Лекция № 6 Spring Data. JPA

Spring Data: Концепция объектно-реляционного отображения. Концепция Entity. Жизненный цикл Entity. Архитектура JPA. Основные интерфейсы. Конфигурация

Spring Data

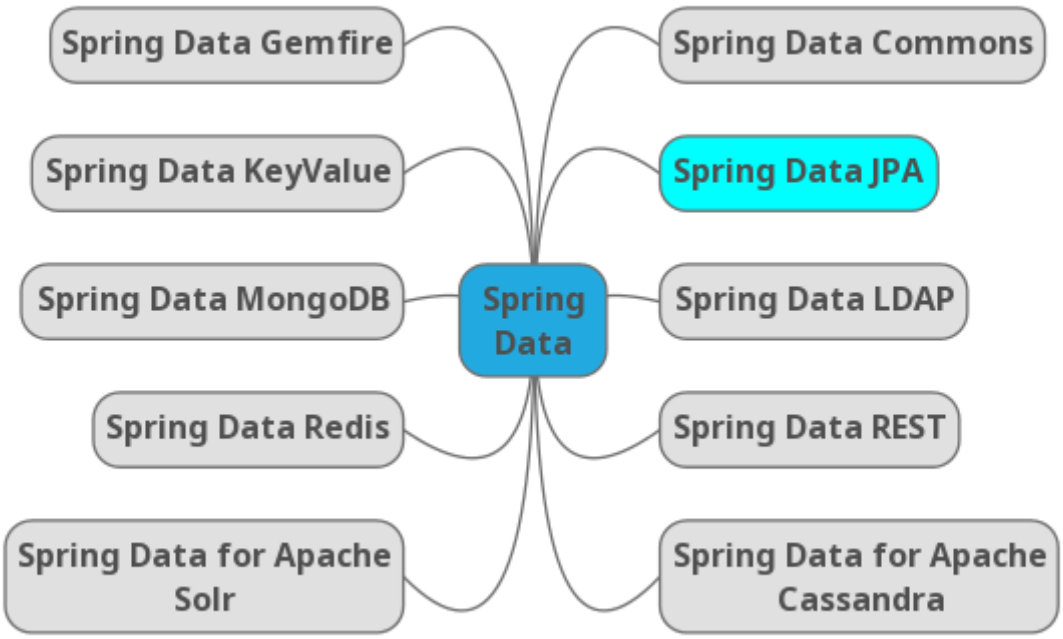
Примеры по Spring Data
https://github.com/spring-projects/spring-data-examples

Spring Data - модель программирования на основе Spring для доступа к данным. Упрощает использование технологий доступа к данным, реляционных и нереляционных баз данных, структур и облачных сервисов данных. Это проект, который содержит множество подпроектов. Проекты разрабатываются в сотрудничестве со многими компаниями и разработчиками.

Основные преимущества:

- Мощный репозиторий и пользовательские абстракции для отображения объектов
- Динамическое получение запросов из имен методов репозитория
- Реализация базовых классов домена, обеспечивающих основные свойства
- Возможность интеграции пользовательского кода репозитория
- Простая интеграция Spring через JavaConfig и пользовательские пространства имен XML
- Расширенная интеграция с контроллерами Spring MVC

Основные модули:



— Spring Data Commons - основные концепции Spring, лежащие в основе каждого модуля Spring Data.

— Spring Data JDBC - поддержка репозитория Spring Data для JDBC.

— Spring Data JDBC Ext - поддержка специфичных для базы данных расширений стандартного JDBC, включая поддержку быстрого аварийного переключения Oracle RAC, поддержку AQ JMS и поддержку использования расширенных типов данных.

— Spring Data JPA - поддержка репозитория Spring Data для JPA.

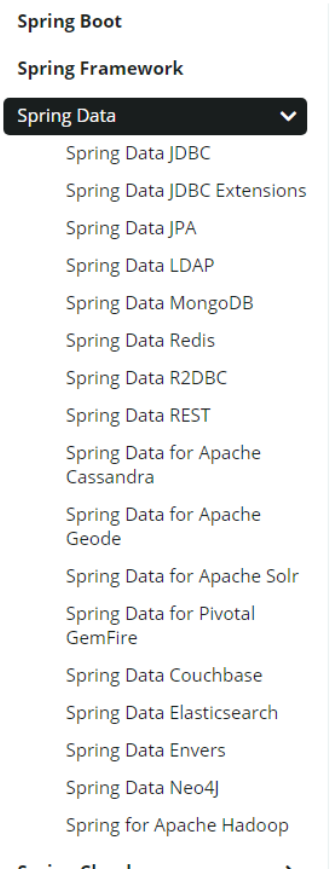
— Spring Data KeyValue - сопоставьте репозитории и SPI, чтобы легко создать модуль Spring Data для хранилищ значений ключей.

— Spring Data LDAP - поддержка репозитория Spring Data для Spring LDAP.

— Spring Data MongoDB - основанная на Spring поддержка объектов и документов и хранилища для MongoDB.

— Spring Data Redis - Простая настройка и доступ к Redis из приложений Spring.

— Spring Data REST - экспортирует репозитории Spring Data в виде управляемых гипермедиа ресурсов RESTful.



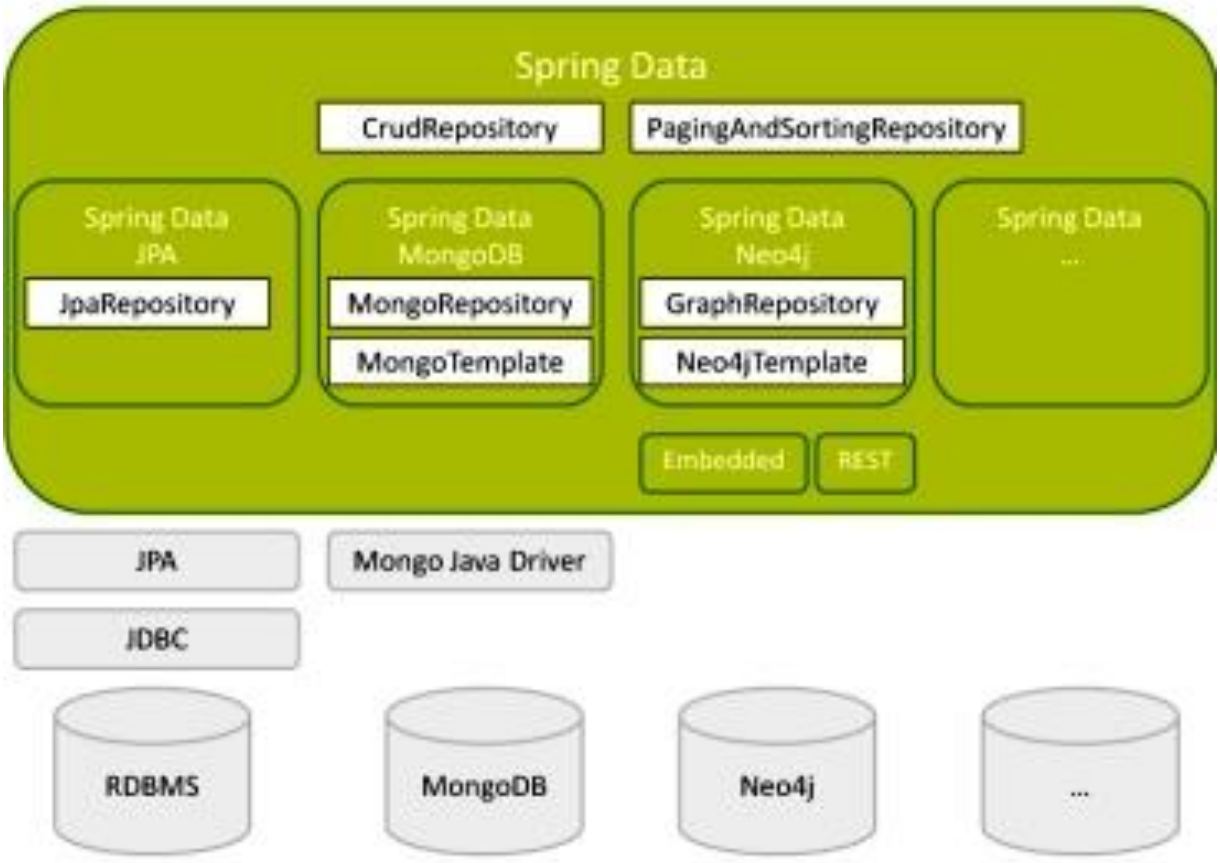
— Spring Data для Apache Cassandra - Простая настройка и доступ к Apache Cassandra или крупномасштабным, высокодоступным.

— Spring Data для Apache Geode - Простая настройка и доступ к Apache Geode для высоконадежных.

— Spring Data для Apache Solr - Простая настройка и доступ к Apache Solr.

— Spring Data для Pivotal GemFire - Простая настройка и доступ к Pivotal GemFire .

https://spring.io/projects/spring-data



ORM

ORM (Object-Relational Mapping или объектно-реляционное отображение) — технология для отображения объектов в структуры реляционных баз данных, чтобы представить java-объект в виде строки таблицы.

Таблица обычно соответствуют классу, строки таблицы - экземплярам этого класса, колонки таблицы ("реляционные атрибуты") при этом отображаются на атрибуты объекта или вызовы методов чтения/записи. Отображение двунаправленно: манипуляции с атрибутами объекта приводят к чтению информации из (и записи в) соответствующие таблицы базы данных.

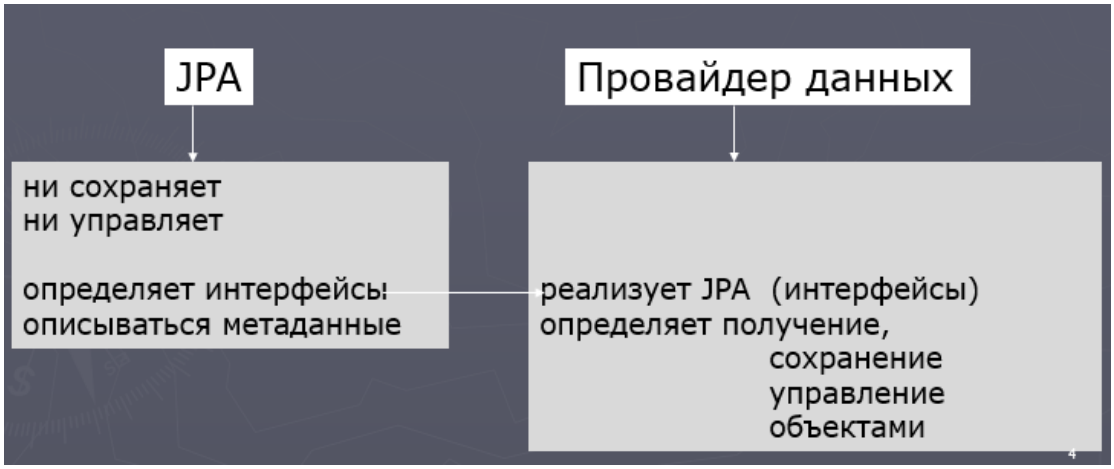
JPA

Java Persistence API, иногда называемый JPA, является платформой Java, управляющей реляционными данными в приложениях с использованием платформы Java Standard Edition (JavaSE) и платформы Java Enterprise Edition (JavaEE). JPA —спецификация, она описывает требования к объектам, в ней определены различные интерфейсы и аннотации для работы с БД.

JSR 338: Java™ Persistence 2.2
https://jcp.org/en/jsr/detail?id=338

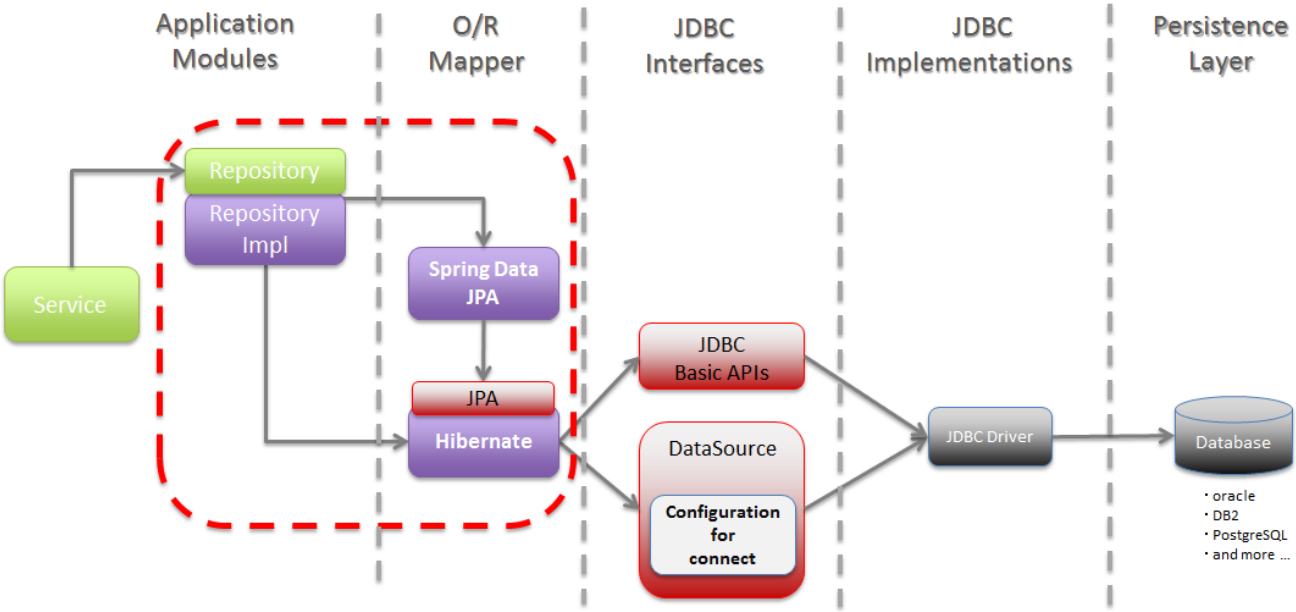
ORACLE

JSR 338: Java™ Persistence API, Version 2.2

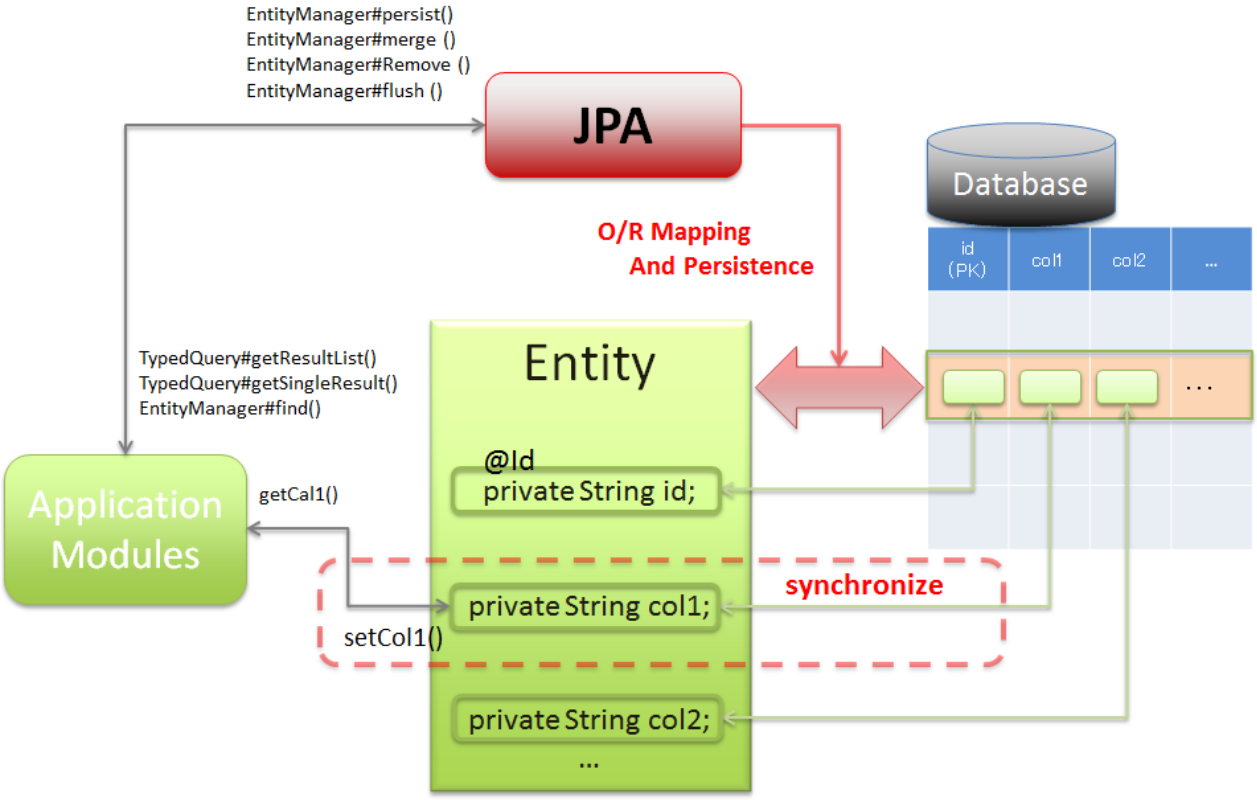


https://download.oracle.com/otndocs/jcp/persistence-2_2-mrel-eval-spec/index.html

JPA является стандартом. Поэтому есть множество конкретных реализаций, одной из которых является **Hibernate**.



Hibernate — реализация спецификации JPA, предназначенная для решения задач объектно-реляционного отображения (ORM).



Основной принцип действия библиотеки Hibernate опирается на интерфейс **Session**, которым управляет компонент **SessionFactory**, представляющий фабрику сеансов в Hibernate.

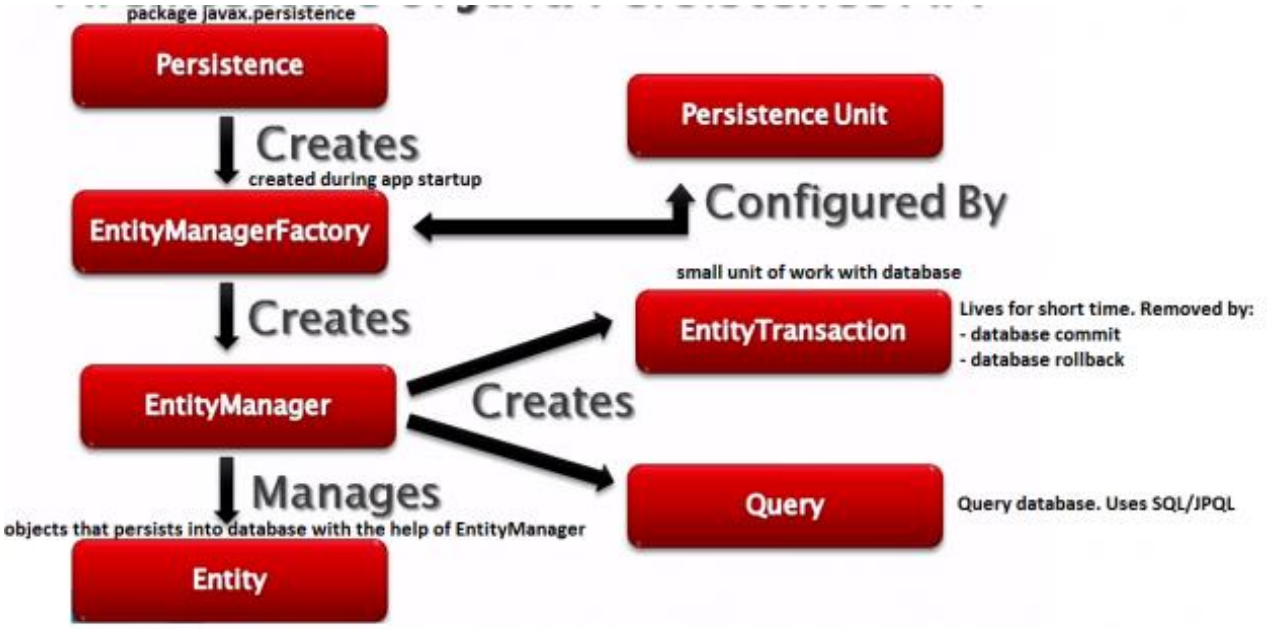
Для транзакционного доступа к данным в фабрике сеансов Hibernate требуется диспетчер транзакций. В каркасе Spring предоставляется диспетчер транзакций HibernateTransactionManager.

Для составления запросов в Hibernate служит язык запросов HQL (Hibernate Query Language). При взаимодействии с базой данных библиотека Hibernate преобразует запросы HQL в операторы SQL. Синтаксис языка HQL очень похож на синтаксис SQL.

Структура JPA 2.2

- API – интерфейсы javax.persistance
- Java Persistence Query Language JPQL – объектный язык запросов (запросы выполняются к объектам)
- Metadata – аннотации над объектами. (XML файл или аннотации)
- Транзакции и механизмы блокировки - Java Transaction API (JTA)
- Обратные вызовы и слушатели

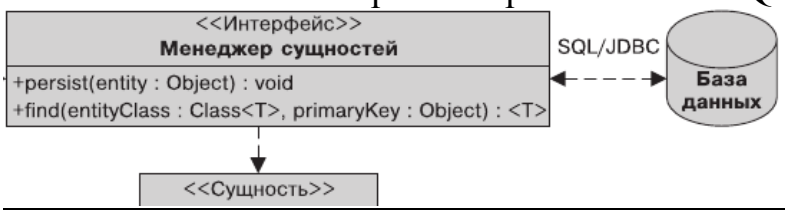
Архитектура JPA



В основу JPA положен интерфейс *EntityManager*, который происходит от фабрик типа *EntityManagerFactory*. Главное назначение интерфейса *EntityManager* - поддержка контекста сохраняемости, в котором будут храниться все экземпляры сущностей, управляемые этим контекстом. Конфигурация интерфейса *EntityManager* определяется как единица сохраняемости, и в приложении их может существовать много. Если применяется библиотека Hibernate, то контекст сохраняемости можно рассматривать таким же образом, как и интерфейс *Session*, а компонент типа *EntityManagerFactory* - как и компонент *SessionFactory*. В библиотеке Hibernate управляемые сущности сохраняются в сеансе, с которым можно взаимодействовать напрямую через компонент *SessionFactory* или интерфейс *Session*. Но в JPA нельзя непосредственно взаимодействовать с контекстом сохраняемости. Вместо этого для выполнения всей необходимой работы используется интерфейс *EntityManager*.

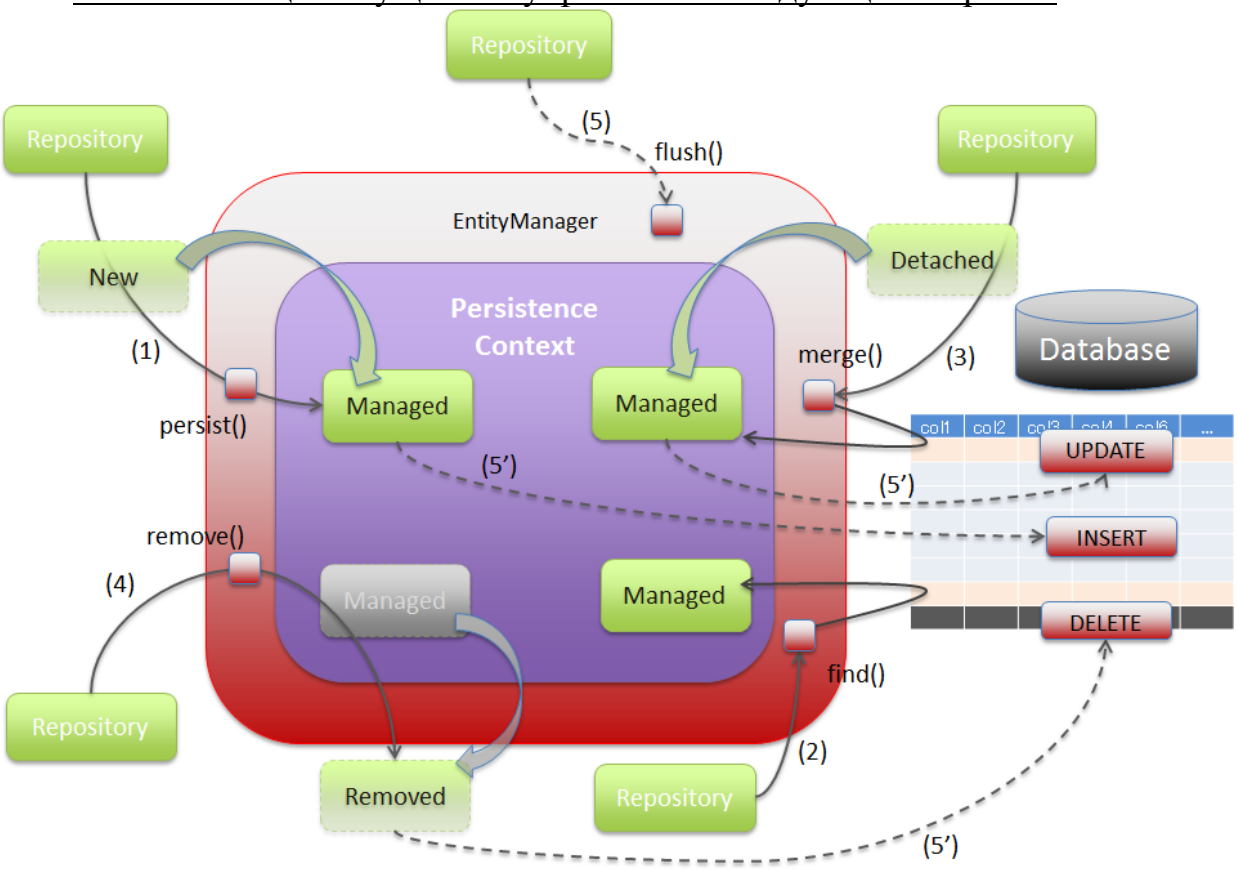
javax.persistence.EntityManager

- скрывает JDBC-вызовы базы данных и операторы SQL
- управление сущностями
- CRUD
- комплексные запросы с применением JPQL (ст. и динамич.)

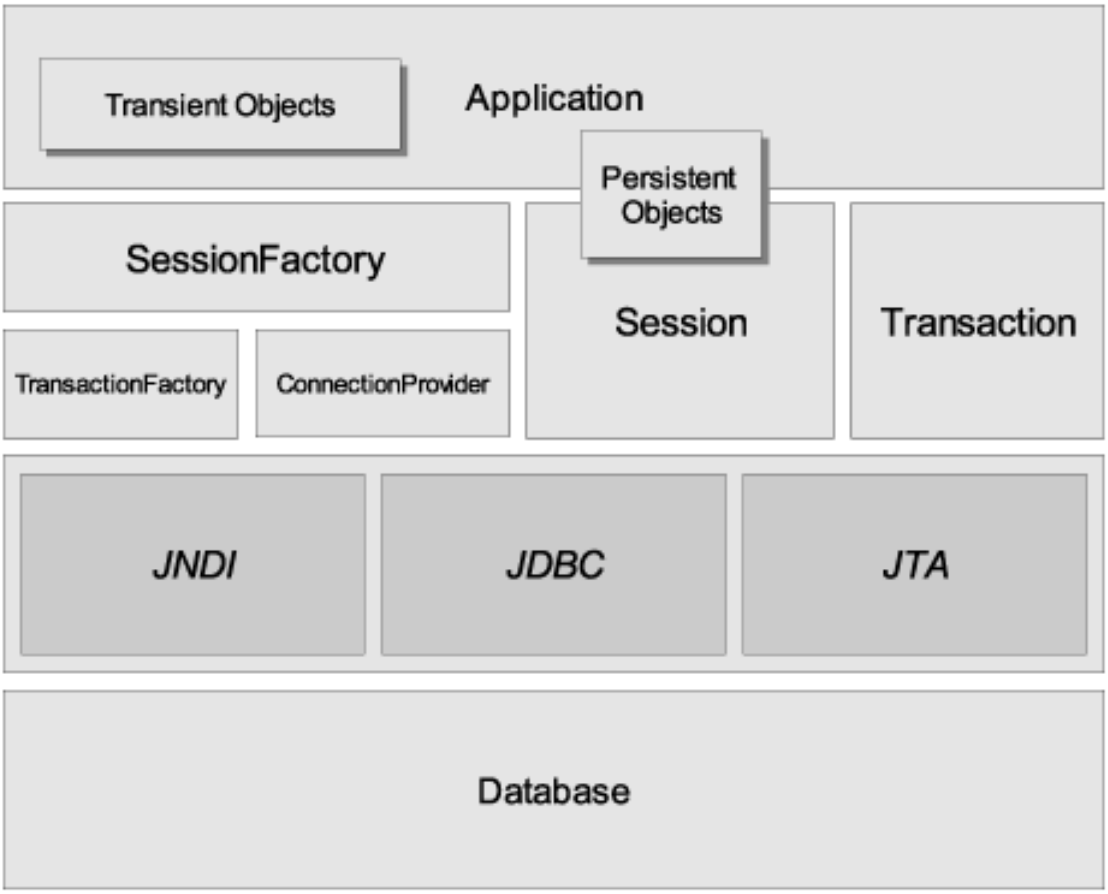



```
EntityManagerFactory emf =  Persistence.createEntityManagerFactory("UTR");
EntityManager em = emf.createEntityManager();
em.persist(card234);
```

Жизненный цикл сущности управляется следующим образом



Внутренняя архитектура Hibernate

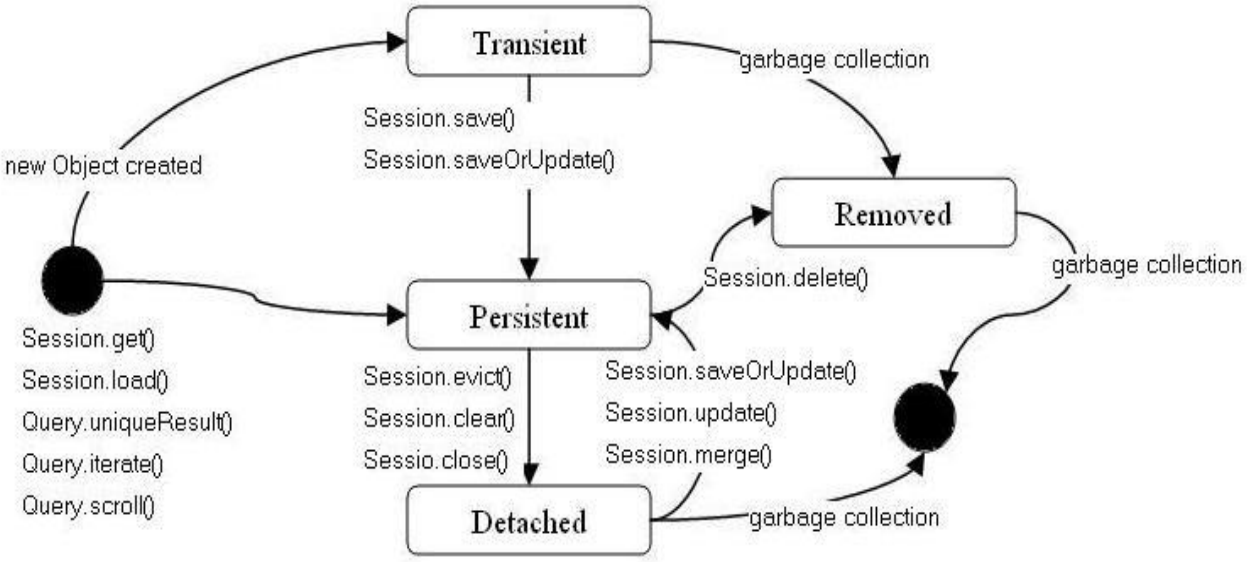


Переходные объекты (Transient Objects):

Экземпляры долгоживущих классов, которые в настоящее время не связаны с Сессией.

Постоянные объекты (Persistent objects):

Короткоживущие, однопоточные объекты, содержащие постоянное состояние и бизнес-функции.



SessionFactory:

Потокобезопасный, неизменный кэш скомпилированных отображений для одной базы данных.

Сессиея (Session):

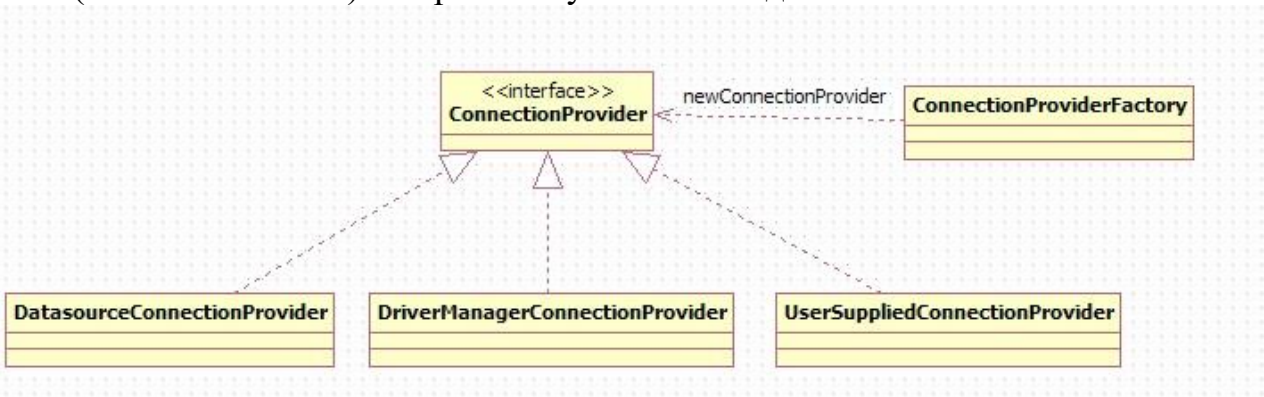
Однопоточный, короткоживущий объект, представляющий взаимодействие между приложением и постоянной памятью.

TransactionFactory:

(Необязательный) Фабрика для экземпляров Transaction.

ConnectionProvider:

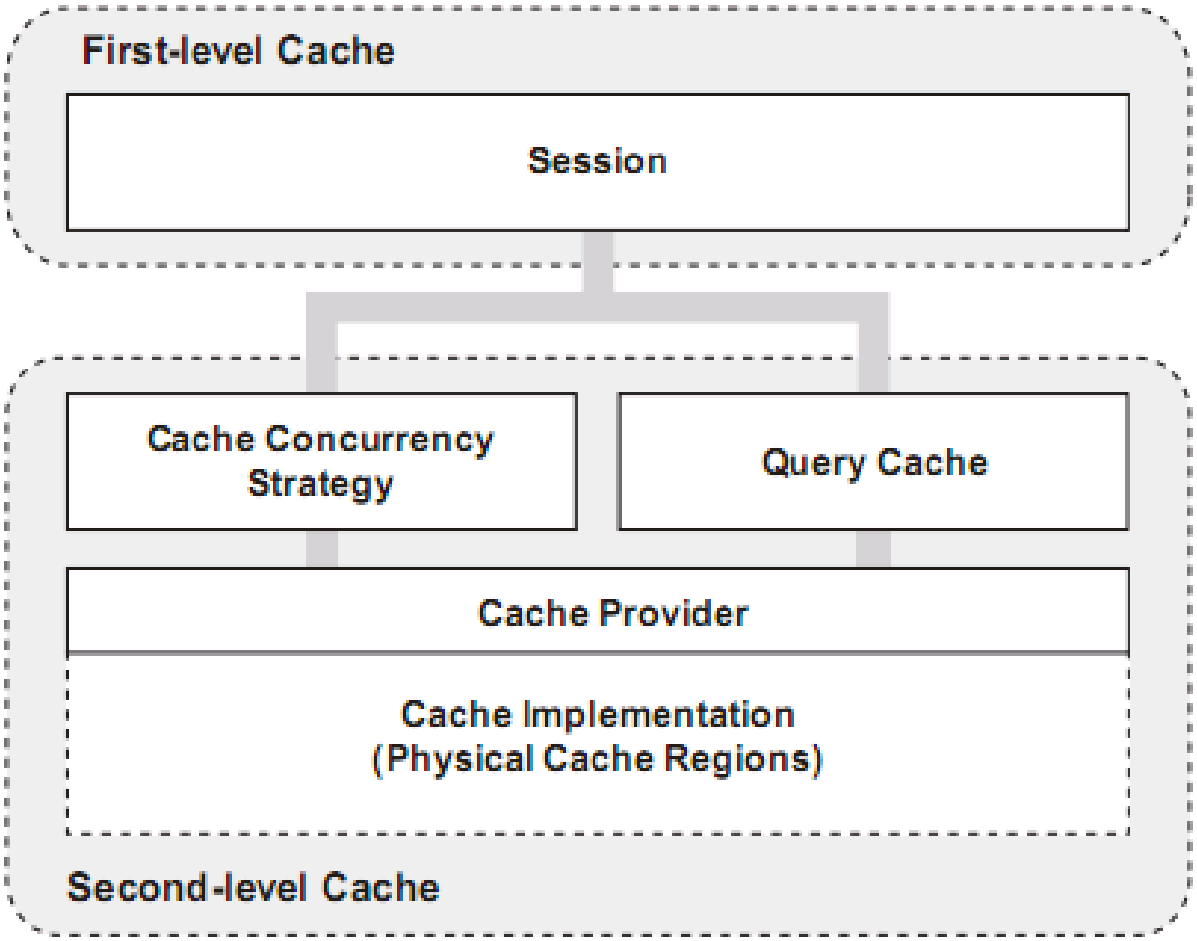
(Необязательный) Фабрика и пул JDBC соединений.



Трансакция (Transaction):

(Необязательный) Однопоточный, короткоживущий объект, используемый приложением для указания atomic переменных работы.

Архитектура КЭШа в Hibernate



Политика КЭШа второго уровня включает в себя настройку следующих параметров:

- Включен ли кэш второго уровня
- Стратегию параллелизма Hibernate
- Политика истекания срока кэширования (такую, как тайм-аут, LRU, зависимую от ОП)
- Физическое устройство КЭШа (в памяти, индексируемые файлы, кластерная репликация

Есть четыре встроенных стратегии параллелизма, представляющие снижение уровня строгости, в терминах изолированности транзакции:

- *Транзакционная* – доступна только в управляемой среде. Она гарантирует полную изоляцию транзакций до повторяемого чтения, если это требуется.
- *Чтение-запись* – поддерживает изоляцию чтения подтвержденного, используя механизм временных меток. Она доступна только в некластерных средах.
- *Нестрогое-чтение-запись* - не дает никакой гарантии согласованности между КЭШем и БД. Если есть возможность одновременного доступа к

одной сущности, то вам необходимо настроить достаточно короткий срок истечения тайм-аута.

— *Только-для-чтения* – данная стратегия подходит для данных, которые никогда не меняются. Используйте её только для справочных данных.

Entity

Если вы хотите чтобы объекты класса могли быть сохранены в базе данных, класс должен удовлетворять ряду условий. В JPA для этого есть такое понятие как **Сущность (Entity)**. Класс-сущность это обыкновенный *POJO* класс, с приватными полями и геттерами и сеттерами для них. У него обязательно должен быть не приватный конструктор без параметров (или конструктор по-умолчанию), и он должен иметь первичный ключ, т.е. то что будет однозначно идентифицировать каждую запись этого класса в БД. Сделать класс сущностью можно при помощи JPA аннотаций.

Annotations

Для конфигурирования любой новой сущности обязательными являются два действия: маркирование класса – сущности аннотацией @Entity, а также выделение поля, которое выступит в качестве ключевого. Такое поле необходимо маркировать аннотацией @Id.

@Entity — указывает на то, что данный класс является сущностью.

@Table — указывает на конкретную таблицу для отображения этой сущности.

@Id — указывает, что данное поле является первичным ключом, т.е. это свойство будет использоваться для идентификации каждой уникальной записи.

@Column — связывает поле со столбцом таблицы. Если имена поля и столбца таблицы совпадают, можно не указывать.

@GeneratedValue — свойство будет генерироваться автоматически, в скобках можно указать каким образом.

Если между сущностями существуют связи, то они тоже конфигурируется при помощи аннотаций уровня полей. Это аннотации @OneToMany, @ManyToOne и @ManyToMany. Соответственно для того, чтобы связать две сущности по некоторому полю, необходимо использовать соответствующие типу связи аннотации.

```
@Entity

public class Customer {
@Id
  private String firstName;
  private String lastName;
  private String email;
  private String phoneNumber;
  private String passort;
  private Date dateOfBirth;
  private Date expDate;

...
```

Мы можем генерировать идентификаторы различными способами, которые определяются аннотацией **@GeneratedValue**.

Можно выбрать одну из четырех стратегий генерации идентификаторов с элементом стратегии. Значение может быть AUTO, TABLE, SEQUENCE или IDENTITY.

```
@Data
@Entity
@Table(name = "person")
@AllArgsConstructor
@NoArgsConstructor
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;
    @Column(name="name")
    private String firstName;
…
```

В большинстве случаев имя таблицы в базе данных и имя объекта не будут совпадать. В этих случаях можно указать имя таблицы с помощью аннотации **@Table**: `@Table(name = "person")`

Как и аннотация @Table, можем использовать @Column, чтобы определить детали столбца в таблице.

Аннотация **@Column** имеет много элементов, таких как имя, длина, обнуляемый и уникальный.

```
@Column(name = "lastname", length=50, nullable=false,
unique=false)
private String lastName;
```

Если не указать @Table, имя поля будет считаться именем столбца в таблице.

@**Transient** (нерезидент) — отмеченные этим модификатором поля не записываются в поток байт при применении стандартного алгоритма сериализации.

```
@Transient
private Double elapsed;
```

Или

```
@Transient
private Integer age;
```

В некоторых случаях может потребоваться сохранить временные значения в нашей таблице. Для этого есть аннотация **@Temporal**:

```
@Temporal(TemporalType.DATE)
private Date birthDate;
```

Для сохранения типа перечисления Java можно использовать аннотацию **@Enumerated**, чтобы указать, следует ли сохранять перечисление по имени или по порядковому номеру (по умолчанию).

Аннотация @**Enumerated** – принимает параметр типа EnumType:

EnumType.STRING – это значит, что в базе будет хранится имя этого enum. То есть если мы зададим role = RoleEnum.ADMIN, то в БД в поле role будет хранится значение ADMIN.

EnumType.ORDINAL – это значит, что в базе будет хранится ID этого enum. ID – это место расположение в списке перечисления начиная с 0.

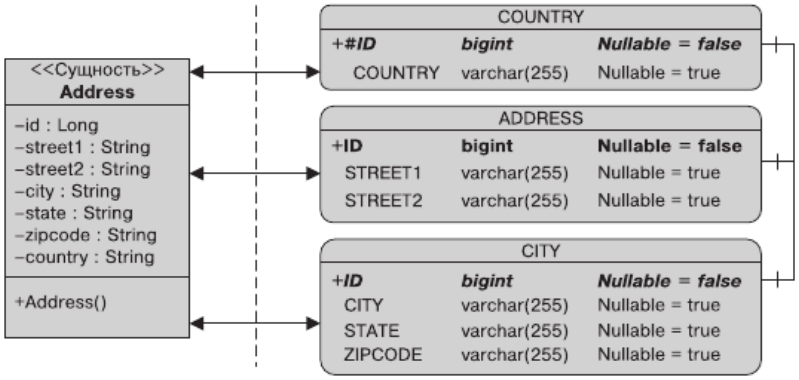
```
@Enumerated (EnumType.STRING)
@Column(name = "role")
private UserRole role;
```

Или

```
@Enumerated (EnumType.ORDINAL)
@Column(name = "role")
private UserRole role;
```

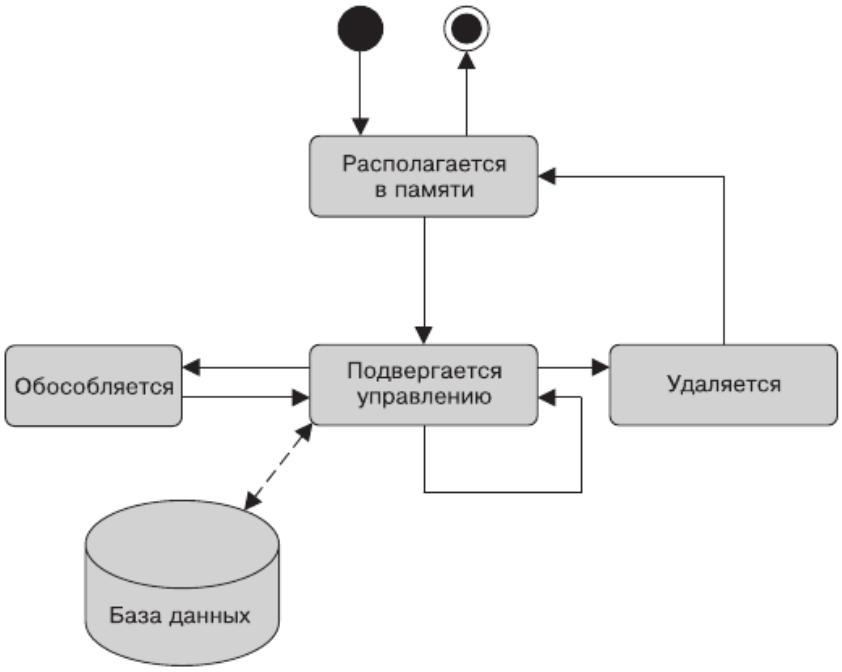
@**Transactional** служит для определения требований к транзакциям.

При отображении нескольких таблиц в одну сущность можно использовать **@SecondaryTables:**




```
@Entity
@SecondaryTables({
        @SecondaryTable(name = "city"),
        @SecondaryTable(name = "country")
})
public class Address {
    @Id
    private Long id;
    private String street1;
    private String street2;
    @Column(table = "city")
    private String city;
    @Column(table = "city")
    private String state;
    @Column(table = "city")
    private String zipcode;
    @Column(table = "country")
    private String country;}
```

Жизненный цикл Entity



Механизм обратных вызовов

В JPA предусмотрен несложный механизм обратных вызовов (*callbacks*) из EntityManager в те моменты, когда он меняет состояние сущностей.

Есть 4 типа callbacks, три из которых вызываются до и после изменения.

@PrePersist — вызывается как только инициирован вызов persist() и исполняется перед остальными действиями.

@PostPersist — вызывается когда сохранение в базу завершено и оператор INSERT выполнен.

@PreUpdate — вызывается перед сохранением изменений в сущности в базу.

@PostUpdate — вызывается, когда данные сущности в базе обновлены и оператор UPDATE выполнен.

@PreRemove — вызывается как только инициирован вызов remove() и исполняется перед остальными действиями.

@PostRemove — вызывается, когда операция удаления из базы завершено и оператор DELETE выполнен.

@PostLoad — вызывается после загрузки данных сущности из БД.

Например,

```
@PostLoad
public void postLoad() {
   estimate = tasks
         .stream()
         .mapToDouble(t -> t.getEstimate() != null ?
t.getEstimate() : 0)
         .sum();

   elapsed = tasks
         .stream()
         .mapToDouble(t -> t.getElapsed() != null ?
t.getElapsed() : 0)
         .sum();
}
```

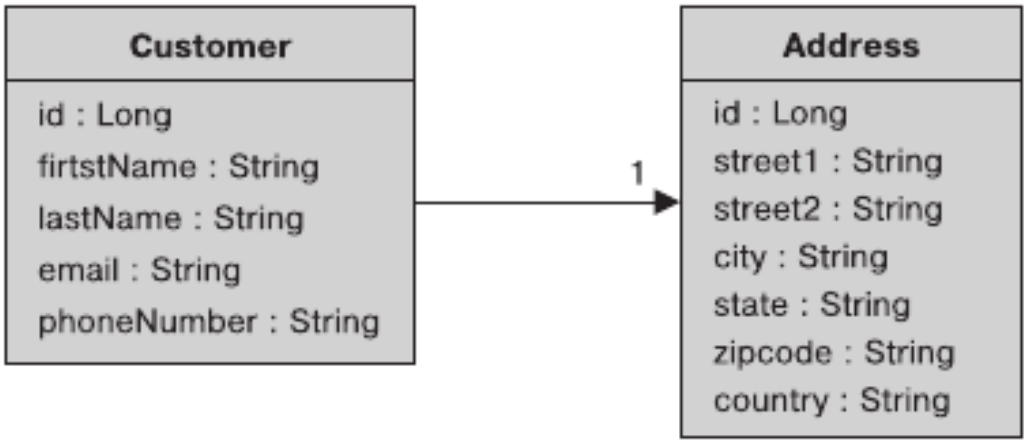
Преобразование

Классы Java сущностей можно преобразовать в исходную структуру реляционных данных двумя способами. Первый - сначала проектируется объектная модель, а затем на ее основе формируются скрипты для базы данных. Например, при **ddl – auto** автоматически экспортируется схема DDL в базу данных. Второй способ состоит в том, чтобы начать с модели данных, а затем построить объекты POJO.

Типы связей

В JPA для определения этих связей используются аннотации: @OneToOne @OneToMany @ManyToOne @ManyToMany Связи могут быть двунаправленными и однонаправленными. При двунаправленной связи оба класса содержат ссылки друг на друга, при однонаправленной — только один класс ссылается на другой. При двунаправленной связи необходимо указывать атрибут другого класса, владеющий связью с данными классом в виде @ManyToMany

@OneToOne




```
@Entity
public class Customer {
    @Id @GeneratedValue
    private Long id;
    private Address address;

}

@Entity
public class Address {
    @Id @GeneratedValue
    private Long id;

}
```

@OneToMany - указывает на наличие отношения «один ко многим». Этой аннотации передается несколько атрибутов. В атрибуте **mappedBy** задается свойство из класса, обеспечивающее связь. Атрибут **cascade** означает, что операция обновления должна распространяться «каскадом» на порожденные записи. Атрибут **orphanRemoval** указывает, что после обновления сведений, которые больше не существуют в наборе, они должны быть удалены из базы данных.

```
@Entity
@Table(name = "project")
public class Project {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(name = "id")
   private Long id;

   @OneToOne(mappedBy = "project", cascade = CascadeType.ALL, orphanRemoval =
true)
   private ProjectInfo projectInfo;

@OneToMany(mappedBy = "project")
  private List<Task> tasks;

  @Transient
  private Double estimate;

…
}
```

@ManyToOne - задает другую сторону для связи. Аннотация **@JoinColumn** определяет столбец с именем внешнего ключа.

```
@Entity
@Table(name = "task")
public class Task {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(name = "id")
   private Long id;

   @ManyToOne
   @JoinColumn(name = "project_id")
   private Project project;
    …}
```

@ManyToMany представляет связь многие ко многим через промежуточную таблицу. Аннотация **@JoinTable** используется для указания промежуточной таблицы для соединения. В атрибуте **name** задается имя промежуточной таблицы для соединения, в атрибуте **joinColumns** определяется столбец с внешним ключом, а в атрибуте **inverseJoinColumns** указывается столбец с внешним ключом на другой стороне устанавливаемой связи.

```
@ManyToMany
@JoinTable(
      name = "user_linked",
      joinColumns = @JoinColumn(name = "user_id"),
      inverseJoinColumns = @JoinColumn(name = "linkeduser_id")
)
private List<User> linkeduser;
```

Подробнее почитайте тут
https://www.baeldung.com/spring-data-annotations

Аудит

Аннотации на основе метаданных аудита: **@CreatedBy** и **@LastModifiedBy** для захвата пользователя, который создал или изменил объект, а также **@CreatedDate** и **@LastModifiedDate** для захвата, когда произошло изменение.

```
@CreatedBy
private User user;

@CreatedDate
private DateTime createdDate;
```

https://docs.spring.io/spring-data/jpa/docs/2.2.5.RELEASE/reference/html/#auditing

Определение Entity через XML (ORM)

```
        <?xml version="1.0" encoding="UTF-8"?>
        <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
3.0//EN"
                "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="students.entity">
<class name="Applicant" table="applicant">
    <id name="applicantId" column="applicant_id">
        <generator class="native"/>
    </id>
    <many-to-one name="profession" column="profession_id"
class="students.entity.Profession"/>
    <bag name="applicantResultList" inverse="true" cascade="all-delete-orphan">
        <key column="APPLICANT_ID"></key>
        <one-to-many class="ApplicantResult"/>
    </bag>
    <property name="firstName" column="first_name"/>
    <property name="lastName" column="last_name"/>
    <property name="middleName" column="middle_name"/>
    <property name="entranceYear" column="entrance_year"/>
</class>
</hibernate-mapping>
```

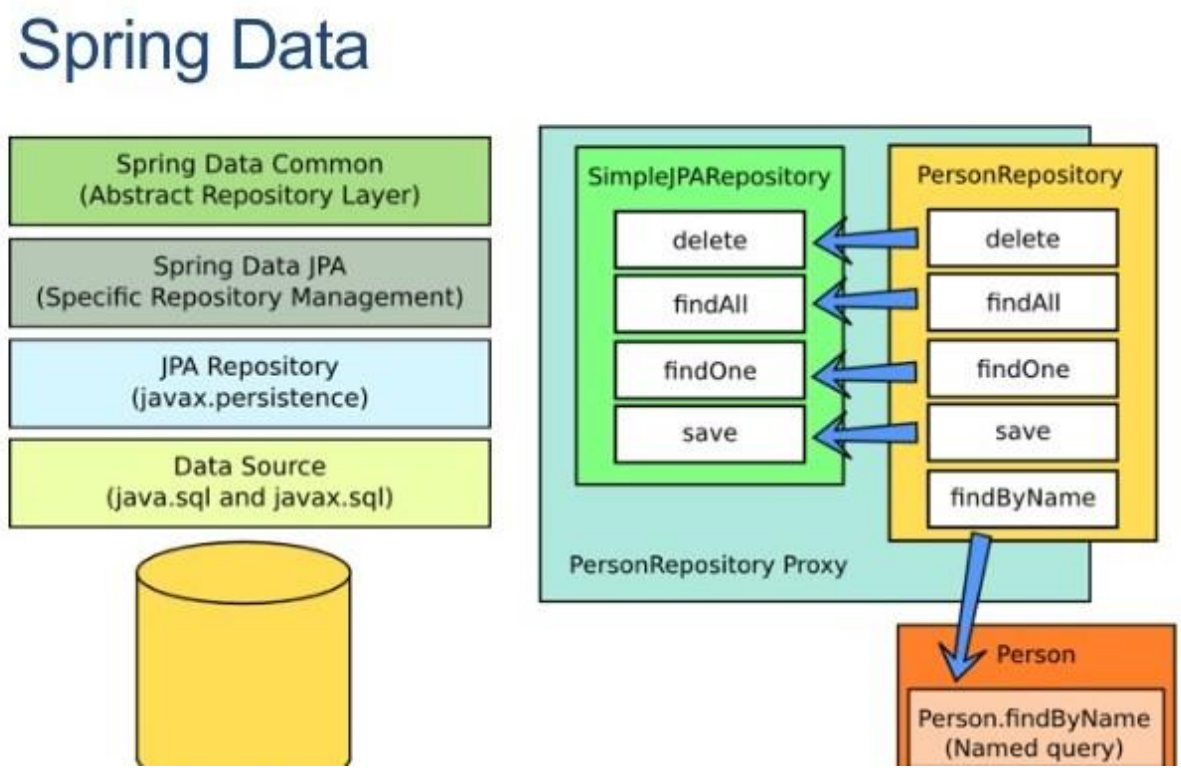
Объектно-реляционный отображение может описываться в виде XML документа:

```
<hibernate-mapping
         schema="schemaName"                                          (1)
         catalog="catalogName"                                        (2)
         default-cascade="cascade_style"                              (3)
         default-access="field|property|ClassName"                    (4)
         default-lazy="true|false"                                    (5)
         auto-import="true|false"                                     (6)
         package="package.name"                                       (7)
 />

<class
        name="ClassName"                                              (1)
        table="tableName"                                             (2)
        discriminator-value="discriminator_value"                     (3)
        mutable="true|false"                                          (4)
        schema="owner"                                                (5)
        catalog="catalog"                                             (6)
        proxy="ProxyInterface"                                        (7)
        dynamic-update="true|false"                                   (8)
        dynamic-insert="true|false"                                   (9)
        select-before-update="true|false"                             (10)
        polymorphism="implicit|explicit"                              (11)
        where="arbitrary sql where condition"                         (12)
        persister="PersisterClass"                                    (13)
        batch-size="N"                                                (14)
        optimistic-lock="none|version|dirty|all"                      (15)
        lazy="true|false"                                             (16)
        entity-name="EntityName"                                      (17)
        check="arbitrary sql check condition"                         (18)
        rowid="rowid"                                                 (19)
        subselect="SQL expression"                                    (20)
        abstract="true|false"                                         (21)
        node="element-name"
```

Spring Data JPA

Документация
https://spring.io/projects/spring-data-jpa#learn

Spring Data JPA не является поставщиком JPA. Это библиотека /фреймворк, которая добавляет дополнительный уровень абстракции поверх провайдера JPA. Если использовать Spring Data JPA, уровень репозитория приложения содержит три уровня:



Spring Data JPA обеспечивает поддержку для создания репозиториев JPA, расширяя интерфейсы репозитория Spring Data.

Spring Data Commons предоставляет инфраструктуру, совместно используемую конкретными проектами Spring Data.

Provider JPA реализует API персистентности Java.

Репозитории

Главными компонентами для взаимодействий с БД в Spring Data являются репозитории. Каждый репозиторий работает со своим классом-сущностью.

Аннотации @Repository и @Service, так же как и @Controller являются производными от @Component.

```
    @Repository
public interface PersonRepository extends CrudRepository<Person, Long> {

    List<Person> findAll();
    Optional<Person> findAllById(Long personaID);
}
```

Типы репозиториев

https://docs.spring.io/spring-data/jpa/docs/2.2.5.RELEASE/reference/html/#repositories

– **CrudRepository<T, ID extends Serializable>** , предоставляет базовый набор методов для доступа к данным. Данный интерфейс является универсальным и может быть использован не только в связке с JPA.

– **Repository <T, ID extends Serializable>** — базовый тип репозиториев, не содержит каких-либо методов, так же является универсальным. Определяется тип сущности и тип id сущности.

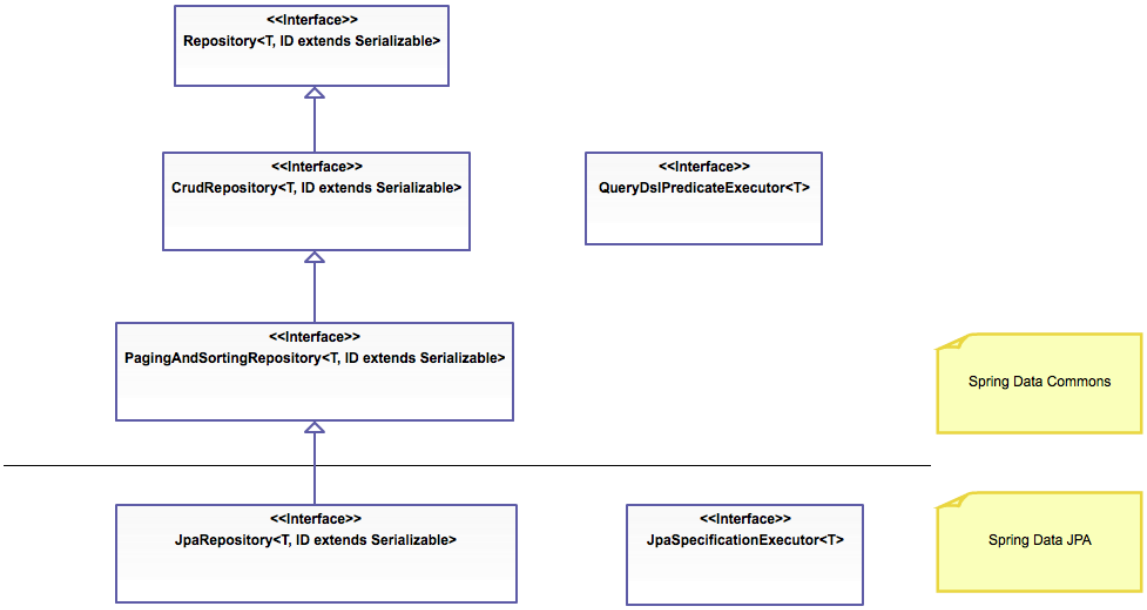
– **PagingAndSortingRepository <T, ID extends Serializable>** — универсальный интерфейс, расширяющий CrudRepository и добавляющий поддержку пагинации и сортировки.

– **JpaRepository <T, ID extends Serializable>** — репозиторий, добавляющий возможности, специфичные для JPA.

– **QueryDslJpaRepositor <T>** — реализация JpaRepository для взаимодействия с QueryDsl.

– **SimpleJpaRepository** — простая реализация JpaRepository.

Иерархия выглядит следующим образом:



Запросы

В сложных приложениях могут потребоваться специальные запросы, которые нельзя автоматически вывести средствами Spring. В таком случае запрос должен быть явно определен с помощью аннотации @**Query**.

Ааннотация @**Param** требуется для того, чтобы сообщить каркасу Spring, что значение данного параметра должно быть внедрено в именованный параметр запроса.

С помощью @**Query** мы можем предоставить реализацию **JPQL** для метода репозитория:

```
@Query("select p from Task p join p.user u join p.taskInfo i " +
"where u.userCredentials.login = :login and i.name = :name")

Optional<Task> findByLoginAndName(@Param("login") String login,
@Param("name") String name);
```

Кроме того, мы можем использовать **собственные запросы SQL**, если для аргумента nativeQuery задано значение true:

```
@Query(value = "SELECT AVG(p.age) FROM person p", nativeQuery =
true)
int getAverageAge();
```

Если мы пишем метод запроса, который должен возвращать более одного результата, то можно вернуть следующие типы:

`List <T>`

`Stream <T>` - метод запроса вернет поток, который можно использовать для доступа к результатам запроса, или пустой поток.

Если мы хотим, чтобы наш метод запроса выполнялся асинхронно, мы должны аннотировать его аннотацией @**Async** и возвращать объект **Future** <T>:

```
@Async
@Query("SELECT t.name FROM Users t where t.id = :id")
Future<String> findNameById(@Param("id") Long id);

@Async
@Query("SELECT t.name FROM Users t where t.id = :id")
Future<Optional<String>> findTitleById(@Param("id") Long id);
```

Аннотация @Query имеет следующие преимущества:

- Она поддерживает как JPQL, так и SQL.
- Вызванный запрос находится над методом запроса. Другими словами, легко узнать, что делает метод запроса.
- Не существует соглашения об именах для имен методов запросов.

Недостатки:

- Нет поддержки динамических запросов.
- Если мы используем запросы SQL, мы не можем изменить используемую базу данных, не проверив, что наши запросы SQL работают должным образом.

Прочитать по запросам можно тут https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation

Существуют также **именованные запросы**

```
@Entity
@NamedNativeQuery(name = "User.findByNameIs”,
    query="SELECT * FROM users t WHERE t.name = 'admin'",
    resultClass = User.class
)
@Table(name = "todos")
 class User {

}
```

JPA Criteria API

Интерфейс **JpaSpecificationExecutor** <T> объявляет методы, которые можно использовать для вызова запросов к базе данных, использующих API критериев JPA. Этот интерфейс имеет один параметр типа T, который описывает тип запрашиваемого объекта.

```
interface UserRepository extends Repository<User, Long>,
JpaSpecificationExecutor<User> {
}
```

Прочитать можно тут https://docs.oracle.com/javaee/6/tutorial/doc/gjitv.html

Создание методов запроса

Генерация запроса из имени метода - это стратегия генерации запроса, в которой вызванный запрос получен из имени метода запроса. Мы можем создавать методы запросов, которые используют эту стратегию, следуя этим правилам:

Имя метода запроса должно начинаться с одного из следующих префиксов: **find… By, read… By, query… By, count… By и get… By**.

Если надо ограничить количество возвращаемых результатов запроса, мы можем добавить ключевое слово **First** или **Top** перед первым словом **By**.

Если хотим выбрать уникальные результаты, то должны добавить ключевое слово **Distinct** перед первым словом. Например, findTitleDistinctBy

или findDistinctTitleBy означает, что мы хотим выбрать все уникальные заголовки, найденные в базе данных.

Keyword	**Sample**	**JPQL Snippet**
And	findByLastnameAndFirstname	...where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	...where x.lastname = ?1 or x.firstname = ?2
Is, Equals	findByFirstnameEquals	...where x.firstname = ?1
Between	findByStartDateBetween	...where x.startDate between ?1 and ?
LessThan	findByAgeLessThan	...where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	...where x.age <= ?1< td>
GreaterThan	findByAgeGreaterThan	...where x.age > ?1
GreaterThanEqual	findByAgeGreaterThanEqual	...where x.age >= ?1
After	findByStartDateAfter	...where x.startDate > ?1
Before	findByStartDateBefore	...where x.startDate < ?1
IsNull	findByAgeIsNull	...where x.age is null
IsNotNull, NotNull	findByAge(Is)NotNull	...where x.age not null
Like	findByFirstnameLike	...where x.firstname like ?1
NotLike	findByFirstnameNotLike	...where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	...where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	...where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	...where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	...where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	...where x.lastname <> ?1
In	findByAgeIn(Collection ages)	...where x.age in ?1
NotIn	findByAgeNotIn(Collection ages)	...where x.age not in ?1
True	findByActiveTrue()	...where x.active = true
False	findByActiveFalse()	...where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	...where UPPER(x.firstame) = UPPER(?1)

```
    public List<User> findFirst3ByNameOrderByGroupAsc(String name);
     public List<User> findTop3ByNameOrderByNameAsc(String name);
}
```

Преимущества:

- Создание простых запросов выполняется быстро.
- Имя метода а запроса описывает выбранные значения и используемые условия поиска.

Недостатки:

- Функции синтаксического анализатора имени метода определяют, какие запросы можно создавать. Если анализатор имени метода не поддерживает обязательное ключевое слово, не можем использовать эту стратегию.
- Имена методов сложных методов запросов получаются очень длинные.
- Нет поддержки динамических запросов.

Хранимые процедуры

```
@NamedStoredProcedureQueries({
        @NamedStoredProcedureQuery(
                name = "count_by_name",
                procedureName = "person.count_by_name",
                parameters = {
                        @StoredProcedureParameter(
                                mode = ParameterMode.IN,
                                name = "name",
                                type = String.class),
                        @StoredProcedureParameter(
                                mode = ParameterMode.OUT,
                                name = "count",
                                type = Long.class)
                }
        )
})
```

После этого мы можем обратиться к нему:

```
@Procedure(name = "count_by_name")
long getCountByName(@Param("name") String name);
```

Пагинация результатов запроса

В Spring Data Commons есть поддержка статической сортировки, которая описывается прямо в имени метода:

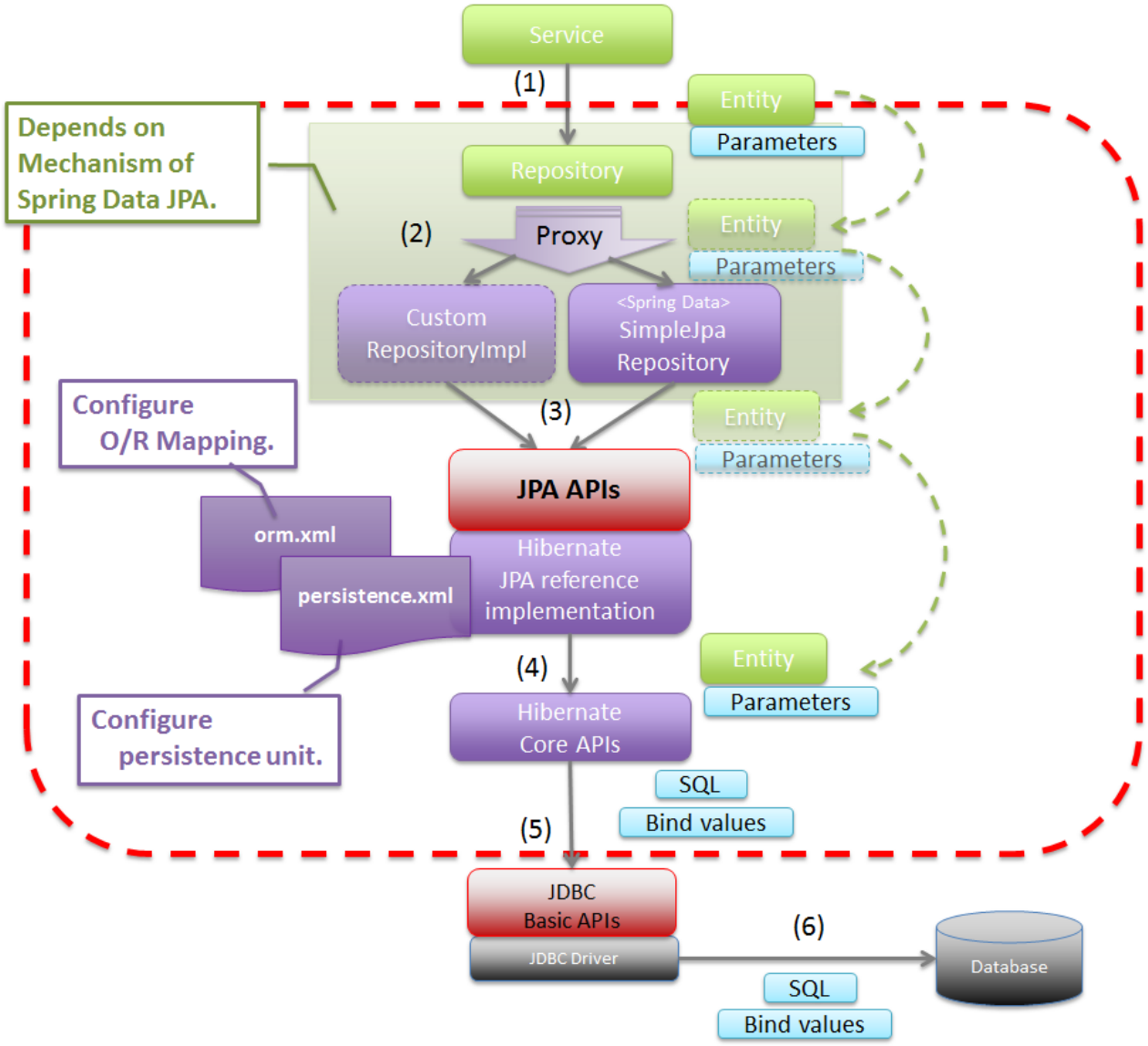
Iterable findByFirstNameOrderByLastNameAsc(String firstName),

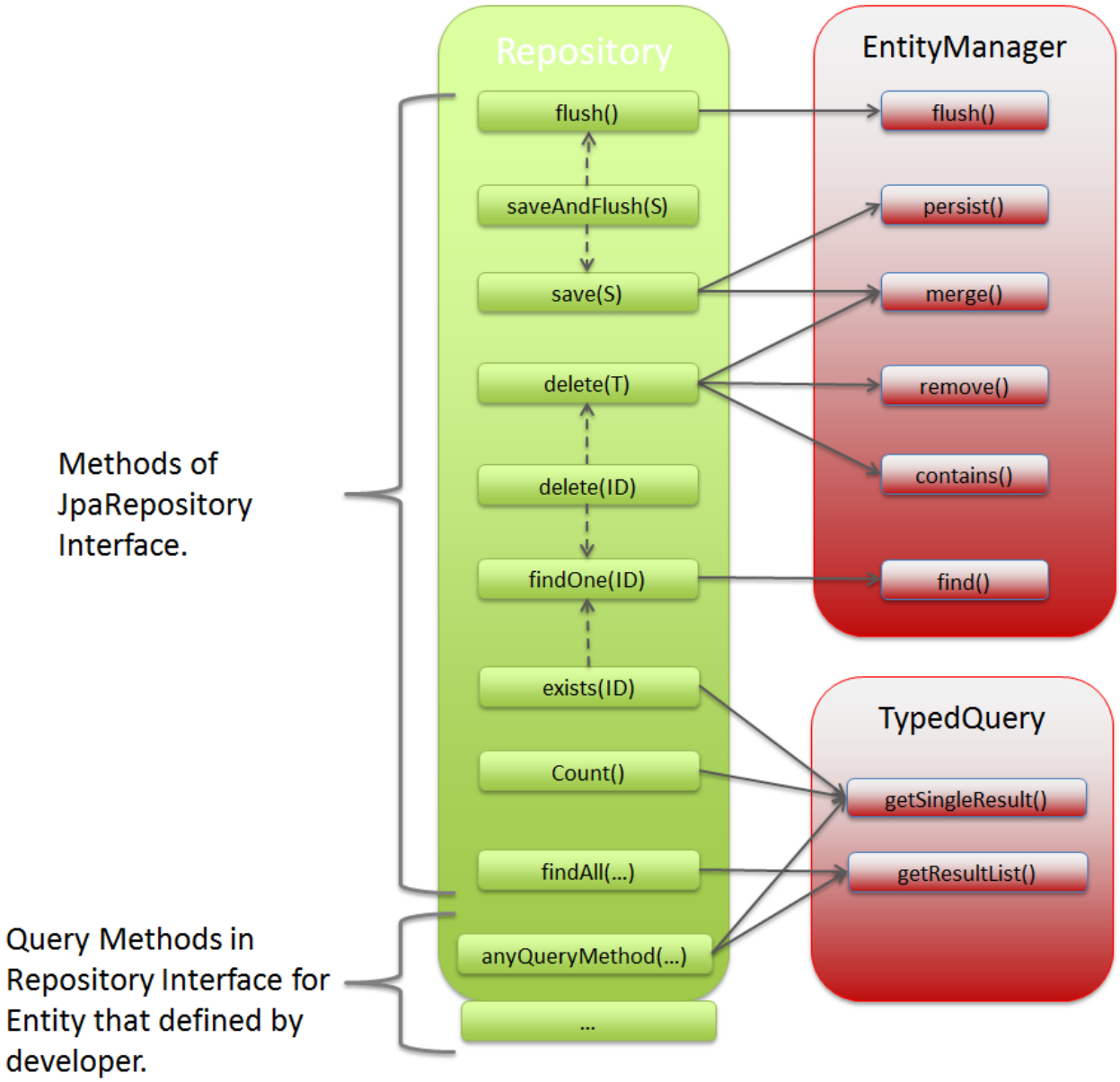
Динамическая сортировка реализована с помощью класса **Sort**, который инкапсулирует описание сортировки.

```
Sort lastNameDOBSort = new Sort(Sort.Direction.Asc, "lastName", "DOB");
```

```
Pageable pageable = PageRequest.of(0, 5, Sort.by(
    User.asc("name"),
    User.desc("id")));


@RestController
@RequiredArgsConstructor
class PagedController {

    private final MovieCharacterRepository characterRepository;

    @GetMapping(path = "/characters/page")
    Page<MovieCharacter> loadCharactersPage(Pageable pageable) {
        return characterRepository.findAllPage(pageable);
    }

}
```

Итого





Конфигурирование Spring JPA с Hibernate

Для реализации сохранения, которая использует Spring Data JPA, нам нужны следующие компоненты: драйвер JDBC (обеспечивает реализацию JDBC API для конкретной базы данных), источник данных (обеспечивает соединения с базой данных, используем источник данных HikariCP), поставщик JPA (реализует API персистентности Java - Hibernate).

Spring Data JPA скрывает используемый JPA-провайдер за абстракцией репозитория.

Если не использовать Spring Boot, то для H2 базе данных нужны были такие зависимости.

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```

```
<!-- DataSource (HikariCP) -->
<dependency>
    <groupId>com.zaxxer</groupId>
    <artifactId>HikariCP</artifactId>
</dependency>

<!-- JPA Provider (Hibernate) -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
</dependency>

<!-- Spring Data JPA -->
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
</dependency>
```

1) Можно создать класс конфигурации, который настраивает уровень персистентности приложения Spring и делает следующее:
   - создает файл свойств, который содержит свойства конфигурации контекста приложения.
   - настраивает компонент источника данных.
   - конфигурирует фабричный компонент диспетчера сущностей.
   - настраивает компонент управления транзакциями.
   - включает управление транзакциями на основе аннотаций.
   - настраивает Spring Data JPA.

   Нужно создать класс конфигурации - PersistenceContext:

```
@Configuration
class PersistenceContext {

    //Configure the required beans here
}
```

2) Файл application.properties (db. properties) содержит конфигурацию, которая используется для настройки приложения. Можно использовать его ли создать новый (но нужно указать на этот файл в конфигурациях)

```
#Database Configuration
db.driver=org.h2.Driver
db.url=jdbc:h2:mem:datajpa
db.username=sa
db.password=

#Hibernate Configuration
hibernate.dialect=org.hibernate.dialect.H2Dialect
hibernate.hbm2ddl.auto=create-drop
hibernate.ejb.naming_strategy=org.hibernate.cfg.ImprovedNamingStrategy
hibernate.show_sql=false
hibernate.format_sql=true
```

3) Конфигурируем компонент источника данных.

```
@Configuration
class PersistenceContext {

    @Bean(destroyMethod = "close")
    DataSource dataSource(Environment env) {
        HikariConfig dataSourceConfig = new HikariConfig();

dataSourceConfig.setDriverClassName(env.getRequiredProperty("db.driver"));
        dataSourceConfig.setJdbcUrl(env.getRequiredProperty("db.url"));
        dataSourceConfig.setUsername(env.getRequiredProperty("db.username"));
        dataSourceConfig.setPassword(env.getRequiredProperty("db.password"));

        return new HikariDataSource(dataSourceConfig);
    }

    //Add the other beans here
}
```

4) Конфигурируем фабрику Entity Manager

```
@Bean
LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource
dataSource,
                                                            Environment env) {
    LocalContainerEntityManagerFactoryBean entityManagerFactoryBean = new
LocalContainerEntityManagerFactoryBean();
    entityManagerFactoryBean.setDataSource(dataSource);
    entityManagerFactoryBean.setJpaVendorAdapter(new
HibernateJpaVendorAdapter());
    entityManagerFactoryBean.setPackagesToScan("by.patsei");

    Properties jpaProperties = new Properties();
    jpaProperties.put("hibernate.dialect",
env.getRequiredProperty("hibernate.dialect"));
    jpaProperties.put("hibernate.hbm2ddl.auto",
        env.getRequiredProperty("hibernate.hbm2ddl.auto")
    );
    jpaProperties.put("hibernate.ejb.naming_strategy",
        env.getRequiredProperty("hibernate.ejb.naming_strategy")
    );
    jpaProperties.put("hibernate.show_sql",
        env.getRequiredProperty("hibernate.show_sql")
    );
    jpaProperties.put("hibernate.format_sql",
        env.getRequiredProperty("hibernate.format_sql")
    );

    entityManagerFactoryBean.setJpaProperties(jpaProperties);

    return entityManagerFactoryBean;
}
```

5) Конфигурирование компонента управления транзакциями.

```
@Bean
JpaTransactionManager transactionManager(EntityManagerFactory
entityManagerFactory) {
    JpaTransactionManager transactionManager = new JpaTransactionManager();
    transactionManager.setEntityManagerFactory(entityManagerFactory);
    return transactionManager;
}
```

6) Включение управления транзакциями на основе аннотаций.

```
@Configuration
@EnableTransactionManagement
class PersistenceContext {
    .... }
```

7) Конфигурирование Spring Data JPA.

```
@Configuration
@EnableJpaRepositories(basePackages = {
    "by.patsei.repository"})
@EnableTransactionManagement
class PersistenceContext {
       ...}
```