Универзитет „Св. Кирил и Методиј" во Скопје
ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

Финален проект по предметот:
Безбедност на компјутерски системи

Тема:

Систем за издавање и верификација на OTP (TOTP/HOTP)

Фисник Лимани, 151027
fisnik.limani@students.finki.ukim.mk

Септември, 2020г.

СОДРЖИНА

Вовед

Во овој проект се имплементира еден автентикациски систем кој користи корисничко име и лозинка и TOTP (Time-based One-Time Password) како дво-факторска автентикација.

Овој систем се имплементира како Spring Boot апликација. Апликацијата користи Spring Security а за зачувување на податоците на регистрираните корисници се користи фајл-базирана H2 база на податоци.

Клиентот е една Angular веб апликација напишана/исходирана со TypeScript и ја користи PrimeNG UI библиотеката. Но фокусот во овој проект е Јава кодот и нема да се дискутира кодот од клиентска страна. Backendot е дизајниран така што може да работи со било каква технологија искористена на клиентската страна.

TOTP

TOTP (Time-based One-Time Password) е механизам кој се додава како втор фактор (слој) во автентикацискиот тек со помош на корисничкото име и лозинката, со цел да се зголеми безбедноста.
TOTP е алгоритам базиран на HOTP (HMAC-based One-time Password), но наместо бројач (counter) користи временски-базирана компонента.
TOTP and HOTP се зависни од една шифра (секрет) која се споделува меѓу двете страни. Шифрата е случаен изгенериран токен кој обично му се покажува на корисникот со помош на Base32 репрезентацијата.
Кога корисникот прв пат се регистрира на системот, серверот ја изгенерира шифрата, ја зачува во базата на податоци и ја покажува на корисникот. Потоа корисникот може таа шифра да ја пишува рачно или да ја копира или да ја скенира во некоја од апликациите кои се нарекуваат Автентикатори (Authenticator) кои поддржуваат TOTP.
Имаме повеќе TOTP апликации достапни за мобилните уреди, за десктоп компјутерите или за пребарувачите. Една од тие е Google Authenticator.

Демо Апликација

Во `server` именикот го имаме Spring Boot апликацијата, додека во `client` се наоѓа Angular апликацијата која може да се стартува со `ng serve` командата.
Прв пат кога се стартува *server* апликацијата, ја креира базата на податоци и ги додава следните 3 корисници:

Username	Password	Secret
admin	admin	W4AU5VIXXCPZ3S6T
user	user	LRVLAZ4WVFOU3JBF
lazy	lazy	

Демо апликацијата поддржува корисници со и без дво-факторска автентикација (2FA).
Треба да се инсталира некој TOTP автентикатор и да се креира нов влез со дадената шифра (secret). Може да се скенира изгенерираниот QR код (од шифрата) или рачно да се внесува.

Основа

Server е обична Spring Boot апликација, креирана со Spring Initializr.
Додадени се security и web зависностите (dependencies). Овие може да се видат во *pom.xml*.

Исто така е додаден и `aerogear-otp` како дополнителна зависност во проектот.

```
<!-- https://mvnrepository.com/artifact/org.jboss.aerogear/aerogear-otp-java -->
<dependency>
    <groupId>org.jboss.aerogear</groupId>
    <artifactId>aerogear-otp-java</artifactId>
    <version>1.0.0</version>
</dependency>
```

Библиотеката ни нуди методи за верификација на TOTP кодовите и изгенерирање на шифрите (secrets). Поради тоа што на овој проект се имплементирани некои дополнителни безбедносни барања, е напишан и сопствен TOTP верификатор, базиран на `aerogear-otp` кодот.

Базата на Податоци

Демо апликацијата ги чува следните информации за секој корисник:

```
@Entity
@Table(name = "app_user")
public class AppUser {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String username;
    @Column(name = "password_hash")
    private String passwordHash;
    private String secret;
    private Boolean enabled;
    @Column(name = "additional_security")
    private Boolean additionalSecurity;
```

- `username` и `password_hash` се користат како обична најава со помош на корисничкото име и лозинката (една-факторска автентикација – 1FA)

- *secret* – е потребен/задолжителен за дво-факторска автентикација со TOTP. Оваа е шифрата која клиентот и серверот треба да ја споделат. Ова споделување може да се гледа во делот на процесот за регистрација на нов корисник.
- *additional_security* е едно важно знаменце кој се користи при процесот на најавување. Првично, ова знаменце е *false*. Кога корисникот внесува погрешен TOTP код, знаменцето се поставува во *true*, и се побара една дополнителна безбедностна верификација. За ова знаменце, повеќе ќе зборуваме подолу, во делот за најавата.
- *enabled* – се користи при процесот на регистрација. Регистрацијата се состои од два чекори:
    - Прв чекор: Корисникот се додава во базата на податоци. *Но во тоа време, апликацијата не знае дали корисникот завршил со процесот на регистрација. Поради тоа, делот кој се справува со додавање на нов корисник во базата на податоци, го поставува ова знаменце на false. Бидејќи ова знаменце е false, корисникот сеуште не може да се најави успешно.*
    - Втор чекор: Кога корисникот ќе го заврши успешно процесот на регистрација (го верификува TOTP кодот за кој ќе зборуваме повеќе подолу), управувачот со ова ја претворува вредноста на ова знаменце на *true*.

Spring Security

За оваа апликација, е напишан сопствен автентикациски систем, па затоа е оневозможена авто-конфигурацијата на Spring Security од страна на Spring Boot со следниов код:

```
@Bean
@Override
protected AuthenticationManager authenticationManager() throws Exception{
    return authentication -> {
        throw new AuthenticationServiceException("Cannot authenticate " + authentication);
    };
}
```

Апликацијата сеуште го користи Spring Security за авторизација. Овој код само го оневозможува делот за автентикација на Spring Security.

Апликацијата користи *Argon2* за хеширање на лозинките:

```
@Bean
public PasswordEncoder passwordEncoder(){
    return new Argon2PasswordEncoder( saltLength: 16,  hashLength: 32,  parallelism: 8,  memory: 1 << 16,  iterations: 4);
}
```

Документацијата на Spring Security препорачува параметрите да се регулираат така што ќе му биде потребна околу една секунда за да ја верификува една лозинка во вашиот систем.

Може да забележиме дека `Argon2PasswordEncoder` е една класа која се нуди од Spring Security библиотеката, но зависи од Bouncy Castle, па затоа треба да се додава и следната зависност (dependency):

```
<!-- https://mvnrepository.com/artifact/org.bouncycastle/bcprov-jdk15on -->
<dependency>
    <groupId>org.bouncycastle</groupId>
    <artifactId>bcprov-jdk15on</artifactId>
    <version>1.66</version>
</dependency>
```

Апликацијата го конфигурира *Spring Security* со следниот код:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.csrf(customizer -> customizer.disable())
            .authorizeRequests(
            customizer -> {
                customizer
                        .antMatchers(
                                ...antPatterns: "/authenticate", "/signin", "/verify-totp",
                                "/verify-totp-additional-security", "/signup", "/signup-confirm-secret",
                                "/welcome"
                                ,"/h2-console/**"
                        ) ExpressionUrlAuthorizationConfigurer<H>.AuthorizedUrl
                        .permitAll() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
                        .anyRequest() ExpressionUrlAuthorizationConfigurer<H>.AuthorizedUrl
                        .authenticated();
            }).logout(customizer -> customizer.logoutSuccessHandler(new HttpStatusReturningLogoutSuccessHandler()));
}
```

- CSRF заштитата е оневозможена само заради демонстративни цели.
- Потоа се конфигурира една листа на ендпоинти кои немаат потреба за автентикација. Овие ендпоинти се сите кои учествуваат на процесот на регистрација и најава на корисници. Подетален опис на овие ендпоинти имаме во следните секции.
- Било кој ендпоинт кој не е излистан во овој дел, не може да се пристапува без автентикација.
- На крај, се конфигурира управувачот со одјавата. Овој управувач, како подразбирливо однесува, враќа назад барање за редирекција/пренасочување, но за SPA (single-page applications) е полесно кога ендпоинтот враќа само HTTP статус код.
  Ова се постигнува со помош на `HttpStatusReturningLogoutSuccessHandler` - овој враќа, by default, статус 200. Ова може да се промени со поставување на друг код/објект во конструкторот.

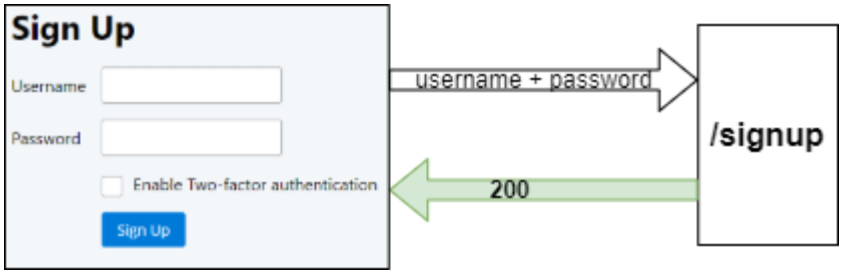
Sign Up

Процесот на регистрација се состои од три страни:

- Првата страница е за внесување на корисничкото име, лозинката и определување дали сакаме да имаме 2FA или не.
- Втората страница е за верификација на регистрацијата
- Третата страница е home-от која се покажува по успешна регистрација

Во првата страна, корисникот ги внесува корисничкото име и лозинката и дали сака да го овозможи 2FA. Ако се селектира 2FA checkbox-от, тогаш се покажува една случајна изгенерирана шифра како QR код на следната страна (1a). Потоа корисникот го додава оваа шифра во неговиот автентикатор (со рачно пишување или скенирање).
После додавањето треба да се верифицира регистрацијата со внесување на кодот на соодветното место што автентикаторот тековно го покажува/изгенерира. На крај, апликацијата прикажува една успешна порака ако внесениот код беше валиден (2).

Корисничко име и Лозинка



Клиентската апликација испраќа корисничко име, лозинка и вредностата на 2FA checkbox-от до `/signup` ендпоинтот. Овој метод прима три вредности како параметри на барањето (request parameters) и враќа назад `SignupResponse` кој се конвертира во JSON.

```
@PostMapping("/signup")
public ResponseEntity<SignupResponse> signup(
        @RequestParam("username") @NotEmpty String username,
        @RequestParam("password") @NotEmpty String password,
        @RequestParam("totp") boolean totp) {
```

`SingupResponse` се состои од овие полиња:

```
public class SignupResponse {

    private final SignupStatus status;
    private final String username;
    private final String secret;
```

```
public enum SignupStatus {
    OK, USERNAME_TAKEN, WEAK_PASSWORD
}
```

`/signup` методата прво проверува дали има некој веќе постоечки корисник со исто корисничко име. Ако да, методата враќа статус `USERNAME_TAKEN`.

```
AppUser appUser = this.appUserRepository.findByUsername(username);
if(appUser != null){
    return new SignupResponse(SignupStatus.USERNAME_TAKEN);
}
```

Потоа, се проверува ако лозинката ги следи некои конфигурирани лозински безбедносни политики. Оваа апликација, за таа намена, го користи *passpol library*.
Методата враќа статус `WEAK_PASSWORD` ако дадената лозинка не ги исполнува некои основни безбедносни проверки.

```
Status status = this.passwordPolicy.check(password);
if(status != Status.OK){
    return new SignupResponse(SignupStatus.WEAK_PASSWORD);
}
```

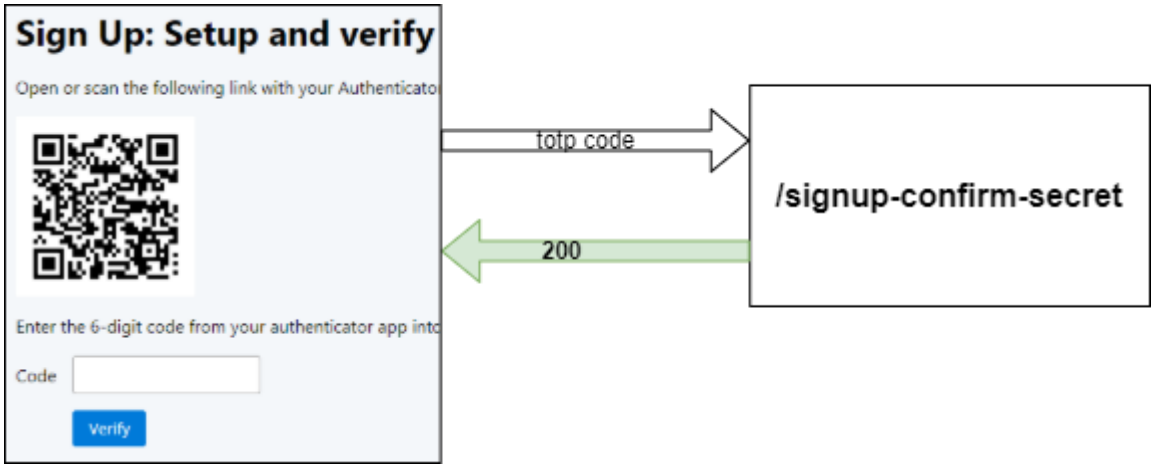
Потоа, методата треба да креира една шифра (secret) ако корисникот го има селектирано 2FA checkbox-от. Потоа дадениот корисник се додава во базата на податоци и се враќа назад еден `OK` одговор со корисничкото име и изгенерираната шифра. Може да забележиме дека апликацијата го постави `enabled` полето на `false`. Корисникот не е потполно регистриран сеуште, бидејќи треба да го валидира TOTP кодот во вториот чекор. Има и други начини како да се имплементира тоа. Можно е, на пример, да се зачуваат корисничките информации на една HTTP сесија и да се додава во базата на податоци само тогаш кога валидацијата успешно ќе помине.

```
if(totp){
    String secret = Base32.random();

    AppUser newUser = new AppUser(username, this.passwordEncoder.encode(password), secret, enabled: false, additionalSecurity: false);

    this.appUserRepository.save(newUser);

    return new SignupResponse(SignupStatus.OK, username, secret);
}
```

Ако корисникот ја оневозможил 2FA, методата го додава корисникот и го постави *enabled* полето на *true*. Потоа, корисникот без дополнителни барања може да се логира/најави.

```
AppUser newUser = new AppUser(username, this.passwordEncoder.encode(password), secret: null, enabled: true, additionalSecurity: false);
this.appUserRepository.save(newUser);

return new SignupResponse(SignupStatus.OK);
```

Верификација

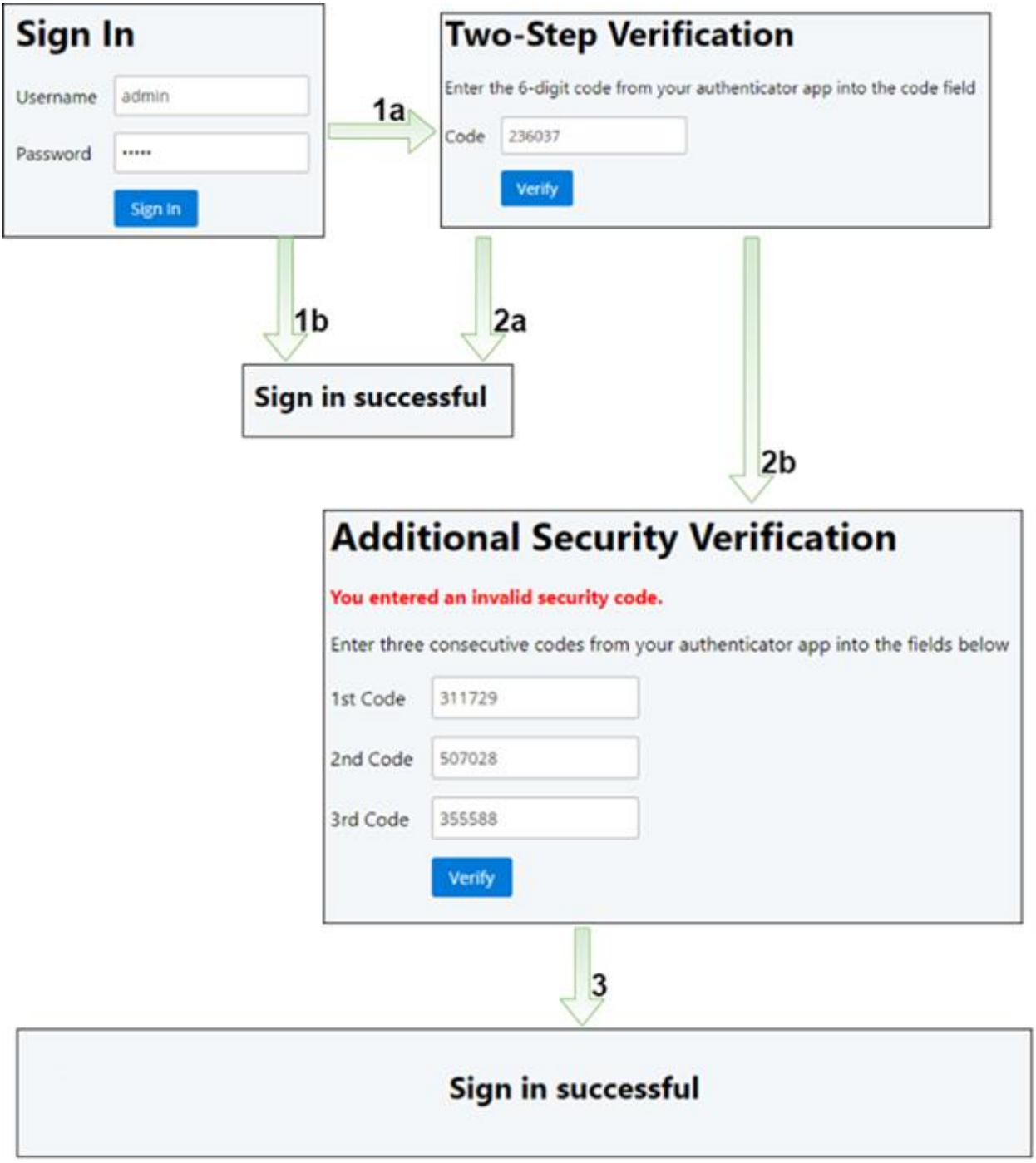


По регистрирање на новата шифра (secret) во автентикаторот, корисникот го внесува во соодветното поле тековниот TOTP код на автентикаторот. Клиентот го испраќа овој код до */signup-confirm-secret* ендпоинтот.

Методата го чита корисникот од базата на податоци и го верифицира кодот. Ако корисникот постои и TOTP кодот е валиден, методата го поставува *enabled* полето на записот на корисникот во базата на податоци на *true* и враќа одговор *true*.

```
public boolean signupConfirmSecret(String username, String code) {
    AppUser appUser = this.appUserRepository.findByUsername(username);
    if(appUser != null){
        String secret = appUser.getSecret();
        Totp totp = new Totp(secret);
        if(totp.verify(code)){
            appUser.setEnabled(true);
            this.appUserRepository.save(appUser);
            return true;
        }
    }
    return false;
}
```

Sign In



Процесот на најава се состои од три страници и home страницата, која се прикажува само доколку имаме успешна најава. Корисникот го внесува неговото корисничко име и лозинка на првата страница, а потоа апликацијата ги испраќа до backend-от.

Ако корисничкото име и лозинката се валидни, апликацијата го редиректира/пренасочува корисникот во следната страница каде тој треба да го внесе неговиот тековен TOTP код (1a). Корисниците без 2FA се редиректираат директно до home страницата (1b).

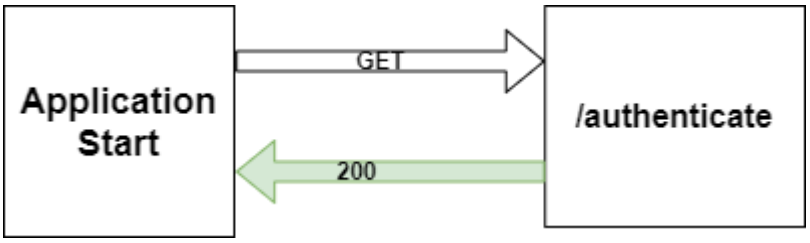
Ако TOTP кодот е валиден, апликацијата ја прикажува home стрницата (2a). Ако корисникот внесува невалиден код, апликацијата го поставува корисникот во состојба на "дополнителна верификација" поставувајќи го полето `additional_security` на базата на податоци во `true` и прикажувајќи една форма каде корисникот треба да ги внесува три последователни TOTP кодови (2b).

Целта на состојбата на "дополнителната верификација" е да ги спречува нападите со "brute force". Да замислиме дека еден напаѓач ги знае корисничкото име и лозинката на еден корисник. TOTP кодот е само шестцифрен број, па таму се 1 милион можни кодови. Кодот се променува секои 30 секунди, и оваа апликација ги смета како валидни кодовите кои што се изгенерирани до 1 минута порано и што ќе се изгенерираат до една минута во иднина, па затоа во секој момент имаме 5 валидни кодови. Еден напаѓач може само да испрати повеќе барања за да ги проба сите TOTP кодови измеѓу 000000 и 999999 и може да се случи да го погоди бараниот код и системот да го автентицира успешно. Веројатноста ова да се случи е голема.

Исто така и еден легитимен корисник може да заврши во состојба на "дополнителна верификација". Ова се случува кога се греши TOTP кодот додека се пишува или кога системскиот часовник на неговиот уред не е синхронизиран со часовникот на серверот. Оваа апликација ја дозволува разликата од +1 до -1 минута.

По внесување на три последователни кодови од страна на корисникот, серверот врши валидација на тие кодови. Оваа валидација го разгледува секој TOTP код во периодот од -25 до +25 часови. Ако тие се валидни и последователни, апликацијата го поставува знаменцето во базата на податоци во `false` и му дозволува на корисникот да продолжи понатаму во апликацијата.

Стартување на Апликацијата

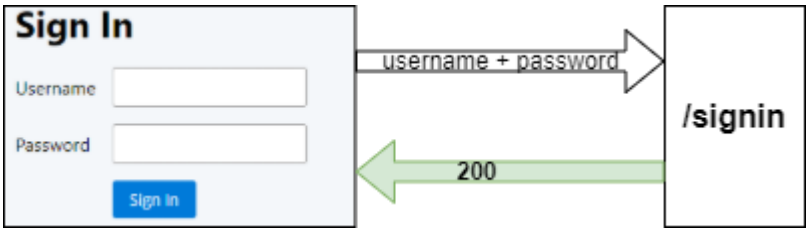


Првата акција која апликацијата ја превземува кога се стартува, е да испраќа GET барање до `/authenticate` ендпоинтот. Овој ендпоинт проверува дали корисникот е најавен или не. Зависно од одговорот на серверот, веб клиентот или го прикажува дијалогот за најава или home страницата (ако веќе е најавен).
За да го провери дали еден корисник е најавен, апликацијата го зема authentication object-от од

security context-от. Овој објект е од типот *AppUserAuthentication* и се додава во security context-от по успешен обид за најава.

```
@GetMapping("/authenticate")
public AuthenticationFlow authenticate(HttpServletRequest request){
    Authentication auth = SecurityContextHolder.getContext().getAuthentication();
    if(auth instanceof AppUserAuthentication){
        return AuthenticationFlow.AUTHENTICATED;
    }

    HttpSession httpSession = request.getSession( b: false);
    if(httpSession != null){
        httpSession.invalidate();
    }

    return AuthenticationFlow.NOT_AUTHENTICATED;
}
```

Корисничко име + Лозинка



`/signin` ендпоинтот прима корисничко име и лозинка. Прво, го зема корисничкиот запис од базата на податоци.

```
AppUser appUser = this.appUserRepository.findByUsername(username);
```

Ако корисникот постои, методата проверува ако дадената лозинка се поклопува со лозинката зачувана во базата на податоци.

```
if(appUser != null){
    boolean pwMatches = this.passwordEncoder.matches(password, appUser.getPasswordHash());
    if(pwMatches && appUser.getEnabled().booleanValue()){
```

Ако лозинката ја поминува проверката успешно, методата креира една инстанца од типот *AppUserAuthentication* и го зачува тоа во HTTP сесија кога корисникот има овозможено 2FA. Ако корисникот е во состојба на "дополнителна верификација", методата го враќа назад

стрингот *"TOTP_ADDITIONAL_SECURITY"* во телото на одговорот (response body), а ако не е во таа состојба се враќа *"TOTP"*.

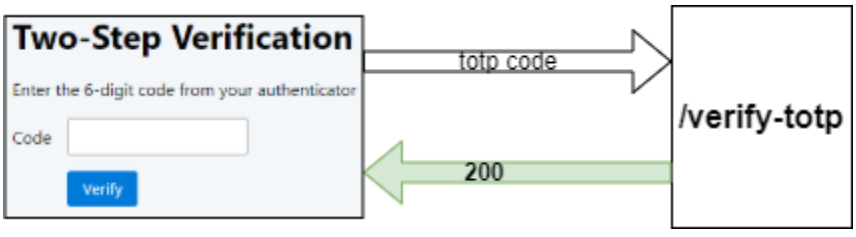
Ако корисникот нема овозможено 2FA, автентикацискиот објект се поставува во security context-от и се враќа назад стрингот *"AUTHENTICATED"*. Од тука понатаму Spring Security се справува со автентикацискиот објект со зачувување на објектот во HTTP сесија и креирање на соодветно сесиско колаче (session cookie).

```
AppUserDetail detail = new AppUserDetail(appUser);
AppUserAuthentication userAuthentication = new AppUserAuthentication(detail);
if(isNotBlank(appUser.getSecret())){
    httpSession.setAttribute(USER_AUTHENTICATION_OBJECT, userAuthentication);

    if(isUserInAdditionalSecurityMode(detail.getAppUserId())){
        return AuthenticationFlow.TOTP_ADDITIONAL_SECURITY;
    }

    return AuthenticationFlow.TOTP;
}

SecurityContextHolder.getContext().setAuthentication(userAuthentication);
return AuthenticationFlow.AUTHENTICATED;
```

Ако корисникот не постои, методата извршува енкодирање на една лажна лозинка. Ова е важно за да се скрие фактот дека корисникот не постои. Ако методата само би се вратила без овој чекор, еден напаѓач со помош на разликите помеѓу времињата на одговорот (response time) може да заклучи дали корисникот постои или не. И на крај, методата врати *"NOT_AUTHENTICATED"* во телото на одговорот.

```
else{
    this.passwordEncoder.matches(password, this.USER_NOTFOUND_ENCODED_PASSWORD);
}

return AuthenticationFlow.NOT_AUTHENTICATED;
```

TOTP кодот



/verify-totp прима TOTP код и проверува ако една инстанца на *AppUserAuthentication* објектот е зачувана во HTTP сесија. Ако таму не постои таков објект, тогаш методата го враќа стрингот *NOT_AUTHENTICATED* во телото на одговорот бидејќи тоа значи дека корисникот не беше најавен со корисничко име и лозинката, туку пристапи директно на овој ендпоинт.

```
AppUserAuthentication userAuthentication = (AppUserAuthentication) httpSession.getAttribute(USER_AUTHENTICATION_OBJECT);

if(userAuthentication == null){
    return AuthenticationFlow.NOT_AUTHENTICATED;
}
```

Следно, методата треба да провери ако корисникот е во состојба на "дополнителна верификација". Ова е, како што беше објаснувано порано, да ги спречи нападите со brute force. Ако корисникот е на оваа состојба, методата го враќа стрингот *"TOTP_ADDITIONAL_SECURITY".*

```
AppUserDetail detail = (AppUserDetail) userAuthentication.getPrincipal();
if(isUserInAdditionalSecurityMode(detail.getAppUserId())){
    return AuthenticationFlow.TOTP_ADDITIONAL_SECURITY;
}
```

Ако корисникот не е во состојба на "дополнителна верификација", се верифицира дадениот TOTP код со тајната (secret) веќе зачувана во базата на податоци. Ако кодот е валиден, методата ја постави *AppUserAuthentication* инстанцата во security context-от и враќа назад *"AUTHENTICATED".*

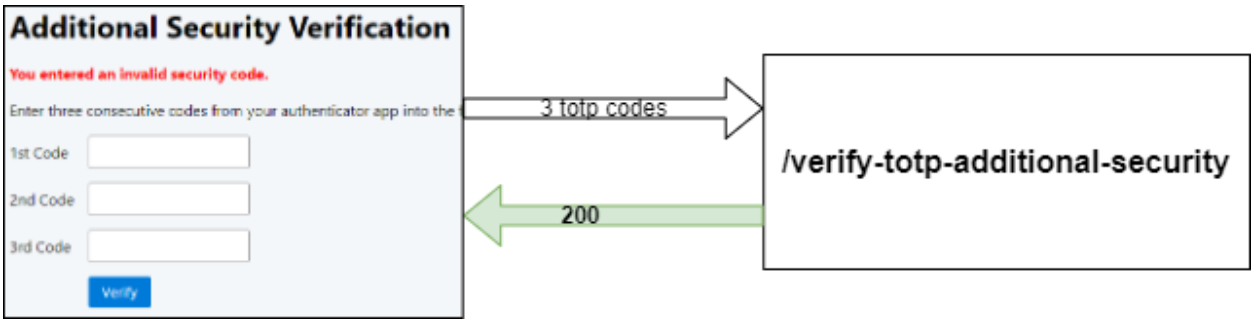
Ако кодот е невалиден, методата го постави корисникот во состојба на "дополнителна верификација" поставувајќи го на *true additional_security* полето во корисничкиот запис, и потоа го враќа назад одговорот *"TOTP_ADDITIONAL_SECURITY".*

```
String secret = ((AppUserDetail) userAuthentication.getPrincipal()).getSecret();
if(isNotBlank(secret) && isNotBlank(code)){
    CustomTotp totp = new CustomTotp(secret);
    if(totp.verify(code, pastIntervals: 2, futureIntervals: 2).isValid()){
        SecurityContextHolder.getContext().setAuthentication(userAuthentication);
        return AuthenticationFlow.AUTHENTICATED;
    }

    setAdditionalSecurityFlag(detail.getAppUserId());
    return AuthenticationFlow.TOTP_ADDITIONAL_SECURITY;
}

return AuthenticationFlow.NOT_AUTHENTICATED;
```

Со вториот и третиот аргумент на методата *verify()*, се овозможува да се подесува колку интервали (по 30-секунди) методата треба да провери во минатото и во иднината за да го

валидира кодот. Во оваа апликација се конфигурира да се провери 2 интервали во минатото и 2 во иднината.

Верификација на "Дополнителна Безбедност"



Ендпоинтот `/verify-totp-additional-security` прима три TOTP кодови како параметри на барањето и проверува ако некоја инстанца на *AppUserAuthentication* објектот е зачувана во HTTP сесија. Ако не, тогаш корисникот не беше најавен со корисничко име и лозинка и методата врати `"NOT_AUTHENTICATED".`

```
AppUserAuthentication userAuthentication = (AppUserAuthentication) httpSession.getAttribute(USER_AUTHENTICATION_OBJECT);
if(userAuthentication == null){
    return AuthenticationFlow.NOT_AUTHENTICATED;
}
```

Следно, методата треба да провери ако трите кодови се валидни и последователни. Тој го прави тоа со помош на `verify()` методата на класата `CustomTotp`. Оваа метода ги прима кодовите во една List податочна структура како прв аргумент. Вториот и третиот аргумент го дефинираат бројот на интервалите (по 30-секунди) кои методата треба да ги провери. Овој пример оди назад 25 часови и напред 25 часови.

```
if(code1.equals(code2) || code1.equals(code3) || code2.equals(code3)){
    return AuthenticationFlow.NOT_AUTHENTICATED;
}

String secret = ((AppUserDetail) userAuthentication.getPrincipal()).getSecret();
if(isNotBlank(secret) && isNotBlank(code1) && isNotBlank(code2) && isNotBlank(code3)){
    CustomTotp totp = new CustomTotp(secret);

    // check 25 hours into the past and future.
    long noOf30SecondsIntervals = TimeUnit.HOURS.toSeconds( duration: 25) / 30;
    Result result = totp.verify(List.of(code1, code2, code3), noOf30SecondsIntervals, noOf30SecondsIntervals);
```

Методата `verify()` ги враќа две вредности, една boolean (valid) која кажува ако кодовите се валидни. Валидни значи дека методата ги најде трите кодови во дадениот временски интервал и тие се последователни.
Ако беа валидни, методата исто така враќа еден integer кој кажува бројот на интервали (по 30-секунди) колку кодовите се во минатото или во иднината. Методата го зачува оваа вредност како сесиски атрибут `"totp-shift"`, ако е надвор од прифатливиот ранг од -2 до 2.

Ако трите кодови се валидни, методата го поставува *additional_security* полето во корисничкиот запис повторно на *false*, ја поставува инстанцата на *AppUserAuthentication* објектот во security context-от и врати назад *"AUTHENTICATED".*

Ако трите кодови се невалидни, методата само врати *"NOT_AUTHENTICATED",* без никакви понатамошни акции.

```
    if (result.isValid()) {
        if (result.getShift() > 2 || result.getShift() < -2) {
            httpSession.setAttribute( s: "totp-shift", result.getShift());
        }

        AppUserDetail detail = (AppUserDetail) userAuthentication.getPrincipal();
        clearAdditionalSecurityFlag(detail.getAppUserId());
        httpSession.removeAttribute(USER_AUTHENTICATION_OBJECT);

        SecurityContextHolder.getContext().setAuthentication(userAuthentication);
        return AuthenticationFlow.AUTHENTICATED;
    }
}
return AuthenticationFlow.NOT_AUTHENTICATED;
```