

Csoportos kiadáskövető alkalmazás fejlesztése Spring és React Native platformon

Dokumentáció

Önálló laboratórium 2 (BMEVIAUML10)

2023

Boros Gergő - IGMEF9

Az alkalmazás célja és jellemzői

Maga a program célja, hogy különböző baráti társaságok vagy csoportok egy Spring Boot és React Native alapú cross platform alkalmazás segítségével könnyedén tudjanak költségeket megosztani és tartozásaikat rendezni. Nyomon követhetőek vele az adósságok, a kiadások és a fizetések, így könnyen eldönthető, hogy ki tartozik kinek, és mennyivel.

Az alkalmazás használata

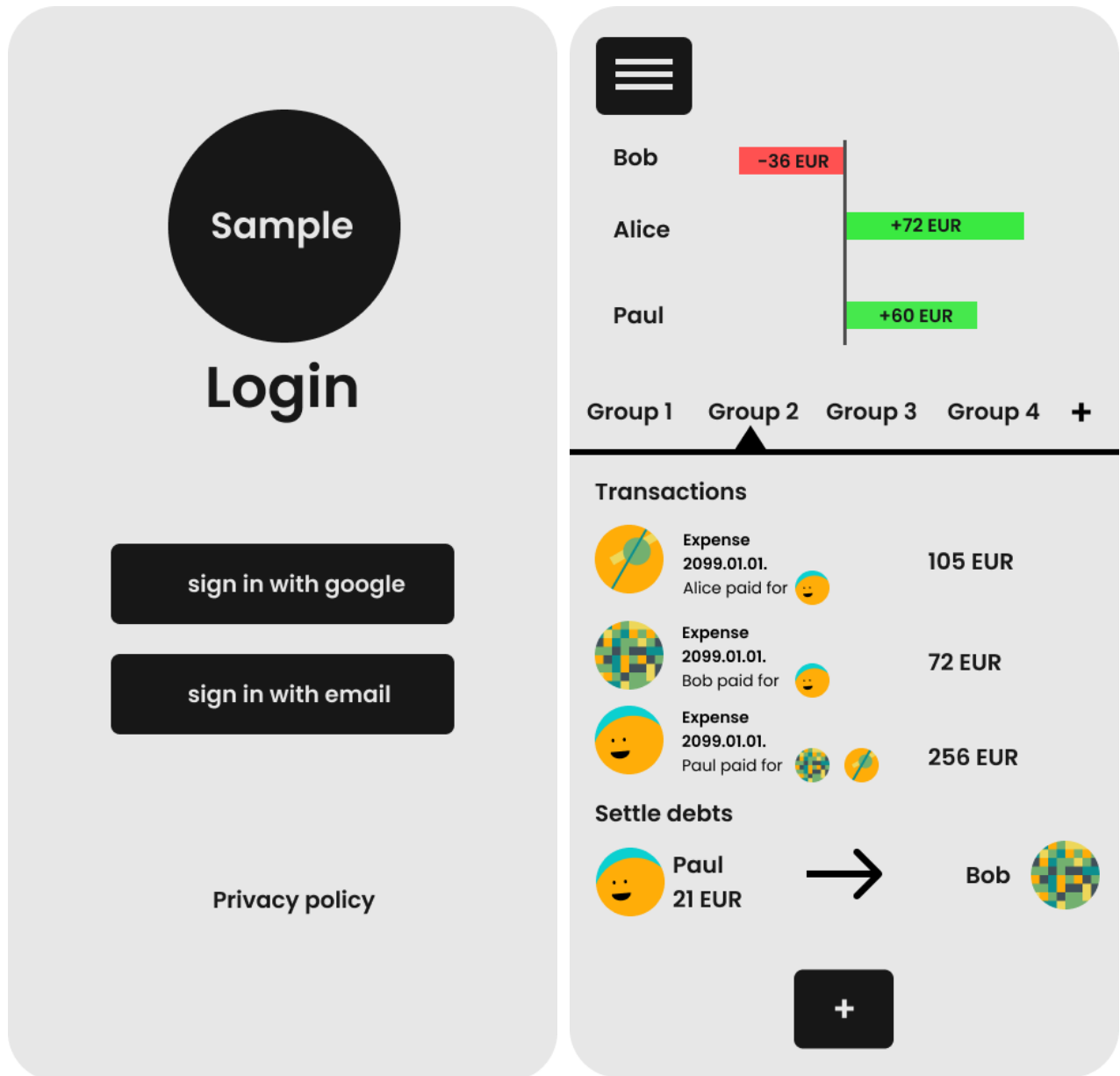
Az alkalmazás lehetővé teszi a felhasználók számára, hogy regisztráljanak egy fiókot, melybe csak az email címük megerősítése után léphetnek be. A regisztrációs folyamat kiváltható azzal, ha a felhasználók a Google fiókjukat használják a belépéshez. Hagyományos email című regisztráció esetében pedig rendelkezésre áll az elfelejtett jelszó szolgáltatás is. A felhasználói adatok titkosított kommunikációval közlekednek a kliens és szerver között. A profilunkból való kijelentkezésre, és annak szerkesztésére is lehetőség nyílik, megváltoztathatjuk az avatarunkat, nevünket.

Az alkalmazásba való bejelentkezés után a felhasználók csoportokat hozhatnak létre, melyekbe meghívhatják az ismerőseiket. Csoportba való csatlakozás után a

felhasználók adósságokat vagy éppen kifizetéseket hozhatnak létre. Az alkalmazás megosztja a költségeket, és kiszámítja, hogy kinek mennyit kell fizetnie a többieknek. Az applikáció a felhasználók közötti egyenlőtlenségeket is kezeli. Például, ha A személy B-nek 10 euróval tartozik, de B-nek C-nek 5 eurót kell fizetnie, akkor az applikáció elvégzi a szükséges számításokat, és meghatározza, hogy mennyit kell fizetniük egymásnak az adósságok rendezése érdekében.

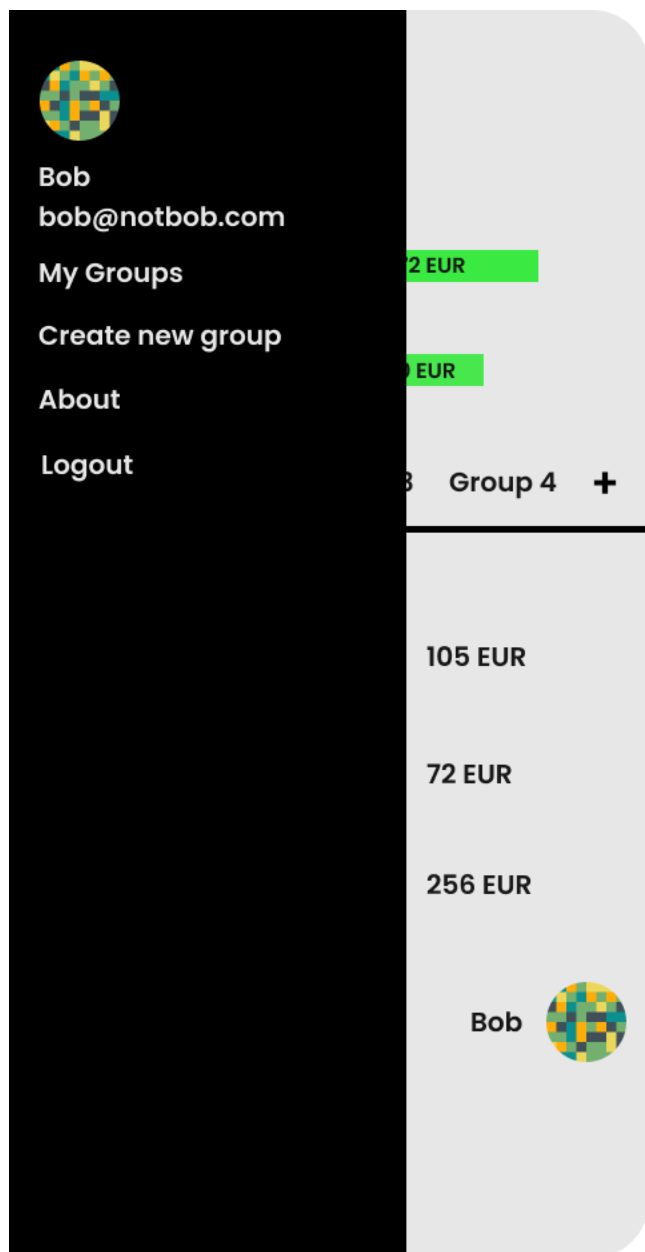
Ezekon felül pedig nyomon követhetőek csoportonként az adott események naplója, melyben megtekinthető, hogy ki küldött pénzt kinek vagy éppen ki szerzett egy újabb tartozást. Összesítve megjelenik a csoporton belüli összes kiadás mennyisége is.

Az alkalmazás mockolt wireframejei

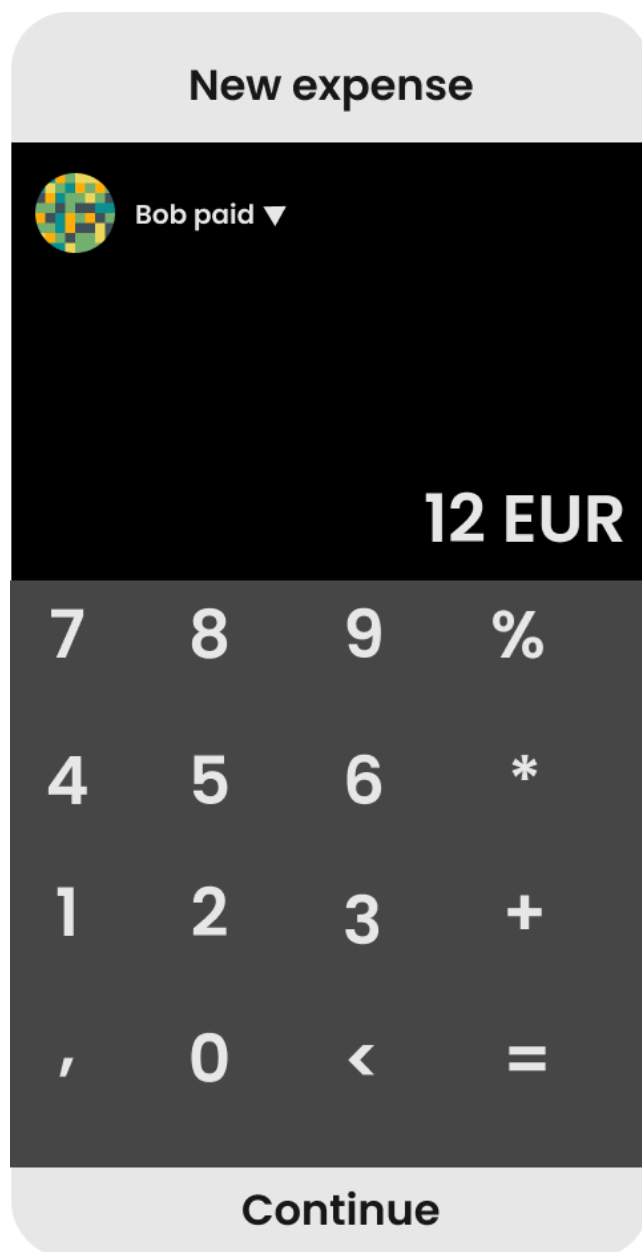


1. Bejelentkezési képernyő

2. Főmenü




3. Felhasználói akciók



4. Új adósság / kifizetés hozzáadása 1


New expense


Who paid


 Bob paid

12 EUR

For whom

 Bob
4 EUR

 Paul
4 EUR

 Alice
4 EUR



Edit ▼

Date & time

2023.03.09. 21:44

Save

5. Új adósság / kifizetés hozzáadása 2



Name

Bob

Email address

bob@notbob.com

Sign out

Delete account

6. Felhasználói profil

Az alkalmazás programozói dokumentációja

Backend

Az alkalmazás a backend oldalon a következő főbb technológiák segítségével készült el:

- **Kotlin**

- Egy erősen típusos programozási nyelv, amely a Java-val teljesen kompatibilis, null-safe és olyan modern, könnyű, tömör szintaktikát alkalmazó nyelvi elemeket is tartalmaz, mellyel könnyen kiválthatóak egy nagyobb Java projekt esetén használt könyvtárak, mint a Lombok és a Mapstruct.

- **Maven**

- Egy parancssori build automatizáló eszköz, amely igen elterjedt és könnyedén segít áthidalni a mindennapi problémáinkat a fejlesztés során. Rengeteg előnye közé tartozik, hogy plugineket importálhatunk be, erősen testreszabható, képes függősek letöltésére, akár tranzitívan is,

- **Spring Framework**

- Application framework, mely segítségével könnyen írhatunk webes alkalmazásokat Java vagy Kotlinban.
- Az alábbi modulok kerültek felhasználásra:
 - **Boot**
 - Maga a backend szerver futtatásához, működéséhez.
 - **Security**
 - Az OAuth2 és JWT alapú autentikáció, védett végpontok meghatározásához szükséges.
 - **Test**
 - Egység és integrációs tesztek írásához és futtatásához.

- Data

- Adatbázissal való kommunikáció és ORM megvalósításához.

- **OAuth2**

- Industry-standard protokoll az felhasználó azonosításhoz.

- **JWT**

- Egy nyílt szabvány alapú token alapú hitelesítési mechanizmus, amelyet gyakran használnak az alkalmazásokban a felhasználók azonosítására és az adatok biztonságos átvitelére. A tokeneket digitálisan aláírjuk, így hitelesíti és ellenőrzi az adatok integritását. A különböző védett adatokhoz a felhasználó csak úgy juthat hozzá, ha a kliens HTTP kéréseinek autorizációs fejléce tartalmazza ezt a JSON Web Token-t.

- **PostgreSQL**

- Relációs adatbázis-kezelő rendszer.

- **Thymeleaf**

- Server-side Java / Kotlin templating engine, HTML dokumentumok generálásához, melyek a felhasználónak küldött email-ekben játszanak nagy szerepet.

- **Swagger**

- Használhatjuk a REST API-nk dokumentálására vagy akár kódgenerálásra is használhatjuk. A Maven pom.xml-jében meghatározhatjuk, hogy milyen osztályokat szeretnénk legenerálni az általunk választott kliens-oldali nyelvre, jelen esetben Typescriptre. Ezekkel az entitásokkal kapcsolatban különösebb teendő nincsen, típusos osztályok jönnek létre belőlük, melyeket a frontend hasznosíthat magának. Ha azonban a REST API-nkat is szeretnénk így megvalósítani extra annotációkkal kell ellátnunk a végpontjainkat.

A backend felépítése

Config

Itt találhatóak az OAuth2, JWT és további Security-hez köthető konfigurációs osztályok. Ennek talán legelengedhetetlenebb 2 része a SecurityFilterChain, amely a bejövő hívások ellenőrzését kezeli különböző szűrők összefűzésével, illetve a CORS Configuration, amely korlátozza ezeknek a kéréseknek az erőforrásokhoz való hozzáférését:

```
@Bean
fun filterChain(http: HttpSecurity): SecurityFilterChain {
    http
        .cors().and() HttpSecurity!
        .csrf().disable()
        .authorizeHttpRequests() AuthorizeHttpRequestsConfigurer<HttpSecurity!>.AuthorizationManagerRequestMatcherRegistry!
        .requestMatchers( ...patterns: "/api/auth/**").permitAll()
        .anyRequest().authenticated().and() HttpSecurity!
        .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS).and()
        .authenticationProvider(authenticationProvider)
        .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter::class.java)
        .logout() LogoutConfigurer<HttpSecurity!>!
        .logoutUrl("/api/auth/logout")
        .addLogoutHandler(logoutHandler)
        .logoutSuccessHandler { request: HttpServletRequest?,
                               response: HttpServletResponse?,
                               authentication: Authentication? -> SecurityContextHolder.clearContext() }

    return http.build()
}

@Bean
fun corsConfigurationSource(): CorsConfigurationSource? {
    val configuration = CorsConfiguration()
    configuration.allowedOrigins =
        listOf("http://localhost:19006", "http://127.0.0.1:19006", "http://[::1]:19006")
    configuration.allowedMethods = listOf("GET", "POST", "PUT", "DELETE")
    configuration.allowCredentials = true
    configuration.allowedHeaders =
        listOf("Authorization", "Cache-Control", "Content-Type", "X-Requested-With")
    val source = UrlBasedCorsConfigurationSource()
    source.registerCorsConfiguration( pattern: "**", configuration)
    return source
}
```


Domain

Ebben a csomagban találhatóak az alkalmazást felépítő entitások halmaza, név szerint a fontosabb elemek:

- **UserEntity**

- A regisztrált felhasználók adataiért kezelős. A Third-Party Google bejelentkezést használó felhasználók ugyanolyan tulajdonsággal rendelkeznek, mint akik az alkalmazásban regisztráltak és bejelentkeztek.

- **Token**

- JWT alapú autentikációhoz szükséges. Ilyen Token-t kap a felhasználó a bejelentkezés után, vagy email-ben, amikor a regisztráláskor aktiválnia kell a felhasználóját, továbbá még akkor is, amikor az elfelejtett jelszó funkciót használja.

- **Wallet**

- Egy felhasználóhoz csatolt, különböző csoportokhoz tartozó egyenlegeket tárolja, melyekkel a felhasználó rendelkezik, ugyanis minden csoporthoz külön egyenleg társul.

- **Balance**

- A felhasználó egy csoporton belüli egyenlegét jellemzi. Lehet pozitív vagy negatív érték, többféle valutában.

- **GroupEntity**

- A csoportokat írja le, melyekhez csatlakoznak a felhasználók és különböző műveleteket végezhetnek el benne, természetesen egymás között. Egy felhasználó egy belépő link-el csatlakozhat hozzá, vagy saját maga is kreálhat egyet, melynek automatikusan tagjává válik

- **Transaction**

- Ez az entitás írja le a különböző pénzügyi műveletek felhasználók között egy csoportban. Három fajtáját különböztet meg, melyeknek megfelelően változik a tranzakcióban résztvevő felhasználók egyenlege:

- **Kiadás**

- Meghatározható, hogy ki fedezte a kiadást, kinek a részére.
Mindkettő halmaz több fős is lehet, tehát több ember is fizethet egy kiadásért, több ember számára.

- **Bevétel**

- A kiadás megfordítottja.

- **Átutalás**

- Egyszerű pénzáutalási műveletet jelképez 2 felhasználó között.

- **Debt**

- Az ilyen entitások jellemzik a felhasználók egymás közötti adósságaikat. A backenden az ilyen adóssági relációkból egy gráf épül, melyet egy rekurzív algoritmussal egyszerűsít.

Repository

A tényleges adatbázissal való interakciókért felelős réteg az alkalmazásban. Minden felsorolt entitáshoz tartozik egy Repository interfész is, amely felelős ahhoz az entitáshoz tartozó adatbázis (CRUD) műveletekért. A réteg az alkalmazás többi rétegével, például a szolgáltatás réteggel (service layer) együttműködve valósítja meg az adatelérést és a logikai műveleteket. A service réteg hívja meg a repository réteget az adatok eléréséhez és az üzleti logika végrehajtásához.

A réteg könnyen implementálható és tesztelhető a Spring Data és a Spring Boot által nyújtott funkciók segítségével.

Service

A bonyolultabb üzleti logikáért felelős osztályok találhatóak a csomagban. A Service osztályok a Repository layerben található osztályokat használják fel, hogy mindenféle bonyolultabb műveletet megvalósítsanak, továbbá ebben a rétegben történik az

entitások és DTO-k közötti mappolás is. Ezeken felül a különböző frontendből jövő adatok, kérések backend számára emészthető formába átalakítása is a feladata.

DTO

Ezek olyan osztályok, melyek a kliens számára használható adatokat csomagolják össze magukba, illetve csak olyan adatot küldenek a kliensnek, amire annak szüksége van. Minden hívható REST-es végponthoz tartozik kettő darab: egy Request (Kérés) és Response (Válasz) DTO objektum. Az üres kérések és válaszok egy-egy osztályból állnak, a `CollectoroVoidReq` és `CollectoroVoidResp`-ből.

Controller

Az alkalmazás webes rétege REST architektúra stílusban íródott, így az API technikailag csak címezhető erőforrások halmaza, JSON formátummal dolgozik. Ez a legfelsőbb layer a programban, a Service rétegre támaszkodva szolgálja ki a bejövő kéréseket a frontend felől.

Minden frontend számára is fontosabb entitáshoz tartozik egy Controller osztály a webes rétegben. Az osztályok metódusai rengeteg OpenAPI-s, Swagger-es annotációval vannak ellátva a minél szebb dokumentáció érdekében.

Ha a Controller-eket a dokumentációs annotációk mellett, extra annotációkkal látjuk el, akkor rengeteg frontend osztály számára fontos kódot tudunk generálni belőle, ami rengeteg időt megspórol számunkra a fejlesztés későbbi szakaszaiban.

```

20     @Api(value = "TransactionController", description = "REST APIs related to Transaction")
21     @Path("/api/transaction")
22     @RestController
23     @RequestMapping("/api/transaction")
24     class TransactionController(
25         private val transactionService: TransactionService
26     ) {
27
28         @POST
29         @PostMapping("/processTransaction")
30         @Path("/processTransaction")
31         @Produces("application/json")
32         @ApiOperation(value = "processTransaction", response = ProcessTransactionResp::class)
33         fun processTransaction(
34             @RequestBody @NotNull @ApiParam(
35                 required = false,
36                 value = "ProcessTransactionReq"
37             ) request: ProcessTransactionReq
38         ): ResponseEntity<ProcessTransactionResp> {
39             return ResponseEntity.ok(transactionService.processTransaction(request))
40         }

```

Ezen annotációk segítségével a swagger-maven-plugin előállít egy swagger.json-t, mely tartalmazza az összes generálandó osztály leírását.

```

308   "/api/transaction/processTransaction" : {
309     "post" : {
310       "tags" : [ "TransactionController" ],
311       "summary" : "processTransaction",
312       "description" : "",
313       "operationId" : "processTransaction",
314       "produces" : [ "application/json" ],
315       "parameters" : [ {
316         "in" : "body",
317         "name" : "body",
318         "description" : "ProcessTransactionReq",
319         "required" : false,
320         "schema" : {
321           "$ref" : "#/definitions/ProcessTransactionReq"
322         }
323       } ],
324       "responses" : {
325         "200" : {
326           "description" : "successful operation",
327           "schema" : {
328             "$ref" : "#/definitions/ProcessTransactionResp"
329           }

```

Ebből a JSON-ból pedig a megfelelően konfigurált swagger-codegen elvégzi a kódgenerálást a végpontokra, melyeket használhat a kliens.

```

/**
 * TransactionControllerApi - factory interface
 * @export
 */
export const TransactionControllerApiFactory = function (configuration?: Configuration, basePath?: string, axios?: AxiosInstance) {
  return {
    /**
     *
     * @summary processTransaction
     * @param {ProcessTransactionReq} [body] ProcessTransactionReq
     * @param {*} [options] Override http request option.
     * @throws {RequiredError}
     */
    async processTransaction(body?: ProcessTransactionReq, options?: AxiosRequestConfig): Promise<AxiosResponse<ProcessTransactionResp>> {
      return TransactionControllerApiFactory(configuration).processTransaction(body, options).then((request) => request(axios, basePath));
    },
  };
};

```

Naplózás

Mindenképp szerettem volna értelmezhető logolást implementálni az alkalmazás backend-jébe, hogy valós időben lehessen követni a beérkező kéréseket és emberi szem számára is könnyen értelmezhető legyen. Ezt a fajta naplózást az AOP megközelítéssel végeztük el. Két lényeges fogalmat érdemes ismerni a megoldás bemutatása előtt:

- **join point:** a program végrehajtásának egy lépése, például egy metódus végrehajtása, vagy kivételkezelés
- **pointcut:** egy predikátum, amely illeszkedik a join point-okra, a pointcut expression nyelvvel lehet leírni

Kettő fontos logikai egységre bonthatjuk a kliens kommunikációját a backenddel, ezek a kérés és válasz. Mindkettőnek minden egyes lépésébe szeretnénk betekintést nyerni. Korábban meghatároztuk, hogy mit jelent egy join point, azonban az alkalmazásnak fogalma sincs, hogy mikor csináljon mit. Itt nyer fontos szerepet ismét két fontos fogalom:

- **aspect:** egy cross-cutting concern, keresztthivatkozás modularizációja
- **advice:** egy művelet egy aspect által egy bizonyos join point-nál

Ezeknek az ismeretében létrehoztunk egy join point-ot, majd meghatároztuk, hogy egy kérés vagy válasz esetén feltétlenül szeretnénk információt kapni az alábbiakról:

- a join point
- a hívott osztály
- annak metódusa
- a metódus argumentumai

A kéréseket és válaszokat naplózó implementáció mellett az alkalmazás az összes végpont végrehajtási idejét is logolja.

Test

Itt találhatóak az üzleti logikai réteghez írt tesztek. A teszteléshez a MockK, Kotlinhoz használt tesztelési könyvtárat használtam. Mind a Repository rétegbeli és a Service rétegbeli osztályok is tesztelve lettek. Egy Repository-hoz tartozó teszt a következőképpen néz ki:

```

24     @Test
25     fun testFindAllNotExpiredOrRevokedTokenByUserEntityAnAndTokenType() {
26         // Create a UserEntity and save it to the repository
27         val userEntity = UserEntity.Builder().build()
28         val savedUserEntity = userRepository.save(userEntity)
29
30         // Create a Token and associate it with the UserEntity
31         val token = Token.Builder()
32             .token("token123")
33             .tokenType(TokenType.BEARER)
34             .userEntity(savedUserEntity)
35             .build()
36         tokenRepository.save(token)
37
38         // Call the repository method to find tokens
39         val foundTokens = tokenRepository.findAllNotExpiredOrRevokedTokenByUserEntityAnAndTokenType(savedUserEntity.id, TokenType.BEARER)
40
41         // Assert that the found tokens list is not empty and contains the expected token
42         Assertions.assertFalse(foundTokens.isEmpty())
43         Assertions.assertTrue(foundTokens!!.contains(token))
44     }

```

Tipikus az Arrange, Act és Assert tesztelési mintát használtam a tesztek írása során.

A Service rétegbeli tesztek írását megelőzte a különböző függőségek mockolt injektálása a tesztelt osztályba:

```

@SpringBootTest
@Transactional
class GroupServiceTest {

    private val groupRepository: GroupRepository = Mockito.mock(GroupRepository::class.java)
    private val userRepository: UserRepository = Mockito.mock(UserRepository::class.java)
    private val balanceRepository: BalanceRepository = Mockito.mock(BalanceRepository::class.java)
    private val transactionService: TransactionService = Mockito.mock(TransactionService::class.java)

    private val groupService = GroupService(
        groupRepository,
        userRepository,
        balanceRepository,
        transactionService
    )
}

```

A függőségek megfeleltetése után már alkalmazható a jól bevált tesztelési minta egy konkrét esetben is:

```

@Test
fun testCreateGroup() {
    // Arrange
    val userEmail = "john.doe@example.com"
    val req = CreateGroupReq()
    req.userEmail = userEmail
    req.name = "Group 1"
    var user = UserEntity.Builder().id(888L).email(userEmail).build()
    var groupEntity = GroupEntity.Builder().id(888L).name("Group 1").users(mutableListOf(user)).build()
    user.groups.add(groupEntity)
    val balance = Balance(
        groupId = groupEntity.id,
        wallet = user.wallet,
        currency = Currency.HUF,
        amount = 0.0
    )

    `when`(userRepository.findByEmail(userEmail)).thenReturn(user)
    `when`(groupRepository.save(any())).thenReturn(groupEntity)
    `when`(balanceRepository.save(any())).thenReturn(balance)

    // Act
    val resp = groupService.createGroup(req)

    // Assert
    verify(userRepository).save(user)
    assertNotNull(resp.group)
    assertEquals(groupEntity, resp.group)
}

```

Frontend

Az alkalmazás kliensoldali része a következő fontosabb technológiákkal készült el.

- **Axios**
 - HTTP kliens, mely a frontend -> backend kommunikációért felelős.
- **Expo**
 - Egy nyílt forráskódú platform és fejlesztői eszközkészlet a React Native keretrendszer számára.
- **NativeBase**
 - UI könyvtár, előre felkonfigurált felületi elemeket tartalmaz.

A frontend felépítése

Shared

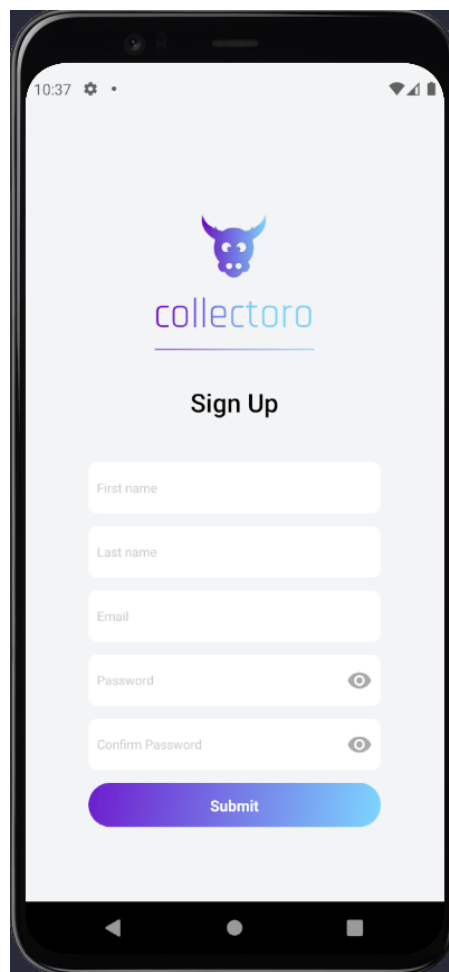
Ebben a modulban található az Axios-hoz tartozó konfigurációk, illetve a különböző komponensek megkapható paraméterlistái is.

Core

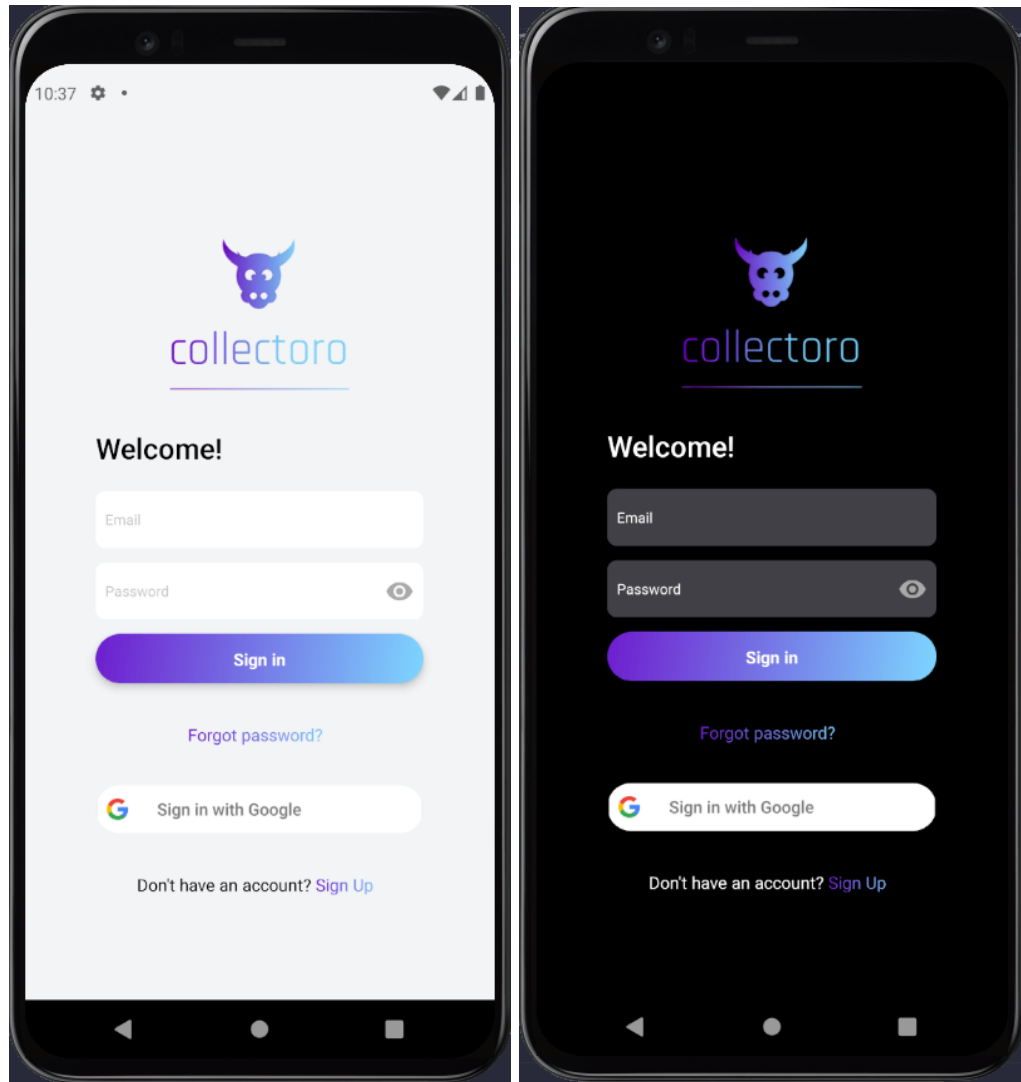
Itt találhatóak az alkalmazásban látható képernyőket jelképező komponensek, illetve a hozzá tartozó Service osztályok is, melyek a backendet hívják és szerzik meg a komponensek számára a szükséges erőforrásokat.

- **Register**

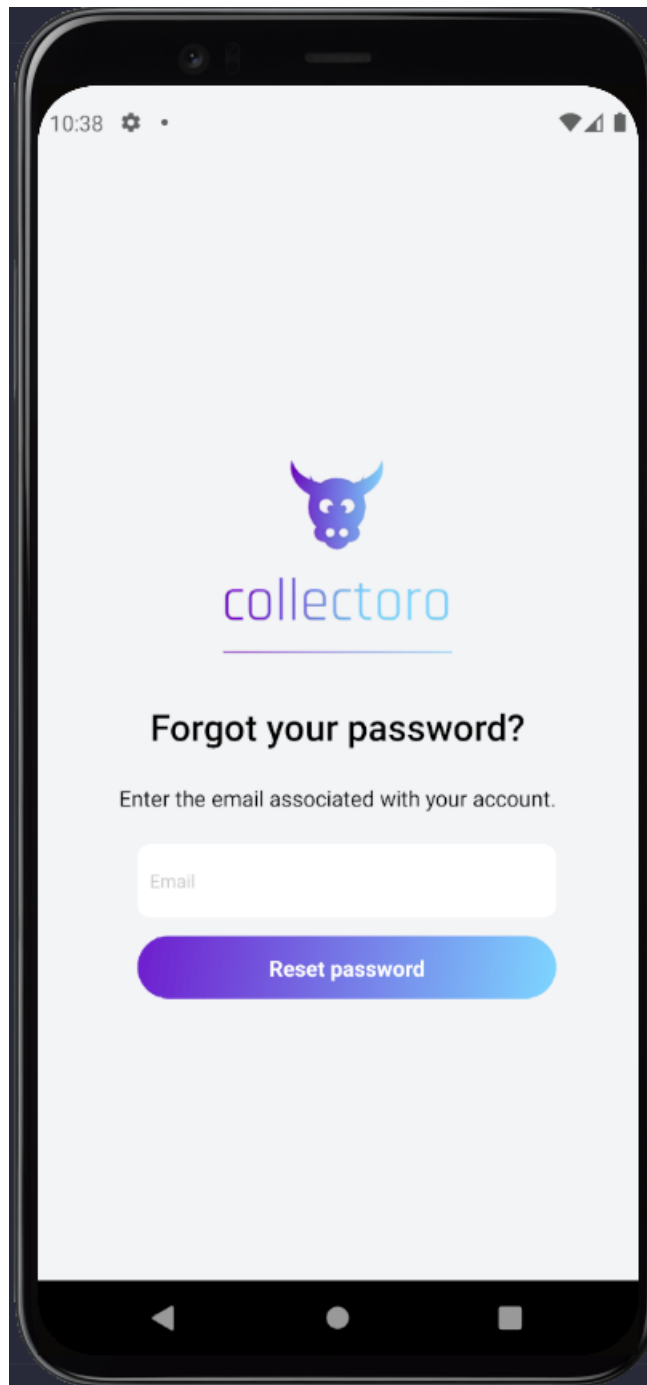
- Regisztrációhoz kapcsolódó képernyő. A felhasználó itt adhatja meg a fiókjának adatait.



- Login
 - Bejelentkezési képernyő, ahonnan kezdeményezhető az elfelejtett jelszó és regisztrálás, továbbá a Google-ös bejelentkezés is.

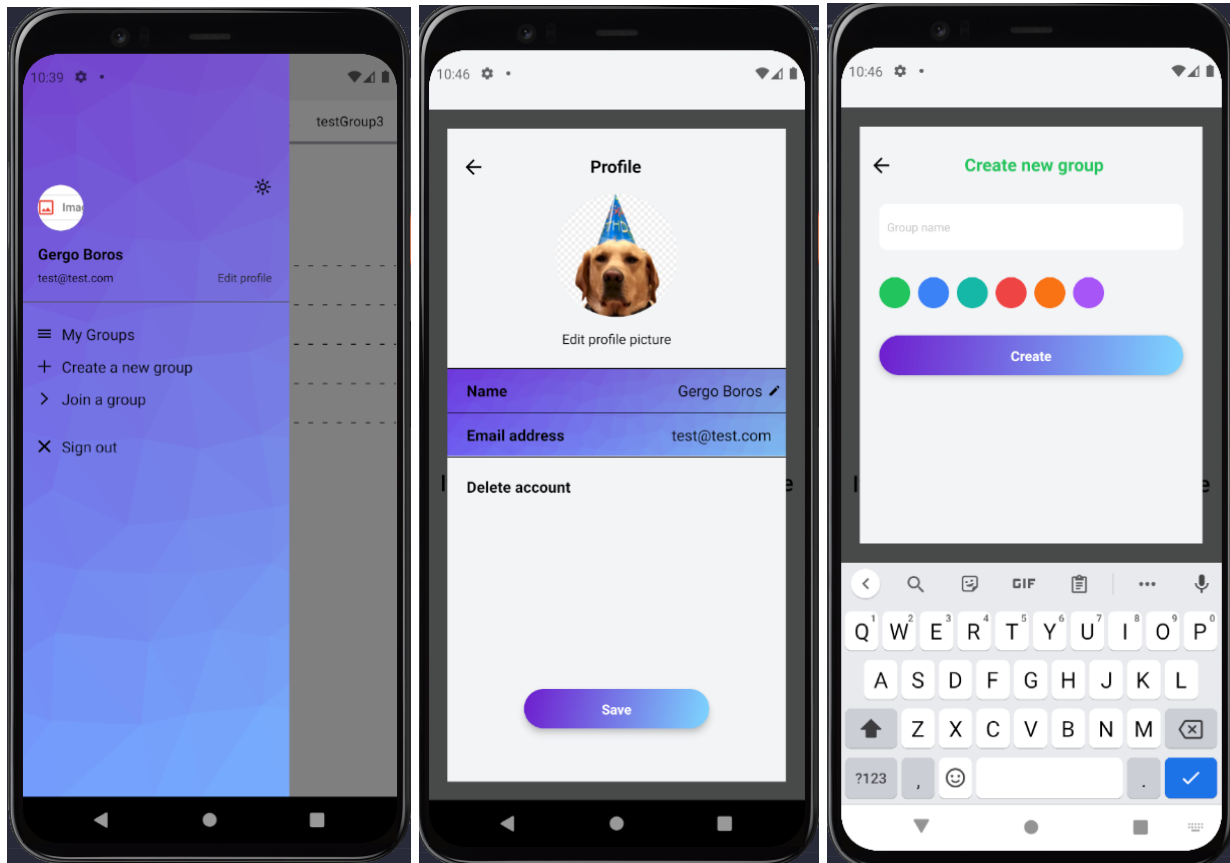


- Forgot Password
 - Elfelejtett jelszó funkcióért felelős.

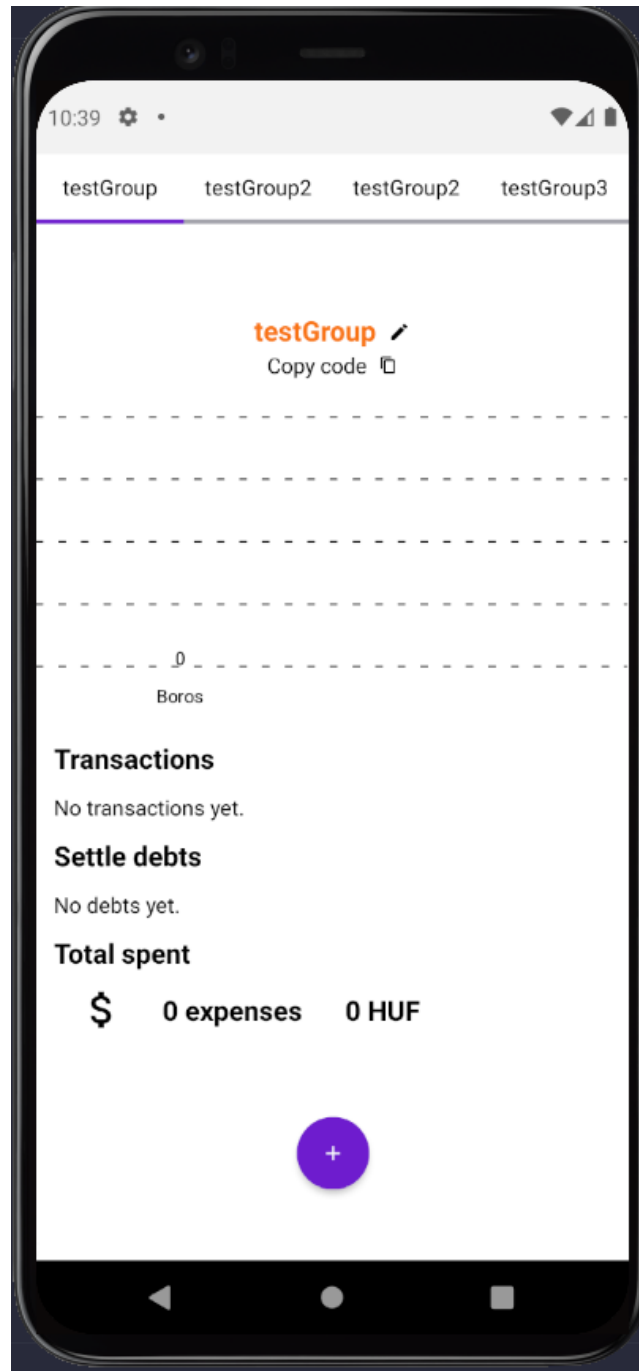


- Sidebar

- Az alkalmazás navigációs sávjáért felel. Itt kijelentkezhet a felhasználó, beléphet csoportokba, vagy készíthet egyet, illetve megtekinthető a már csatlakoztatottakat, továbbá ki is léphet belőlük. Ezen felül válthat éjszakai módra is.



- Home
 - A “fő” képernyője az alkalmazásnak, a felhasználó itt találkozhat a csoportok képernyőjével és adataival, melyekbe belépett.



- Transaction Editor
 - Tranzakciókat rögzíthetünk ezeken a képernyőn.

