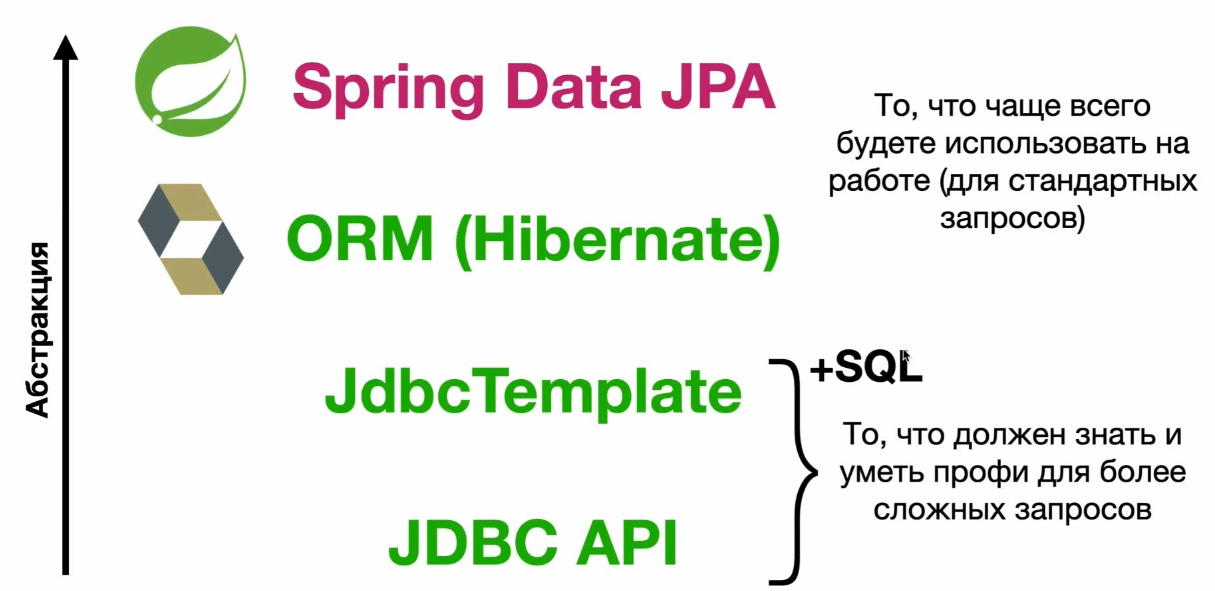
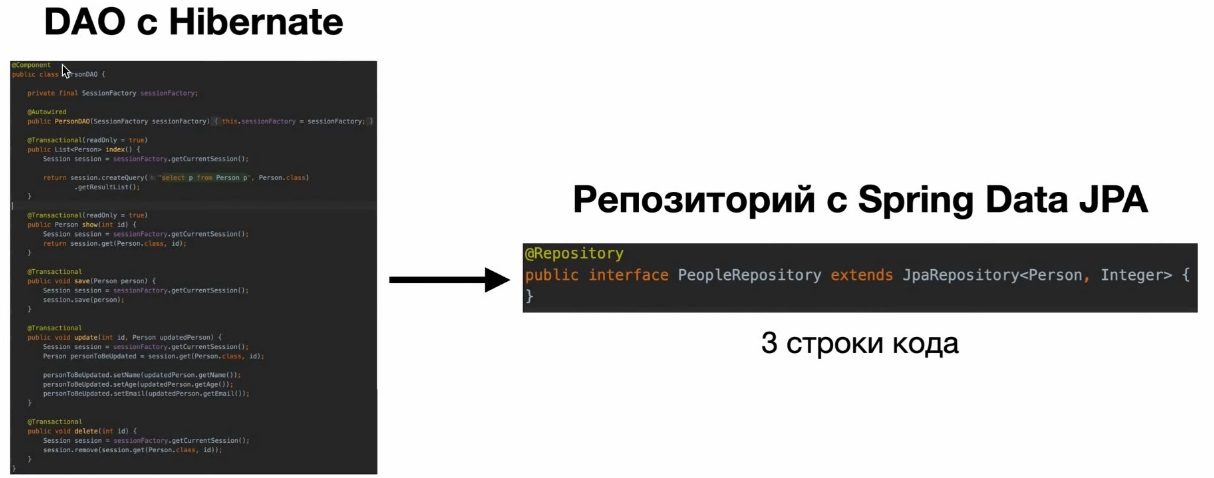
**Spring Data JPA**

**Spring Data JPA** максимально упрощат работы с базой данных.

**Spring Data JPA** под капотом использует Hibernate, а Hibernate под капотом использует JDBC API.

В **Spring Data JPA** все методы CRUD (и не только) генерируются автоматически. Тоесть если раньше при использовании **Hibernate** мы описывали все CRUD методы самостоятельно, то с помощью **Spring Data JPA** нам достаточно использовать аннотацию **@Repository** и унаследоваться от класа **JpaRepository**. В итоге все CRUD операции (и не только) будут доступны всего лишь написав 3 строчки кода.



При работе с базой данных существует несколько паттернов: **DAO** и **Repository**.

**1. DAO** и **Repository** похожие понятия (и то и другое для работы с данными).

**2. Repository** обычно более высокоуровненый, ближе к бизнес логике (не пишем SQL запросы, работаем с сущностями).

**3. DAO** обычно более низкоуровневый, ближе к БД (можем писать SQL запросы даже если используем **Hibernate**).

**4.** В сложных приложениях обычно есть и то и другое.

**Repository –** для стандартных операций с данными (CRUD, например).

**DAO** – для более сложных/не стандартных маниуляций с данными и с БД, где обычно нужно вручную писать SQL/HQL код.

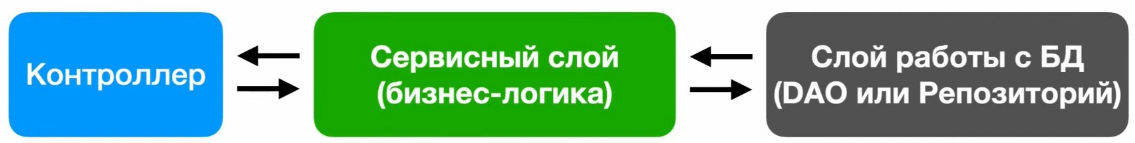
**Сервисный слой. Аннотация @Service**

В предыдущих уроках мы делали так: у нас был какой -то контроллер и в этом контроллере мы вызывали методы из **DAO**. **DAO** – это был слой работы с базой данных, тоесть было всего лишь 2 слоя: слой контроллера и слой работы с базой данных.

Но такое решение является **не** **правильным**.

  
  
  
  
  
  
  
**Правильным** решением является:  
Помимо слоя контроллера и слоя работы с базой данных должен быть еще один слой который лежит между ними. Этот слой называется сервисный слой (бизнес логика).

В предыдущих уроках у нас не было никакой бизнес логики. Мы просто получали данные из базы данных или записывали данные в базу данных, тоесть никакой бизнес логики не было, **НО** в реальных приложениях обычно всегда есть какая то бизнес-логика, тоесть например, достаем какие то данные из базы данных и с ними что то делаем (считаем что-то, рассчитываем, преобразуем и т.д). Эту бизнес логику мы не можем помещать в контроллер так как в контроллере нужно иметь минимум кода и поэтому между слоем базы данных и слоем контроллера есть еще сервисный слой в который мы помещаем всю бизнес-логику.

 **1.** В контроллере мы хотим иметь минимум кода. В идеале — только простейшая логика обработки запросов и ответов от пользователя.

**2.** Любая работа с данными, полученными из БД должна лежать в **сервисном слое**. Здесь мы и описываем бизнес логику нашего приложения, а в контроллере только вызываем нужные методы.

**3.** В один сервисный класс может быть внедренно несколько DAO, репозиториев.

**4.** Клас для сервисного слоя помечается аннотацией **@Service.**

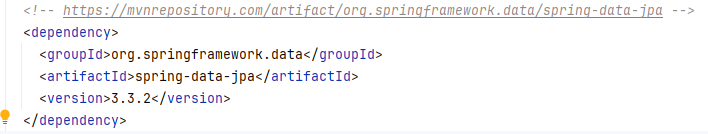
**ИТОГО:**

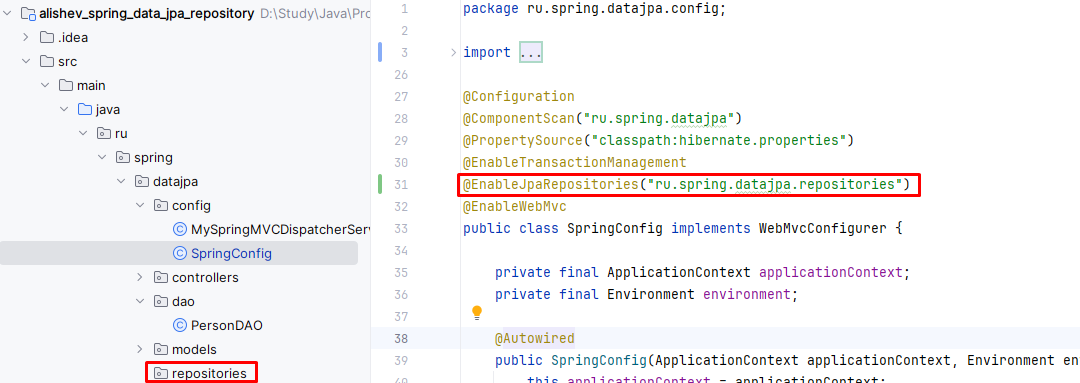
Боевое Spring приложение в реальной работе выглядит примерно так:

Spring приложение с Hibernate, Sprimg Data JPA и сервисный слой. Так же будет DAO со сложными SQL запросами.

**Пример работы Spring Data JPA.**

**1.** Нужно добавить завизимость **Spring Data JPA** в pom.xml.

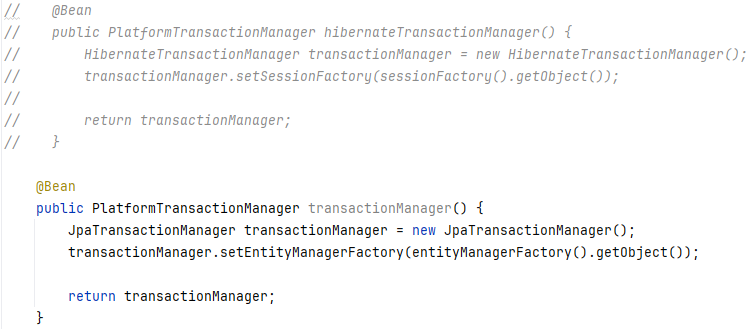


**2.** В класс**SpringConfig** добавить анотацию **@EnableJpaRepositories()**.  
С помощью этой аннотации мы включаем поддержку JPA репозиториев. Внтури аннотации мы должны указать путь к папке где будут лежать Jpa репозитории. **Репозиторий** — это тот интерфейс который помечен аннотацией репозиторий.

**3.** Чтобы у нас работал **Spring Data Jpa** нужно изменить SpringConfig.

Меняем создание бина **sessionFactory()** на создание бина **entityManagerFactory()**. Эти два понятия очень похожи, они делаеют пратически одно и тоже. Но зачем тогда менять на **entityManagerFactory() ?** Дело в том, что это понятие которое есть в библиотеке Hibernate, тоесть создатели Hibernate сами создали такие понятия как **session** и **sessionFactory.** Но помимо библиотеки Hibernate есть еще спецификация JPA и в спецификации JPA нет понятий **session** и **sessionFactory.** В спецификации JPA есть понятия

**manager** и **entityManagerFactory.** Другими словами **sessionFactory** делают пратически тоже самое, что и **entityManagerFactory** только **sessionFactory** находится в Hibernate, а  **entityManagerFactory** в Jpa**.** Поэтому чтобы работал **Spring Data Jpa** нужно изменить **sessionFactory** на **entityManagerFactory**, но при этом у нас будет работать и Hibernate и **Spring Data Jpa.**



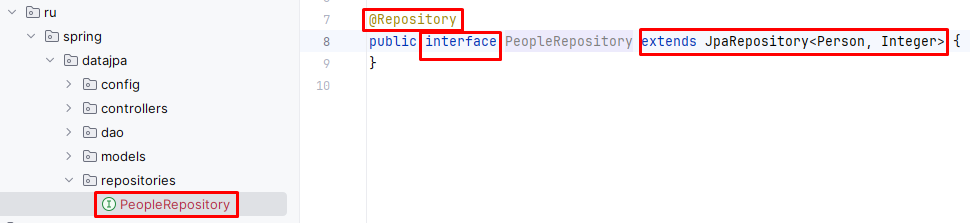
**4.** После настройки конфигурации создадим репозиторий.

Репозиорий должен иметь аннотацию **@Repository**, быть интерфейсом и наследоваться от **JpaRepository**. Этот **JpaRepository** мы должны параметризовать:

1. классом который является сущностью, тоесть для какой сущности будут созданы наши методы.

2. нужно указать тип первичного ключа.

За счет того, что мы указали эти два параметра Jpa repository автоматически сгенерирует все необходимые операции для сущности которая указана в параметрах в пункте 1.

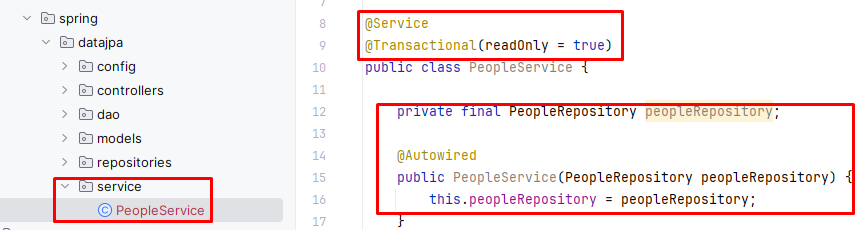


**5.** Теперь нужно создать Сервис.

Для этого создадим новый пакет где у нас будут лежать сервисы, он будет называться **services**.

В этом пакете создадим новый класс **PeopleService**. В этом классе будет лежать бизнес логика. У нас в примере пока бизнес логики не будет, здесь будут обычные вызовы методов из репозитория, но в целом здесь могут быть внедрены несколько репозиториев, здесь можно работать с данными. Этот класс нужно пометить аннотацией **@Service**. Так же сервис берет на себя еще и работу с транзакциями поэтому помечаем этот класс так же аннотацией **@Transactional**. Если целый класс помечается аннотацией **@Transactional**, то все публичные методы внутри этого класса автоматически будут иметь аннотацию **@Transactional.**

Так же в классе **PeopleService** внедрим репозиторий с помощью конструктора.

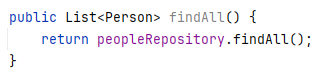
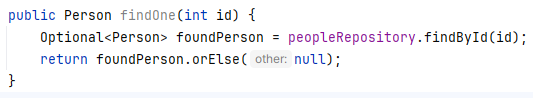


**6.** Теперь внедрим в наш контроллер наш сервис, чтобы могли использовать наш сервис в контроллере в качестве прослойки между контроллером и репозиторием.

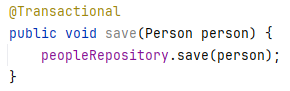


**7.** Теперь опишем методы в сервисе.

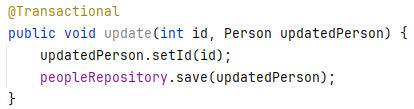
1. Вернем всех людей из списка. Для этого используем метод ***findAll()***. Этот метод возвращает все сущности из таблицы. Репозиторий видит, что наша сущность **Person** связана с таблицей **person**.

  
  
  
  
  
2. Найдем человека по его id.

3. Сохраним человека.



4. Обновим человека. В репозитории есть такое соглашене, что для добавления сущности и для обновления сущности используется один и тот же метод ***save()***. Поэтому в этот метод просто передаем обновленную сущность и метод ***save()*** увидит, то что передается сущность с **id** который уже есть в таблице и поэтому эту чущность он не будет сохранять, он просто обновит значения у существующей сущности которая уже есть в таблице. Именно поэтому мы указали **id** в сеттере, чтобы метод ***save()*** понимал, что эта сущность уже есть в таблице.



5. Удаление челвоека.

