Hibernate

?

- ORM framework
- JPA provider(v2.1)

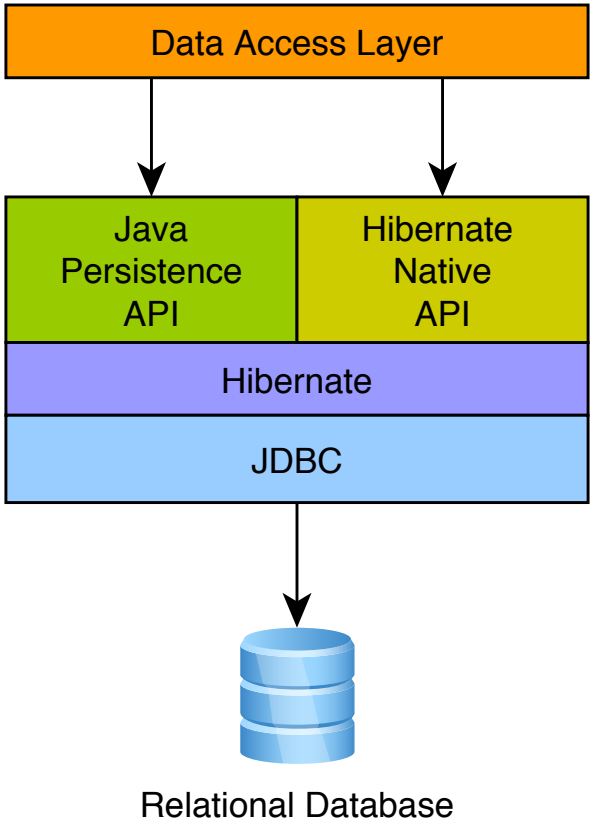
JPA

- specifikace
- implementace:
  - Hibernate
  - eclipseLink
  - OpenJpa

Použití

```
@Entity
@Table(name = "person")
public class Person {

    @Id
    @SequenceGenerator(name = "person_id_seq",
        sequenceName = "person_id_seq", allocationSize = 1)
    @GeneratedValue(generator = "person_id_seq", strategy = GenerationType.SEQUENCE)
    private Integer id;

    @Column(name = "name")
    private String name;
```

Hibernate architektura



Hibernate architektura

- Hibernate - SessionFactory
  - reprezentuje mapování obj do DB
  - poskytuje Session
  - threadSafe
  - jedna pro každou DB - velká náročnost vytvoření
- JPA - EntityManagerFactory

Hibernate architektura

- Hibernate - Session
  - žije jen po dobu potřeby, pak zaniká
  - poskytuje metody nad persistent-contextem
- JPA - EntityManager

Konfigurace

- Vytvoření 'javax.persistence.EntityManagerFactory'
  - META-INF/persistence.xml

- Získání entityManager
  - EE - @PersistenceContext
  - SE - entityManagerFactory.createEntityManager()

Konfigurace - native

- http://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html#bootstrap-native

Konfigurace - jpa

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
             http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
             version="2.1">

    <persistence-unit name="PERSISTENCE">
        <description>Hibernate JPA Configuration Example</description>
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

        <properties>
            <property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver"/>
            <property name="javax.persistence.jdbc.url" value="jdbc:postgresql://localhost:5432/
            <property name="javax.persistence.jdbc.user" value="postgres"/>
            <property name="javax.persistence.jdbc.password" value="postgres"/>
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
        </properties>

    </persistence-unit>

</persistence>
```

Entita

- objekt reprezentující řádek tabulky
- třída se anotuje @Entity
- musí obsahovat public/protected no-args constructor
- musí být top-level class
- třída a persistované proměnné nesmí být final
- musí být ve style java bean - tedy obsahovat get/set
- musí mít identifikátor (@Id)

Entita - mapování

- @Entity(name=..) - označuje entitu + ji může pojmenovat
- @Table - specifikuje mapování v DB

```
@Entity(name = "Book")
@Table(
    schema = "store",
    name = "book"
)
public class Book {
```

Entita - přístup

- field-based
- jednodušší přidávání metod
- pokud chceme nějaký field nepersistovat označí se @Transient

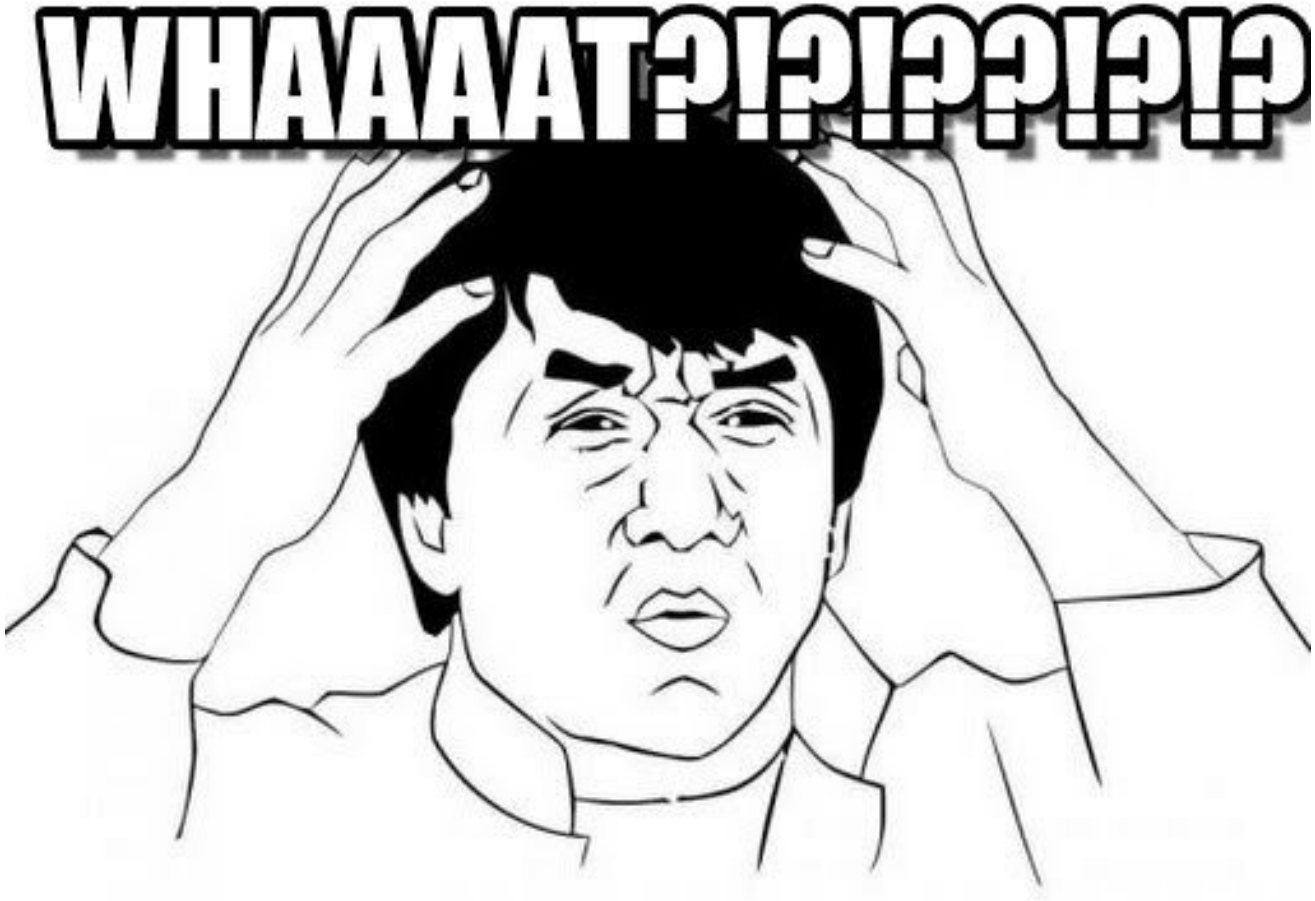
```
@Entity(name = "Book")
public static class Book {

    @Id
    private Long id;

    private String title;

    private String author;

    //Getters and setters are omitted for brevity
}
```

Entita - přístup

- property-based
- pomocné metody musí být @Transient
- metody mohou ovlivňovat persistovaná data

```
@Entity(name = "Book")
public static class Book {

    private Long id;

    private String title;

    private String author;

    @Id
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }
```

Entita - bez tabulky



Entita - bez tabulky

- Proč?
  - entita která obsahuje data z více tabulek
  - podmnožina existující tabulky

- Jak? - @Subselect + @Synchronize

```
@Entity
@Subselect("select p.id as id, p.phone as num from Person p")
@Synchronize({"Person"})
public class Phone {
```

Basic types

- typy mapované 1:1 s buňkou v DB
- anotované @Basic (lze vynechat, je použita defaultně)

```
@Entity(name = "Product")
public class Product {

    @Id
    @Basic
    private Integer id;

    @Basic
    private String name;
}
```

```
@Entity(name = "Product")
public class Product {

    @Id
    private Integer id;

    private String name;
}
```

Basic types

- optional
  - true(default) - nepovinný (nullable)
  - false - povinný (not-null)
- fetch
  - lazy - načte až při přístupu
  - eager(default) - načten okamžitě

Basic types

- @Column
  - name - jmeno sloupce v DB
  - nullable - nenulová data
  - updatable - properta je/není zahrnuta v update query
  - insertable - properta je/není zahrnuta v insert query

```
@Entity(name = "Product")
public class Product {

    @Id
    private Integer id;

    @Column(name = "NAME", nullable = false)
    private String name;
}
```

Basic types

- mapování typů (javatype-hibernatetype-sqltype)
  - default podle 'BasicTypeRegistry'
  - explicitně pomocí @Type(type="")

```
@Entity
@Table(name = "car")
public class Car {

    @Id
    private Long id;

    @Type(type = "text")
    private String name;

    private String description;
```

Basic types Enum

- Mapovat enum lze pomocí 2 strategií
  - ORDINAL(default) - pomocí pořadí(0, 1, 2, ...)
  - STRING - pomocí nazvu enumu
- Strategii zapisujeme následovně

```
@Enumerated(value = EnumType.ORDINAL)
private CarType carType;
```

Basic types Enum

- Pokud chceme nějaké "vlastní" mapování použijeme @Convert

```
@Convert(converter = CarTypeConverter.class)
private CarType carType;
```

- a @Converter

```
@Converter
public class CarTypeConverter implements AttributeConverter<CarType, String> {

    @Override
    public String convertToDatabaseColumn(CarType attribute) {

        return attribute.name() + "car";
    }

    @Override
    public CarType convertToEntityAttribute(String dbData) {

        String carType = dbData.replaceAll("/car$/", "");
        return CarType.valueOf(carType);
    }
}
```

Basic types LOB

- ukládání velkých dat
  - BLOB - binární data
  - CLOB - posloupnost znaků každý znak = 1 byte
  - NCLOB - posloupnost znaků každý znak = tolik byte kolik zabírá v dané znakové sadě

- anotujeme @Lob
- @Lob je lazyLoad

Basic types DateTime

- Date - datum
- Time - čas
- Timestamp - datum a čas
- při použití java.sql se typy mapují automaticky
- při použití java.util se mapujeme na timestamp
  - nebo lze použít @Temporal pro explicitní mapování

```
@Entity
@Table(name = "item")
public class Item {

    @Id
    private Long id;

    private Date sqlDate;

    @Temporal(TemporalType.DATE)
    private java.util.Date utilDate;
```

Basic types DateTime

- java 8
  - mapování mezi SQL a java8 typy je implicitní = bez anotace @Temporal
  - pro hibernate < 5.2 je třeba přidat dependency

```
 <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-java8</artifactId>
    <version>${version.hibernate}</>
</dependency>
```

Basic types DateTime

- TimeZone
  - Default se bere z JVM
  - lze nastavit v hibernate pro factory - "hibernate.jdbc.time_zone"

```
<property name="hibernate.jdbc.time_zone" value="UTC"/>
```

Basic types

- použití rezervovaných slov('user') jako název
  - escapovat jeden název - lokálně
  - escapovat všechny - globálně

```
@Column(name = "`user`")
@Column(name = "\"user\"")
private String user;
```

```
<property name="hibernate.globally_quoted_identifiers" value="true"/>
```

Basic types

- transformace hodnoty
  - @ColumnTransformer

```
@ColumnTransformer(
    read = "pgp_sym_decrypt(pwd::bytea, 'myPass')",
    write = "pgp_sym_encrypt(?, 'myPass')"
)
private String pwd;
```

Basic types

- transformace hodnoty
  - @Formula
    - jen read operace
    - není persistována

```
@Column(name = "money")
private int mon;

@Formula(value = "money * 20")
private int realValue;
```

Basic types

- mapování polymorfusmu
- @Any a @AnyDef

```
@AnyMetaDef(name = "PropertyMetaDef", metaType = "string", idType = "long",
        metaValues = {
                @MetaValue(value = "sprop", targetEntity = StringProp.class),
                @MetaValue(value = "lprop", targetEntity = LongProp.class)
        }
)

@Any(
    metaDef = "PropertyMetaDef",
    metaColumn = @Column(name = "property_type")
)
@JoinColumn(name = "property_id")
private Property property;
```

Embeddable

- forma jak zapsat kompozici jak ji známe z OOP

```
public class User {

    private Long id;

    private String city;

    private String street;
}
```

```
public class User {

    private Long id;

    private Address;
}

public class Address {

    private String city;

    private String street;
}
```

Embeddable

- @Embeddable

```
@Entity
@Table(schema = "embeddable")
public class User {

    @Id
    private Long id;

    @Embedded
    private Address address;

}

@Embeddable
public class Address {

    private String city;

    private String street;
}
```

Embeddable

- více stejných emebedded typů

```
@AttributeOverrides({
        @AttributeOverride(name = "companyAddress.city",
                column = @Column(name = "company_city")),
        @AttributeOverride(name = "companyAddress.street",
                column = @Column(name = "company_street")),
        @AttributeOverride(name = "address.city",
                column = @Column(name = "home_city")),
        @AttributeOverride(name = "address.street",
                column = @Column(name = "home_street"))
})
@Entity
@Table(schema = "embeddable", name = "user")
public class User {

    private Long id;

    private Address address;

    private Address companyAddress;
}
```

Embeddable

- Co když chceme/potřebujeme mít embeddable Interface?

```
public interface Coordinates {

    double x();
    double y();
}

@Entity
@Table(schema = "embeddable", name = "user")
public class User {

    @Id
    private Long id;

    @Embedded
    private Address address;

    @Embedded
    @Target(GPS.class)
    private Coordinates coordinates;
}
```

Identifikátory

- PK v DB - jednoznačný identifikátor entity
- musí být:
  - not null
  - imutable
  - unique

```java
@Entity
@Table(schema = "identifier", name = "user")
public class User {

    @Id
    private Long id;

    private String name;
}
```

Identifikátory

- generování
  - JPA - pouze číslo
- auto(default)
- identity
- sequence
- table

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```

Identifikátory

- IDENTITY
  - podle JDBC - tedy dle DB
    - v postgresql je to sequence
    - v mysql je to autoincrement

Identifikátory

- SEQUENCE
  - JPA preferovaná varianta
    - nejjednoznačnější a uživatelsky čitelné
  - base varianta(ale pozor hibernate vytvori sequenci do default schematu..

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE)
private Long id;
```

Identifikátory

- SEQUENCE
  - vlastni jmeno
  - parametry initialValue, allocationSize

```
@Entity
@Table(schema = "identifier", name = "user_gen")
public class UserGenerated {

    @Id
    @GeneratedValue(
            strategy = GenerationType.SEQUENCE,
            generator = "moje_seq"
    )
    @SequenceGenerator(
            schema = "identifier",
            name = "moje_seq",
            sequenceName = "user_ger_id_seq",
            initialValue = 10,
            allocationSize = 50
    )
    private Long id;
}
```

Identifikátory

- table identifier
  - vytvori tabulku kde drzi "pseudo sequence" kterou by default pouzije pro každý table generator
- ma/melo nejhorsi performance ze vsech

```
@Id
@TableGenerator(
    schema = "identifier",
    name = "table-generator",
    table = "table_identifier",
    pkColumnName = "table_name",
    valueColumnName = "user_id",
    allocationSize = 1
)
@GeneratedValue(
    strategy = GenerationType.TABLE,
    generator = "table-generator"
)
private Long id;
```

Identifikátory

- odvozené identifikátory
  - umožňuje použít jedno ID jako společný identifikátor
  - @MapsId

```
@Entity
@Table(schema = "identifier", name = "user_deriv")
public class UserDeriv {

    @Id
    private Long id;

    @OneToOne
    @MapsId
    private Wallet wallet;
}
@Entity
@Table(schema = "identifier", name = "wallet")
public class Wallet {

    @Id
    private Long id;
}
```

Identifikátory

- složené identifikátory
  - skládá se z jednoho nebo více persistovaných prvků
  - reprezentován třídou která musí:
    - mít public no-arg kontruktor
    - musí definovat equals & hashCode
    - musí implementovat Serializable

Identifikátory

- složené identifikátory - @EmbeddedId
  - použití embeded typu jako klíče

```
@Embeddable
public class BasicEmbedKey implements Serializable {

    private String key;
    private String lang;

    ... get/set equals&hashCode
}

@Entity
@Table(schema = "identifier", name = "basic_embed_user")
public class BasicEmbedUser {

    @EmbeddedId
    private BasicEmbedKey basicEmbedKey;
}
```

Identifikátory

- složené identifikátory - @EmbeddedId
  - embedded může obsahovat i jiné entity

```
@Embeddable
public class AdvanceEmbeddedKey implements Serializable {

    private String lang;

    @ManyToOne
    private Property property;
}

@Entity
@Table(schema = "identifier", name = "property")
public class Property {

    @Id
    private Long id;

    private String key;

    private String value;
}
```

Identifikátory

- složené identifikátory - @IdClass

```
@Entity
@Table(schema = "identifier", name = "advance_property")
@IdClass(ClassKey.class)
public class AdvanceProperty {

    @Id
    private String key;

    @Id
    private String lang;

    @Id
    @GeneratedValue
    private Long version;
}

public class ClassKey implements Serializable {

    private String key;
    private String lang;
    private Long version;
```

Identifikátory

- složené identifikátory - pomocí asociace
  - umožní implicitně vytvořit složený klíč

```
@Entity
@Table(schema = "identifier", name = "book")
public class Book implements Serializable {

    @Id
    @ManyToOne
    private Author author;

    @Id
    @ManyToOne
    private Publisher publisher;

    private String name;
}
```

Asociace

- Jak spojit dvě a více entit
  - OneToOne
  - ManyToOne
  - ManyToMany
  - OneToMany

Asociace

- @ManyToOne

```
@Entity
@Table(schema = "associate", name = "phone")
public class Phone {

    @Id
    private Long id;

    private String number;

    @ManyToOne
    @JoinColumn(name = "person_id",
            referencedColumnName = "key",
            foreignKey = @ForeignKey(name = "FK_PERSON_ID")
    )
    private Person person;
}
```

Asociace

- @OneToMany - unidirection

```
@Entity
@Table(schema = "associate", name = "department")
public class Department {

    @Id
    private Long id;

    private String name;

    @OneToMany
    @JoinColumn(name = "DEP_ID")
    private List<Employee> employees;
}
```

Asociace

- @OneToMany - bidirection

```
@Entity
@Table(schema = "associate", name = "employee")
public class Employee {

    @Id
    private Long id;

    private String name;

    @ManyToOne
    private Department department;
}
```

Asociace

- @OneToOne - jednosměrné

```
@Entity
@Table(name = "phone")
public class Customer {

    @Id
    private Long id;

    @OneToOne
    @JoinColumn(name = "order_id")
    private Order order;
}
```

Asociace

- @OneToOne - obousměrné

```
@Entity
@Table(name = "customer")
public class Customer {

    @Id
    private Long id;

    @OneToOne(mappedBy = "phone")
    private Order order;
}

@Entity
@Table(name = "order")
public class Order {

    @Id
    private Long id;

    @OneToOne
    @JoinColumn(name = "customer_id")
    private Customer customer;
```

Asociace

- @ManyToMany - jednosměrná

```
@Entity
@Table(name = "person")
public class Person {

    @Id
    private Long id;

    @ManyToMany
    private List<Department> department;
}

@Entity
@Table(name = "department")
public class Department {

    @Id
    private Long id;

    private String name;
}
```

Asociace

- @ManyToMany - obousměrná

```
@Entity
@Table(name = "person")
public class Person {

    @Id
    private Long id;

    @ManyToMany
    private List<Department> department;
}

@Entity
@Table(name = "department")
public class Department {

    @Id
    private Long id;

    @ManyToMany(mappedBy = "department")
    private List<Person> persons;
}
```

Asociace

- @ManyToMany
  - synchronizace při manipulaci s daty

```
public void addDepartment(Department dep) {
    departments.add(dep);
    dep.getPersons().add(this);
}

public void removeDepartment(Department dep) {
    department.remove(dep);
    dep.getPersons().remove(this);
}
```

Asociace

- @ManyToMany - mazání a performance
  - Co stane když někomu odeberem např. jedno oddělení?
  - proč?
  - co s tím?

Asociace

- @ManyToMany - link table entity

```
@Entity
public class Address {

    @Id
    private Long id;

    @OneToMany(mappedBy = "address", cascade = CascadeType.ALL)
    private List<PersonAddress> persons;
}
```

```
@Entity
public class Person {

    @Id
    private Long id;

    @OneToMany(mappedBy = "person", cascade = CascadeType.ALL)
    private List<PersonAddress> addresses;

    public void removeAddress(Address address) {
        PersonAddress personAddress = new PersonAddress(this, address);
        address.getPersons().remove(personAddress);
        addresses.remove(personAddress);
        personAddress.setPerson(null);
        personAddress.setAddress(null);
    }
}
```

Asociace

- @ManyToMany - link table entity

```
@Entity
public class PersonAddress {

    @Id
    @ManyToOne
    private Person person;

    @Id
    @ManyToOne
    private Address address;
}
```

Asociace

- Pokud nemáme v DB FK restrikci

```
@ManyToOne
@NotFound(action = NotFoundAction.IGNORE)
private City city;
```

Fetching

- proces získávání dat z DB
- má asi největší vliv na performance
- dvě hlavní otázky:
  - Kdy?
  - Jak?

Fetching

- Kdy se mají která data načíst
  - EAGER - teď, při volání dotazu
  - LAZY - později, při první potřebě dat

Fetching

- Jak se mají data načíst
  - direct fetch / query

```
@Entity
public class Person {

    @Id
    private Long id;

    @ManyToOne(fetch = FetchType.EAGER)
    private Department department;
}
```

```
entityManager.find(Person.class, 12L);

entityManager
    .createQuery("select p from PersonF p where p.id=12")
    .getSingleResult();
```

Fetching

- jak na dynamické načítání?
  - "manuální" inicializace - volání metod size() atp..
  - pomocí query
  - pomocí grafu

Fetching

- dynamické načítání pomocí query

```
person = entityManager.createQuery("select p from PersonF p " +
    "left join fetch p.department " +
    "where p.id=12", Person.class)
    .getSingleResult();
```

Fetching

- dynamické načítání pomocí grafu

```
@NamedEntityGraph(name = "project.employees",
    attributeNodes = @NamedAttributeNode(
        value = "employees",
        subgraph = "project.employees.department"
    ),
    subgraphs = @NamedSubgraph(
        name = "project.employees.department",
        attributeNodes = @NamedAttributeNode( "department" )
    )
)
```

Fetching

- Batch
- Dávkový load při lazy inicializaci

```
@BatchSize(size = 3)
private List<Person> persons = new ArrayList<>();
```

Immutable entita

- entita označená @Immutable
- ignoruje změny
- lepší performance díky nepotřebě dirty-checkingu
- lze anotovat i kolekci

```
@OneToMany
@Immutable
private List<Event> events;
```

NaturalId

- jde o business identifikátor
- atribut který je pro entitu jedinečný
- k označení využíváme anotace @NaturalId
- může to být primitivní atribut, embedded i třída

```
@Entity
@Table(name = "person")
public class Person {

    @Id
    private Long id;

    @NaturalId
    private String name;

}
```

Persistence context

- session/entitymanager jsou Api pro práci s persistence contextem

Persistence context

- Stavy entit:
  - transient - nově instanciovaný objekt, není reprezentován v DB ani v contextu
  - managed/persist - entita v contextu, může a nemusí být v DB
  - detached - entita má identifikátor ale už není asociována s contextem
  - removed - entita je v contextu, ale je označena ke smazání

Persistence context

- persistence
  - session.save(entity)
  - entityManager.persist(entity)
- delete
  - session.delete(entity)
  - entityManager.remove(entity)
- obtain
  - entityManager.find(class, id)
  - session.get(class, id)

Persistence context

- obtain
  - session.byId(class).load(id)
  - session.byId(class).loadOptional(id)
  - session.bySimpleNaturalId(class).load(naturalId)
  - session.byNaturalId(class).using("key", "value").load()

Persistence context

- reference na entitu
  - session.load(entityClass, id)
  - entityManager.getReference(entityClass, id)

Persistence context

- změna entity
  - managed entita
    - automatická propagace změny stavu
  - detached entita
    - entityManager.merge(entity)
    - session.merge(entity)

Persistence context

- kontrola stavu entity
  - session.contains(entity)
  - entityManager.contains(entity)
- kontrola inicializace při lazyloadingu

```
PersistenceUtil persistenceUtil = Persistence.getPersistenceUtil();
persistenceUtil.isLoaded(entity);
persistenceUtil.isLoaded(entity, "property");

Hibernate.isInitialized(entity);
Hibernate.isPropertyInitialized(entity, "property");
```

Persistence context

- Cascade - umožňuje propagovat stav z vlastníka na potomka
  - PERSIST
  - MERGE
  - REMOVE
  - REFRESH
  - DETACH
  - ALL

Flushing

- akce kdy dochází k synchronizaci stavu mezi persistence contextem a databází
  - AUTO - default
  - COMMIT
  - ALWAYS - jen nad session
  - MANUAL - jen nad session

Flushing

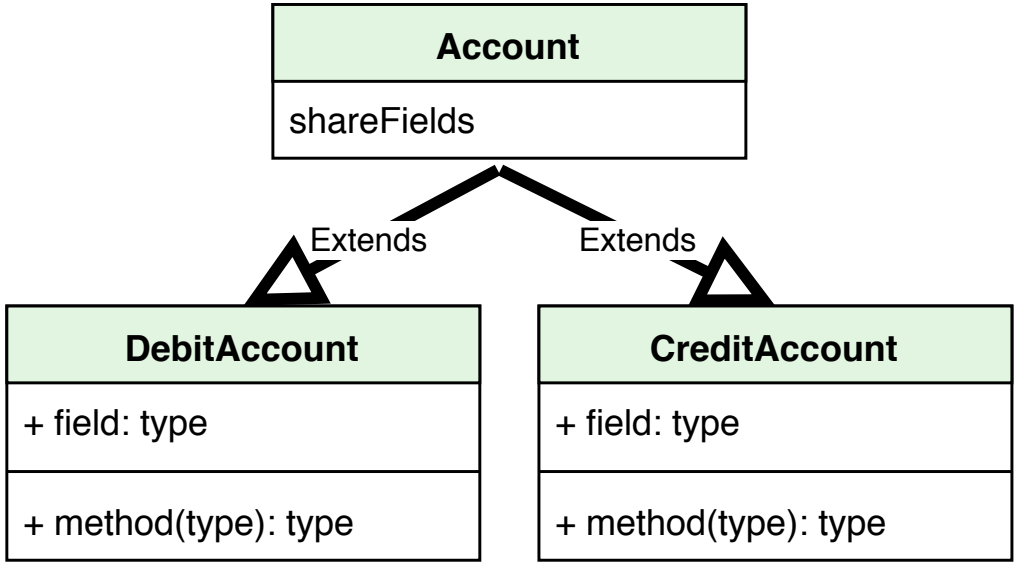
- AUTO
  - při potvrzení transakce
  - při překrytí dotazů
  - před jakýmkoli native SQL
- COMMIT
  - až při potvrzení transakce
- ALWAYS
  - před každým query
- MANUAL
  - až při manuálním zavolání flush()

Dědičnost

- Relační Db "dědičnost" nemají
- Hibernate umožňuje několik možností jak dědičnosti dosáhnout
- mappedSuperclass
  - jen v jave - mimo db
- singletable
  - jedna tabulka s daty pro více entit
- joinedtable
  - rodič a všechny děti mají své tabulky
- tableperclass
  - každý potomek má svou tabulku

Dědičnost

- MappedSuperclass

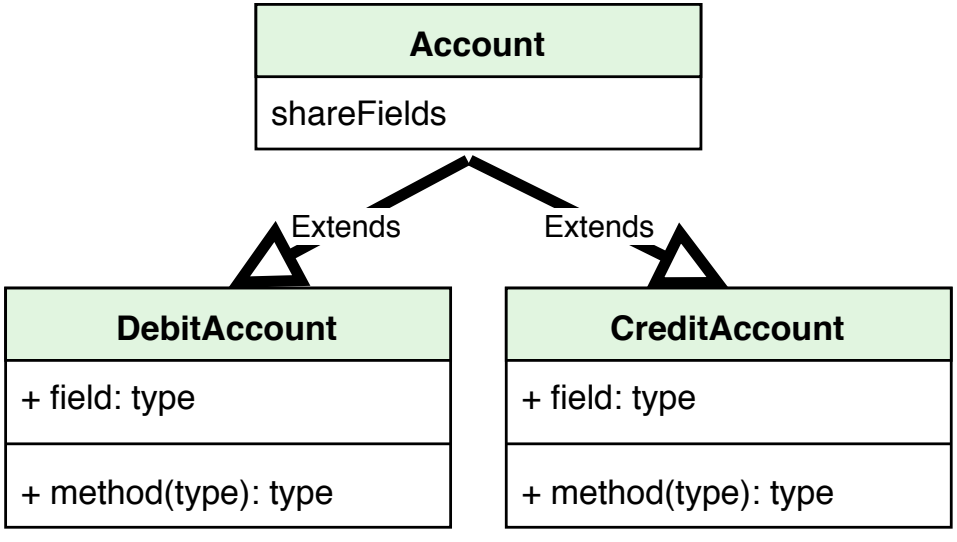


CreditAccount
Row 1
Row 2
Row 3

DebitAccount
Row 1
Row 2
Row 3

Dědičnost

- SingleTable - vše v jedné tabulce
- nejlepší performance



Account
DTYPE
creditAccFields
debitAccFields

Dědičnost

- SingleTable - discriminator (DTYPE)

```
@Entity
@Table(schema = "inheritance", name = "account")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "class")
public class Account {
}

@DiscriminatorValue(value = "D")
public class DebitAccount extends Account {
}
```

	class character varying (31)	**id** bigint	**name** character varying (255)	**limit** integer	**overdraft** integer
1	C	1	credit	1000	[null]
2	D	2	debit	[null]	4321

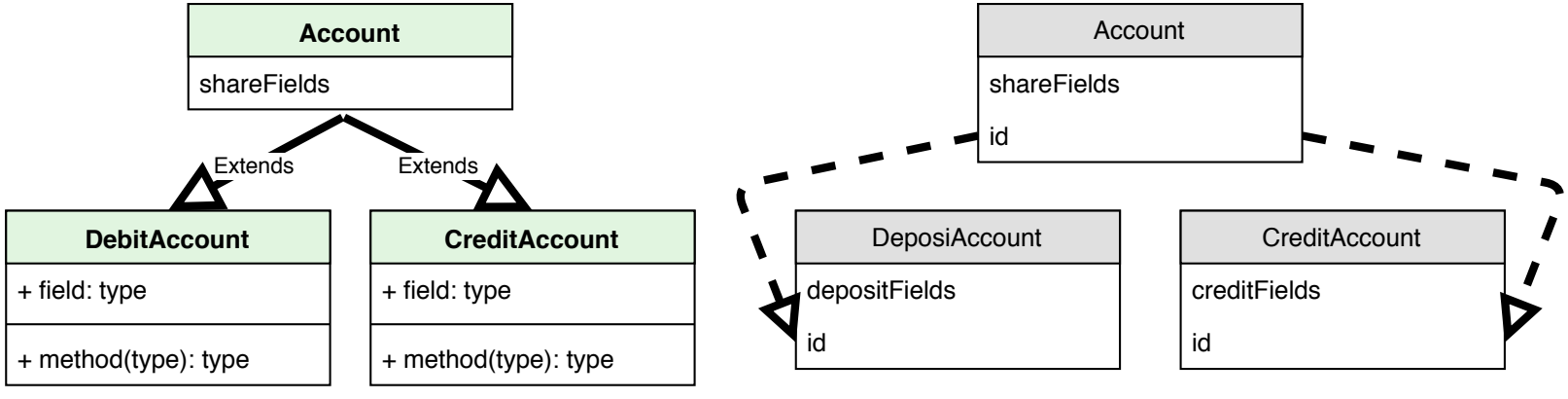
Dědičnost

- SingleTable - virtuální discriminator
- @DiscriminatorFormula

```
@Entity
@Table(schema = "inheritance", name = "account")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorFormula(
        "case when \"limit\" is not null " +
        "then 'C' " +
        "else ( " +
        "   case when overdraft is not null " +
        "   then 'D' " +
        "   else 'NA' " +
        "   end ) " +
        "end ")
public class Account {
```

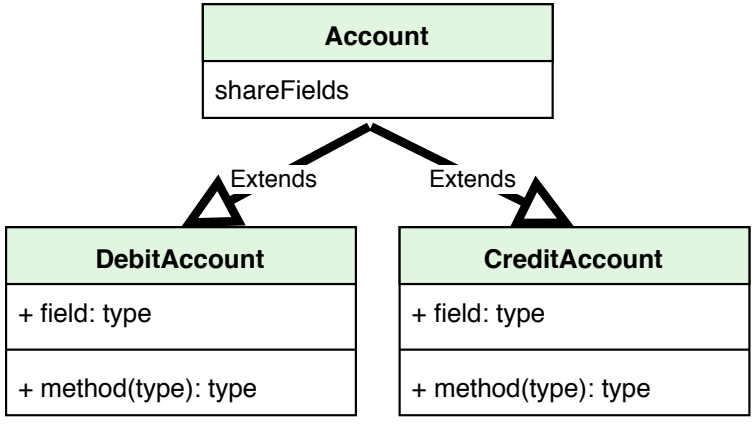
Dědičnost

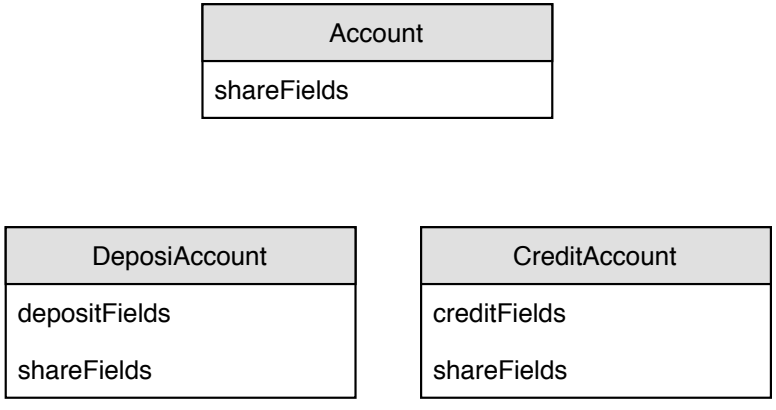
- JoinedTable - každá třída má svoji tabulku



Dědičnost

- tablePerClass - plnohodnotná tabulka pro každou entitu





Transakce

- probráno ve školení Springu

Locking

- způsob jak "chránit" data mezi časem načtení a použitím

- Optimistic
  - předpokládá že transakce mohou běžet souběžně
  - před potvrzením transakce zkontroluje že data nebyla změněna
- Pessimistic
  - předpokládá že transakce jsou konfliktní
  - uzamkne zdroje a nedovolí změny

Locking

- OptimisticLockTypes
  - VERSION(default) - používá spec. atribut
  - NONE - vypne locking
  - ALL - kontroluje všechny prvky entity
  - DIRTY - kontroluje změněné prvky

Locking

- @Version

```
@Version
@Source(SourceType.DB)
private Timestamp version;

@Version
private long version;

@OptimisticLock(excluded = true)
private String name;
```

Locking

- OptimisticLockType
  - ALL
  - DIRTY

```
@Entity
@Table
@OptimisticLocking(type = OptimisticLockType.DIRTY)
@DynamicUpdate
public class Customer {
```

Locking

- Pessimistic
  - READ - zamkne data pro zápis
  - WRITE - zamkne data pro všechny akce

```
session.find(Customer.class, 12L, LockModeType.PESSIMISTIC_READ)
```

2nd lvl Cache

- příznivě ovlivňuje performace cachováním opakovaně používaných částí aplikace

2nd lvl Cache

- hibernate umí ingerovat více 2nd lvl cachí
  - ehcache
  - jchahe
  - infinispan

2nd lvl Cache

- ehcache

```
<property name="hibernate.cache.use_second_level_cache" value="true"/>
<property name="hibernate.cache.region.factory_class"
    value="org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory"/>
```

2nd lvl Cache

- entity nejsou defaultně cachované a to díky shared cached mode
  - ENABLE_SELECTIVE - default
    - cachuje entity označené @Cachable
  - DISABLE_SELECTIVE
    - cachuje vše co není expicitně označeno @Cachable(false)
  - ALL
    - cachuje vše
  - NONE
    - necachuje nic

2nd lvl Cache

- concurrency strategy
  - READ_ONLY - nepovolí update nad entitou
  - READ_WRITE - povolí i update

```
@Entity
@Table(schema = "cache", name = "order")
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.READ_ONLY)
public class Order {
```

2nd lvl Cache

- load pomocí naturalId funguje stejným způsobem

```
Person person = session
    .byNaturalId( Person.class )
    .using( "code", "unique-code")
    .load();
```

2nd lvl Cache

- cache kolekcí
  - pokud jde o basic kolekci(@ElementCollection) zapíše se standartně do cache
  - pokud jde o associace(ManyToOne...) do cache se zapíší jen ID

2nd lvl Cache

- hinty pro ovlivnění cache
  - USE - čte i zapisuje do/z cache
  - BYPASS - nečte/nezapisuje do cache
  - REFRESH - zapíše do cache a vynutí si select

```
Map<String, Object> hints = new HashMap<>(  );
hints.put( "javax.persistence.cache.retrieveMode " , CacheRetrieveMode.USE );
hints.put( "javax.persistence.cache.storeMode" , CacheStoreMode.REFRESH );
Person person = entityManager.find( Person.class, 1L , hints);
```

2nd lvl Cache

- eviction
  - přes entityManagerFactory/SessionFactory

```
entityManager.getEntityManagerFactory().getCache().evict( Person.class );
```

JPQL

- Java persistence query language

```
@NamedQueries({
    @NamedQuery(
        name = "get_phone_by_number",
        query = "select p " +
                "from Phone p " +
                "where p.number = :number"
    )
})

Phone phone = entityManager
    .createNamedQuery( "get_phone_by_number", Phone.class )
    .setParameter( "number", "123-456-7890" )
    .getSingleResult();
```

Criteria

- criteria api pro tvorbu dotazů

```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();

CriteriaQuery<Person> criteria = builder.createQuery( Person.class );
Root<Person> root = criteria.from( Person.class );
criteria.select( root );
criteria.where( builder.equal( root.get( Person.name ), "John Doe" ) );

List<Person> persons = entityManager.createQuery( criteria ).getResultList();
```

Native

- možnosta native sql query

```
List<Person> persons = entityManager.createNativeQuery(
    "SELECT * FROM Person", Person.class )
.getResultList();
```

Batching

- postup jak bezpečně vkládat velké množství dat

```
<property name="hibernate.jdbc.batch_size" value="10"/>


    transaction.begin();

    int batchSize = 5;

    for (int i = 10; i > 1; i--) {
        Info info = new Info();
        entityManager.persist(info);

        if (i % batchSize == 0) {
            entityManager.flush();
            entityManager.clear();
        }

        System.out.println("end");
    }

    transaction.commit();
```

Spring integration

- Co je třeba nakonfigurovat

Spring integration

- datasource

```
@Bean
@Profile("!junit")
public DataSource dataSource() {

    String jndiName = propertyResolver.getProperty(JNDI_NAME_PROPERTY_KEY, DEFAULT_VALUE);
    JndiObjectFactoryBean jndiObjectFactoryBean = new JndiObjectFactoryBean();
    jndiObjectFactoryBean.setJndiName(jndiName);
    jndiObjectFactoryBean.setResourceRef(true);
    jndiObjectFactoryBean.setProxyInterface(DataSource.class);

    return (DataSource) jndiObjectFactoryBean.getObject();
}
```

Spring integration

- entityManagerFactory

```
@Bean
@DependsOn("flyway")
public EntityManagerFactory entityManagerFactory(DataSource dataSource) {

    LocalContainerEntityManagerFactoryBean factory = new LocalContainerEntityManagerFactoryBean()

    HibernateJpaVendorAdapter hibernateJpaAdapter = new HibernateJpaVendorAdapter();
    factory.setJpaVendorAdapter(hibernateJpaAdapter);
    factory.setPackagesToScan("packege.entity", "another.entity.package");
    factory.setDataSource(dataSource);
    factory.setJpaProperties(hibernateProperties());
    factory.afterPropertiesSet();

    return factory.getObject();
}
```

Spring integration

- hibernate properties

```
private Properties hibernateProperties() {

        String hbm2dll = propertyResolver.getProperty(HBM2DLL_PROPERTY_KEY, DEFAULT_VALUE);
        String schema = propertyResolver.getProperty(SCHEMA_PROPERTY_KEY, DEFAULT_VALUE);
        String dialect = propertyResolver.getProperty(DIALECT_PROPERTY_KEY, DEFAULT_VALUE);
        String logSql = propertyResolver.getProperty(LOG_SQL_PROPERTY_KEY, DEFAULT_VALUE);

        return new Properties() {
            {
                setProperty("hibernate.hbm2dll.auto", hbm2dll);
                setProperty("hibernate.default_schema", "[" + schema + "]");
                setProperty("hibernate.dialect", dialect);
                setProperty("hibernate.physical_naming_strategy",
                        "dtag.reservation.chatbot.server.common.UpperCaseNamingStrategy");
                if (logSql != null && Boolean.valueOf(logSql)) {
                    setProperty("hibernate.show_sql", "true");
                    setProperty("hibernate.format_sql", "true");
                    setProperty("hibernate.use_sql_comments", "true");
                }
            }
        };
```

Spring integration

- transaction manager

```
@Bean
public PlatformTransactionManager transactionManager(DataSource dataSource) {

    JpaTransactionManager txManager = new JpaTransactionManager();
    txManager.setEntityManagerFactory(entityManagerFactory(dataSource));

    return txManager;
}
```

Spring integration

- jpa repository