**Spring Security Аутентификация**

**1.** Создадим новую базу данных и в ней создадим новую таблицу пользователей:

CREATE TABLE Person(

id int PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,

username varchar(100) NOT NULL,

year\_of\_birth int NOT NULL,

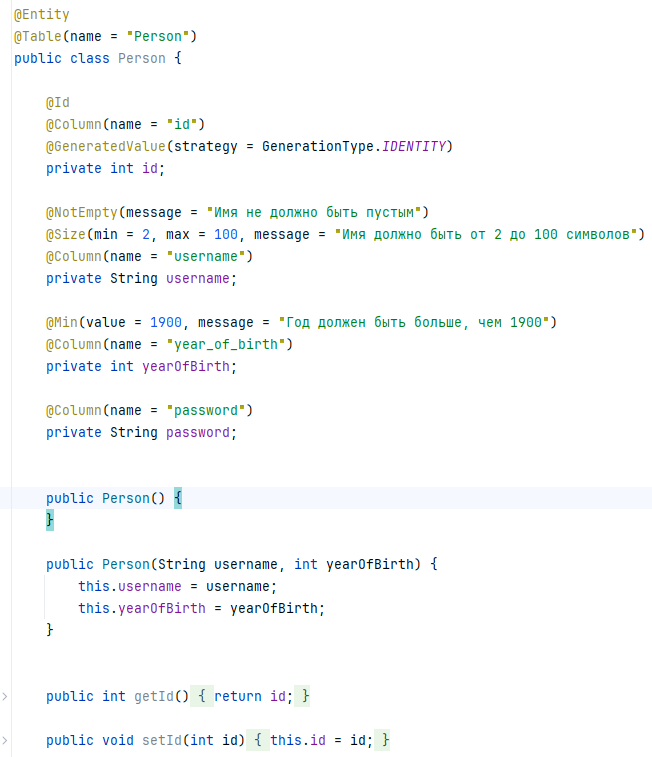
password varchar NOT NULL

);

INSERT INTO Person(username, year\_of\_birth, password) VALUES ('test\_user1', 1991, 'testpass');

INSERT INTO Person(username, year\_of\_birth, password) VALUES ('test\_user2', 1993, 'testpass');

**2.** Создадим сущность **Person** в джава приложении для связи с таблицей базы данных.



**3.** Создадим отдельный пакет **security**. В этом пакете будут лежать классы котоыре нужны для **Spring Security**.

В этом пакете создадим специальный класс **PersonDetails**, который нужен специально для **Spring Security**. Мы могли бы использовать саму сущность **Person** и на объекте **Person** которого мы достали из базы данных вызвать геттер и получать пароль из самой сущности и таким образом проводить аутентификацию, НО в **Spring Security** так не принято, а принято использовать специальный класс, который является классом обёрткой над нашей сущностью. Тоесть когда мы работаем с **Security** – мы не напрямую работаем с сущностью, а работаем с классом обёрткой.

Класс **PersonDetails** должен реализовывать специальный интерфейс который называется **UserDetails** и соответственно нужно реализовать все методы этого интерфейса. Реализация этого интерфейса нужна по большей части для стандартизации, так как в этом интерфейсе есть стандартные методы с стандартными названиями и сигнатурами (***getPassord()***, ***getUserName()*** и т. д.).

Так как класс **PersonDetails** это обёртка над нашей сущностью **Person**, то в качестве поля у нас будет человек, тоесть будем хранить сущность **Person** в качестве поля. Его мы будем передавать в качестве аргумента в конструкторе.

**Описание методов интерфейса UserDetails:**

***getAuthorities() -*** нужен для авторизации. В будующем будем получать роли которые есть у пользователя (например из базы данных) и будем возвращать коллекцию прав которые есть у пользователя.

***getPassword()*** - возвращает пароль пользователя.

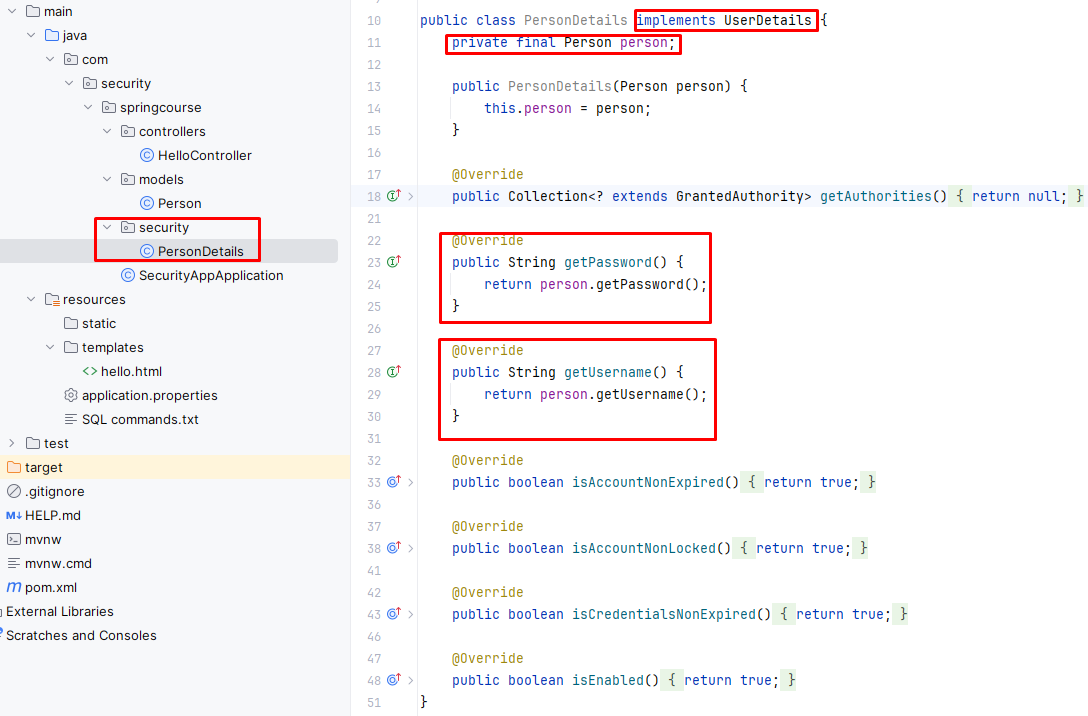
***getUserName()*** - возвращает имя пользователя.

***isAccountNonExpired()*** - проверяет если аккаунт не просрочен.

***isAccountNonLocked()*** - проверяет если аккаунт не заблокирован.

***isCredentialIsNonExpired()*** - проверяет если пароль не просрочен.

***isEnabled()*** - проверяет если аккаунт работает.



**4.** После реализации обёртки над сущностью можно приступить к реализации конфига, конфигурации **Spring Security**.

Для этого создадим отдельный пакет, который будет называться **config**.

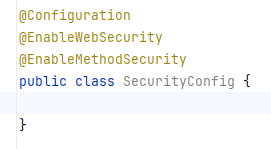
В этом пакете создадим новый класс который будет называться **SecurityConfig**. Этот класс является главным классом, где мы настраиваем наш **Spring Security**. Здесь мы будем настраивать аутентификацию и здесь же мы будем настраивать авторизацию и вообще всё, что касается **Spring Security.**

Здесь мы будем использовать специальные аннтоции:

**@EnableWebSecurity** – эта аннотация даёт понять спрингу, то что это конфигурационный класс для **Spring Security**.

**@EnableMethodSecurity –** переход на использование конфигурация с помощью бинов.

Для самой просто конфиграции этого хватит.



**5.** Дальше нужно реализовать **AuthnticationProvider.** Он будет лежать в пакете security.

Создадим новый джава класс, который будет называться **AuthProviderImpl.**

В этом классе будет следующая логика: мы будем смотреть на таблицу **Person** в базе данных и сравнивать пароль и username который будет введен с формы с паролем из базы данных.

Над классом добавим аннотацию **@Component**, чтобы мы могли внедрять его с помощью аннотации **@Autoawired**.

Внедрим в этот класс сервис **PersonDetailsService**, чтобы взаимодйствовать с базой данных для получения данных о пользователе если он существует.

Класс **AuthProviderImpl** долженреализовывать интерфейс **AuthnticationProvider.** В этом интерфейсе есть два метода:

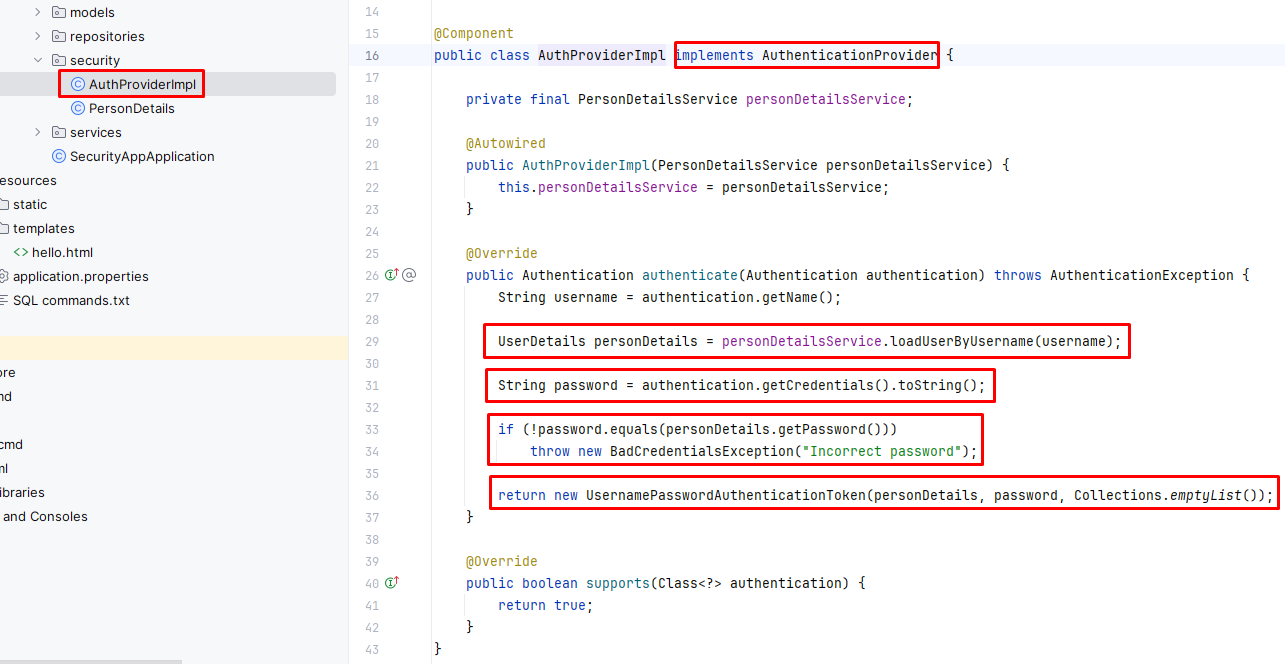
***authentificate()*** - принимает объект **Authntication** и возвращает тоже объект Authntication.Здесь будет лежать логика аутентификации.

Здесь на нашем сервисе вызовем метод ***loadUserByUsername() -*** он нам вернет **UserDetails**.

Получим пароль который был введен на форме — ***authentication.getCredantials().toString()***.

Далее сравним пароль из формы и пароль который есть у нашего пользователя, Если пароль не совпадает выкинем исключение, если совпадает, тогда вернем объект **Authentication**. Для этого создадим объект **UsernamePasswordAuthenticationToken** ипередаем в качестве аругментов PersonDetails (это будет в качетсве Principal), password и в качестве третьего аргумента передается список прав этого пользователя.

***supports()* -** чисто технический метод, нужен спрингу, чтобы понять для какого объекта наш класс работает.



**П.С.** мы можем не создавать этот класс, но тогда нам нужно будет в классе конфигурации изменить конфигурацию, чтобы аутентфикация проходила стандатным методом:



**6.** Дальше нужно реализовать репозиторий и сервис для поиска человека по имени.

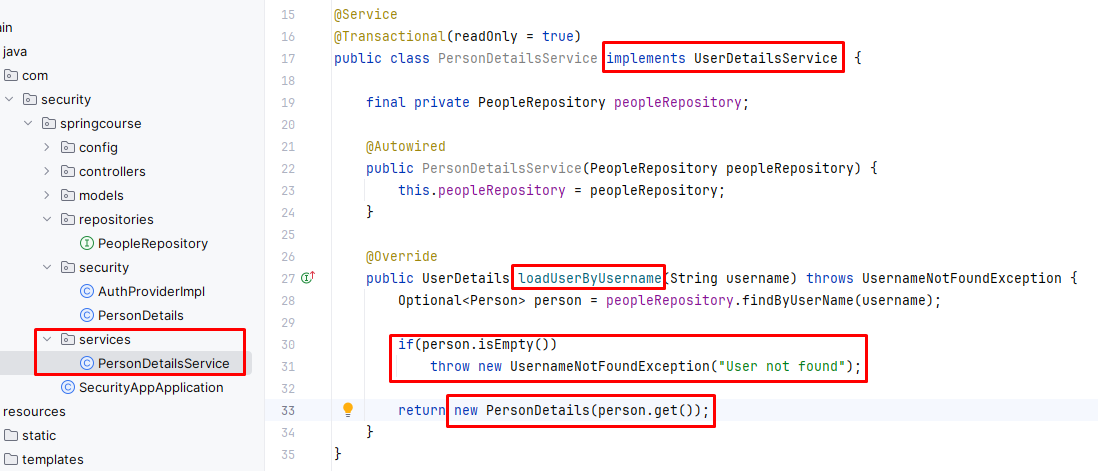
Репозиторий и сервис создается как обычно, но с небольшим отличичием:

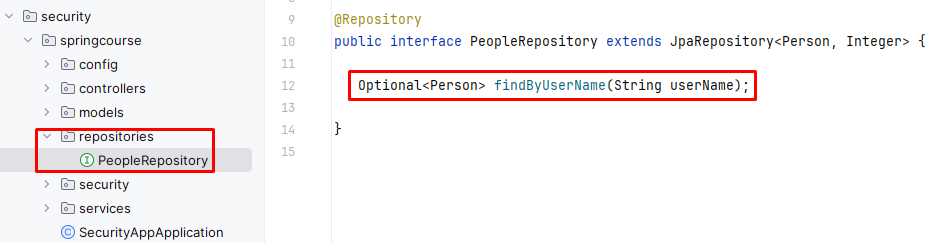
Сервис будет предназначатся для **Spring Security**. Чтобы **Spring Security** знал, что этот сервис загружает пользователя нам нужно реализовать интерфейс **UserDetailsService**. В этом интерфейсе есть всего лишь одна сигнатура ***loadUserByUserName()***. В этом методе мы просто вызываем метод из репозитория, чтобы получить челвоека по userName.

После получения пользователя, проверим если он существует с помощью метода **isEmpty()**.

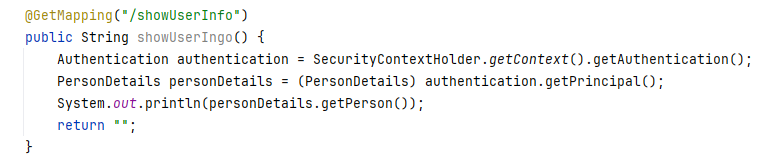
Если такой пользователь не найден. То выкинем исключение **UserNameNotFoundException(“”)**. Это исключение будет поймано самим **Spring Security** и **Spring Security** покажет сообщене которое было в исключении на странице логина.

Если такой человек найден, то мы просто оборачиваем этот объект класса Person в класс **PersonDetails** (класс который мы реализовали ранее) и просто возвращаем этот **PersonDetails** из этого метода.

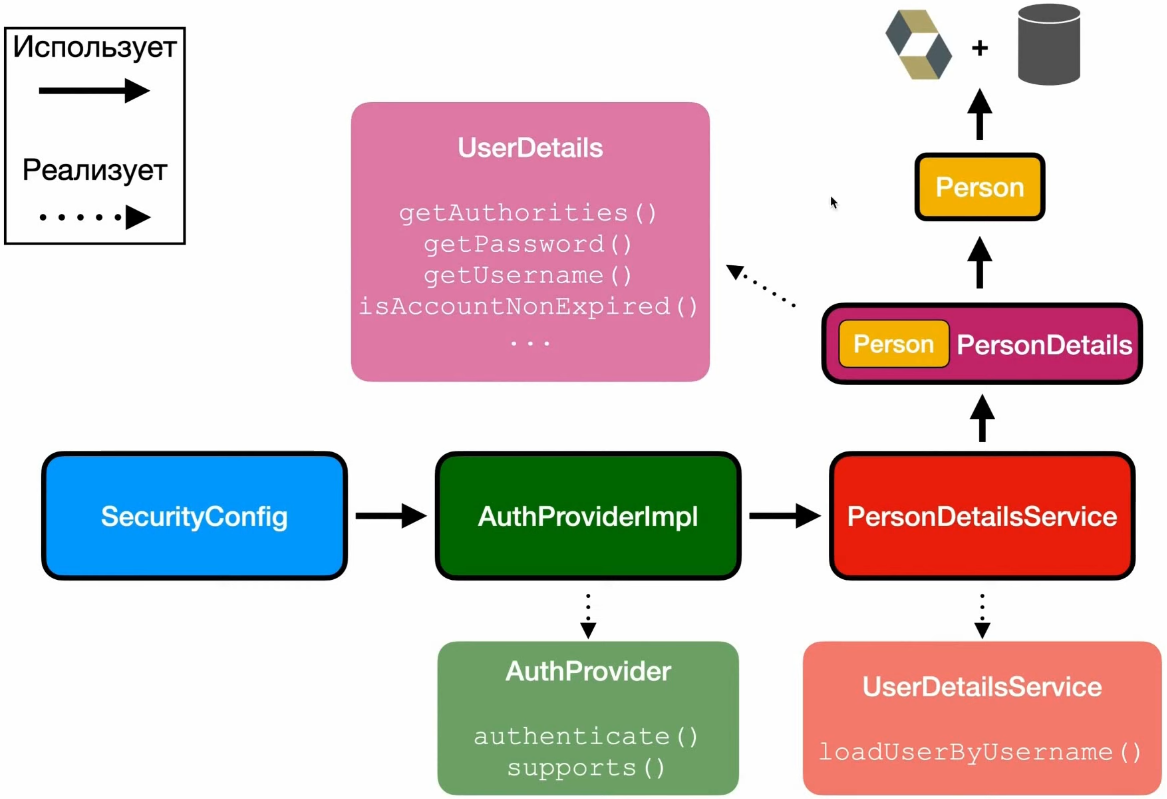




**7.** Чтобы получить получить из потока объект аутентифицированного пользователя нужно получить доступ к объекту **Authentication** и с помощью специального **SecurityContextHolder** вызываем методы ***getContext().getAuthentication()***, чтобы получить доступ к этому объекту из нашего потока.



**Схема приложения которое мы реализовали выше**

В ярких красках показаны классы, в не яскрих красках показаны интерфейсы.  
  
Описание приложения (схемы):

У нас есть класс **SecurityConfig** – в котором содержится вся конфигурация Spring Security.

Класс **AuthProviderImpl** в себе содержит логику аутентификации и этот класс реализует интерфейс AuthProvider, который есть в Spring Security. В этом интерфейсе описаны два метода, которые мы реализовали в классе **AuthProviderImpl**.   
В нашем классе **AuthProviderImpl** мы используем сервис PersonDetailsService потому-что мы обращаемся к базе данных и и щем пользователя по его имени

В сервисе **PersonDetailsService** есть всего лишь один метод, который называется ***loadUserByUsername()***. Этот сервис не похож на другие сервисы, потому-что в этом сервисе мы реализуем интерфейс **UserDetailsService**.

В сервис **PersonDetailsService** внедрен JpaRepository. Через JpaRepository мы делаем запрос к базе данных, получаем человека по имени и получаем объект Person. Затем в сервисе **PersonDetailsService** объект Person оборачиваем в наш класс **PersonDetails**.

**PersonDetails** – является обёрткой над нашим человеком и он реализует интерфейс **UserDetails**, в котором содержаться разные методы.

**Текущая логика AuthProviderImpl**

**1.** Получаем объект с данными о пользователе из БД по имени пользователя. Тоесть мы с помощью сервиса вызываем метод ***loadUserByUsername()***, передаем имя пользователя, ищем его в базе с помощью hibernate, получаем этого пользователя , оборачиваем его в класс **PersonDetails** и возвращаем в **AuthProviderImpl** объект класса PersonDetails с найденным пользователем если он конечно есть в базе данных.

**2.** Сравниваем имя пользователя с тем именем которые было передано на форме.

**3.** Если с именем пользоватля всё впорядке, то сравниваем пароли из БД и с паролем который был передан на форме.

**4**. Если имя пользователя и пароль совпадает, то мы возвращаем объект Authentication с **Principal**`ом внутри. **Principal**`ом у нас является объект класса **PersonDetails**.

**НО**

Оказываестся, что последние три пунтка логики нашего **AuthProviderImpl** можно не реализовывать, так как они весьма стандартные. Тоесть стандартную логику сравнения имени пользователя, пароля и логику возвращения объекта за нас может реализовать Spring Security. Для этого нам нет необходимости реализовывать свой **AuthProvider,** чтобы такую стандартную логику реализовать. Единственное, что нам самим нужно сделать это получить объект с данными о пользователе из БД по имени.

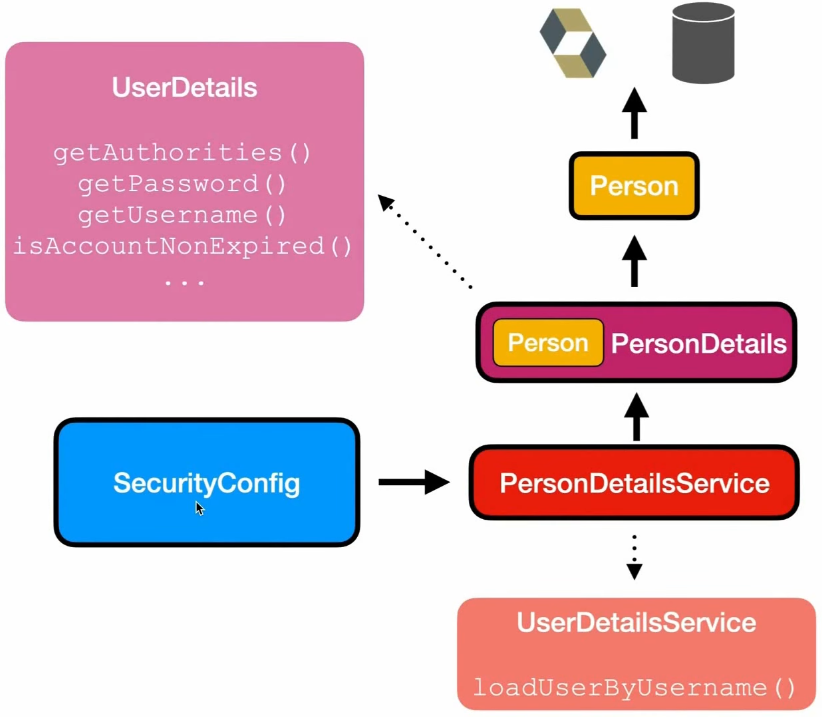
**Когда надо реализовывать свой AuthenticationProvider ?**

Если у нас простая аутентификация (с помощью логина и пароля), которая проводится на нашем сервере (логин и пароль хранится на нашем сервере), то можно использовать упрощенный вариант - без реализации собственного **AuthenticationProvider.**

Если у нас какая-то кастомная аутентификация или своя логика, то нужно реализовывать свой **AuthenticationProvider.**

Например, аутентификация может проводится на удаленном центральном сервере (CAS) .

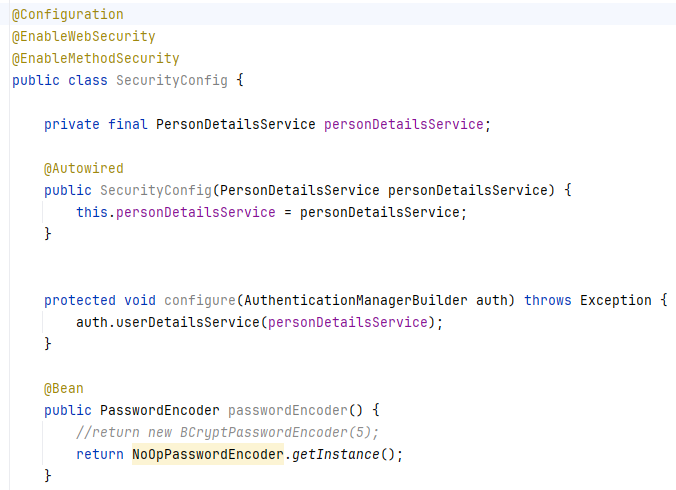
**Пример приложения без AuthenticationProvider**



Класс SecurityConfig останется, но в нём будет наш сервис (**PersonDetailsService**), который возвращает человека по имени, а всё остальное реализует Spring Security.

**1.** Просо удалим класс **AuthProviderImpl.**

**2.** Изменим SecurityConfig

****