фреймворк (система фреймворков, настолько он большой)

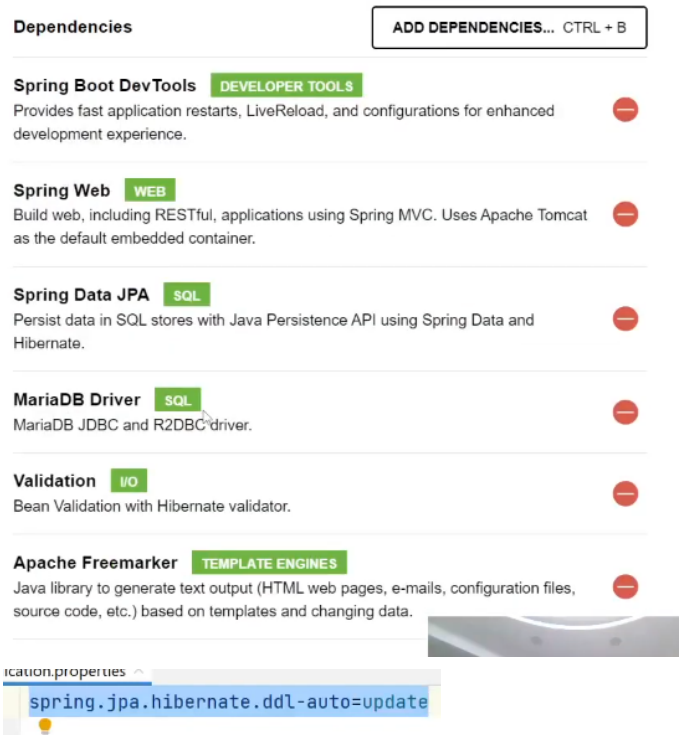
принцип convention over configuration
вместо того чтобы в конфигурационных файлах много чего настраивать мы просто называем файлы определенным образом и определенным образов кладем их в папки и все работает

spring boot -- фрейворк второго порядка, нужен для управления ("оркестрации") spring приложением, облегчает его настройку
используя spring boot можно легко построить веб приложение
еще более высокий уровень -- spring initialiser -- сайт на котором покликаешь какие нужны зависимости, он отдаст архив где будет готовое для spring boot приложение? со всеми зависимостями

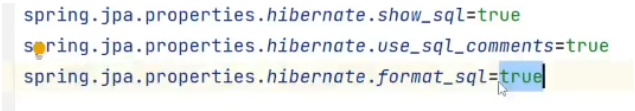
jar -- обычно самодостаточный файл который можно просто запустить и в нем уже будет какой-нибудь tomcat и он заработает (в противопоставление war)

spring web -- классическая часть спринга, позволяет создавать классические mvc приложения

JPA -- раньше мы использовали jdbc, он был довольно низкоуровневым (резалтсеты, соединения и тп).
JPA это более высокоуровневый API для взаимодействия нашего объектно-орентированного кода с персистент-слоем, в нашем случае -- реляционнной базой данных
такие вещи называются ORM
Hibernate -- одна из реализаций (стандарта) JPA



означает что схема бд будет автоматически обновляться в зависимости от декларативной разметки вокруг наших исходных текстов



-кидать нащих запросы в логи
-комментарии??
-отформатируй (чтобы нам было проще читать)

```
.datasource.url=jdbc:mariadb
.datasource.username=u00
.datasource.password=p021701
```

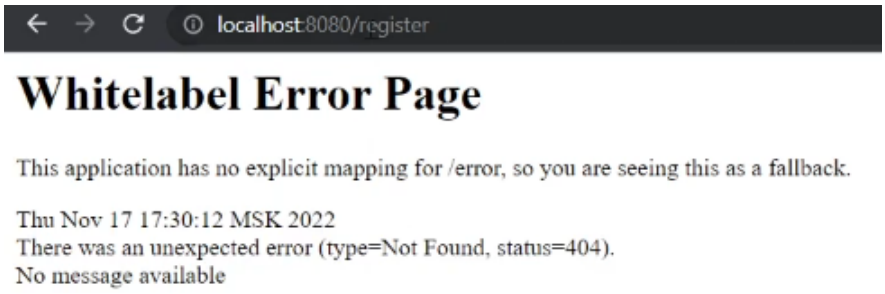
это плохо, по хорошему нужно в отдельном файлике sensitive information писать и каким-нибудь плагином сюда через переменные подгружать

```
@Controller
public class IndexPage {
}
```

соответствующий экземпляр класса будет создан чисто спрингом

```
@GetMapping({"", "/"})
public String index() {

}
```



заглушка на непомаппленный урл

```
@PostMapping("/register")
public String registerPost(
        @ModelAttribute("registerForm") UserCredentials registerForm) {
    return "RegisterPage";
}
```

ну я не совсем понял как оно само помапило переменные с запроса но короче он говорит что в запросе нам кинули аттрибуты логин пароль и что нужно чтобы их превратили (побайндили) в объект UserCredentials
типа оно работало без аннотаций ваще я вахуях
#todo почитать поэксперементировать

UPD: а, вот как:

```
<input id="login" name="login" value="${registerForm.login!}"/>
```

```
2 usages
@NotNull
@NotEmpty
@Size(min = 2, max = 20)
@Pattern(regexp = "[a-z]+", message = "Exp")
private String login;
2 usages
private String password;
```

String **message**() default "{javax.validation.constraints.Patt
Class<?>[] **groups**() default {}
Class<? extends Payload>[] **payload**() default {}

аннотации для валидации

```
@Valid @ModelAttribute("registerForm") UserCredentials registerForm) {
```

и добавили аннотацию @Valid

```
@PostMapping("/register")
public String registerPost(
        @Valid @ModelAttribute("registerForm") UserCredentials registerForm,
        BindingResult bindingResult) {
    if (bindingResult.hasErrors()) {
        return "RegisterPage";
    }

    // todo...
    return "RegisterPage";
}
```

@Valid должен быть до **@ModelAttribute**, **BindingResult** после того что мы байндим

```
<#import "/spring.ftl" as spring>
```

файл с набором специальных макросов которые обеспечивают интеграцию спринга для фримаркера

```
<#macro error field>
    <@spring.bind field/>
    <#if spring.status.errorMessage??>
        <div class="error">
            ${spring.status.errorMessage}
        </div>
    </#if>
</#macro>
```

```
    <div class="field">
        <div class="name">
            <label for="login">Login</label>
        </div>
        <div class="value">
            <input id="login" name="login"/>
        </div>
        <@error "registerForm.login"/>
    </div>
       <input id="password" name="passwor
   </div>
   <@error "registerForm.password"/>
```

```
ublic class RegisterPage {
    @GetMapping("/register")
    public String registerGet(Model model) {
        model.addAttribute( attributeName: "registerForm", new UserCredentials());
        return "RegisterPage";
    }
```

здесь model это модель для view

```
ublic class RegisterPage {
    1 usage
    private final UserService userService;

    public RegisterPage(UserService userService) {
        this.userService = userService;
    }
```

теперь добавится зависимость от UserService и спринг автоматически все разрулит

```
@Entity
public class User {
}
```

```
@GeneratedValue
```
-- сквозные айдишники по всем пользователям

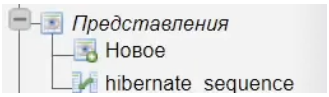
```
@CreationTimestamp
```

```
@Table(indexes = {@Index(columnList = "creationTime"),
        @Index(columnList = "login", unique = true)})
public class User {
```
-- раньше мы это делали руками

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
}
```
Long -- соответсвующий primary-ключ

```
public class UserService {
    1 usage
    private final UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
```
-- говорим спрингу что нам нужен UserRepository и даем ему простой способ его внедрить



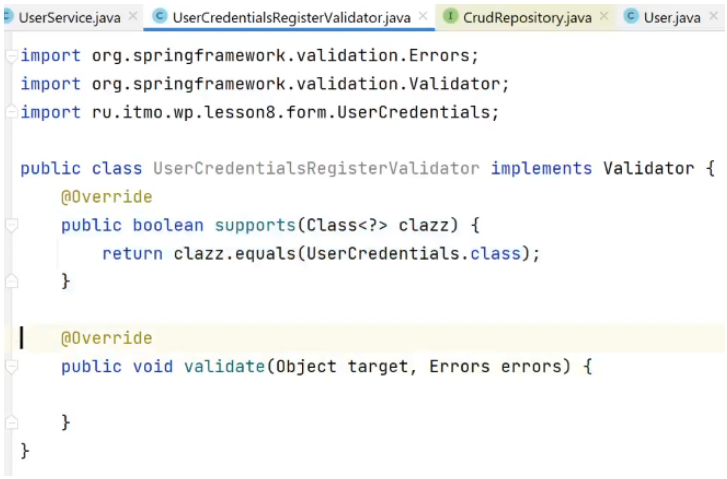
-- служебная информация, хранит возрастающую последовательность индексов

```
@Query(value = "UPDATE user SET passwordSha=SHA1(CONCAT('cddb3fc705370d0193ce', ?1, ?2
        nativeQuery = true)
void updatePassword(long id, String password);
```

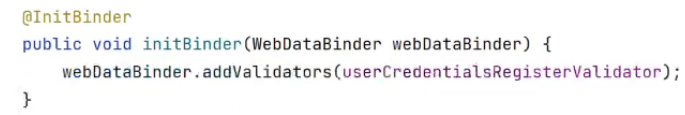
-делаем в repository метод обновления пароля, пишем запрос (декларативно)

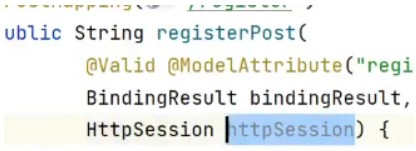
в @Repository: `long countByLogin(String login);` -- JPA сам распарсит название и сделает функцию вау вот это реально магия

@Component (он потом дописал)

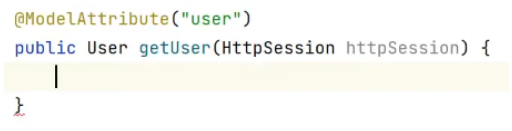


кастомный валидатор

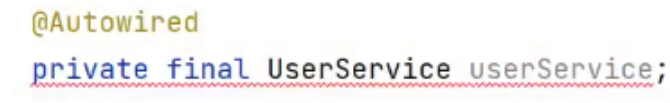




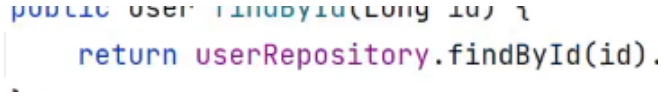
-- мы попросили сессию и нам ее дадут магия спринга



-- за счет того что мы это написали в Page этот юзер будет класться во все контроллеры что нам нужны???



-- способ сказать что поле заинжекшено



-- возвращает optional

IoC, DI (Inversion of Control, Dependency Injection)

IoC -- передача управления от кода к фреймворку
DI -- реализация IoC, например передача в конструктор интерфейса
в спринге мы при этом должны использовать атрибуты (@autowired)

хабр:

DI (Dependency Injection, внедрение зависимости) — процесс при котором построение одного объекта, предоставляется внешнему объекту. Или точнее это то место, где зависимость будет внедрена другим объектом.

https://habr.com/ru/sandbox/124305/ -- хорошие примеры мб ошибки в использовании терминов

javarush:

IoC - это принцип ООП, используемый для уменьшения связанности между классами и объектами. Программист в нужные точки программы разместит необходимый код и не волнуется как и когда должен работать размещенный код. Говоря простым языком, при использовании IoC кодом будет управлять фреймворк а не программист. **DI** - делает объекты приложения слабо зависимым друг от друга. То есть об инициализации объектов будет заботится внешний механизм разработанный программистом. При

использовании DI программист будет работать не на "уровне классов" а на "уровне интерфейсов". Таким образом зависимости между объектами будут сведены к минимуму.

dependency-injection принцип
IoC -- **inversion of control**
https://habr.com/ru/articles/131993/

Inversion of Control (IoC)

Objects do not create other objects on which they rely to do their work. Instead, they get the objects that they need from an outside source (for example, an xml configuration file).