

Работа со бази на податоци

JPA и Spring Data JPA



Универзитет „Св. Кирил и Методиј“ во Скопје
**ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ
И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО**



JPA



Универзитет „Св. Кирил и Методиј“ во Скопје
**ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ
И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО**



Животен циклус на ентитет

- Состојби на ентитет во зависност од неговата позиција во перзистентниот контекст
 - *Транзитен* (анг. Transient) - ентитетот е надвор од перзистентниот контекст
 - добиен преку директно инстанцирање со new()
 - *Управуван* (анг. Managed) - ентитетот е во перзистентниот контекст
 - управуван од страна на EntityManager
 - Било која промена врз ентитетот ќе се синхронизира со базата на податоци.
 - *Откачен* (анг. Detached) - ентитетот бил во перзистентниот контекст но е привремено острнет
 - Промените врз ентитетот нема да се синхронизираат со базата
 - *Отстранет* (анг. Removed) - ентитетот е отстранет од контекстот и означен за бришење.
 - При синхронизацијата ќе биде избришана соодветаната редица од табелата

Методи на EntityManager

- `persist(entity)` – внесува ентитет `entity` во перзистентен КОНТЕКСТ

```
@Transactional
public Address create(String street, String city, int postalCode){
    Address addr = new Address();
    addr.setStreet(street);
    addr.setCity(city);
    addr.setPostalCode(postalCode);
    return em.persist(addr)
}
```

Методи на EntityManager

- `find(Entity.class, id)` – пребарува инстанца на ентитет од класата `Entity` со идентификатор `id` во базата на податоци
 - Прави cast во соодветната класа
 - Пронајдениот објект се сместува во перзистентниот контекст
 - null ако не постои таков објект

```
@Transactional
public Address findById(int id){
    return em.find(Address.class,id)
}
```

- `contains(entity)` – пребарува инстанца на ентитет `entity` во перзистентниот контекст

Методи на EntityManager

- `refresh(entity)` – присилно ажурира ентитет `entity` со податоци од базата на податоци
- Уредување на ентитет
 - Ентитетот треба да биде перзистентен
 - Доволно е да се променат неговите својства
 - Синхронизација по излез од методот (`@Transactional`, AOP)

```
@Transactional
public Address update(int id, String street, int zip, String city){
    Address addr = em.find(Address.class,id);
    if (addr != null){
        addr.setStreet(street);
        addr.setZip(zip);
        addr.setCity(city);
    }
    return addr;
}
```

Методи на EntityManager

- `detach(entity)` – откачување на ентитет `entity` од перзистентен контекст
 - Било какви промени врз ентитетот не се синхронизираат
- `clear()` – откачување на сите ентитети од перзистентен контекст
 - Откако EntityManager ќе заврши со сите трансакции, го повикува овој метод за да го испразни перзистентниот контекст
- `remove(entity)` – отстранување на ентитет `entity` од перзистентен контекст и маркирање за бришење

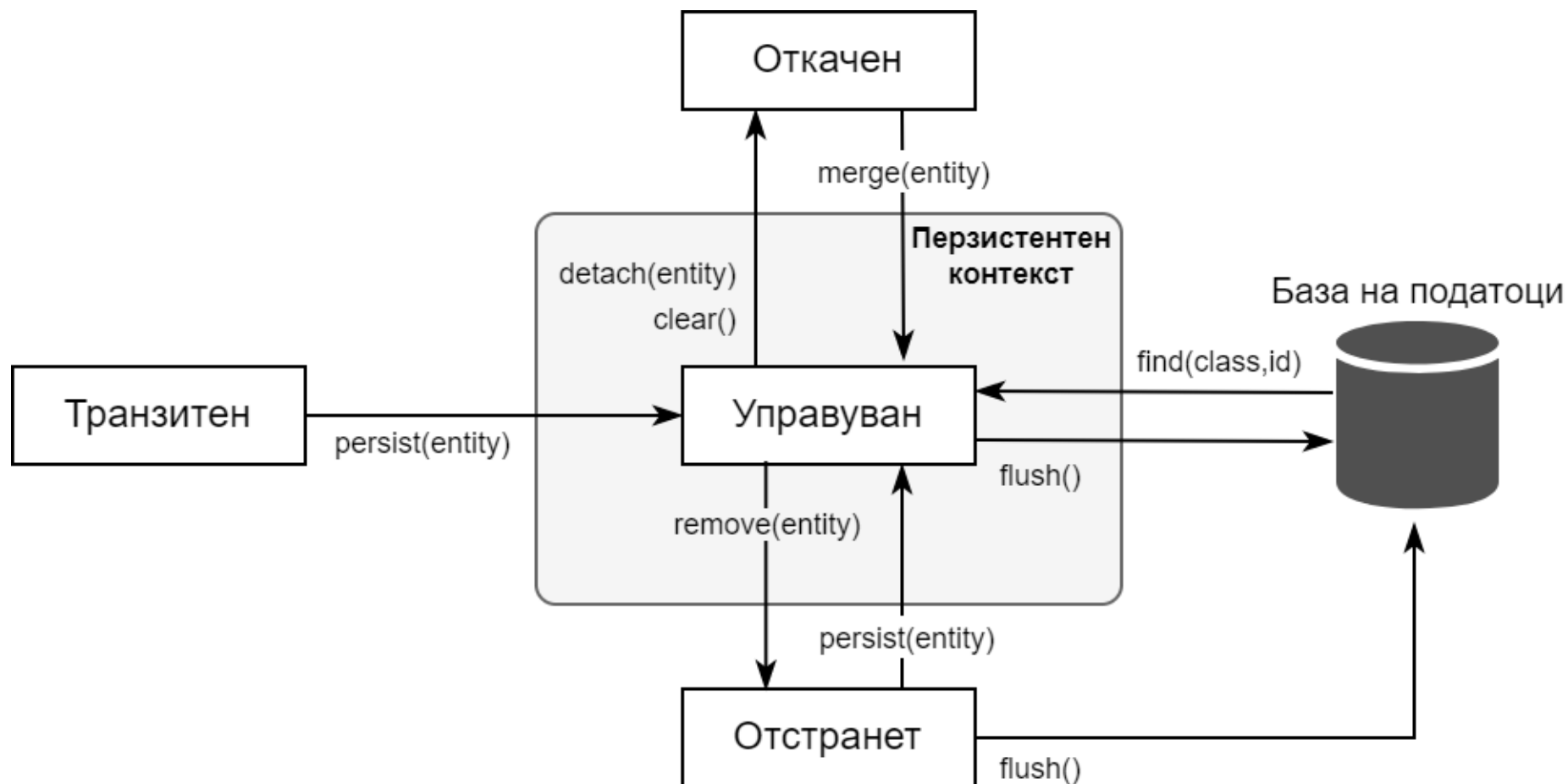
Методи на EntityManager

- `merge(entity)` – враќање на откачен ентитет `entity` во перзистентен контекст
 - Ентитетот бил претходно откачен (пр. пристигнува објект од фронтенд кој треба да се ажурира во база)

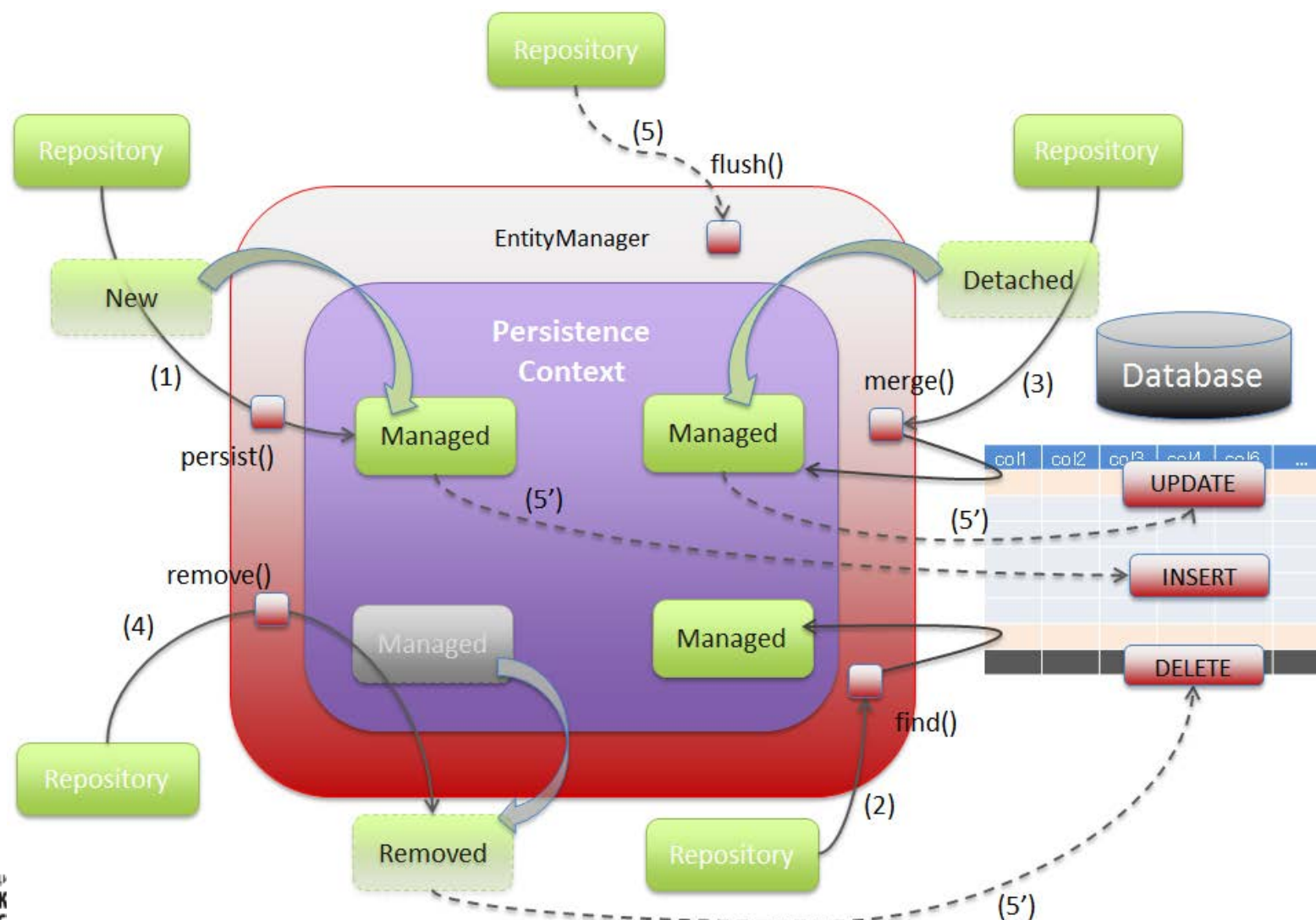
```
@Transactional
public void update(Address addr){
    if (addr.id != null){
        em.merge(addr)
    }
}
```

- `flush()` – присилно синхронизирање на сите промени на ентитетите од перзистентниот контекст со базата

Животен циклус на ентитет и методи



Животен циклус на ентитет и методи



Кориснички дефинирани барања

- Барања напишани во JPQL или SQL преку класи изведени од `Query`
 - `TypedQuery` - динамички барања во JPQL
 - `createQuery(query, class)` - креирање на `TypedQuery` со команди во `query`
 - `NamedQuery` – статички барања во JPQL зачувани како функција
 - `createNamedQuery(namedQuery, class)` - креирање на `NamedQuery` со команди во `query`
 - `NativeQuery` – динамички барања во SQL
 - `createNativeQuery(query, class)` - креирање на `NamedQuery` со команди во `query`
- Секоја класа нуди методи за извршување на барањата
 - `getResultList()` - враќа листа од елементи
 - `getSingleResult()` - враќа само еден елемент
 - `executeUpdate()` - извршува барање за ажурирање или бришење кое не враќа конкретни резултати

Пример за репозиториум со JPA

```
@Repository
public class AddressRepository {
    @PersistenceContext
    EntityManager em;

    @Transactional
    public Address findById(int id){
        return em.find(Address.class,id);
    }

    @Transactional
    public Address create(Address address){
        em.persist(address);
        return address;
    }

    @Transactional
    public Address update(Address address){
        em.merge(address);
        return address;
    }
}
```

```
@Transactional
public void delete(int id){
    Address address = em.find(Address.class,id);
    em.remove(address);
}

@Transactional
public List<Address> findAll(){
    TypedQuery<Address> query = em.createQuery("select a from Address
        a", Address.class);
    return query.getResultList();
}

@Transactional
public List<Address> findByPostalCode(int pc){
    TypedQuery<Address> query = em.createQuery("select a from Address
        where a.postalCode = :postalCode", Address.class);
    query.setParameter("postalCode", pc);
    return query.getResultList();
}
```



Пример за репозиториум со JPA

```
Address address;  
address = new Address();  
address.setStreet("Happiness Blvd.");  
address.setCity("Joy town");  
address.setPostalCode(1000);  
addressRepository.create(address);  
System.out.println("----New address-----");  
System.out.println(address.toString());
```

Hibernate: select nextval ('hibernate_sequence')
Hibernate: insert into address (city, country, postal_code, street, id)
↪ values (?, ?, ?, ?, ?)

----New address-----

Address(id=1, street=Happiness Blvd., city=Joy town, postalCode=1000,
↪ country=null)

```
address.setCountry("Dreamland");  
addressRepository.update(address);  
System.out.println("----Updated address-----");  
System.out.println(address.toString());
```

Hibernate: select ... from address address0_ where address0_.id=?
Hibernate: update address set city=?, country=?, postal_code=?, street=?
↪ where id=?

----Updated address-----

Address(id=1, street=Happiness Blvd., city=Joy town, postalCode=1000,
↪ country=Dreamland)

Пример за репозиториум со JPA

```
System.out.println("----All addresses-----");  
System.out.println(addressRepository.findAll());
```

```
System.out.println("----Filtered addresses-----");  
System.out.println(addressRepository.findByPostalCode(1000));
```

Hibernate: select address0_.id as id1_0_, address0_.city as city2_0_,
↪ address0_.country as country3_0_, address0_.postal_code as
↪ postal_c4_0_, address0_.street as street5_0_ from address address0_

Hibernate: select ... from address address0_ where address0_.postal_code=?

Читање на ентитети во релација

- Кога се читаат дестинациските ентитети ако од базата се побара изворен ентитет?
- Начини на читање на дестинациски ентитети од базата
 - **LAZY** - дестинацискиот ентитет(и) не се чита(ат) од базата со самото читање на изворниот ентитет
 - За да се прочита(ат) потребно е експлицитно да се повика `getter`
 - При повик на `getter` се генерира ново барање до базата
 - Предефинирано однесување за **@*ToMany** врските
 - Може да се промени преку својството **fetch** како аргумент на анотацијата за релација
 - Дефиниција на LAZY однесување: **fetch=FetchType.LAZY**

```
@OneToMany(fetch=FetchType.LAZY)  
private List<Order> orders = new ArrayList<>();
```


Читање на ентитети во релација

- **EAGER** - дестинацискиот ентитет(и) се чита(ат) од базата со самото читање на изворниот ентитет
 - Предефинирано однесување за **@*ToOne** врските
 - Ако се користи кај **@*ToMany** може да резултира со слаби перформанси ако изворниот ентитет е во врска со голем број дестинациски ентитети
 - Дефиниција на EAGER однесување: **fetch=FetchType.EAGER**

```
@ManyToOne(fetch=FetchType.EAGER)
@JoinColumn(name = "customer_id")
private Customer customer;
```


Уредување на ентитети во релација

- Потребно е да се дефинира како промените на изворниот ентитет во однос на перзистентниот контекст ќе се афектираат кај дестинацискиот(те) ентитет(и)
 - Пример: ако транзиентен изворен ентитет кој има транзиентни дестинациски ентитети се перзистира, што ќе се случи со дестинациските ентитети?
 - Ако се перзистираат, нема никаков проблем
 - Ако не се перзистираат, настанува проблем бидејќи перзистентен објект се врзува во релација со неперзистентни објекти (не постојат во базата)
 - Пример за бришење на ентитет во релација
- Може да се дефинира начинот на пренесување (каскада) на состојбата на изворниот ентитет кон дестинацискиот ентитет преку аргументот **cascade** на анотацијата за релацијата

Уредување на ентитети во релација

- Елементи на енумерација **CascadeType** за дефинирање на каскада:
 - **PERSIST** - повик на методот `persist()` кај изворниот ентитет се пренесува и кај дестинациските ентитети
 - **MERGE** - повик на методот `merge()` се пренесува и кај дестинациските ентитети.
 - **DETACH** - повик на методот `detach()` се пренесува и кај дестинациските ентитети.
 - **REMOVE** - повик на методот `remove()` се пренесува и кај дестинациските ентитети.
 - Внимателна употреба: ако се избрише изворниот ентитет, неповратно се бришат дестинациските ентитети
 - **REFRESH** - повик на методот `refresh()` се пренесува и кај дестинациските ентитети.
 - **ALL** - Било која методите за промена на состојбата на изворниот ентитет се пренесува на дестинациските ентитети (ги вклучува сите горенаведени)
- Може да се дефинираат комбинации од разните типови на каскади

Уредување на ентитети во релација

- Предефинирано однесување - зависи од имплементацијата, но често не вклучува ниеден тип на каскада
- Пример
 - пренесување на сите методи

```
@Entity
public class Customer {
    @Id @GeneratedValue
    private int id;
    private String firstName;
    private String lastName;
    @OneToMany(cascade = CascadeType.ALL)
    private List<Order> orders = new ArrayList<>();
}
```

- Пренесување само на merge() и persist()

```
@OneToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
private List<Order> orders = new ArrayList<>();
```

Справување со нерференцирани ентитети

- Отстранување на дестинациски ентитет во `@OneTo*` релација кај изворен ентитет (пр. бришење на `order` од `customer.orders`)
- Дестинацискиот ентитет не се брише од базата
 - Надворешниот клуч на дестинацискиот ентитет се поставува на вредност `null`
 - Дестинацискиот ентитет станува сирак (orphan) и останува нерференциран
- Се користи аргумент `orphanRemoval` на анотацијата за `@OneTo*` релацијата
 - Предефинирана вредност `orphanRemoval = false`

Справување со нерференцирани ентитети

- Пример

```
customer.getOrders().remove(order)
```

- Предефинирано однесување

- Избришаната инстанца `order` станува сирак со вредност `null` за надворешниот клуч `customer_id` во базата

- Отстранување на сираци

```
@OneToMany(orphanRemoval=true, cascade = CascadeType.ALL)  
private List<Order> orders = new ArrayList<>();
```

- Ако се отстрани `order` од `customer.orders`, при перзистирање на `customer` ќе се отстрани редицата која одговара на таа инстанца на `order`
- `CascadeType.ALL` не означува дека треба да се избрише дестинациската инстанца бидејќи во овој случај изворната инстанца `customer` се перзистира (откако од неа е отстанета една инстанца `order`), не се брише

Додавање и бришење ентитети кај двонасочни релации

- Ажурирањето на ентитетите треба да се направи во двете насоки од врската преку додавање референци
- Пример Order-Customer
 - Додавање на инстанца `order` од ентитетот `Order` кај инстанца `customer` со својство `orders`

```
order.setCustomer(customer);  
orders.add(order);
```

- Бришење на инстанца `order` од ентитетот `Order` кај инстанца `customer` со својство `orders`

```
order.setCustomer(null);  
orders.remove(order);
```

Spring Data JPA



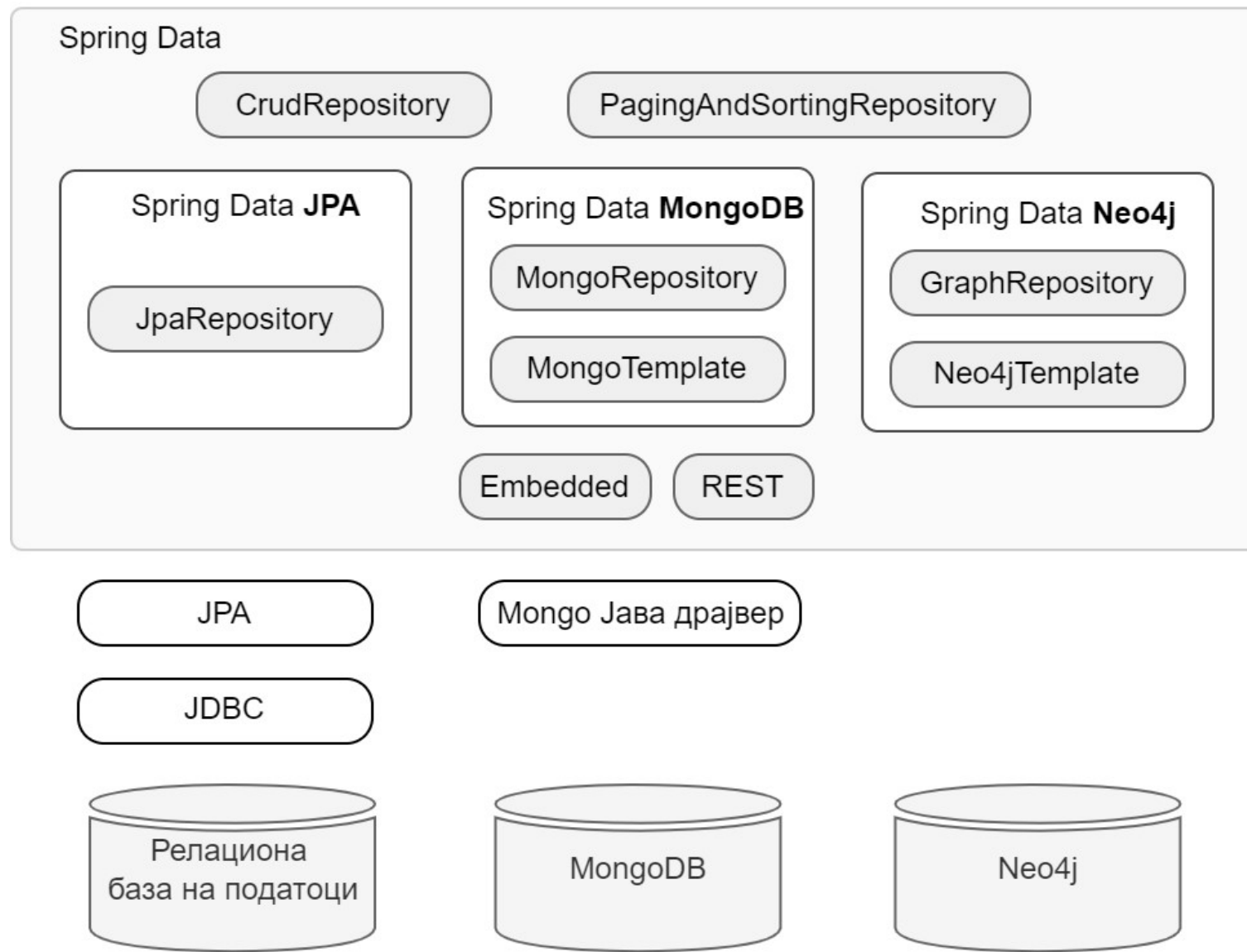
Универзитет „Св. Кирил и Методиј“ во Скопје
**ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ
И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО**



Spring Data

- Од претходно, за CRUD операции со ентитети, потребно е да се генерираат различни репозиториуми со сличен код
 - Кодот за CRUD операции се разликува според аргументите на методите и типот на податоци кој го враќаат
 - Дуплирање на код
 - Поголеми можности за грешки
 - Помала продуктивност
- Spring Data – проект на Spring за работа со податочен слој, независно од податочното складиште
 - Автоматски креира код за CRUD операции со ентитети преку интерфејси за репозиториуми
 - Spring Data JPA – модул за работа со релациони бази на податоци

Spring Data



Интерфејси за репозиториуми

- CrudRepository

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {  
    <S extends T> S save(S entity);  
    Optional<T> findById(ID primaryKey);  
    Iterable<T> findAll();  
    long count();  
    void delete(T entity);  
    boolean existsById(ID primaryKey);  
}
```

- Проширување на основниот интерфејс `Repository` кој служи само за маркирање на типот на податоци
 - Класа на ентитет со кој ќе управува (`T`)
 - Перзистентен идентитет на ентитетот (`ID`)
- Важи за било кое поставено складиште (не само за JPA)



Интерфејси за репозиториуми

- Методи на `CrudRepository`
 - `save` го зачувува ентитетот `S` кој може да биде проширување на `T`.
 - `findById` пребарува ентитет со идентификатор `ID`.
 - Враќа `Optional`, објект кој може, но и не мора да има вредност
 - `findAll` враќа листа од сите ентитети од типот `T`
 - `count` го враќа бројот на ентитети од типот `T`
 - `delete` го брише ентитетот `T`
 - `existsById` враќа `true` ако постои ентитет `T` со идентификатор `ID`
- `T` е генерички тип и се заменува со ентитетот за кој е дефиниран репозиториумот

Интерфејси за репозиториуми

- Репозиториум за конкретен ентитет се дефинира само преку проширување на `CrudRepository`

- Пример: репозиториум за ентитет Order

```
interface OrderRepository extends CrudRepository<Order,Integer>{  
}
```

- Со инјектирање на репозиториум дефиниран за ентитет, Spring Data креира креира имплементација на сите методи од интерфејсот `CrudRepository`

Интерфејси за репозиториуми

- Имплементација со JPA провајдер
 - Обврска на програмерот
- Имплементација со [CrudRepository](#)
 - Веќе е имплементирана без линија код

```
@Transactional
public Order save(Order order){
    if (order.getId() == null){
        em.persist(order);
        return order;
    } else
        return em.merge(order);
}
```



```
Order save(Order order)
```



Интерфејси за репозиториуми

- **PagingAndSortingRepository**

```
public interface PagingAndSortingRepository<T, ID> extends  
↳ CrudRepository<T, ID> {  
    Iterable<T> findAll(Sort sort);  
    Page<T> findAll(Pageable pageable);  
}
```

- Проширување на интерфејсот **CrudRepository** со функционалности за
 - Подредување
 - Страничење
- Методи на **PagingAndSortingRepository**
 - **findAll(Sort sort)** – враќа подредена листа од ентитети со подредување дефинирано преку објектот **sort**
 - насока (**Sort.Direction.ASC** или **Sort.Direction.DESC**) и
 - својство за подредување
 - Пример: **findall(Sort.by(Sort.Direction.DESC, "date"))**

Интерфејси за репозиториуми

- `findAll(Pageable pageable)` – враќа објект од типот `Page` кој содржи само дел од целокупната колекција на ентитети
 - `getContent()` – враќа колекција од ентитети кои се дел од страницата
 - `getTotalElements()` – го враќа вкупниот број на ставки (од сите страници!)
 - `getTotalPages()` – го враќа вкупниот број на страници
 - Објектот од типот `Page` овозможува дефинирање на
 - големина на страница
 - бројот на страница која треба да се врати
 - Подредување преку назначување на објект од типот `Sort`
 - Пример: `findall(1, 10, Sort.by(Sort.Direction.DESC, "date"))`
 - Страна со индекс 1 (втора страна) и големина од 10 ставки
 - Ги содржи ставките со индекси од 10 до 19
 - Подредување во опаѓачка насока според својство `date`

Интерфејси за репозиториуми

- JpaRepository

```
public interface JpaRepository<T, ID> extends PagingAndSortingRepository<T, ID> {  
    List<T> findAll();  
    List<T> findAll(Sort sort);  
    List<T> findById(Iterable<ID> ids);  
    void flush();  
    <S extends T> List<S> saveAll(Iterable<S> entities);  
    <S extends T> saveAndFlush(S entity);  
    <S extends T> List<S> saveAllAndFlush(Iterable<S> entities);  
    void deleteInBatch(Iterable<T> entities);  
    ...  
}
```

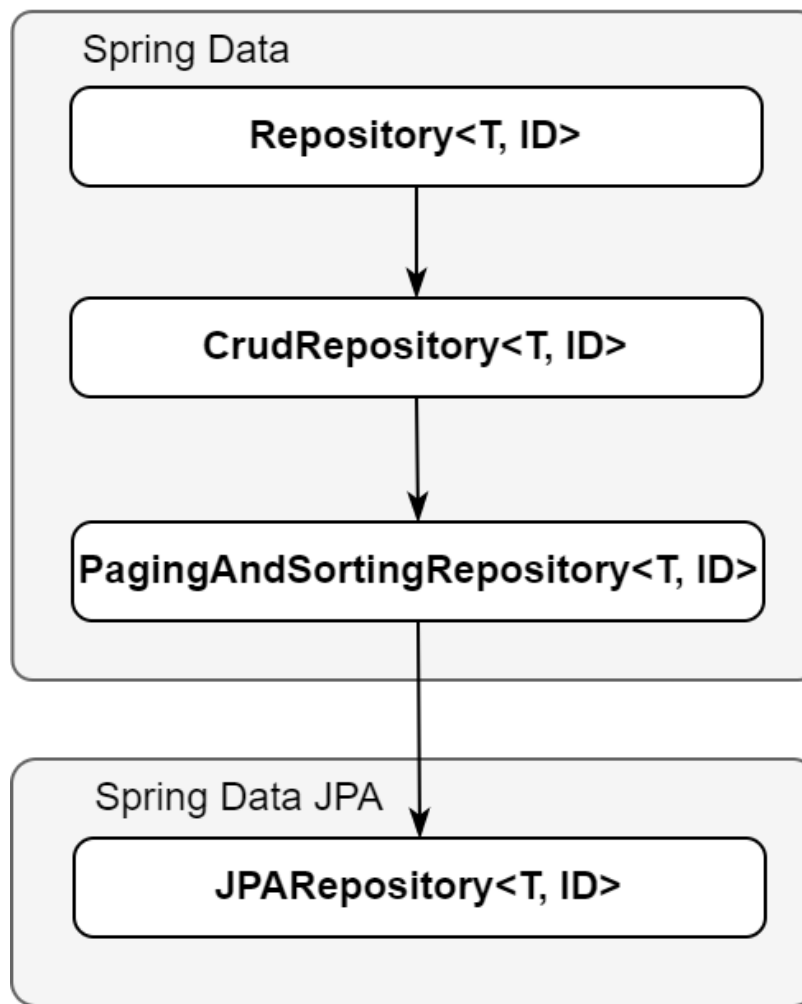
- Проширување на [PagingAndSortingRepository](#) со JPA специфични методи
 - Дел од Spring Data JPA



Интерфејси за репозиториуми

- Методи на `JpaRepository`
 - `findAll()` – враќа листа од објекти `List`
 - Не враќа `Iterable` како `JpaRepository` (може да се користи и метод кој враќа `Iterable`, бидејќи тој е дел од репозиториумот од кој е изведен `JpaRepository`)
 - `Iterable` има методи `hasNext()` и `next()` vs. `List` има методи `add()`, `remove()`, `get()`
 - `flush()` - го повикува `flush()` метод во JPA за моментална синхронизација на перзистентниот контекст
 - `saveAll(Iterable<S> entities)` – зачувува цела листа од ентитети
 - `saveAndFlush(S)` - ги повикува `save()` и `flush()` за да постигне инстантно зачувување на ентитетот во база
 - `saveAllAndFlush(Iterable<S> entities)` – инстантно зачувува цела листа од ентитети во база
 - `DeleteInBatch(Iterable<T> entities)` – брише листа од ентитети

Преглед на интерфејси за репозиториуми



Кориснички дефинирани методи

- Spring Data овозможува креирање на специфични барања за ентитети само преку дефиниција на името на методот
- Пример
 - Имплементација со JPA провајдер

```
@Transactional
public List<Order> findByCustomerId(Integer id){
    TypedQuery<Order> query = em.createQuery("select o from Order o
    ↪ where o.customer.id = :id", Order.class);
    query.setParameter("id",id);
    return query.getResultList();
}
```

- Имплементација со `JpaRepository`

```
interface OrderRepository extends JpaRepository<Order,Integer>{
    public List<Order> findByCustomerId(Integer id);
}
```

Кориснички дефинирани методи

- Конвенција за именување на методи
 - *findBy*, *readBy*, *getBy*, *searchBy* и *queryBy* - враќаат листа од ентитети кои го задоволуваат условот кој следи во името
 - *countBy* враќа број на ентитети кои го задоволуваат условот во името
 - *deleteBy* и *removeBy* – бришат ентитети кои го задоволуваат условот во името
 - *existsBy* – true ако постојат ентитети кои го задоволуваат условот во името
 - *...First<number>...* и *...Top<number>...* - се користат помеѓу *find* и *By* за да се ограничи бројот на ставки во резултатот
 - Пример, *findFirst10ByName* и *findTop10ByName*
 - *...Distinct...* - се користи помеѓу *find* и *By* за да се вратат само уникатните ентитети
 - Пример: *findDistinctByName*

Кориснички дефинирани методи

- После дефинирањето на типот на пребарувањето следуваат предикати кои ги дефинираат условите.
- Секој предикат започнува со својство на ентитетот за кој е дефиниран репозиториумот и еден или повеќе клучни зборови
- Секој збор кој започнува со голема буква се третира како својство сè до појавање на клучен збор или до крајот на името на методот.
- Ако постојат повеќе зборови со голема буква, сите тие се третираат како својство со истото име, но со мала почетна буква
 - Ако не постои, се проверува постоење на својства одвоени со точка
 - Пример: `findByCustomerId(Integer id)`
 - Се проверува постоење на својство `customerId`
 - Ако не постои, се проверува постоење на својство `customer.id`



Кориснички дефинирани методи

- За експлицитно пребарување по својство кој е дел од сложено својство, се користи знакот долна црта (_).
 - Пример: кај `findByCustomer_Id(Integer id)` се пребарува по `customer.id`
- На секое својство во условот одговара аргумент од истиот тип чија позиција зависи од позицијата на својството во името.
- Ако станува збор за бинарен оператор (пр. `between`) се користат два аргументи

Клучни зборови за предикати

Клучен збор	Пример	JPQL репрезентација
And	<code>findByLastNameAndFirstName</code>	<code>select distinct ... where x. .lastName = ?1 and x.firstName = ?2</code>
Or	<code>findByLastNameOrFirstName</code>	<code>... where x.lastName = ?1 or x.firstName = ?2</code>
True, IsTrue	<code>findBySubscribedTrue()</code>	<code>... where x.subscribed = true</code>
False, IsFalse	<code>findBySubscribedFalse()</code>	<code>... where x.subscribed = false</code>
Not	<code>findByNameNot</code>	<code>... where x.name <> ?1</code>
In	<code>findByNameIn(Collection<String> names)</code>	<code>... where x.name in ?1</code>
NotIn	<code>findByNameNotIn(Collection<String> names)</code>	<code>... where x.name not in ?1</code>
Empty, IsEmpty	<code>findByOrdersIsEmpty()</code>	<code>... where x.orders is empty</code>
NotEmpty, IsNotEmpty	<code>findByOrdersNorEmpty()</code>	<code>... where x.orders is not empty</code>



Клучни зборови за предикати

Is, Equals	findByName, findByNameEquals, findByNameIs	... where x.name = ?1
IsNull, Null	findByNameIsNull, findByNameNull	... where x.name is null
IsNotNull, findByNameNotNull	findByNameIsNotNull, findByNameNotNull	... where x.name not null
LessThan	findByAmountLessThan	... where x.amount < ?1
LessThanEqual	findByAmountLessThanEqual	... where x.amount <= ?1
GreaterThan	findByAmountGreaterThan	... where x.amount > ?1
GreaterThanEqual	findByAmountGreaterThanEqual	... where x.amount >= ?1
Before	findByPurchaseDateBefore	... where x.purchaseDate < ?1
After	findByPurchaseDateAfter	... where x.purchaseDate > ?1
Between	findByPurchaseDateBetween	... where x.purchaseDate between ?1 and ?2

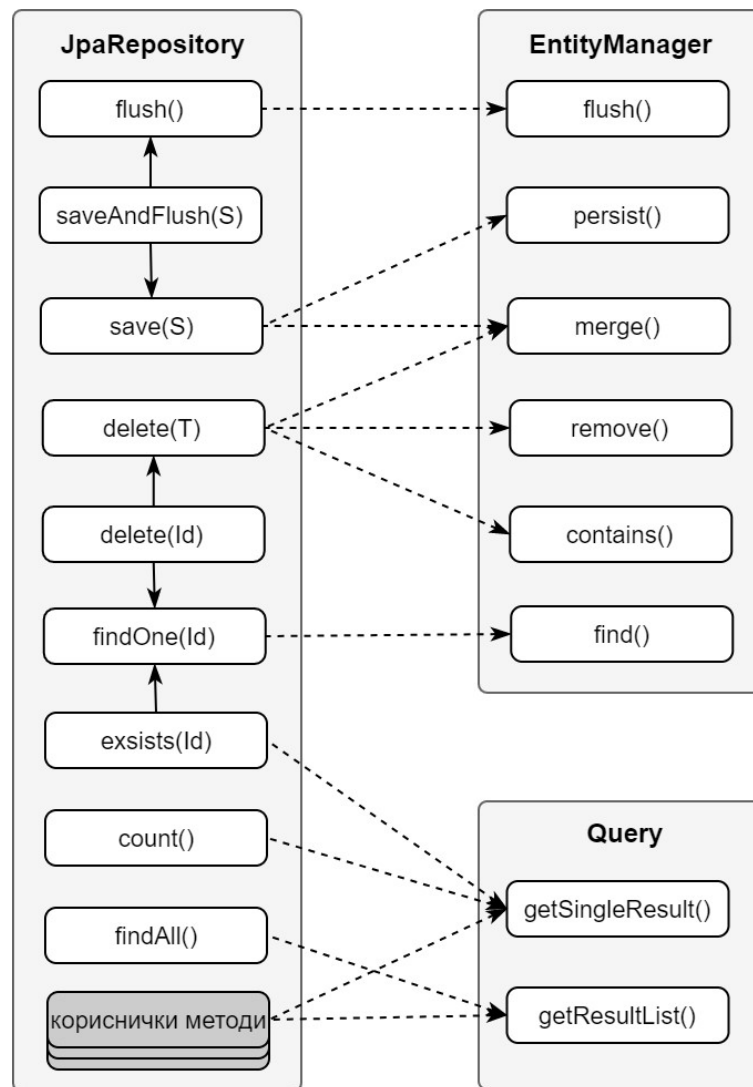


Клучни зборови за предикати

Containing	findByNameContaining	... where x.name like '%?1%'
Like	findByNameLike	... where x.name like ?1
NotLike	findByNameNotLike	... where x.name not like ?1
StartingWith, StartsWith, IsStartingWith	findByNameStartingWith	... where x.name like '?1%'
EndingWith, EndsWith, IsEndingWith	findByNameEndingWith	... where x.name like '%?1'
IgnoreCase, IgnoringCase	findByNameIgnoreCase	... where UPPER(x.name) = UPPER(?1)
AllIgnoreCase, AllIgnoringCase	findByNameAndTitleAllIgnoreCase	where UPPER(x.name) = UPPER(?1) and UPPER(x.title) = UPPER(?2)
OrderBy	findByNameOrderByAgeDesc	... where x.name = ?1 order by x. .age desc



Споредба на методи од JpaRepository и JPA



Кориснички дефинирани барања

- Некои покомплескни барања не можат да се дефинираат преку кориснички дефинирани методи
 - Пример Join на табели
- Spring Data JPA овозможува механизам за дефинирање на барања со користење на JPQL или SQL
- Се користи анотација `@Query` над методите
 - Барањето се пренесува како аргумент name на анотацијата
 - Аргумент `nativeQuery = false`, барањето се третира како JPQL (предефинирано)
 - Аргумент `nativeQuery = true`, барањето се третира како SQL
 - Се игнорира името на методот
- Методи со кориснички дефинирани барања се дефинираат во репозиториум кој го наследува `JpaRepository`

Кориснички дефинирани барања

- Пример

- Онзачување на параметри преку име

- `:ime` во барање
 - `@Param("ime")` пред аргумент во метод

```
interface OrderRepository extends JpaRepository<Order,Integer>{  
    @Query("select o from Order o where o.customer.id = :id and o.status =  
        ↳ :status ")  
    public List<Order> findByCustomerIdAndStatus(@Param("id") Integer  
        ↳ customerId, @Param("status") OrderStatus satus);  
}
```

- Означување на параметри преку позиција

- `?n` во барање, каде `n` е позиција (индекс) на аргумент во методот

```
@Query(value="select * from eshop_order o where o.customer_id = ?1 and  
    ↳ o.status = ?2 ", nativeQuery=true)  
public List<Order> findByCustomerIdAndStatus(Integer customerId,  
    ↳ OrderStatus status);  
}
```

Кориснички дефинирани барања

- Пример
 - Напредно корисничко барање со страничење
 - Pageable

```
@Query("select o from Order o where o.customer.id = ?1 and o.status = ?2 ")  
public Page<Order> findByCustomerIdAndStatus(Integer customerId,  
    ↪ OrderStatus status, Pageable pageable);  
}
```

Кориснички дефинирани барања

- Кориснички дефинирани барања кои прават промена во база
 - Update, create, delete
 - Анотација `@Modifying`

```
@Modifying
@Query("update Order o set o.amount = (1-:discount*1.0/100)*o.amount where
↪ o.customer.id = :id ")
public List<Order> updateCustomerIdAmount(@Param("id") Integer customerId,
↪ @Param("discount") Integer discount);
}
```

Други функционалности

- *Складирани процедури* (анг. Stored Procedures) - процедури напишани во SQL код и се креирани и зачувани во базата на податоци.
- *Спецификации* (анг. Specification) - класа која овозможува динамичко градење на пребарување додавајќи предикати во пребарувањето
- *Ревизија* (анг. Auditing) - водење на дневник за модификација на ентитетите кој вклучува кориснички и временски податоци за модификацијата
- *Именувани ентитетски графови* (Named Entity Graphs) за подобрување на перформансите при читање на податоците

Пример за репозиториум со Spring Data JPA

```
public interface AddressJpaRepository extends JpaRepository<Address,
    Integer> {
}

final AddressJpaRepository addressSpringRepository;
public Ch4ExamplesApplication(AddressRepository addressRepository,
    CustomerRepository customerRepository, OrderRepository orderRepository,
    AddressJpaRepository addressSpringRepository) {
    ...
    this.addressSpringRepository = addressSpringRepository;
}


List<Address> addressList = new ArrayList<>();
addressList.add(new Address("Happiness Blvd.", 2, "Joy Town", 1000,
    "Dreamland"));

...
addressList.add(new Address("Late home Sq.", 80, "Jobville", 6120,
    "Realand"));
```



Пример за репозиториум со Spring Data JPA


```
System.out.println("----Spring Data JPA: saveAll----");  
addressSpringRepository.saveAll(addressList);
```



```
----Spring Data JPA: saveAll----  
... sequence generation ...  
Hibernate: insert into address (city, country, number, postal_code, street,  
↪ id) values (?, ?, ?, ?, ?, ?)  
... insert for each instance
```

```
Address address = addressList.get(0);  
System.out.println("addressList[0]: "+address.toString());  
address.setNumber(9);
```


```
System.out.println("----Spring Data JPA: save----");  
addressSpringRepository.save(address);
```



```
----Spring Data JPA: save----  
Hibernate: select ... from address address0_ where address0_.id=?  
Hibernate: update address set city=?, country=?, number=?, postal_code=?,  
↪ street=? where id=?
```


Пример за репозиториум со Spring Data JPA

```
System.out.println("----Spring Data JPA: findById----");  
address = addressSpringRepository.findById(address.getId()).get();
```




```
----Spring Data JPA: findById----  
Hibernate: select ... from address address0_ where address0_.id=?
```

```
System.out.println("----Spring Data JPA: delete----");  
addressSpringRepository.delete(address);
```



```
----Spring Data JPA: delete----  
Hibernate: select ... from address address0_ where address0_.id=?  
Hibernate: delete from address where id=?
```

```
System.out.println("----Spring Data JPA: count----");  
System.out.println("Total items:" + addressSpringRepository.count());
```



```
----Spring Data JPA: count----  
Hibernate: select count(*) as col_0_0_ from address address0_
```

Пример за репозиториум со Spring Data JPA

```
Pageable pageable = PageRequest.of(1,3);
System.out.println("----Spring Data JPA: findAll(pageable) no sort----");
Page<Address> page = addressSpringRepository.findAll(pageable);
System.out.println("page 1: "+page.toList());
```



```
----Spring Data JPA: findAll(pageable) no sort----
Hibernate: select ... from address address0_ limit ?, ?
Hibernate: select count(address0_.id) as col_0_0_ from address address0_
```

```
System.out.println("----Spring Data JPA: findAll(pageable) with
    sort----");
pageable = PageRequest.of(0,3, Sort.by("number").descending());
page = addressSpringRepository.findAll(pageable);
System.out.println("page 0, sortBy number: "+page.toList());
System.out.println("Total elements: "+page.getTotalElements());
```



```
----Spring Data JPA: findAll(pageable) with sort----
Hibernate: select ... from address address0_ order by address0_.number desc
    limit ?
Hibernate: select count(address0_.id) as col_0_0_ from address address0_
```



Пример за репозиториум со Spring Data JPA

```
public interface AddressJpaRepository extends JpaRepository<Address,
    Integer> {
    public List<Address> findByStreetIsEndingWithIgnoreCase(String end);
    public List<Address> getByNumberBetweenAndPostalCodeGreaterThan(Integer
        startNumber, Integer endNumber, Integer postalCode);
    public Page<Address>
        readByCountryNotInOrderByPostalCodeDesc(List<String> countryList,
            Pageable pageable);

    @Query("select a from Address a where a.city like %?1%")
    public List<Address> myFindByCityContaining(String content);

    @Modifying
    @Transactional
    @Query("update Address a set a.city = :city where
        a.postalCode=:postalCode")
    public void updateByPostalcode(@Param("postalCode") Integer postalCode,
        @Param("city") String city);
}
```



Пример за репозиториум со Spring Data JPA

```
addressList =  
    ↳ addressJpaRepository.findByStreetIsEndingWithIgnoreCase("St.");  
System.out.println("Addresses with street ending with 'St.':");  
System.out.println(addressList.toString());
```



```
----Spring Data JPA: findByStreetIsEndingWithIgnoreCase() ----  
Hibernate: select ... from address address0_ where upper(address0_.street)  
    ↳ like upper(?) escape ?
```

```
System.out.println("----Spring Data JPA:  
    ↳ getBetweenAndPostalCodeGreaterThan(50,85,1200) ----");  
addressList = addressJpaRepository |  
    ↳ .getBetweenAndPostalCodeGreaterThan(50,85,1200);
```



```
----Spring Data JPA:  
    ↳ getBetweenAndPostalCodeGreaterThanOrderByNumberDesc() ----  
Hibernate: select ... from address address0_ where (address0_.number between  
    ↳ ? and ?) and address0_.postal_code > ? order by address0_.number desc
```

Пример за репозиториум со Spring Data JPA

```
System.out.println("----Spring Data JPA: updateByPostalcode(6120,  
↪ Jobtown) ----");  
addressJpaRepository.updateByPostalcode(6120, "Workburg");
```



```
----Spring Data JPA: updateCityByPostalcode() ----  
Hibernate: update address set city=? where postal_code=?
```

```
System.out.println("----Spring Data JPA:  
↪ myFindByCityContaining(Town) ----");  
addressList = addressJpaRepository.myFindByCityContaining("Work");
```



```
----Spring Data JPA: myFindByCityContaining(Town) ----  
Hibernate: select ... from address address0_ where address0_.city like ?
```

