernate - auf einen Blick



- Extreme Skalierbarkeit
  Cluster-support
  2-Level Caching
- Abfragesprache
  In Code Queries
  Materialisierung, Dematerialisierung
- Datenkonsistenz und -integrität
  Type/Key/Nullable-Validierung
  Automatisches Locking und Transaktionen