Лекция 3 № Многоуровневая архитектура web приложений

The layered architecture of the web application. MVC Application of design patterns. Repository level. Data Access Object. Service level. Command level (business logic).

Архитектура программного обеспечения - это структура высокого уровня, а также дисциплина ее создания и документация.

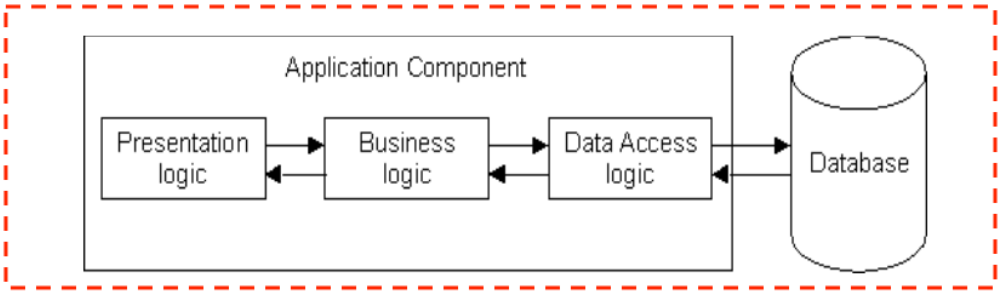
1) **The Separation of Concerns (SoC) Principle**
   Принцип разделения интересов - это принцип разработки для разделения компьютерной программы на отдельные уровни, так что каждый уровень посвящен отдельной проблеме.
   Принцип поможет определить необходимые уровни и обязанности каждого уровня.
2) **Принцип Keep It Simple Stupid (KISS)**
   Большинство систем работают лучше, если они остаются простыми, а не сложными; поэтому простота должна быть ключевым принципом в дизайне, и следует избегать ненужной сложности.
   У каждого слоя есть цена и если мы создадим сложную архитектуру, которая имеет слишком много слоев, эта цена будет слишком высокой.

MULTI-TIER (2-TIER, 3-TIER), MVC

N-уровневые архитектуры имеют одинаковые компоненты

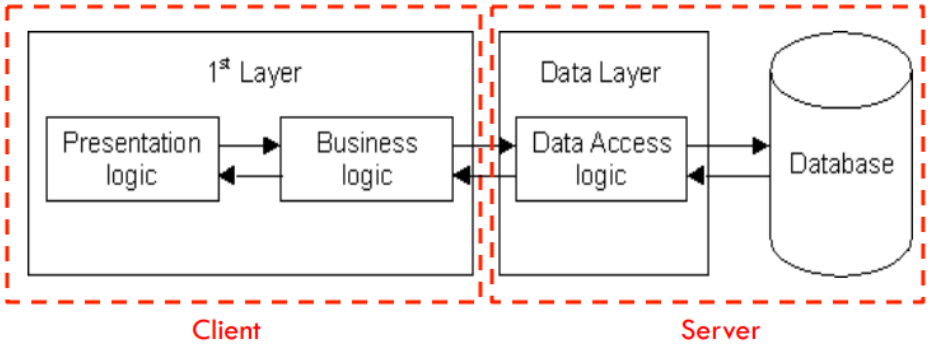
- Presentation
- Business/Logic
- Data

1-Tier Architecture

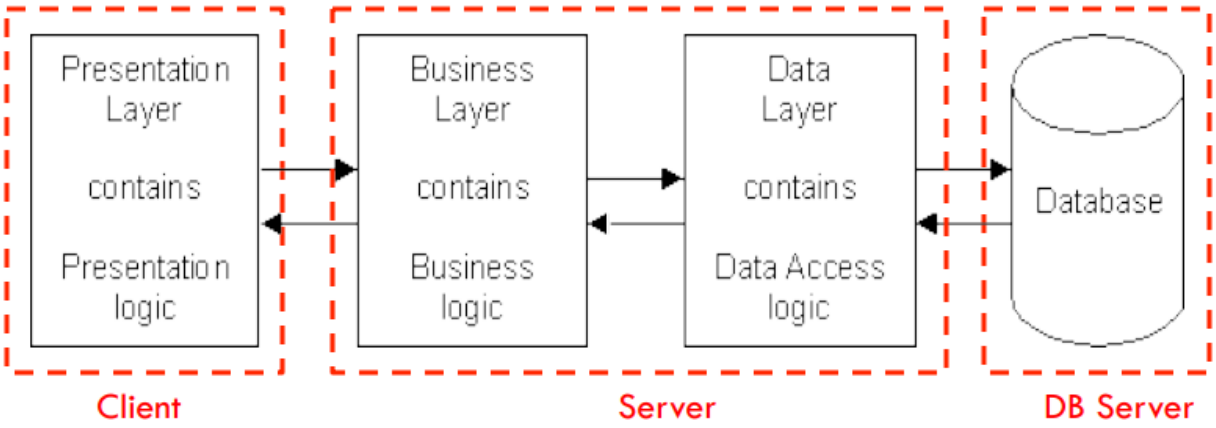


Presentation, Logic, Data layers тесно связаны

2-Tier Architecture



3-Tier Architecture



3-Tier Architecture for Web Apps

Presentation Layer

Статический или динамически генерируемый контент, отображаемый браузером (front-end)

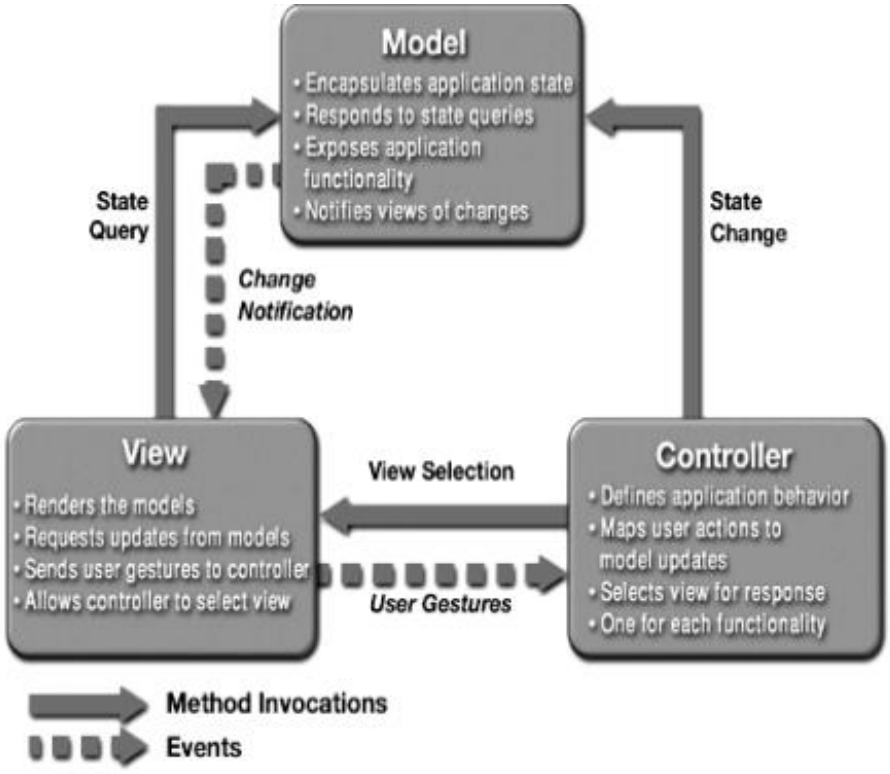
Logic Layer

Динамическая обработка контента и генерация сервером приложений (Java EE, Spring, ASP.NET, PHP) (middleware)

Data Layer

База данных, содержащая как наборы данных, так и систему управления, которое управляет и обеспечивает доступ к данным (back-end)

MVC for Web Applications



3-tier Architecture vs. MVC Architecture

Коммуникации

3-tier: уровень представления никогда не связывается напрямую с уровнем данных только через логический уровень (линейный топология)

MVC: все слои взаимодействуют напрямую (топология треугольника)

Использование

3-tier: в основном используется в веб-приложениях, где клиент, промежуточное ПО и уровни данных работают физически раздельно от платформы

MVC: исторически используется в приложениях, работающих на одной рабочей станции.

Catalog of Patterns of Enterprise Application Architecture
https://martinfowler.com/eaaCatalog/index.html

Трех слоев хватает всем. Если подумать об обязанностях веб-приложения, мы заметим, что у веб-приложения есть следующие «проблемы»:

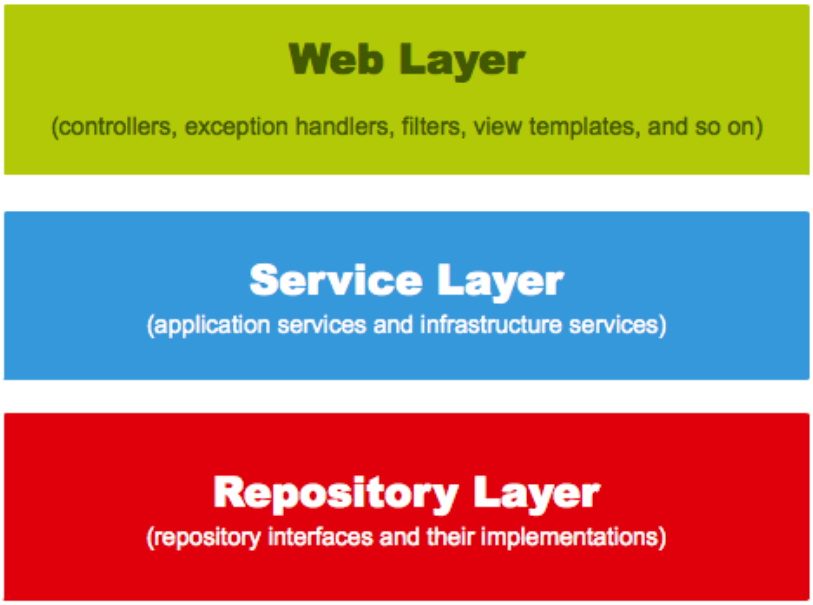
1) нужно обработать ввод пользователя и вернуть правильный ответ обратно пользователю ;
2) требуется механизм обработки исключений, который даст разумные сообщения об ошибках пользователю;
3) нужна стратегия управления транзакциями;
4) обработка как аутентификации, так и авторизации;
5) необходимо реализовать бизнес-логику приложения.
6) нужно общаться к хранилищу данных и другими внешними ресурсам.

слои:

Web - веб-слой (слой контроллеров) - это самый верхний слой веб-приложения. Он отвечает за обработку ввода пользователя и возврат правильного ответа обратно пользователю. Веб-слой также должен обрабатывать исключения, создаваемые другими слоями. Поскольку веб-слой является точкой входа в наше приложение, он должен заботиться об аутентификации и действовать как первая линия защиты от неавторизованных пользователей.

Service - сервисный слой находится ниже веб-слоя. Он действует как граница транзакции и содержит как *прикладны*е, так и *инфраструктурные* сервисы. Службы приложений предоставляют общедоступный API уровня служб. Они также действуют как граница транзакции и отвечают за авторизацию. Инфраструктурные сервисы содержат «соединительный код», который связывается с внешними ресурсами, такими как файловые системы, базы данных или почтовые серверы. Часто эти методы используются несколькими службами приложений.

Repository - уровень хранилища. Это самый нижний уровень веб-приложения. Он отвечает за связь с используемым хранилищем данных.

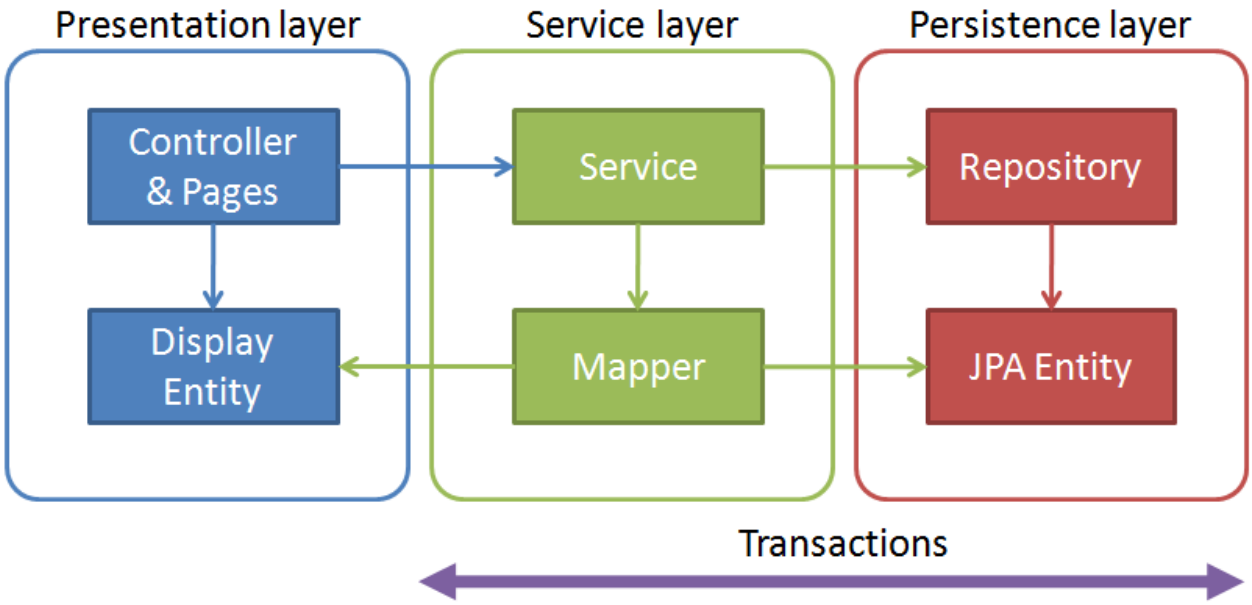


Spring web application

Repository

Эрик Эванс: «Repository представляет собой все объекты определенного типа в виде концептуального множества. Его поведение похоже на поведение коллекции, за исключением более развитых возможностей для построения запросов».

Репозиторий отличается от коллекции, если рассматривать возможности для построения запросов. Имея коллекцию объектов в памяти, довольно просто перебрать все ее элементы и найти интересующий нас экземпляр. Репозиторий работает с большим набором объектов, чаще всего, находящихся вне оперативной памяти в момент выполнения запроса. Нецелесообразно загружать все в память, если нам необходим один объект. Вместо этого, мы передаем репозиторию критерий, с помощью которого он сможет найти один или несколько объектов. Репозиторий может сгенерировать SQL запрос в том случае, если он использует базу данных в качестве бекэнда, или он может найти необходимый объект перебором, если используется коллекция в памяти.



Spring 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** или **JpaRepository**. 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.

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.

Domain model

Отвечает за представление концепций, содержит информацию о ситуации и бизнес-правила.

Domain model состоит из трех разных объектов:

1) **Доменная служба** (**domain service**) - это класс без состояния, который предоставляет операции связанные с концепцией домена, но не являются «естественной» частью объекта или объекта значения.
2) **Entity (сущность)** - это объект, определяемый идентичностью, которая остается неизменной на протяжении всего жизненного цикла.
3) **Value object (объект значения)** описывает свойство или вещь. Эти объекты не имеют своей собственной идентичности или жизненного цикла. Жизненный цикл объекта значения связан с жизненным циклом объекта.

Entity

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

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.

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.

Value objects vs Entities

Value objects

1) Используются в качестве дескрипторов для элементов модели.

Примеры: деньги, валюта, имя, высота и цвет.

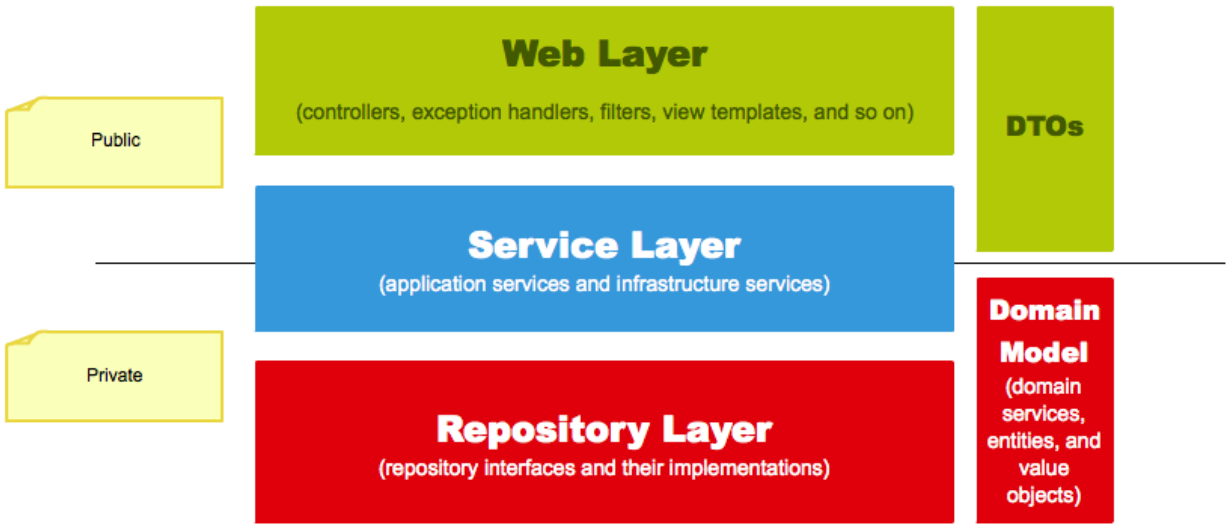
2) Value objects в одном домене могут использовать value objects в другом и наоборот.
3) Не имеют идентичности.
4) Неизменны; их значения не могут измениться.
5) Являются связными; они могут обернуть несколько атрибутов для полной инкапсуляции единой концепции.
6) Могут быть объединены для создания новых значений без изменения оригинала.
7) Являются самовалидирующимися; никогда не должны быть в недопустимом состоянии.

Entities

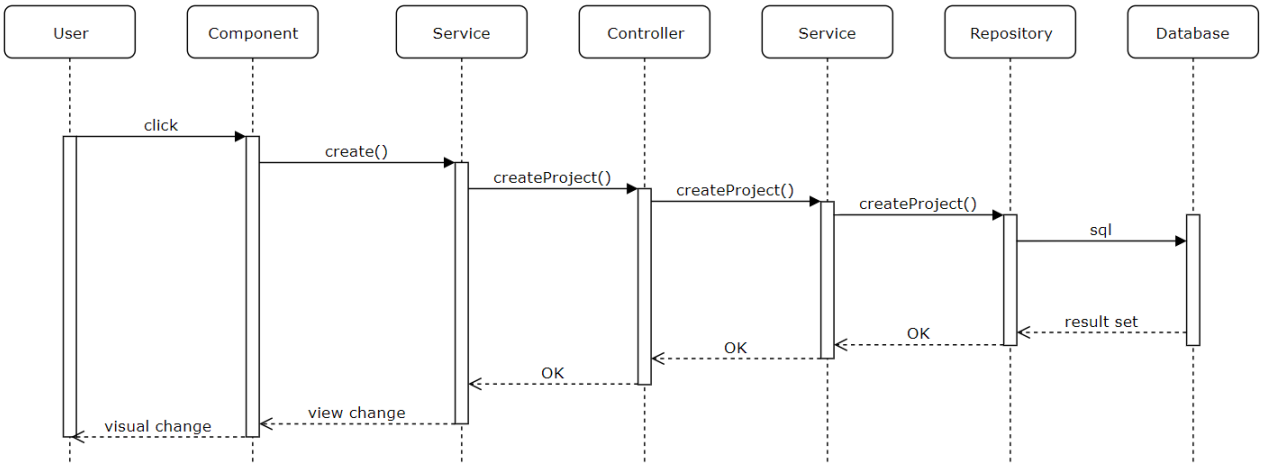
1) доменные понятия, которые имеют уникальную идентичность в проблемной области.
2) Ключевые понятия из проблемной области, генерируются приложением и хранилищем данных.
3) Наличие жизненного цикла.
4) Всегда должны быть действительными для данного контекста.

Теперь можем перейти к разработке интерфейса для каждого слоя. Выделим обязанности слоев:

1) Веб-слой должен обрабатывать только DTO.
2) Сервисный уровень принимает DTO (и основные типы) в качестве параметров метода. Он может обрабатывать объекты domen model , но может возвращать только DTO обратно на веб-слой.
3) Уровень хранилища принимает entity (и основные типы) в качестве параметров метода и возвращает entity (и основные типы).



Процесс обработки показан на диаграмме последовательности

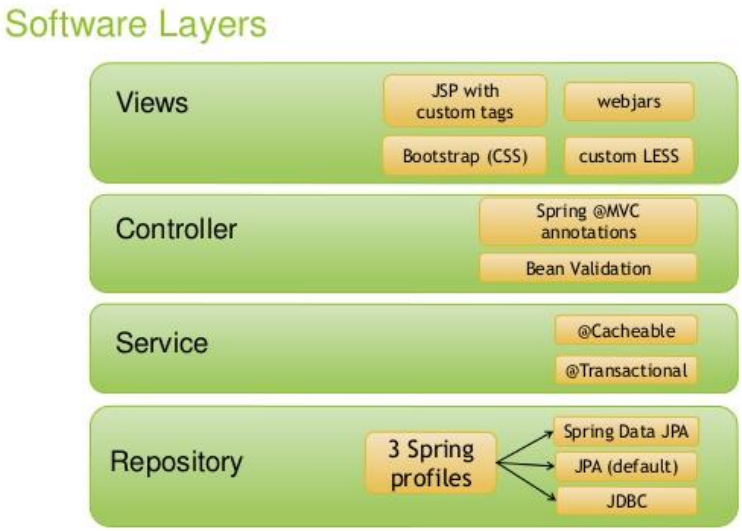


Обработка запроса состоит из 7 уровней:

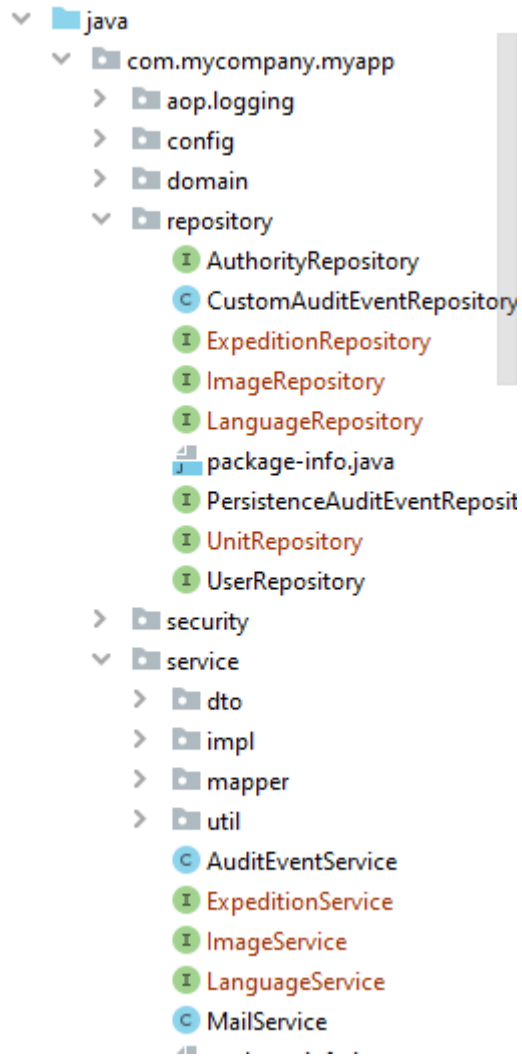
1. Пользователь
2. Компонент (фронтенд)
3. Сервис (фронтенд)
4. Контроллер (бэкенд)
5. Сервис (бэкенд)
6. Репозиторий (бэкенд)
7. База данных

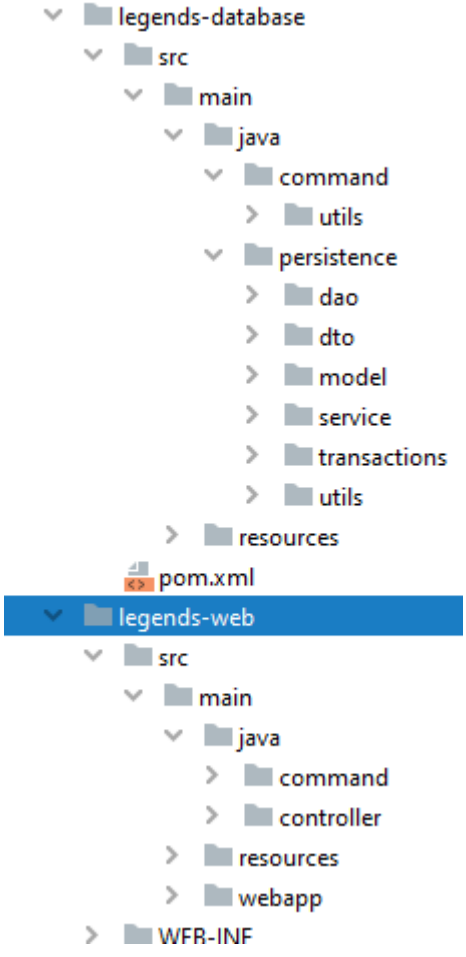
У нас будет **Repository**, отвечающий за работу с данными, **Service**, где будет разная логика и **Controller** будет только обрабатывать запросы и вызывать нужные методы сервиса.

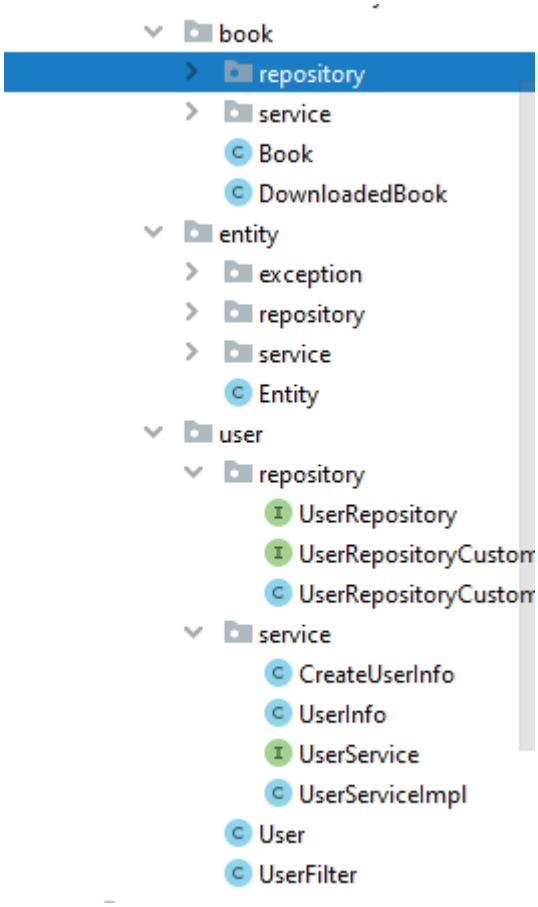
Варианты

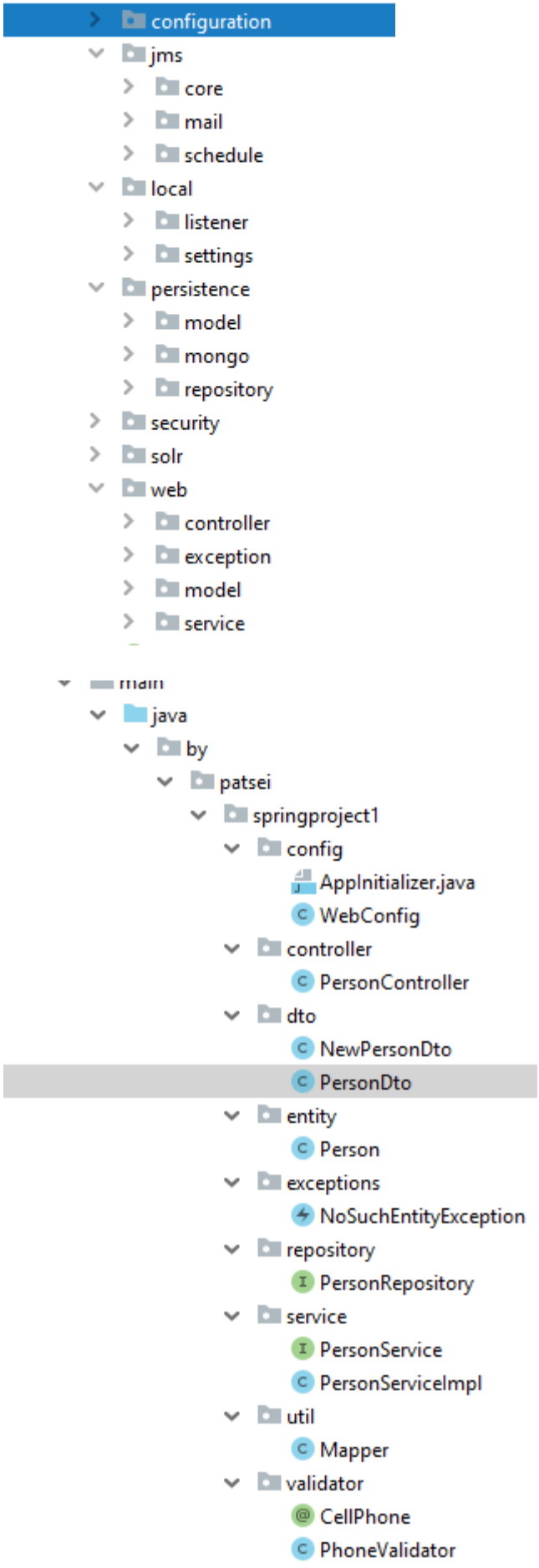


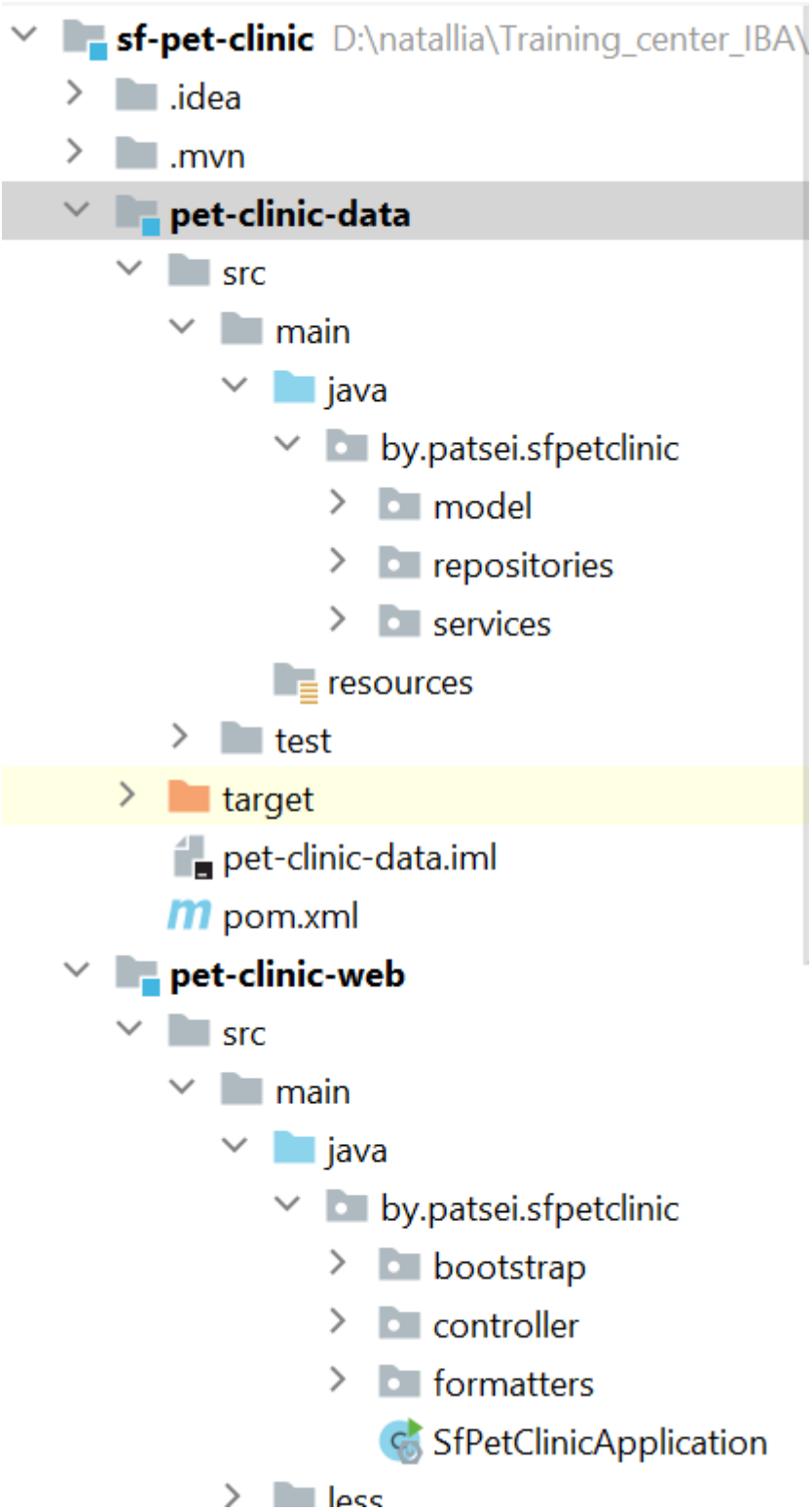
Структура проекта может быт такой




legends-database
src
main
java
command
utils
persistence
dao
dto
model
service
transactions
utils
resources
pom.xml
legends-web
src
main
java
command
controller
resources
webapp
WEB-INF


book
repository
service
Book
DownloadedBook
entity
exception
repository
service
Entity
user
repository
UserRepository
UserRepositoryCustom
UserRepositoryCustom
service
CreateUserInfo
UserInfo
UserService
UserServiceImpl
User
UserFilter


configuration
jms
core
mail
schedule
local
listener
settings
persistence
model
mongo
repository
security
solr
web
controller
exception
model
service
main
java
by
patsei
springproject1
config
AppInitializer.java
WebConfig
controller
PersonController
dto
NewPersonDto
PersonDto
entity
Person
exceptions
NoSuchEntityException
repository
PersonRepository
service
PersonService
PersonServiceImpl
util
Mapper
validator
CellPhone
PhoneValidator



https://github.com/PatseiBSTU/sf-pet-clinic

https://github.com/PatseiBSTU/SpringProject1