Тема №5 JPA, Entity. Шаблон Service, Repository, Controller

https://www.baeldung.com/the-persistence-layer-with-spring-and-jpa

5.1 ORM, JPA, Entity

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. Благодаря ORM можно не заботиться о написании SQL-скриптов и сосредоточиться на работе с объектами.

JPA (Java Persistence API) - реализация ORM концепции. JPA — спецификация, она описывает требования к объектам, в ней определены различные интерфейсы и аннотации для работы с БД. 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).

Между *Hibernate* и JPA сложились тесные отношения. Начиная с версии 3.2 в Hibernate предоставляется реализация прикладного интерфейса *JPA*. Это означает, что при разработке приложений с помощью библиотеки *Hibemate* в качестве поставщика услуг сохраняемости данных можно выбирать между собственным прикладным интерфейсом API библиотеки Hibernate и прикладным JPA API с поддержкой Hibernate.

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.

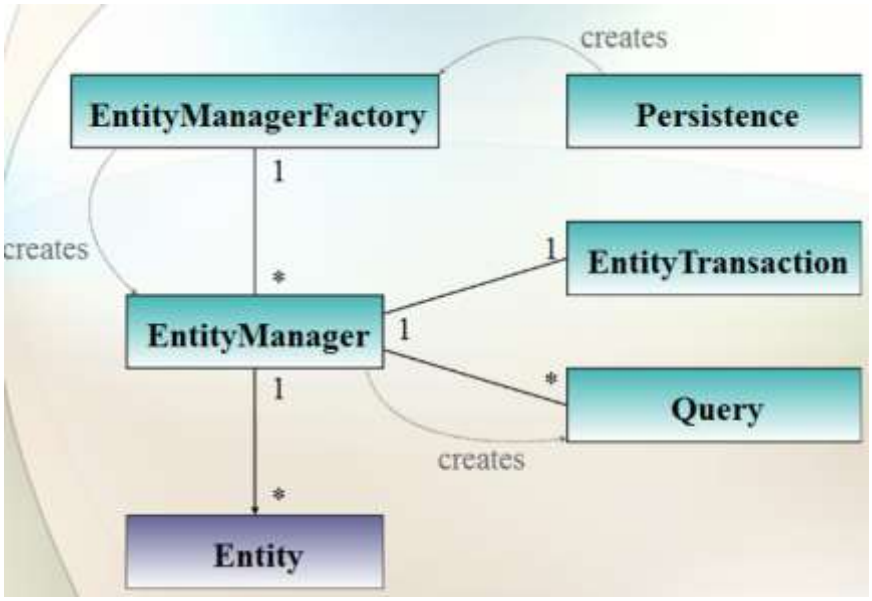
Для составления запросов в Hibernate служит язык запросов HQL (Hibernate Query Language). При взаимодействии с базой данных библиотека Hibernate преобразует запросы HQL в операторы SQL. Синтаксис языка HQL очень похож на синтаксис SQL.

Структура JPA 2.1

- API – интерфейсы javax.persistance
- 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*.

Каркасу Spring предписывается просмотреть интерфейсы информационного хранилища, а также передать компонент типа *EntityManagerFactory* и диспетчера транзакций соответственно.

По умолчанию репозитории Spring Data JPA являются компонентами Spring. Они являются singleton и легко инициализируются. Во время запуска они уже взаимодействуют с JPA *EntityManager* для проверки и анализа метаданных. Spring Framework поддерживает инициализацию JPA *EntityManagerFactory* в фоновом потоке, поскольку этот процесс обычно занимает значительное время запуска..

Начиная с Spring Data JPA 2.1 вы можете настроить BootstrapMode (либо через аннотацию @**EnableJpaRepositories**, либо через пространство имен XML), которая принимает следующие значения:

— **DEFAULT** (по умолчанию) - хранилища создаются с энтузиазмом, если явно не указано @Lazy. Лазификация действует только в том случае, если ни одному клиентскому компоненту не требуется экземпляр репозитория, поскольку для этого потребуется инициализация компонента репозитория.

— **LAZY** - неявно объявляет все bean-компоненты репозитория ленивыми. Это означает, что репозитории не будут созданы, если клиентский бин просто хранит экземпляр и не использует репозиторий во время инициализации. Экземпляры репозитория будут инициализированы и проверены при первом взаимодействии с репозиторием.

— **DEFERRED** - в основном тот же режим работы, что и в LAZY, но инициирующий инициализацию репозитория в ответ на ContextRefreshedEvent.

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.

Spring Data 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.

Можно для каждого свойства еще дополнительно указать много чего еще, например, что должно быть не нулевое, или уникальное, указать значение по-умолчанию, максимальный размер и т.д. Это пригодится если нужно сгенерировать таблицу на основании этого класса, с Hibernate есть такая возможность.

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.

Произведя отображение сущностей, ими можно управлять посредством JPA. Вы можете обеспечить постоянство сущности в базе данных, удалить ее, а также выполнять запросы к этой сущности с использованием языка запросов Java Persistence Query Language, или JPQL. Объектно-реляционное отображение позволяет вам манипулировать сущностями при доступе к базе данных «за кадром».

Итак, как было сказано ранее чтобы объявить класс POJO сущностью, его нужно отметить аннотацией **@Entity**:

```
@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;

...
```

Это позволит поставщику постоянства признать его постоянным классом, а не простым POJO. Кроме того, аннотация @javax.persistence.Id будет определять уникальный идентификатор этого объекта. Поскольку JPA используется для отображения объектов в реляционные таблицы, объектам необходим идентификатор, который будет отображен в первичный ключ.

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;
…
```

Если мы укажем GenerationType.AUTO, провайдер JPA будет использовать любую стратегию.

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

Как и аннотация @Table, мы можем использовать аннотацию @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 укажем @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. При десериализации объекта такие поля инициализируются значением по умолчанию.

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

Или

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

Существует ряд ситуаций, в которых необходимо использовать данный модификатор. Значение поля класса может быть вычислено после десериализации на основании значений остальных полей. Примером является объект, который кэширует результаты внутренних вычислений. Значение поля корректно только в рамках текущего контекста. Некоторые поля могут не сериализоваться из соображений безопасности, например, поле password некоторого класса.

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 можем 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.

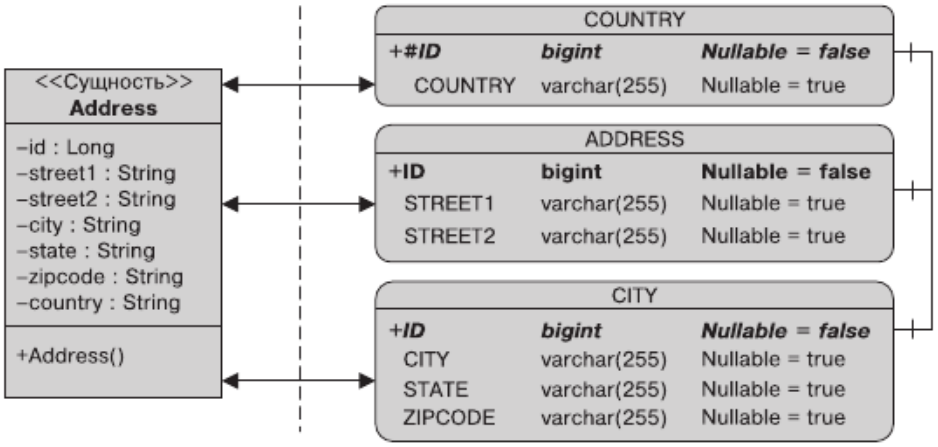
```
@Enumerated (EnumType.STRING)
@Column(name = "role")
private UserRole role;
```

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;}
```

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 информационных системах сущности обычно связаны друг с другом различными типами связей. JPA также определяет связи между классами, если один класс объявляется в качестве поля другого класса, т. е. является его частью. Классы могут соотноситься как «один к одному», «один ко многим»,

«многие к одному» и «многие ко многим». 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 @ManyToMan

@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.

```
@OneToMany(mappedBy = "project", cascade = CascadeType.ALL,
orphanRemoval = true)
private List<Role> roles;
```

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

```
@ManyToOne
@JoinColumn(name = "user_id")
private User user;
```

@ManyToMany представляет связь многие ко многим черз 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_follower",
      joinColumns = @JoinColumn(name = "user_id"),
      inverseJoinColumns = @JoinColumn(name = "follower_id")
)
private List<User> followed;
```

Механизм обратных вызовов

В JPA предусмотрен несложный механизм обратных вызовов (*callbacks*) из EntityManager в те моменты, когда он меняет состояние сущностей. Это позволяет управлять процессом перехода сущности из состояния в состояние и реализовывать дополнительный функционал, например проверку уровней доступа, учёт обращений, ведение истории изменений и так далее. Идеологически механизм JPA *callbacks* похож на механизм событий в Hibernate. *Callback* может быть определён как в классе сущности, так и в отдельном классе, связанным с классом сущности.

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.

Стоит отметить, что помеченные @Pre функции вызываются всегда, когда вызывается метод, инициирующий изменение состояния сущности. В то время как @Post вызываются только тогда, когда операция уже была выполнена. И если операция не выполняется, то @Post метод не будет вызван. Например, если вы изменяете значение нескольких полей в сущности, а потом её удаляете вызовом remove(), то будет @PreUpdate метод будет вызван несколько раз, @PreRemove и @PostRemove будут вызываны по одному разу и @PostUpdate может быть не вызыван ни разу.

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();
}
```

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 Project p join p.user u join p.projectInfo
i " +  "where u.userCredentials.login = :login and i.name =
:name")

Optional<Project> 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();
```

С помощью Spring Data JPA мы можем вызывать хранимые процедуры из репозиториев. Нужно объявить хранилище для класса сущностей, используя стандартные аннотации JPA:

```
@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);
```

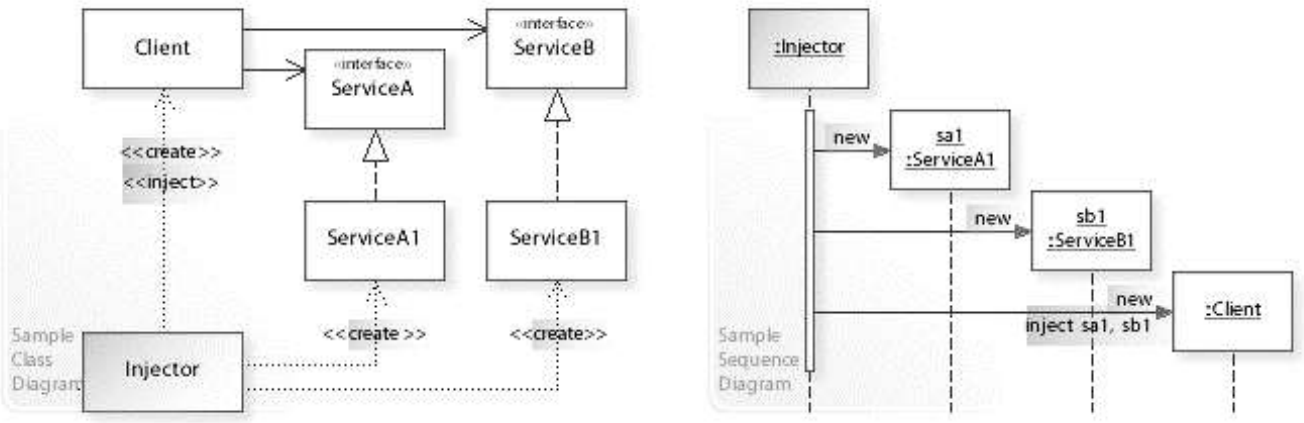
Подробнее почитайте тут
https://www.baeldung.com/spring-data-annotations

https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation

5.1 Шаблон Service, Repository, Controller

Dependency Injection

Паттерн «инверсия управления» (Inversion of Control - IoC), а конкретнее паттерн «внедрения зависимости» (Dependency Injection - DI).

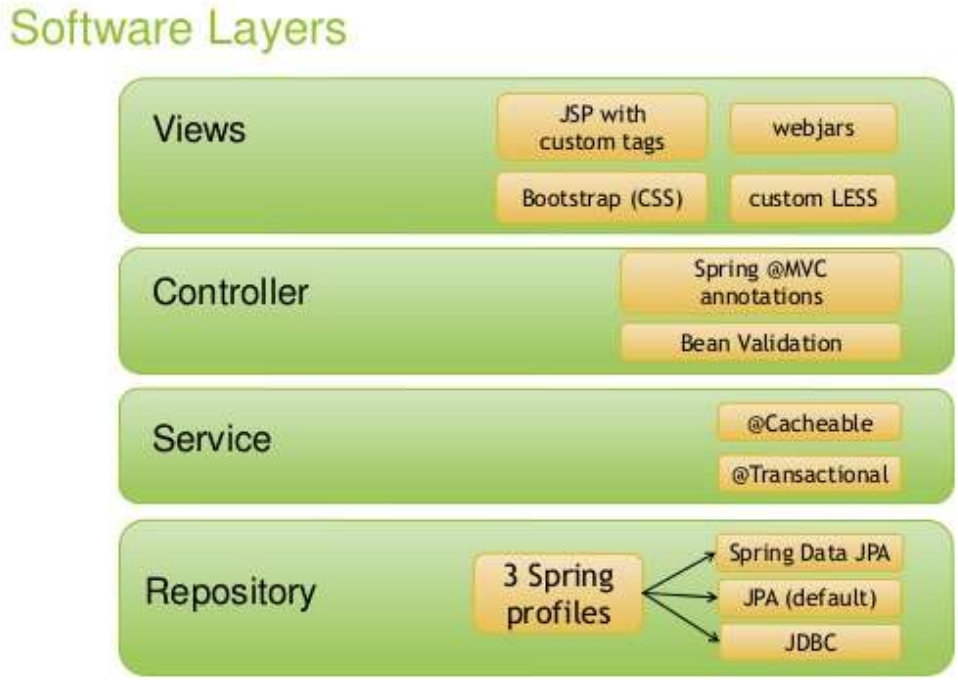


Паттерн используется для связи друг с другом контроллеров, сервисов и репозиториев. Рассмотрим пример обращения клиента (фронтенд) к серверу (бэкенд):

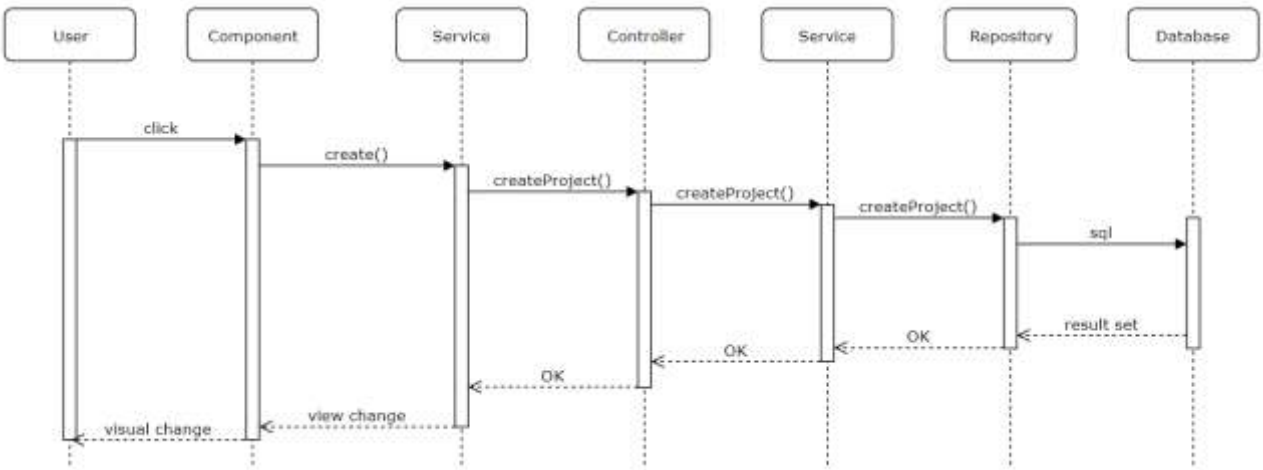
1) Запрос клиента имеет определенный путь, который должен совпадать с конкретным и единственным сервлетом сервера. Сервлет – интерфейс Java, выполняющий роль сервера для единственного вида запросов. Для проверки совпадения и передачи управления сервер сверяет путь запроса с сервлетами из контейнера сервлетов
2) Контейнер сервлетов вызывает метод контроллера, сверяя тело запроса, параметры, заголовки и возвращаемый тип. Контроллер, произведя базовые преобразования типов передает управления сервису(-ам)

3) Сервис в свою очередь выполняет всю бизнес-логику приложения. В зависимости от поставленной задачи он может производить вычисления и/или передавать выполнения репозиторию(-ям)
4) Репозиторий является прослойкой между бизнес-логикой приложения и источником данных (сервером СУБД в нашем случае). Он преобразует программные задачи в SQL-запросы, выполняемые сервером СУБД
5) После выполнения всей иерархии компонентов (все классы, которые могут выступать в роли инжектора в паттерне внедрения зависимости называются компонентами) выполнение передается обратно вверх (отсюда и названия паттерна – «инверсия управления») и клиент либо получает ответ с требуемыми данными, или ошибку с кодом и сообщением

Архитектура паттерна «Controller, Service, Repository»



Процесс обработки показан на диаграмме последовательности



Обработка запроса состоит из 7 уровней:

1. Пользователь
2. Компонент (фронтенд)
3. Сервис (фронтенд)
4. Контроллер (бэкенд)
5. Сервис (бэкенд)
6. Репозиторий (бэкенд)
7. База данных

Entity

Сущности необходимы для того, чтобы работать в коде с объектами предметной области. Созданные классы сущностей должны совпадать с данными.

Для доступа к данным будем использовать технологию JPA, а в качестве провайдера - Hibernate.

Итак, мы хотим, чтобы объекты класса могли быть сохранены в базе данных. Для этого 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.

Например

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

```
    private String email;
    @Column(name = "bday")
    private Date birthday;
    @Column(name = "phone")
    private String phone;


}
```

DTO

Для передачи данных между слоями внутри приложения будем использовать один из шаблонов проектирования – Data Transfer Object(DTO). Класс DTO, содержит данные без какой-либо логики для работы с ними. DTO обычно используются для передачи данных между различными приложениями, либо между слоями внутри одного приложения. Их можно рассматривать как хранилище информации, единственная цель которого — передать эту информацию получателю. Поэтому для каждого класса сущности нужно создать соответствующий класс DTO.

В них же для недопущения получения неверных данных используется валидация, которая реализована с помощью Validator.

Например

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class NewPersonDto {

    private Long personId;
    @NotNull(message="{valid.name.notNull}")
    @Size(min=3, message="{valid.firstname.size.min3}")
    private String firstName;
    @NotBlank(message="{valid.lastname.notBlank}")
    private String lastName;
    @NotBlank(message="{valid.street.notBlank}")
    private String street;
    @NotBlank(message="{valid.city.notBlank}")
    private String city;
    @Digits(integer=6, fraction=0, message="{valid.zip.digits}")
    private String zip;
    @NotBlank(message="{valid.email.notBlank}")
    @Email (message = "{valid.email.email}")
    private String email;
    @DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
    @Past(message = "{valid.birthday.past}")
    private Date birthday;
    @CellPhone (message = "{valid.phone.cellphone}")
    private String phone;


}
```

Repository

Для доступа к данным используется спецификация JPA. Данная спецификация описывает систему управления сохранением Java объектов в таблицы реляционных баз данных в удобном виде.

Для каждой сущности нужно создать соответствующий класс репозиторий наследуемый от CrudRepository. Он обеспечивает основные операции по поиску, сохранения, удалению данных (CRUD операции): T save(T entity), Optional findById(ID primaryKey), void delete(T entity) и др. операции.

```
S save(S var1);
Iterable<S> saveAll(Iterable<S> var1);
Optional<T> findById(ID var1);
boolean existsById(ID var1);
Iterable<T> findAll();
Iterable<T> findAllById(Iterable<ID> var1);
long count();
void deleteById(ID var1);
void delete(T var1);
 void deleteAll(Iterable<? extends T> var1);
void deleteAll();
```

Также класс CrudRepository позволяет строить запросы к сущности прямо из имени метода. Для этого используется механизм префиксов find…By, read…By, query…By, count…By, и get…By, далее от префикса метода начинает разбор остальной части. Вводное предложение может содержать дополнительные выражения, например, Distinct. Далее первый By действует как разделитель, чтобы указать начало фактических критериев. Можно определить условия для свойств сущностей и объединить их с помощью And и Or.

Существует несколько типов репозиториев, различающихся по набору возможностей.

Типы репозиториев

- **CrudRepository**, предоставляет базовый набор методов для доступа к данным. Данный интерфейс является универсальным и может быть использован не только в связке с JPA.
- **Repository** — базовый тип репозиториев, не содержит каких-либо методов, так же является универсальным.
- **PagingAndSortingRepository** — универсальный интерфейс, расширяющий CrudRepository и добавляющий поддержку пейджинации и сортировки.
- **JpaRepository** — репозиторий, добавляющий возможности, специфичные для JPA.
- **QueryDslJpaRepository** — реализация JpaRepository для взаимодействия с QueryDsl.
- **SimpleJpaRepository** — простая реализация JpaRepository.

Например

```
@Repository
```

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

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

Service

Все классы этого уровня взаимодействуют с базой данных посредством экземпляров классов репозиториев соответствующих классам сущностей. Каждый класс этого уровня имеет свои специфичные методы бизнес-логики.

Например,

```
public interface PersonService {
    List<Person> getAllPerson();
    void addNewPerson(Person person);
    void deletePerson(Person person );
    void editPerson(Person person);
    Optional<Person> getById(long id);
}
```

```
@Service
@Transactional
public class PersonServiceImpl implements PersonService{
    private final PersonRepository personRepository;
    @Autowired
    public PersonServiceImpl(PersonRepository personRepository)
{
        this.personRepository = personRepository;
    }
    public List<Person> getAllPerson() {
        return personRepository.findAll();
    }
    public  void addNewPerson(Person person){
        personRepository.save(person);
    }
    public  void deletePerson(Person person ){
        personRepository.delete(person);

    }

    public  void editPerson(Person person){
                personRepository.save(person);
    }
    public Optional<Person> getById(long id) {
        return personRepository.findAllById(id);
    }
}
```

Controller

Классы контроллеров состоят из набора методов, которые обрабатывают запросы, поступающие от клиента. В данном случае это запросы, отправляемые с web части проекта. При помощи аннотаций, предоставляемых фреймворком Spring, необходимо соотнести данные методы с URI и методом запроса.

Контроллеры не содержат сложной логики, а лишь пользуются методами, определенными на уровне бизнес-логики.

Например

```
@Slf4j
@RestController
@RequestMapping
public class PersonController {

    private  final PersonService personService;
   @Value("${welcome.message}")
    private String message;

    @Value("${error.message}")
    private String errorMessage;

    @Autowired
    public PersonController(PersonService personService) {
        this.personService = personService;
    }

    @GetMapping(value = {"/", "/index"})
    public ModelAndView index(Model model) {
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.setViewName("index");
        model.addAttribute("message", message);
        log.info("index was called");
        return modelAndView;
    }
```

H2 база данных

База данных H2 часто называется дефолтовая, потому что она используется только на этапе разработки приложения, пока не определились с конкретной базой данных. Это делается для того, чтобы не приходилось настраивать полное окружение для запуска приложения, которая к тому же может поменяться.

Конфигурации в Spring

Spring Boot настраивает Hibernate в качестве поставщика JPA по умолчанию, поэтому нет необходимости определять компонент EntityManagerFactory.

Spring Boot также может автоматически настраивать bean-компонент dataSource в зависимости от базы данных, которую мы используем. В случае базы данных в памяти типа H2, HSQLDB и Apache Derby Boot автоматически настраивает источник данных, если соответствующая зависимость базы данных присутствует в пути к классам.

Если мы хотим использовать JPA с базой данных MySQL, нам нужна зависимость mysql-connector-java,

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.15</version>
</dependency>
```

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

а также определения конфигурации DataSource.

Чтобы настроить источник данных с помощью файла свойств, мы должны установить свойства с префиксом spring.datasource:

```
spring.jpa.hibernate.ddl-auto
spring.datasource.url=
spring.datasource.username=
spring.datasource.password=
```

Или мы можем сделать это в классе @Configuration

```
@Configuration
public class WebConfig implements WebMvcConfigurer{
    @Bean


...
  //   добавлем конфигурацию
    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();

        dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
        dataSource.setUsername("root");
        dataSource.setPassword("root");
        dataSource.setUrl(
```

```
"jdbc:mysql://localhost:3306/projectDb?createDatabaseIfNotExist=true");

        return dataSource;
    }
```

Если Spring-Boot не используется нужно настроить **EntityManager**.

Сделать это можно с помощью фабричного компонента Spring. Нужно явно определить bean-компонент DataSource и компоненты TransactionManager.

```
@Configuration
@EnableTransactionManagement
public class PersistentJPAConfig {

         @Bean
        public LocalContainerEntityManagerFactoryBean entityManagerFactory()
{
            LocalContainerEntityManagerFactoryBean em
                    = new LocalContainerEntityManagerFactoryBean();
            em.setDataSource(dataSource());
            em.setPackagesToScan(new String[] {
"by.patsei.springproject1.entity" });

            JpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
            em.setJpaVendorAdapter(vendorAdapter);
            em.setJpaProperties(additionalProperties());

            return em;
        }


    @Bean
    public DataSource dataSource(){
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");

dataSource.setUrl("jdbc:mysql://localhost:3306/projectDb?createDatabaseIfNotE
xist=true");
        dataSource.setUsername( "root" );
        dataSource.setPassword( "root" );
        return dataSource;
    }

    @Bean
    public PlatformTransactionManager transactionManager(EntityManagerFactory
emf) {
        JpaTransactionManager transactionManager = new
JpaTransactionManager();
        transactionManager.setEntityManagerFactory(emf);

        return transactionManager;
    }

    @Bean
    public PersistenceExceptionTranslationPostProcessor
exceptionTranslation(){
        return new PersistenceExceptionTranslationPostProcessor();
    }
```

```
    Properties additionalProperties() {
        Properties properties = new Properties();
        properties.setProperty("hibernate.hbm2ddl.auto", "create-drop");
        properties.setProperty("hibernate.dialect",
"org.hibernate.dialect.MySQL5Dialect");

        return properties;
    }
}
```

Подробней

https://www.baeldung.com/persistence-with-spring-series