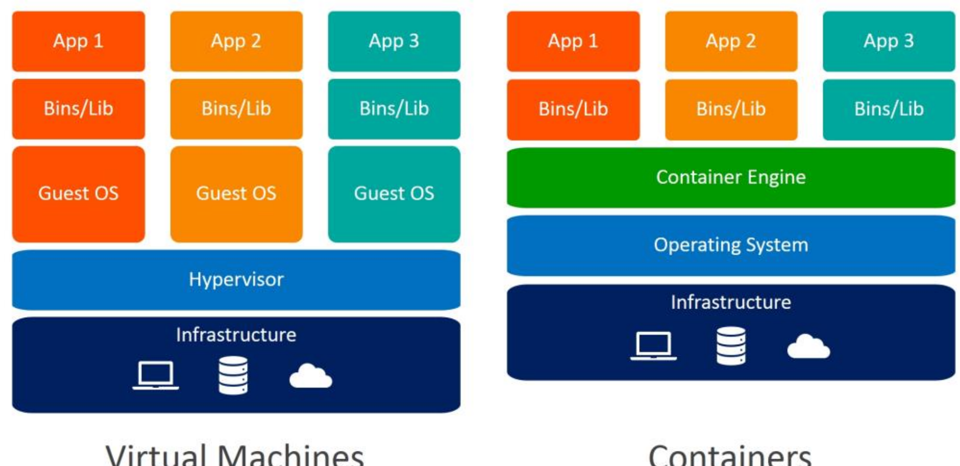
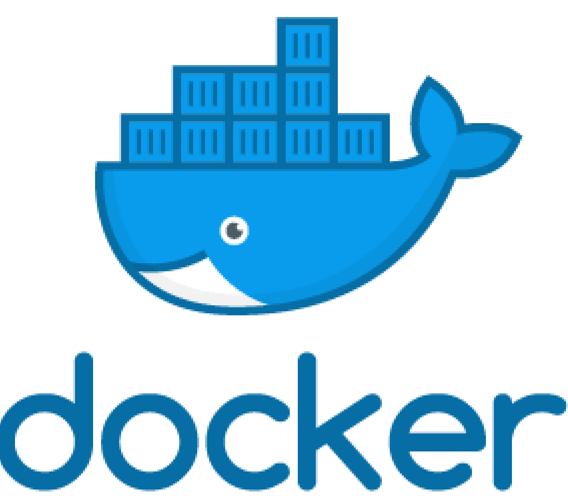
docker
Docker, časť. 1
Projekt 1
Prednáška 2

Kontajnerizácia



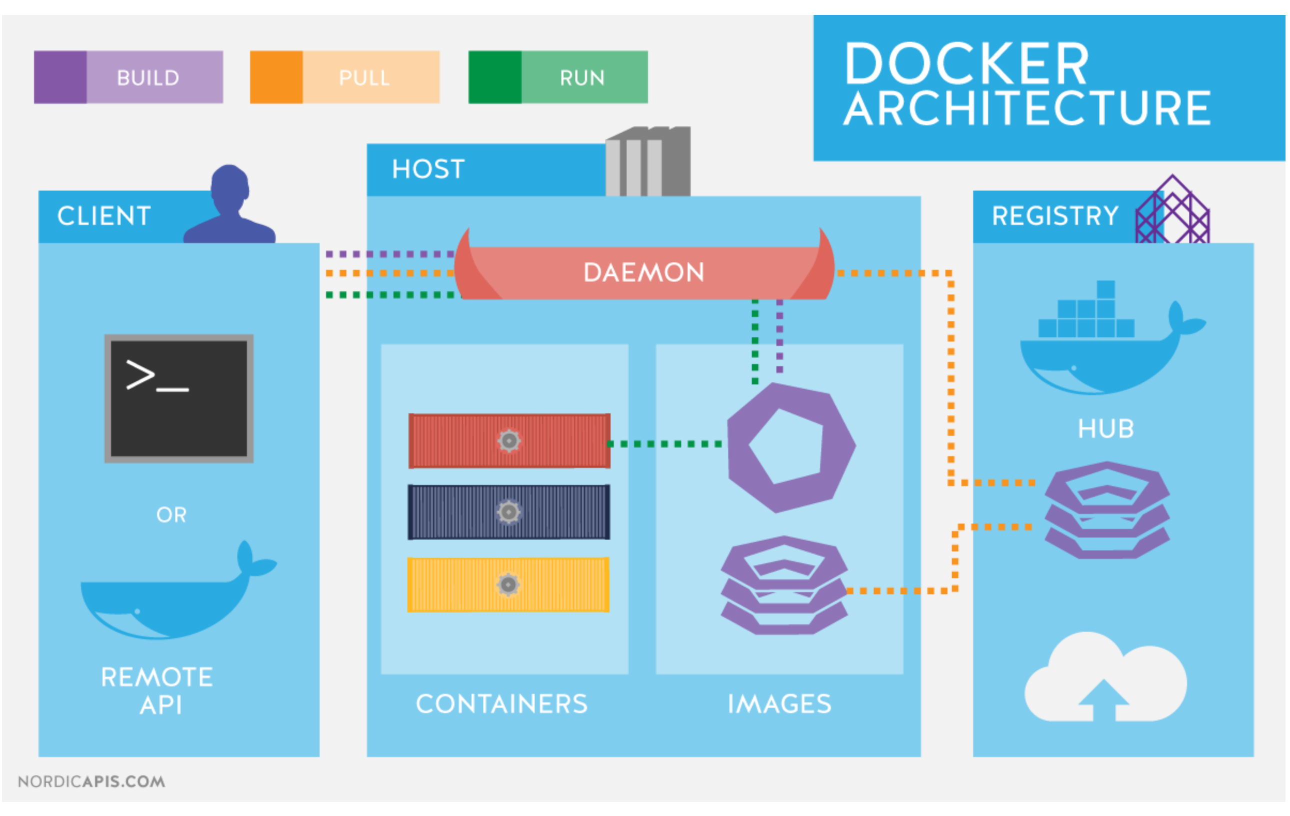
- Virtuálne stroje
  - Viac OS (OS jadier) na jednom PC súčasne
  - OS jadrá môžu byť rôzne (NT, Linux, XNU)
- Kontajnery
  - Viac užívateľských priestorov v jednom OS súčasne
  - Máme jedno OS jadro
  - Nad tým istým OS jadrom pustíme viacero OS distribúcií
    - Ubuntu, Debian, vlastné
    - Windows edície atď.

Docker



- Najpoužívanejší nástroj na kontajnerizáciu
- Užívateľsky prívetivá nadstavba nad OS kontajnerizáciou
- Umožňuje ľahko vytvárať špecifické kontajnery
  - MySQL kontajner
  - NodeJS kontajner
  - GCC kontajner
  - Java kontajner
  - ...
- Špecifické kontajnery spustia aplikáciu izolovane od hlavného OS
- Z hlavného OS si berú iba jadro (kernel)
- Ostatné veci si kontajner pribalí sám
- Ak kontajner funguje na jednom stroji, tak funguje na všetkých strojoch

Architektúra Dockera



- Kľúčové komponenty
  - démon, klient, registre
- Objekty
  - Obrazy (images)
    - MySQL, Java, GCC, NodeJS, moja appka
  - Kontajnery
    - Spustené obrazy
  - Siete
  - Úložiská (volumes)

Inštalácia

Docker Engine

- Linuxový program
- V repozitároch
- Iba CLI
- Apache 2 licencia
- Natívne linuxové kontajnery

Docker Desktop

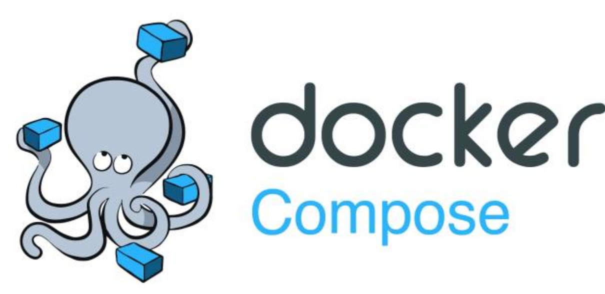
- Funguje na Win, Mac, Linux
  - Linuxová VM pod kapotou
    - podpora GPU paravirtualizácie vo Win (WSL 2)
- Z webu, Chocolatey, Homebrew
- CLI + GUI
- Platená, okrem
  - Nekomerčné použitie
  - Školstvo
  - Komerčné použitie s <$10 mil alebo <250 zamestnancami

Docker CLI

- Základné príkazy
  - docker pull – stiahni obraz
  - docker build – vytvor obraz mojej appky
  - docker run – spusti obraz (vytvor kontajner)
  - docker ps – ukáž kontajnery
- Dockerfile: skript na vytvorenie vlastného obrazu
  - Vstupný súbor pre `docker build`

```
docker pull mysql:latest

docker run --name runningMySQL -e MYSQL_ROOT_PASSWORD=verysecret -d mysql:latest
```

Docker Compose



- Jednoduchšie spustenie kontajnerov
  - Najmä ak chceme spustiť viac naraz
- YAML formát
- Skvelý pre vývoj, integračné testovanie, non-HA produkčné nasadenia
  - Jedným príkazom vieme spustiť celý aplikačný stack
    - Databázy, brokery, frontend, backend, ...
  - Pre HA (high-availability) nasadenia používame Kubernetes alebo Docker Swarm

Ukážka – MySQL ako kontajner

docker-compose.yaml

```
services:
  mysql:
    image: 'mysql:latest‘
    environment:
      - 'MYSQL_DATABASE=entrance'
      - 'MYSQL_PASSWORD=secret'
      - 'MYSQL_ROOT_PASSWORD=verysecret'
      - 'MYSQL_USER=myuser'
    ports:
      - '3306:3306'
#    volumes:
#      - ./init.sql:/docker-entrypoint-initdb.d/init.sql
```

stiahni najnovší MySQL

konfigurácia

mapuj port host<-kontajner

(voliteľne) sprístupni súbor init.sql do priečinka */docker-entrypoint-initdb.d v kontajneri. MySQL ho načíta pri prvom spustení*

Spustíme príkazom `docker compose up`

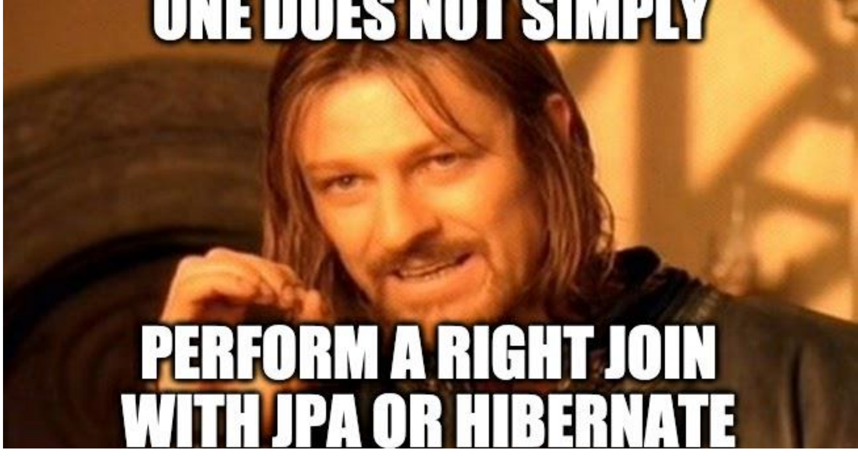
alebo `docker compose up -f docker-compose.yaml`

spring
Spring Data JPA
Projekt 1
Prednáška 2

ORM – Objektovo-relačné mapovanie

- Prepojenie objektov v prog. jazyku a SQL databázy
- Entitná tabuľka v SQL = Entitná trieda v Java
- Relačná tabuľka v SQL = inštančné premenné typu List/Set/... v entitných triedach
- Relačný model a objektovo orientovaný model nie sú veľmi kompatibilné
- **Problém cyklických referencií**
  - Užívatelia majú prístup k čítačkám kariet. Čítačka kariet potrebuje vedieť akého užívateľa môže pustiť.
  - V SQL -> relačná tabuľka users_cardreaders
  - V Jave -> trieda User má field List<CardReader> a/alebo trieda CardReader má field List<User>
    - Zacyklenie
- **N+1 query problém**
  - Entita má relácie s N ďalšími entitami.
  - SELECT-nem entity a jej relácie na-JOIN-ujem v jednom über dopyte?
  - Alebo SELECT-nem iba entity a relácie budem SELECT-ovať samostatne pre každý riadok?
- Na Paz1c sme ORM robili ručne v DAO triedach
  - Trieda ResultSet predstavovala výsledok príkazu SELECT
  - Každý riadok z ResultSet sme previedli na náš objekt (User, CardReader)

JPA – Jakarta Persistence



- Pred 2019 známe ako Java Persistence API
- Vysokoúrovňová práca s SQL v Jave
- Nadstavba nad JDBC
- Má vlastný dopytovací jazyk JPQL
  - SQL, ale nad Java triedami namiesto tabuľkami

SQL
```
SELECT u.* FROM user u JOIN user_card_readers ucr ON u.id = ucr.user_id
```

JPQL
```
SELECT u FROM User u WHERE u.cardReaders IS NOT EMPTY
```

- Tiež môžeme SQL dopyty robiť programovo a objektovo
  - Criteria Query API
- Generovanie SQL tabuliek, primárnych a cudzích kľúčov z Java entitných tried
- ...

JPA je rozhranie

- JPA je špecifikácia a banda rozhraní
- Potrebuje implementáciu aby fungovalo
- Má reálne viac implentácií

- Hibernate ORM – asi najznámejšia implementácia JPA
  - Hibernate je paradoxne starší ako JPA
  - Jeho vlastné API vyzerá úplne inak ako JPA
  - Asi najpopulárnejšie Java ORM pred JPA (a nepriamo aj po JPA)

- Rôzne frameworky používajú rôzne implementácie JPA
  - Ale to vás nemusí v súčasnosti veľmi zaujímať

Spring Data JPA

- Použitie **Repository** vzoru namiesto DAO
- Vytvoríme si interface UserRepository extends JpaRepository<User, Long>
- Automaticky dostaneme, **findById**, **findAll**, s voliteľným stránkovaním a ORDER BY
- Spring automaticky implementuje hlavičky našich metód
  - Podľa názvu/návratového typu/vstupných parametrov metódy
  - Pr.: `List<User> findByName(String name);`
    - `SELECT * FROM users WHERE name = ?;`
  - Pr.: `User findByUsernameOrEmail(String username, String email);`
    - `SELECT * FROM users WHERE username = ? OR email = ? LIMIT 1;`
  - https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html
- Zložitejšie dopyty cez anotáciu @Query
  - implicitne JPQL, alebo voliteľne SQL
  - Alebo implementujeme ručne cez Criteria Query API, alebo cez surové JDBC

Poďme si vytvoriť JPA demo

- V IntelliJ IDEA Ultimate, alebo cez https://start.spring.io/ si vytvorme nový *Spring Boot Project*
- ->File/New/Project/Spring Initializr
  - Language: Java
  - Type: Maven
  - JDK: cesta k JDK 21, alebo nechajte IDE stiahnuť
  - Java: 21
  - Packaging: Jar
- Klikneme na *Next*

Komponenty pre spring boot

- V okne *Dependencies* si vyklikáme
  - Lombok
  - Spring Web
  - Spring Data JPA
  - MySQL Driver
- Klikneme dole na *Create*

Spustíme si MySQL v Dockeri

docker-compose.yaml

```yaml
services:
  mysql:
    image: 'mysql:latest‘
    environment:
      - 'MYSQL_DATABASE=entrance'
      - 'MYSQL_PASSWORD=secret'
      - 'MYSQL_ROOT_PASSWORD=verysecret'
      - 'MYSQL_USER=myuser'
    ports:
      - '3306:3306‘
```

Spustíme príkazom `docker compose up`

- `docker compose stop` – zastaví kontajnery
- `docker compose down` – zastaví a zmaže kontajnery

Nakonfigurujeme Spring

src/main/resources/application.properties

```
# url databázy, tiež nastaviteľná cez premennú prostredia
spring.datasource.url=${DB_JDBC:jdbc:mysql://localhost:3306/entrance}

# konto v databáze, tiež nastaviteľné cez premennú prostredia
spring.datasource.username=${DB_JDBC:myuser}

# heslo k databáze, tiež nastaviteľné cez premennú prostredia
spring.datasource.password=${DB_JDBC:secret}

# z entitných tried generuj tabuľky v databáze
spring.jpa.generate-ddl=true

# zapni JPA repozitáre (namiesto DAO)
spring.data.jpa.repositories.enabled=true
```

Modelujme entity

```
@Data
@Entity
public class User {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;

   private String name;

   @Column(unique=true)
   private String email;

   private boolean active;

   private int sex;

   @ManyToMany
   private List<CardReader> cardReaders;
}
```

- @Data nám vygeneruje gettre a settre
- @Entity táto trieda bude tabuľka v SQL
- @Id stĺpec s primárnym kľúčom
- @GeneratedValue(...) bude automaticky generovať hodnoty ID. Na vstup vieme určiť spôsob generovanie. *IDENTITY* znamená použite AUTO_INCREMENT v MySQL
- Inštančné premenné bez anotácie budú normálne stĺpce
- @Column(unique=true) nám zabezpečí unikátnosť
- Pohlavie ľudí v DB sa môže riadiť štandardom https://en.wikipedia.org/wiki/ISO/IEC_5218
- @ManyToMany predstavuje vzťah M:N s entitou CardReader. Pomocná relačná tabuľka v SQL sa vygeneruje automaticky na pozadí a v Jave ju neriešime (ale môžeme ak chceme)

Modelujme entity

```
@Data
@Entity
public class CardReader {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;

   @Column(unique=true)
   private String position;

   @ManyToMany(mappedBy = “cardReaders”)
   @JsonIgnore
   private List<User> users;
}
```

- @ManyToMany predstavuje vzťah M:N s entitou User.

  Vstupný parameter určuje, že inštančná premenná cardReaders v triede User je „vlastníkom“ vzťahu.

  V praxi to znamená, že na inštančnú premennú s takouto anotáciou **nešahajte**.
- Modifikácia listu a následné uloženie entity **nebude fungovať korektne**.

- @JsonIgnore nám nebude serializovať tento stĺpec aby sme sa vyhli nekonečnej rekurzii
  - User má CardReader, CardReader má User, ...
  - V praxi sa to rieši ináč
    - Na tejto strane relácie nebudeme mať inštančnú premennú users
    - Použitie tzv. DTO objektov

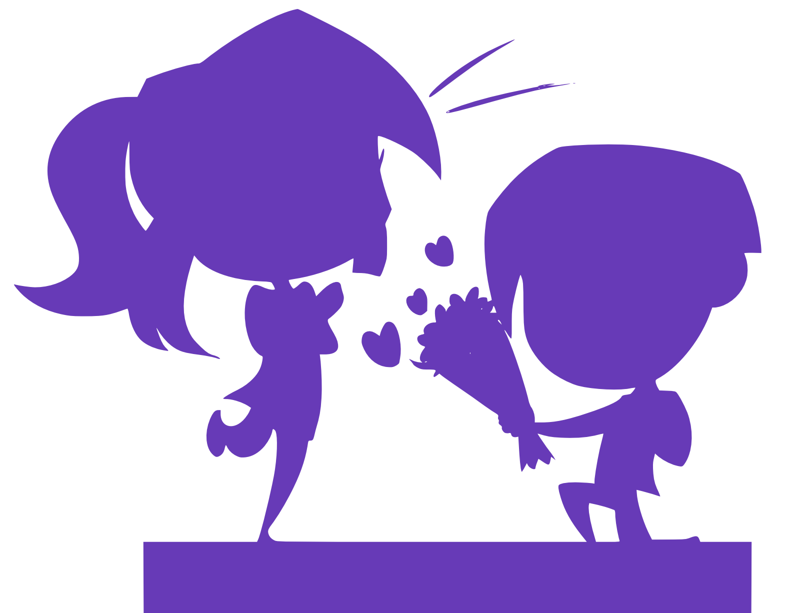
Repozitáre

- V PAZ1c ste na načítanie/ukladanie entít používali DAO triedy
- Spring JPA má však pre nás chuťovku – rozhranie JpaRepository

```
public interface UserRepository extends JpaRepository<User, Long> {

    Optional<User> findByEmail(String email);

    List<User> findByActiveTrue();
    Page<User> findByActiveTrue(Pageable pageable);

    @Query("SELECT u FROM User u WHERE u.cardReaders IS NOT EMPTY")
//    @Query(nativeQuery = true, value = "SELECT u.* FROM user u JOIN user_card_readers ucr ON u.id = ucr.users_id")
    List<User> findWithCardReaderAccess();

    List<User> findByCardReaders(CardReader cardReader);

}
```

Ďalšie možnosti vzťahov

- Ešte máme relačné anotácie @OneToMany+@ManyToOne a @OneToOne
- V inštančnej premennej, ktorá je vlastníkom vzťahu (t.j. nemá v relačnej anotácií atribút mappedBy) môžeme
  - definovať kaskádové operácie
  - definovať načítavanie vzťahov (N+1 problém)
    - okamžité načítavanie – FetchType.EAGER
    - načítavanie podľa potreby – FetchType.LAZY (defaultne)



Načítanie vzťahov

Ak robíme SELECT na entitu (User), tak

- FetchType.EAGER nám urobí JOIN až k druhej entite (CardReader) a naplní user-ovú inštančnú premennú typu List<CardReader>

- FetchType.LAZY nechá List<CardReader> „prázdny“
  - Ak budeme následne „prázdny“ zoznam iterovať, tak sa „magicky“ naplní
  - **Pozor**, „magické“ (odborne *lenivé*) napĺňanie funguje iba kým je otvorené spojenie k DB
  - Ak používate SpringBoot + Spring Web + JPA, tak spojenie je aktívne počas celej HTTP požiadavky a lenivé napĺňanie funguje bez problémov

  - Mnohé projekty však majú nastavenú inú politiku spojenia k DB
    - Spojenie začína a končí v Repository/DAO
    - Po vrátení entity z Repository/DAO, môžete pri následnom prístupe k List-u dostať LazyInitializationException
    - Ak vám nejaká metóda hádže túto výnimku, použite na ňu @Transactional
      - To vám (okrem iného) bude udržiavať spojenie s DB počas volania metódy

S veľkou mocou...

JDBC (pre absolventov PAZ1c)

JPA

Ďakujem za pozornosť

Nech vás sprevádza Sila