

# Documento di sviluppo dell'app

<b>Scopo del documento</b>	3
<b>Funzionalità implementate</b>	3
<b>User flow</b>	4
User flow 1 - Autenticazione	4
User flow 2 - Creazione evento	4
User flow 3 - Navigazione nella homepage (o bacheca)	6
User flow 4 - Navigazione nella pagina profilo utente	7
<b>API dell'applicazione</b>	8
Estrazione risorse class diagram	8
Resource diagram	9
Resource Diagram 1	9
Resource Diagram 2	10
Resource Diagram 3	10
<b>Implementazione del backend</b>	11
Struttura del progetto	11
Dipendenze	12
Modelli nel database	13
Tabella utenti	14
Tabella per il reset della password	14
Tabella eventi	14
Tabella place per gli eventi	15
Tabella per la sessione di autenticazione degli eventi	15
Tabella flyway e migrazioni	16
Configurazione delle variabili d'ambiente	17
Script per la generazione del file .env	19
Dependency Injection	19
Web server e API	22
Configurazione del web server	22
Routing	22
Autenticazione	24
<b>Documentazione delle API</b>	27
Business logic degli endpoint	29
Monitoring metrics	29
Login	29
Password forgotten	29
Logout	32
Richiesta dell'utente loggato	32

Richiesta utente specifico	32
Interazione con PostgreSQL / Redis	32
⭐ Id tipizzati	34
Lista degli eventi	35
Ricerca di un singolo evento	35
Creazione di un evento	36
Validazione dei dati	36
Modifica di un evento	37
Eliminazione di un evento	37
<b>Testing</b>	<b>38</b>
<b>Monitoring</b>	<b>40</b>
Monitoring delle performance	40
Monitoring e tracking degli errori	41
<b>Implementazione del frontend</b>	<b>43</b>
Homepage (o bacheca)	43
Profilo utente	43
Autenticazione	43
Creazione evento	43
Dettagli evento	44
<b>Deployment e CICD</b>	<b>45</b>
Esecuzione in locale	48
Dashboard	49

# **Scopo del documento**

Il presente documento fornisce informazioni dettagliate delle risorse e dei servizi impiegati nello sviluppo di specifiche componenti dell'applicazione E-20. In esso sono delineati gli strumenti necessari alla realizzazione delle interfacce utente previste, quali la homepage (o bacheca), la schermata del profilo utente, l'interfaccia di autenticazione, la funzionalità per la creazione di un evento e la pagina di dettaglio relativa a ciascun evento.

Saranno descritti i test adottati per garantire l'affidabilità e la sicurezza dell'applicazione, assicurando che la stessa risponda efficacemente alle esigenze degli utenti e mantenga un elevato standard di performance.

Il documento include inoltre le direttive per il deployment del software, offrendo le indicazioni necessarie per l'installazione e l'esecuzione dell'applicazione in ambienti locali su diversi dispositivi, al fine di facilitarne la distribuzione e l'utilizzo.

## **Funzionalità implementate**

Solo una parte delle funzionalità descritte dai documenti precedenti sono state effettivamente implementate, questo per evitare di rendere il progetto troppo complesso ed anche per questioni di tempistiche.

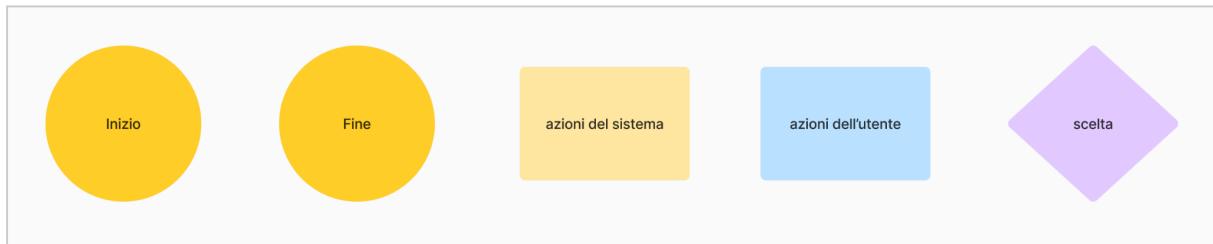
Questa la lista di funzioni implementate:

- visualizzazione del feed degli eventi (senza filtro per localizzazione)
- visualizzazione dettagli di un evento
- creazione di un evento
- visualizzazione pagina profilo
- autenticazione tramite email e password
- gestione reset della password

# User flow

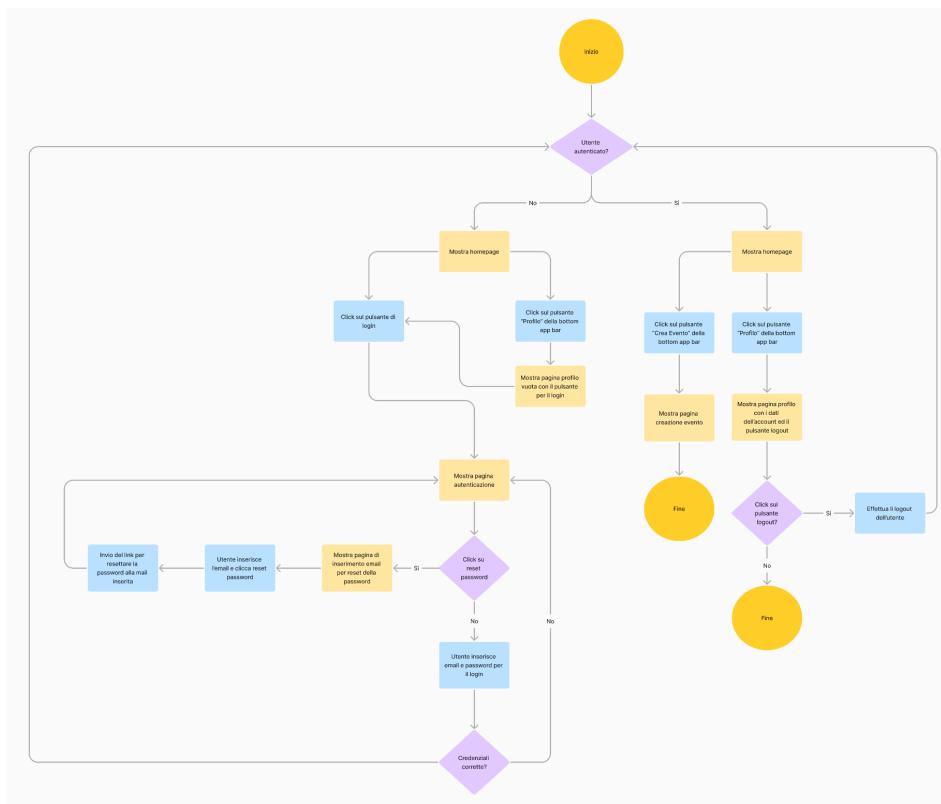
Inizieremo questo documento con lo user flow diagram, una rappresentazione grafica del percorso compiuto dagli utenti all'interno della nostra applicazione.

In questa prima immagine sono indicati gli elementi grafici dello user flow ed è annesso il loro funzionamento:



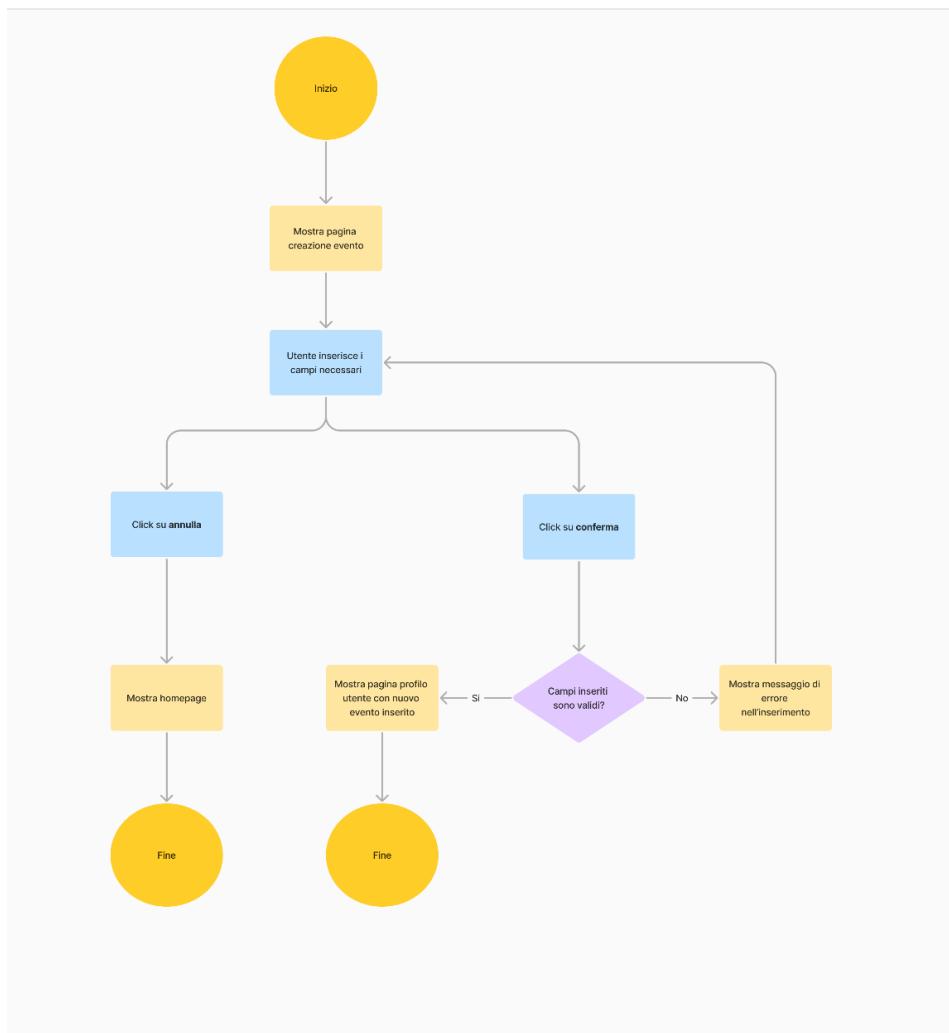
Per semplicità abbiamo deciso di partizionare il nostro user flow diagram in sotto-diagrammi in modo da favorirne la leggibilità.

## User flow 1 - Autenticazione

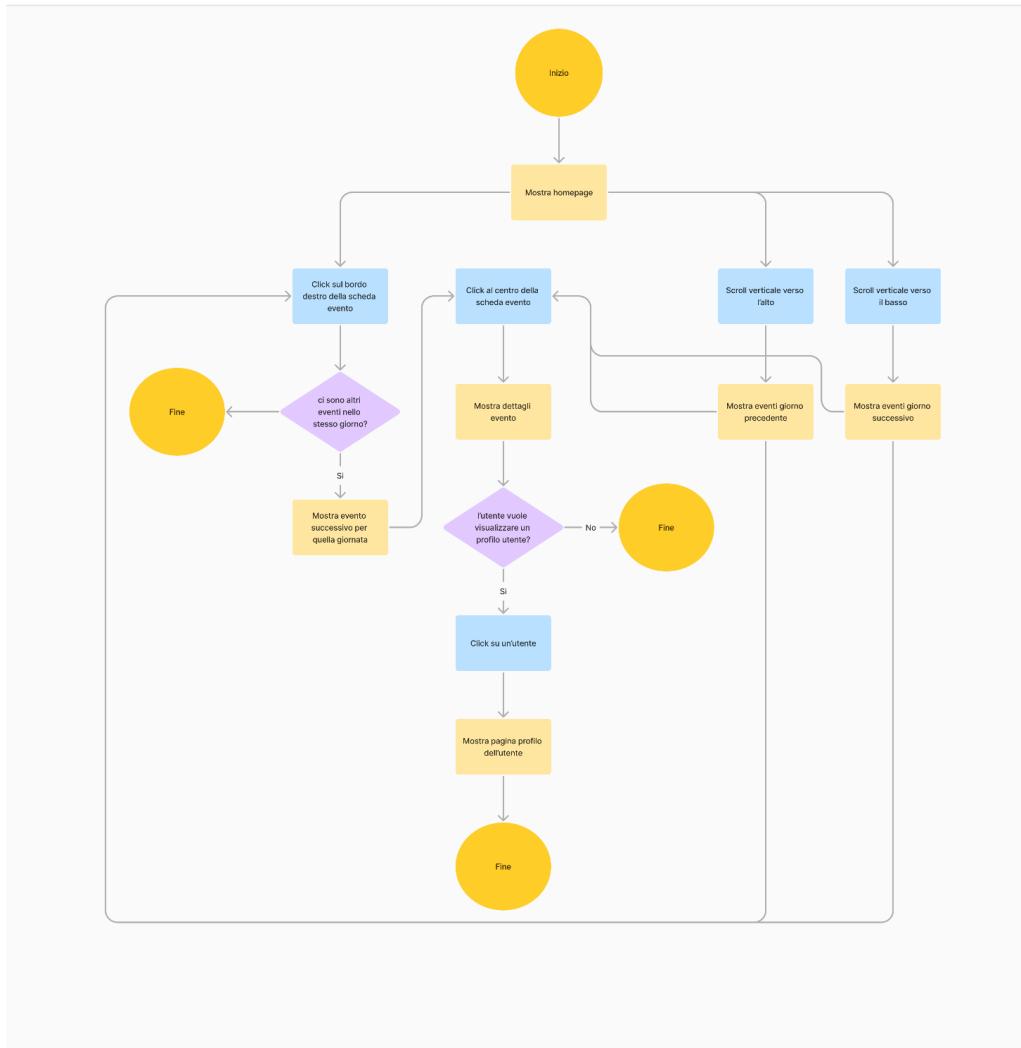


## User flow 2 - Creazione evento

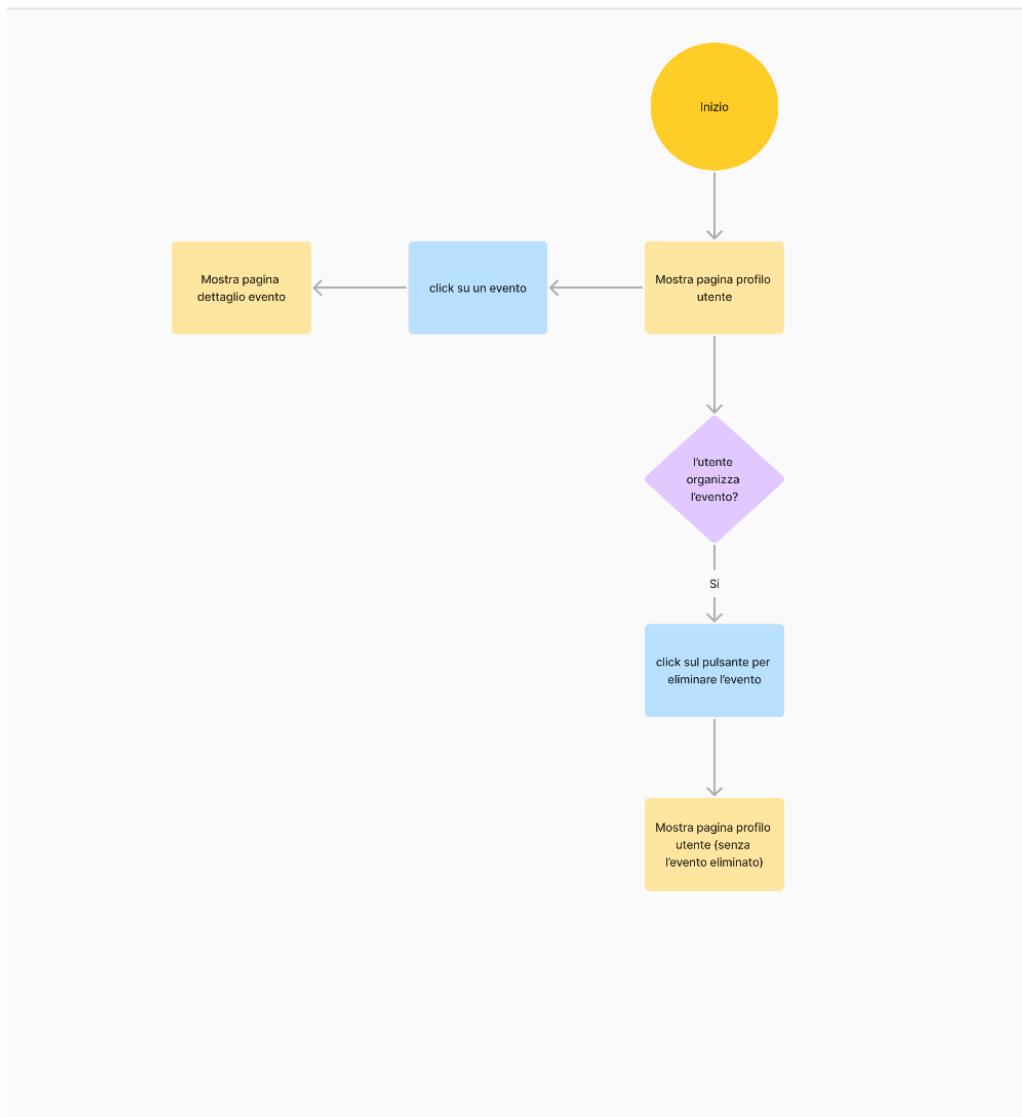
In questo user flow è necessario che l'utente sia già autenticato correttamente.



## User flow 3 - Navigazione nella homepage (o bacheca)



## User flow 4 - Navigazione nella pagina profilo utente

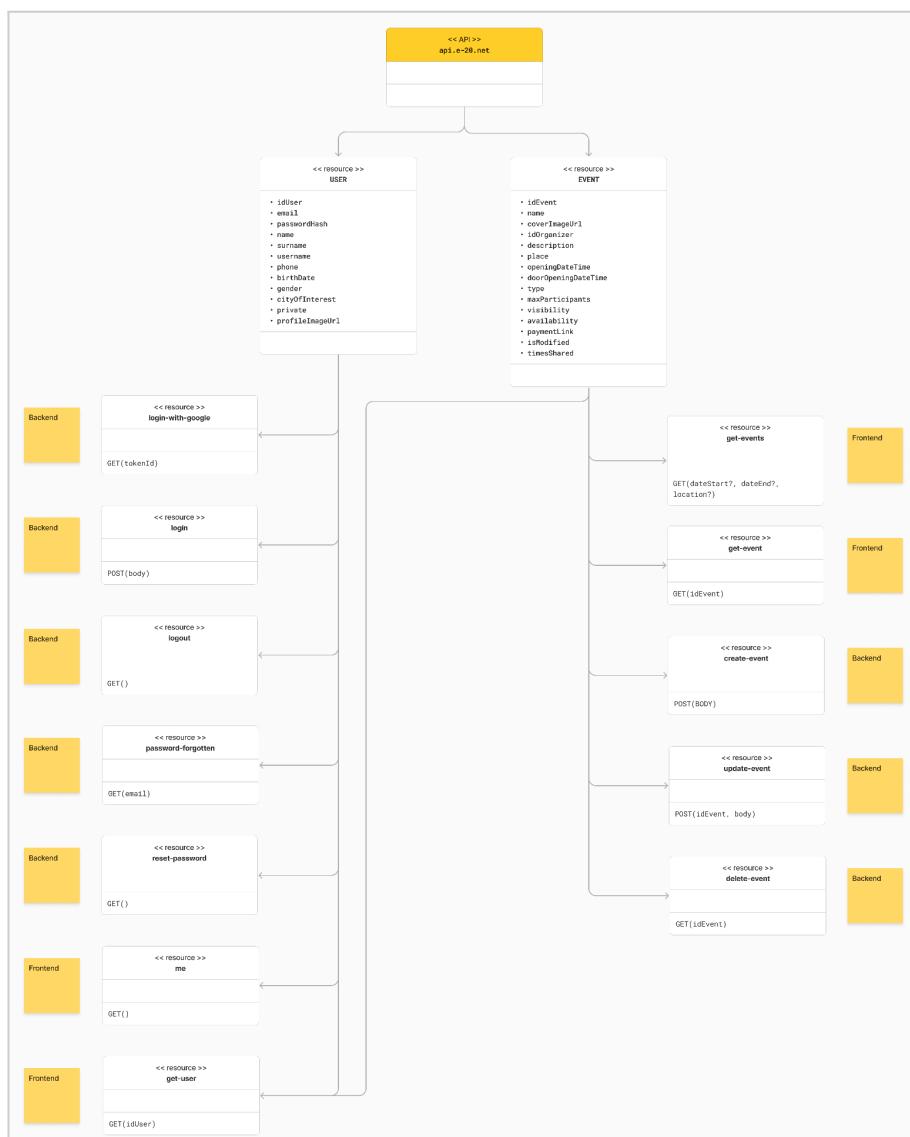


# API dell'applicazione

## Estrazione risorse class diagram

In questo paragrafo viene allegato il diagramma di estrazione delle risorse. Si tratta di un processo attraverso il quale le informazioni e le definizioni di risorse software vengono estratte da un diagramma delle classi per essere utilizzate in altre fasi dello sviluppo software, come l'implementazione, il deployment o la configurazione di un'applicazione.

Sono stati identificati e selezionati i datatype e gli attributi fondamentali che meglio si adattano alle necessità specifiche del progetto corrente, contribuendo allo sviluppo di una iterazione dell'applicazione. Per ciascuna entità e interazione delineata, è stata definita la metodologia HTTP appropriata per la comunicazione con l'endpoint corrispondente (GET, POST), selezionata in funzione degli obiettivi desiderati dell'operazione. In aggiunta, sono stati specificati i parametri necessari per ciascuna richiesta, che possono essere trasmessi tramite l'URL (per GET) o incluso nel corpo della richiesta (per POST).

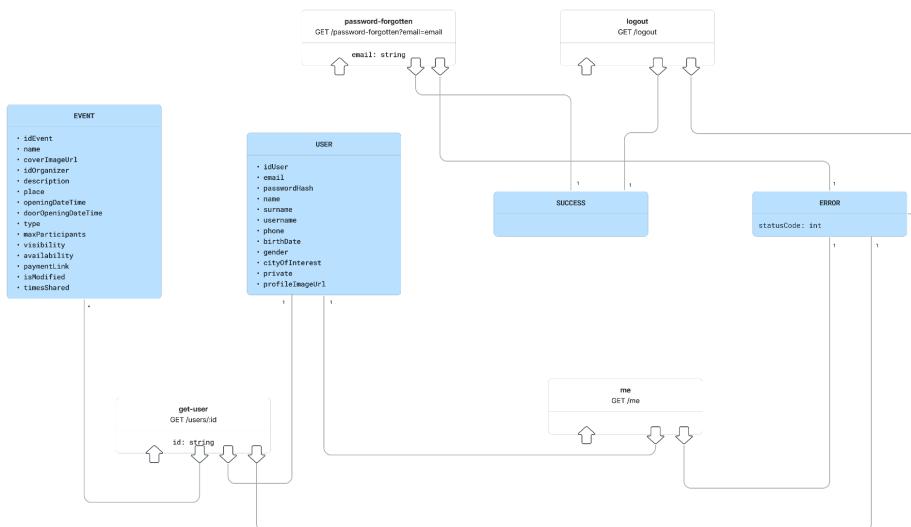


# Resource diagram

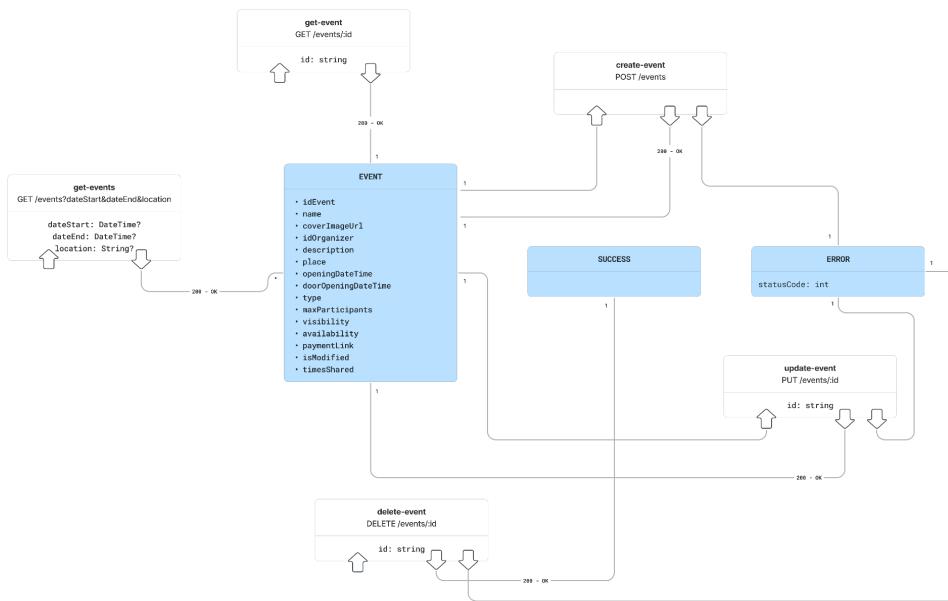
In seguito viene riportato il resource diagram, utilizzato per la rappresentazione delle API presenti in *E-20*. In questo diagramma viene specificato per ogni operazione che deve essere eseguita su una risorsa all'interno del database di quali input e quali output necessita. Ogni API può tornare un diverso errore a seconda del comportamento ottenuto dal database. Queste specifiche costituiranno il fondamento per lo sviluppo e la configurazione delle interfacce API, assicurando che la manipolazione delle risorse sia eseguita in conformità con le definizioni stabilite e che sia previsto un adeguato trattamento degli scenari di errore.

I diagrammi sono stati divisi per entità per semplificare la visualizzazione.

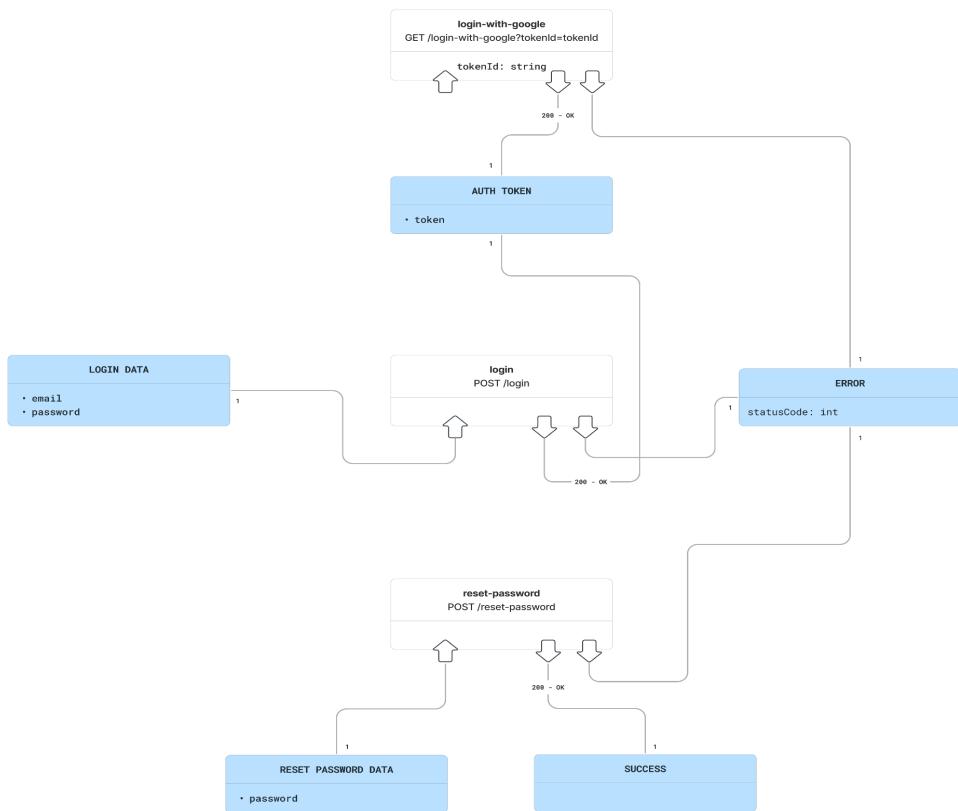
## Resource Diagram 1



## Resource Diagram 2



## Resource Diagram 3



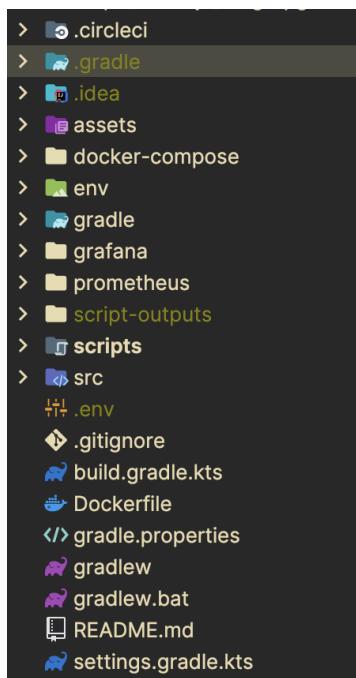
# Implementazione del backend

Il [backend](#) è stato sviluppato usando le seguenti tecnologie:

- **Kotlin**: linguaggio di programmazione
- **Gradle**: gestione delle dependencies
- **Brevo**: servizio per email transazionali
- **PostgreSQL**: database per la maggior parte dei dati dell'applicazione
- **Redis**: storage veloce ed efficiente di dati temporanei come la sessione di autenticazione degli utenti
- **Prometheus**: storage dei dati di monitoraggio dell'applicativo backend
- **Grafana**: visualizzatore dei dati di Prometheus
- **Sentry**: per il monitoraggio specifico degli errori riscontrati a runtime
- **Docker**: creazione di un'immagine Docker per l'applicativo backend e Docker compose per la creazione di tutto l'environment necessario ad esso
- **GitHub**
- **CircleCI**: cicd per la creazione della Docker image ad ogni git commit sul main branch della repository di GitHub
- **GCP**: storage delle Docker images
- **VPS** Hetzner: un semplice Virtual Private Server di Hetzner per l'hosting dell'applicativo

## Struttura del progetto

La struttura delle cartelle è illustrata nell'immagine seguente:



Ci sono diverse cartelle per la configurazione dei vari servizi che vengono spiegati in seguito. Il codice dell'applicazione è contenuto nella cartella [src](#). Vi sono inoltre alcuni script utilitari nella cartella [scripts](#).

## Dipendenze

Le dipendenze del progetto sono gestite tramite Gradle. È presente una cartella `/gradle` che contiene la lista di tutte le dipendenze disponibili a livello globale nel file `libs.versions.toml`. Gli altri file all'interno della cartella `/gradle/wrapper` specificano la versione di Gradle utilizzata dal progetto e ne contengono il relativo eseguibile (assieme ai file `gradlew` e `gradlew.bat`).

Le dipendenze, dopo essere state dichiarate nel file `/gradle/libs.versions.toml` possono essere utilizzate nel progetto "implementandole" all'interno del file [`build.gradle.kts`](#).

Le dipendenze utilizzate sono:

- **reflections**: permette di effettuare delle operazioni analizzando direttamente il codice, utilizzata per la gestione delle variabili di configurazione che vengono spiegate in seguito
- **kotlin-logging** e **logback**: per il logging
- **sentry**: per l'integrazione con Sentry
- **dotenv**: permette di leggere variabili d'ambiente della macchina o da un file
- **kotlinx-datetime**: per le operazioni sulle date ed sul tempo
- **koin**: per la dependency injection
- **exposed** e **postgresql**: per interagire con PostgreSQL utilizzando i "tipi" di Kotlin
- **flyway**: per le migrazioni di PostgreSQL
- **jedis**: per interagire con Redis
- **ktor** ed i relativi plugin: per la gestione delle richieste http e tutte le altre funzioni utilizzate da un tipico web server
- **konform**: per la validazione degli oggetti ricevuti dai client che mandano richieste al server (in questo caso la validazione dei dati provenienti dal frontend)
- **ktor-client**: client http per interagire con le API di servizi terzi come Brevo
- **spring-security-crypto**: per effettuare l'hashing delle password degli utenti
- **junit**: testing

# Modelli nel database

Per garantire una corretta modellazione dei dati essenziali per la nostra applicazione, abbiamo intrapreso la definizione di modelli di dati nel database, basandoci inizialmente sul diagramma delle classi preesistente. La definizione di questi modelli assicura coerenza e struttura nell'impiego dell'applicazione da parte degli utenti. Sono stati delineati cinque modelli fondamentali, i quali verranno ora descritti con maggiore precisione.



Le tabelle `events`, `eventplace`, `users` e `passwordreset` sono tabelle di PostgreSQL create tramite la libreria exposed.

## Tabella utenti

La tabella `users` contiene le informazioni di ogni utente. Un campo particolare è `password_hash` che corrisponde alla password hashata dell'utente, che quindi non verrà mai salvata in chiaro.

```
object UsersTable : UUIDTable() {
    val email = varchar("email", 150).uniqueIndex()
    val passwordHash = varchar("password_hash", 100).nullable()
    val name = varchar("full_name", 150)
    val surname = varchar("surname", 150)
    val username = varchar("username", 100)
    val phone = varchar("phone", 20)
    val birthDate = date("birth_date")
    val gender = enumerationByName<UserData.UserGender>("gender", 20)
    val cityOfInterest = varchar("city_of_interest", 150)
    val private = bool("is_private")
    val profileImageUrl = varchar("profile_image_url", 200)
}
```

## Tabella per il reset della password

La tabella `passwordreset` contiene i token per il reset della password (tutto il flow per effettuare l'operazione viene spiegato successivamente).

Anche in questo caso il `token` è hashato. Inoltre vi è un campo `expires_at` che fino a quando questo token per resettare la password è valido tramite una timestamp in millisecondi.

```
object PasswordResetTable : IntIdTable() {
    val token = varchar("token", 100).uniqueIndex()
    val user = reference(
        name = "id_user",
        foreign = UsersTable,
        onDelete = ReferenceOption.CASCADE
    ).index()
    val createdAt = long("created_at")
    val expiresAt = long("expires_at")
}
```

## Tabella eventi

Contiene gli eventi creati dagli utenti.

```

object EventsTable : UUIDTable() {
    val name = varchar("event_name", 150)
    val coverImageUrl = varchar("cover_image_url", 200)
    val idOrganizer = EventsTable.reference(
        name = "id_organizer",
        foreign = UsersTable,
        onDelete = ReferenceOption.CASCADE
    ).index()
    val description = varchar("description", 500)
    val openingDateTime = datetime("opening_date_time")
    val doorOpeningDateTime = datetime("door_opening_date_time")
    val type = enumerationByName<EventData.EventType>("event_type", 20)
    val maxParticipants = integer("max_participants").nullable()
    val visibility = enumerationByName<EventData.EventVisibility>("event_visibility", 20)
    val availability = enumerationByName<EventData.EventAvailability>("availability", 20)
    val paymentLink = varchar("event_link", 200).nullable()
    val isModified = bool("is_modified")
    val timesShared = long("times_shared")
}

```

## Tabella place per gli eventi

Contiene le informazioni relative ai luoghi degli eventi. L'`address` in questo caso sarà un classico indirizzo leggibile, mentre `url` è un campo opzionale in caso l'organizzatore dell'evento voglia linkare una pagina web con le istruzioni su come arrivare al luogo dell'evento.

```

object EventPlaceTable: IntIdTable() {
    val event = reference(
        name = "id_event",
        foreign = EventsTable,
        onDelete = ReferenceOption.CASCADE,
    ).index()
    val name = varchar("place_name", 150).nullable()
    val address = varchar("address", 200)
    val url = varchar("url", 200).nullable()
}

```

## Tabella per la sessione di autenticazione degli eventi

Questa tabella in realtà non è salvata in PostgreSQL ma bensì in Redis. Redis è una cache molto veloce che utilizziamo per salvare i dati sulle sessioni di autenticazione degli utenti. I dati sono organizzati in coppie key-value. Noi utilizziamo come chiave l'id della sessione e come valore un json che contiene: `iat` ossia una timestamp di quando la sessione è stata creata, `deviceName` e l'indirizzo `ip`:

```
@Serializable
data class UserAuthSessionData(
    @Contextual val id: IxId<UserAuthSessionData>,
    @Contextual val userId: IxId<UserData>,
    val iat: Long,
    val deviceName: String?,
    val ip: String
) : Principal
```

## Tabella flyway e migrazioni

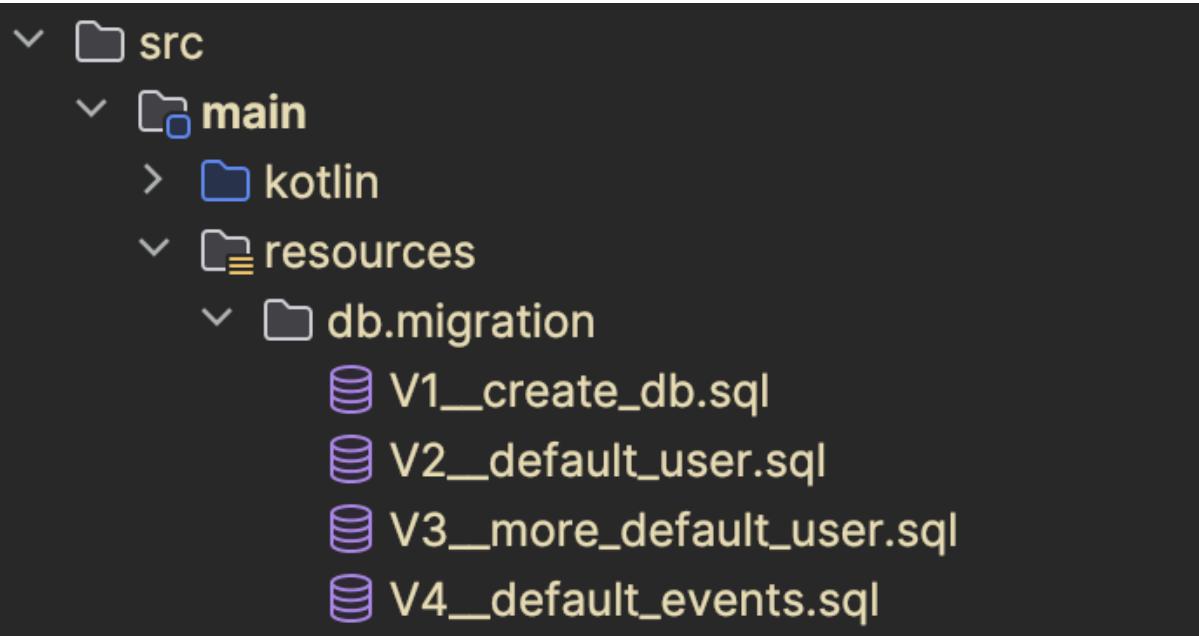
La tabella `flyway_schema_history` è una tabella speciale generata dalla libreria flyway. Serve a tenere traccia delle migrazioni effettuate sul database. È infatti essenziale avere un modo per gestire i cambiamenti del database model per facilitare i futuri aggiornamenti dell'applicazione.

Tutte le migrazioni vengono quindi salvate nella cartella

`/src/main/resources/db/migration`, ognuna identificata da una versione ed un nome, separati da due underscores.

Al momento sono presenti 4 migrazioni:

- **create\_db**: crea effettivamente tutte le tabelle del database, questa è la migrazione più importante ed è stata creata attraverso lo script [`/scripts/src/main/kotlin/GenerateFirstPostgresMigration.kt`](#). Esso utilizza la libreria exposed per creare tutti gli statements SQL necessari a creare le tabelle e relazioni a partire dalle tabelle definite dal codice Kotlin.
- **default\_user**: crea l'utente di default per l'applicazione.
- **more\_default\_user**: crea utenti di default aggiuntivi. Questo perchè non abbiamo implementato la registrazione per gli utenti, quindi abbiamo creato degli account di default per permettere ai professori di testare l'app.
- **default\_events**: contiene degli eventi di default.



Qui di seguito lo script che genera la migrazione `V1__create_db.sql`:

```
val statements = transaction {
    SchemaUtils.createStatements(
        UsersTable,
        PasswordResetTable,
        EventPlaceTable,
        EventsTable
    )
}

val folder = createScriptOutputsFolderIfNotExisting()
val file = File(folder, "V1__create_db.sql")
file.writeText(statements.joinToString("\n") { "$it;" } )

log.info { "Generated V1__create_db.sql file in /script-outputs" }
```

L'id di ogni migrazione viene salvato nella tabella creata da flyway insieme ad altri metadati così da poter tenere traccia di quali sono state effettuate. All'avvio del web server infatti viene creato il client per Postgres che applica tutte le migrazioni mancanti.

## Configurazione delle variabili d'ambiente

L'applicazione richiede delle specifiche configurazioni per:

- connettersi a PostgreSQL: username, password e nome del database
- connettersi a Redis: stringa di connessione
- sessioni di autenticazione jwt: secret, audience, issuer e realm assieme ad una massima durata delle sessioni
- connettersi a Sentry: chiave api
- connettersi a Brevo: chiave api

Tutte queste configurazioni devono essere specificate tramite variabili d'ambiente o in un file `.env`.

Per semplificare il processo di capire che variabili sono richieste è stato creato un sistema che va a definire tutte queste configurazioni con codice Kotlin.

Sono infatti presenti diversi file di configurazione in `/src/main/kotlin/app/e20/config`. Ogni `object` annotato con `@Configuration` può contenere all'interno delle variabili annotate con `@ConfigurationProperty`. Quando l'applicazione viene avviata, vengono letti tutte le variabili di questi `object` tramite la libreria reflections. Per ogni variabile trovata si va a cercare nel file `.env` o nelle variabili d'ambiente il corrispettivo valore ed il valore ottenuto lo si salva nella variabile di Kotlin. In questo modo si ottengono variabili d'ambiente tipizzate (che possono avere anche dei valori di default), e viene notificato a runtime se mancano delle specifiche variabili d'ambiente.

Qui di seguito un esempio delle variabili di configurazioni per postgres, che possono poi essere ottenute a runtime in modo safe semplicemente con `PostgresConfig.url` ad esempio.

```
@Configuration("postgres")
object PostgresConfig {
    /**
     * Postgres server url
     */
    @ConfigurationProperty("url")
    var url: String = "jdbc:postgresql://localhost:5432/e20devdb"

    /**
     * Postgres' authentication user
     */
    @ConfigurationProperty("user")
    var user: String = "e20DevUser"

    /**
     * Postgres' authentication password
     */
    @ConfigurationProperty("password")
    var password: String = "e20DevPassword"
}
```

Il codice che va ad eseguire queste operazioni è leggermente complesso e lungo quindi non verrà incluso in questo documento ma il file che lo contiene è `src/main/kotlin/app/e20/config/core/ConfigurationManager.kt`

Con questo sistema, all'avvio basterà utilizzare il seguente codice per leggere tutte le variabili d'ambiente e renderle pronte all'uso al resto dell'applicazione:

```
val configInitializer = ConfigurationManager(  
    packageName = ConfigurationManager.DEFAULT_CONFIG_PACKAGE,  
    ConfigurationReader::read  
)  
  
configInitializer.initialize()
```

## Script per la generazione del file .env

Siccome non è comodo andare ad impostare a mano le variabili d'ambiente, soprattutto per lo sviluppo in locale, è stato inoltre creato un piccolo script che prende tutte le configurazioni necessarie e crea un file `.env` di template pronto all'uso: [/scripts/src/main/kotlin/GenerateTemplateEnvFile.kt](#)

## Dependency Injection

Per semplificare la gestione dei client utilizzati dall'applicazione, si è utilizzato [koin](#) per la dependency injection. Questo consente di creare i client necessari una volta sola all'avvio dell'app ed utilizzarli dove necessario senza doverli ricreare ogni volta.

Il fulcro della dependency injection sono i "component", ovvero i singoli componenti gestiti da koin.

Per dichiarare una classe come "component" basta aggiungere l'annotazione `@Single` o `@Factory`. Single in caso si voglia che esso sia unico per tutto il lifetime dell'app, Factory quando si vuole che venga creato un nuovo componente ogni volta che viene utilizzato.

Qui ad esempio `PostgresClient` è indicato come Single perché vogliamo un solo client connesso al database, e viene anche specificato che deve essere creato all'avvio dell'app con `createdAtStart`:

```
@Single(createdAtStart = true)  
class PostgresClient {
```

Invece qui un esempio di Factory che permette di creare un semplice client Http ogni volta che serve:

```

@Factory
fun httpClient() = HttpClient(Apache) {
    install(Logging)
    install(ContentNegotiation) {
        json(Json)
    }
    install(HttpRequestRetry) {
        retryOnServerErrors(maxRetries = 3)
        exponentialDelay()
    }
    defaultRequest {
        contentType(ContentType.Application.Json)
        accept(ContentType.Application.Json)
    }
}

```

Noi abbiamo poi suddiviso i componenti in moduli. Qui ad esempio viene definito il modulo che contiene tutti i componenti necessari alle operazioni sui dati.

```

@Module(includes = [LogicModule::class, ClientModule::class])
@ComponentScan("app.e20.data")
class DataModule

```

Esso indica a Koin di andare a cercare tutti i componenti nel package (cartella in sostanza) `app.e20.data` ed gli dice che alcuni componenti possono richiedere componenti di altri moduli (in questo caso `LogicModule` e `ClientModule`).

Qui di seguito un componente del DataModel responsabile dello storage delle sessioni degli utenti nella cache Redis (CM = cache manager). Questo componente richiede infatti un `RedisClient` che permette di connettersi a Redis (che si trova nel `ClientModule`) ed un `ObjectMapper` utilizzato per la serializzazione e deserializzazione dei dati da/a json (`LogicModule`). Viene anche settato un “alias” a questo componente tramite il parametro binds. Koin quindi saprà che quando viene richiesta la classe di tipo `UserSessionCM` dovrà in realtà restituire un’istanza della classe `UserSessionCMImpl` (la prima sarebbe un’interfaccia mentre la seconda l’implementazione di essa).

```

@Single(createdAtStart = true, binds = [UserSessionCM::class])
class UserSessionCMImpl(
    redisClient: RedisClient,
    objectMapper: ObjectMapper
) : UserSessionCM,

```

Una volta definiti tutti i moduli necessari, all’avvio verrà inizializzato Koin e verranno create tutte le istanze dei componenti a cui è stato settato `createdAtStart = true`.

Inoltre abbiamo implementato una comoda interfaccia per indicare tutti quei componenti che hanno bisogno di rilasciare delle risorse (ad esempio chiudere una connessione) quando l'applicazione viene spenta.

Ogni componente potrà quindi implementare questa interfaccia e definire la sua logica prima dello shutdown nel metodo `close`.

```
interface IClosableComponent {  
    suspend fun close()  
}
```

Qui il codice in cui viene inizializzata la dependency injection.

Si può vedere che quando viene notificato lo stato di

`KoinApplicationStopPreparing` vengono presi tutti i componenti che implementano l'interfaccia descritta prima, e per ognuno viene eseguito il metodo `close`.

```
fun Application.configureDI() {  
    install(Koin) {  
        slf4jLogger(Level.valueOf(ApplicationConfig.logLevel.levelStr))  
  
        modules(LogicModule().module, ClientModule().module, DataModule().module)  
  
        this.createEagerInstances()  
    }  
  
    environment.monitor.subscribe(KoinApplicationStarted) {  
        logger.info { "Koin application started" }  
    }  
  
    environment.monitor.subscribe(KoinApplicationStopPreparing) {  
        logger.info { "Shutdown started" }  
  
        val closableComponents by lazy {  
            getKoin().getAll<IClosableComponent>()  
        }  
  
        closableComponents.forEach {  
            runBlocking {  
                it.close()  
            }  
        }  
    }  
  
    environment.monitor.subscribe(KoinApplicationStopped) {  
        logger.info { "Shutdown completed gracefully" }  
    }  
}
```

# Web server e API

## Configurazione del web server

La maggior parte delle funzioni del web server sono gestite dalla libreria Ktor.

Noi abbiamo definito i vari plugin necessari nella package `app.e20.api.plugin`:

- **DI**: plugin che inizializza la dependency injection, descritto sopra
- **HTTP**: definisce le impostazioni per i CORS ed il rate limiting. Di default abbiamo impostato dei rate limit che permettono al massimo 60 richieste al minuto ad ogni client.
- **Monitoring**: viene detto al web server di effettuare un log nella console ad ogni richiesta e viene inoltre configurato Micrometer per il monitoring (spiegato successivamente)
- **Serialization**: viene aggiunto il supporto alla serializzazione e deserializzazione di dati in formato json
- **Validator**: configura tutte le validazioni che devono essere effettuate sui body delle richieste ricevute
- **StatusPages**: imposta delle risposte http di default che vengono utilizzate quando certe eccezioni avvengono, ad esempio se viene lanciata un'eccezione `AuthenticationException` in qualsiasi momento dell'elaborazione della richiesta http, il server risponderà con il codice HTTP `401` (non autorizzato)
- **Security**: qui vengono definiti i metodi di autenticazioni che si possono applicare ai vari endpoint del nostro webserver
- **Swagger**: impostazioni per la generazione del file `openapi.json`, che verrà poi utilizzato per la visualizzazione della pagina Swagger

## Routing

Per definire gli endpoint del nostro web-server utilizziamo delle funzioni dedicate sempre di Ktor.

Diciamo innanzitutto a Ktor che utilizzeremo uno dei suoi plugin chiamato Resources, che ci permette di definire le route come classi con i parametri tipizzati (ad esempio i path e query parameters).

Poi specifichiamo le nostre route:

- **monitoringRoutes**: endpoint per i dati di monitoraggio di micrometer
- **authRoutes**: endpoint per l'autenticazione
- **userRoutes**: tutti gli endpoint utili per le operazioni sugli utenti
- **eventRoutes**: endpoint relativi alle operazioni sugli eventi

```

fun Application.configureRouting() {
    // Needed for typed queries
    install(Resources) {
        serializersModule = IdKotlinXSerializationModule
    }

    routing {
        monitoringRoutes()
        authRoutes()
        userRoutes()
        eventRoutes()
    }
}

```

Prendiamo ad esempio `userRoutes`.

Vediamo che sono state dichiarate delle classi con l'annotazione `@Resource`. Ogni resource rappresenta un possibile endpoint. La resource `PasswordForgottenRoute` accetta ad esempio un query parameter chiamato `email`, mentre `UserRoute` accetta un path parameter chiamato `id` per specificare l'id dell'utente.

Le route vengono poi suddivise nuovamente per tipologia, la cosa importante è che `passwordOperationRoutes` e `userRoute` non sono protette dall'autenticazione, mentre le altre necessitano che l'utente sia autenticato per essere utilizzate.

```

@Resource("/logout")
class LogoutRoute

@Resource("/me")
class MeRoute

@Resource("/password-forgotten")
class PasswordForgottenRoute(val email: String)

@Resource("/reset-password")
class ResetPasswordRoute(val token: String)

@Resource("/reset-password-webpage")
class ResetPasswordWebpageRoute(val token: String)

@Resource("/users/{id}")
class UserRoute(@Contextual val id: IxId<UserData>)

fun Route.userRoutes() {
    passwordOperationRoutes()
    userRoute()

    authenticate(AuthenticationMethods.USER_SESSION_AUTH) {
        logoutRoute()
        meRoutes()
    }
}

```

Qui di seguito un esempio di un endpoint.

Questa route gestisce le richieste di tipo GET che matchano la risorsa **MeRoute** (che corrisponde a `/me`).

Vengono definite le informazioni per Swagger, in questo caso ci si aspettano due possibili risposte, o una **200** con nel body della risposta i dati dell'utente autenticato, oppure un **401** in caso l'utente non sia autenticato.

```
fun Route.meRoutes() {
    val userDao by inject<UserDao>()

    get<MeRoute>({
        tags = listOf("user")
        operationId = "me"
        summary = "get the logged in user data"
        response {
            HttpStatusCode.OK to {
                description = "user data"
                body<UserData>()
            }

            HttpStatusCode.Unauthorized to {
                description = "user not logged in"
            }
        }
    }) {
        val user = userDao.get(userIdFromSessionOrThrow())
        ?: throw AuthenticationException()

        call.respond(user)
    }
}
```

## Autenticazione

Per quanto riguarda l'autenticazione, il tutto viene definito appunto nel plugin **Security**.

```

fun Application.configureSecurity() {
    val userSessionDao by inject<UserSessionDao>()

    install(Authentication) {
        jwt(AuthenticationMethods.USER_SESSION_AUTH) {
            realm = ApiConfig.jwtRealm

            verifier(JWT
                .require(Algorithm.HMAC256(ApiConfig.jwtSecret))
                .withAudience(ApiConfig.jwtAudience)
                .withIssuer(ApiConfig.jwtIssuer)
                .build()
            )

            validate { credentials ->
                val userId: IxId<UserData> = credentials.payload.getClaim(JwtClaims.JWT_USER_ID_CLAIM)
                    .asString()
                    .takeIf { it.isNotEmpty() }
                    ?.toIxId()
                ?: return@validate null
                val sessionId: IxId<UserAuthSessionData> = credentials.payload.getClaim(JwtClaims.JWT_SESSION_ID_CLAIM)
                    .asString()
                    .takeIf { it.isNotEmpty() }
                    ?.toIxId()
                ?: return@validate null

                val session = userSessionDao.get(userId, sessionId)

                // If there is no session or if it has expired
                if (session == null || (DatetimeUtils.currentTimeMillis() - session.iat) >= (ApiConfig.sessionMaxAgeInSeconds * 1000))
                    null
                else
                    session
            }

            challenge { _, _ ->
                call.respond(HttpStatusCode.Unauthorized, "Token is not valid or has expired")
            }
        }
    }
}

```

Qui stiamo creando tramite Ktor un metodo per autenticarsi tramite JWT. Passiamo innanzitutto le configurazioni per JWT, quindi il `realm`, l'`audience`, l'`issuer` e soprattutto il `jwtSecret`

Specifichiamo poi come devono essere verificate le richieste HTTP, per endpoint che richiedono questo tipo di autenticazione, tramite la funzione `validate`. Quando riceviamo una richiesta andiamo a prendere l'`id` della sessione e dall'utente dal payload del jwt. Andiamo a cercare se esiste una sessione con quegli id, se non esiste facciamo un return dal valore `null` (che equivale a dire che l'utente non è autenticato), se invece esiste ma la durata di questa sessione ha ecceduto la durata massima che abbiamo impostato nelle configurazioni allora ritorniamo ancora `null`, altrimenti l'utente è autenticato e ritorniamo la sessione trovata, che potremo utilizzare in seguito per prendere ad esempio l'id dell'utente autenticato.

Notiamo che per cercare se esiste la sessione utilizziamo l'apposita classe `UserSessionDao`, che si occuperà di andare a fare una ricerca nella cache Redis per quella sessione. Questa classe è stata ottenuta tramite dependency injection all'inizio del file con il metodo `inject`.

Abbiamo deciso di salvare le sessioni in Redis, invece di utilizzare solo la sicurezza crittografica di JWT, per poter cancellare e revocare le sessioni in ogni momento (come ad esempio dopo un reset della password).

Per poi specificare quali route necessitano di essere autenticati basta utilizzare il metodo `authenticate`, come visto in precedenza nella descrizione del routing.

Si può poi prendere l'`id` dell’utente autenticato durante l’elaborazione di una risposta con il metodo `userIdFromSessionOrThrow`, che va a cercare se è stata trovata una sessione ed in caso contrario lancia l’eccezione `AuthenticationException`, e il plugin StatusPages descritto sopra risponderà con il codice 401.

```
/**  
 * Gets the [IxId] of a [UserData] from the auth-user-session [UserAuthSessionData]  
 *  
 * @throws AuthenticationException  
 */  
fun PipelineContext<Unit, ApplicationCall>.userIdFromSessionOrThrow(): IxId<UserData> = call.principal<UserAuthSessionData>()?.userId  
?: throw AuthenticationException()
```

# Documentazione delle API

Le interfacce API create per l'applicazione, le cui specifiche sono state dettagliate in precedenza, sono state documentate attraverso l'uso di Swagger. Questo strumento offre una visione complessiva delle API, fornendo dettagli aggiuntivi su ciascuna di esse. Il collegamento per accedere alla documentazione delle API è il seguente: <https://api.e-20.net/swagger/index.html>.

Il file openapi.json e la relativa pagina Swagger viene generata a runtime tramite le informazioni settate su ogni route. Vengono inoltre configurate delle impostazioni globali nell'apposito [plugin](#).

Qui una preview degli endpoint sugli eventi:

The screenshot shows the Swagger UI interface for the /events endpoint. It lists five methods: DELETE /events/{id}, GET /events, GET /events/{id}, POST /events, and PUT /events/{id}. The POST and PUT methods are highlighted with green backgrounds, indicating they are used for creating or updating events. Each method has a description and a lock icon.

Ed il metodo per aggiornare un evento:

The screenshot shows the detailed view for the PUT /events/{id} endpoint. It includes parameters for 'id' (required, string, path) and a request body example. The request body example is a JSON object representing an event update, showing fields like availability, coverImageURL, description, doorOpeningDateTime, maxParticipants, name, openingDateTime, paymentLink, places, type, and visibility.

```
{ "availability": "AVAILABLE", "coverImageURL": "string", "description": "string", "doorOpeningDateTime": { "value": "2024-02-04T23:09:38.451Z" }, "maxParticipants": 0, "name": "string", "openingDateTime": { "value": "2024-02-04T23:09:38.451Z" }, "paymentLink": "string", "places": [ { "address": "string", "name": "string", "url": "string" } ], "type": "CONCERT", "visibility": "string" }
```

Responses		
Code	Description	Links
200	updated event Media type <code>application/json</code> Controls Accept header. Example Value   Schema <pre>EventData &lt; {     availability      EventAvailability &gt; [...]     coverImageUrl    &gt; [...]     description      &gt; [...]     doorOpeningDateTime LocalDateTime &gt; [...]     idEvent          &gt; [...]     idOrganizer      &gt; [...]     isModified        &gt; [...]     maxParticipants   &gt; [...]     name              &gt; [...]     openingDateTime   LocalDateTime &gt; [...]     paymentLink       &gt; [...]     place             EventPlaceData &gt; {...}     timesShared       &gt; [...]     type              EventType &gt; [...]     visibility        EventVisibility &gt; [...] }</pre>	No links
401	not logged in, only logged in users can update event	No links
404	event doesn't exist or logged in user is not the organizer of the event	No links

Anche tutti gli schema vengono automaticamente generati dalle classi scritte in Kotlin.

Schemas

```
LoginCredentials >

PasswordResetRequestBody >

EventAvailability < string
Enum:
  [ AVAILABLE, RUNNING_OUT, SOLD_OUT, CANCELED ]

EventData < {
    availability      EventAvailability string
    Enum:
      [ AVAILABLE, RUNNING_OUT, SOLD_OUT, CANCELED ]
    coverImageUrl    string
    description      string
    doorOpeningDateTime LocalDateTime > [...]
    idEvent          string
    idOrganizer      string
    isModified        boolean
    maxParticipants integer($int32)
    name              string
    openingDateTime   LocalDateTime > [...]
    paymentLink       string
    place             EventPlaceData < {
        address      string
        name         string
        url          string
    }
    timesShared       integer($int64)
    type              EventType string
    Enum:
      [ CONCERT, FESTIVAL, BAR, CLUB, PARTY, HOUSEPARTY, PRIVATEPARTY, WORKSHOP, CONGRESS ]
    visibility        EventVisibility string
}
```

# Business logic degli endpoint

Qui viene descritta la logica con cui si risponde alle richieste HTTP sui vari endpoint. Viene inserito anche il codice di alcune route, principalmente per mostrare le principali logiche utilizzate in tutto il backend. Ovviamente abbiamo cercato di includere meno codice possibile per non occupare troppo spazio nel documento (in ogni caso è disponibile su [GitHub](#)).

## Monitoring metrics

Le metriche che vengono prese da Prometheus e successivamente visualizzate da Grafana, sono il risultato della libreria Micrometer descritta in precedenza. Non si fa altro che rispondere alla richiesta HTTP con i dati di micrometer.

## Login

- Si ricevono le credenziali (`email` e `password`) nel body della richiesta, che vengono quindi criptate tramite https.
- Ricerca di un account nel database con la email ricevuta nelle credenziali
- Hash della password ricevuta e confronto con la password hashata ottenuta dal database
- Se l'hash della password non corrisponde si risponde con `401` (l'eccezione mostrata nel codice è gestita dalle StatusPages)
- Creo la sessione per l'utente che viene salvata nella cache Redis
- Creo il token JWT, a cui associo sia l'user id che la session id tramite i claim del payload JWT. Viene impostata anche una scadenza e poi criptato con un algoritmo standard per JWT.
- Si risponde con un codice 200 alla richiesta, includendo nel body un json del format `{ "token": "valore_del_token_jwt" }` (in questo caso si usa una semplice funzione `HashMapOf` invece di andare a creare una classe a posta solo per un valore).

```
val loginData = call.receive<LoginCredentials>()
val user = userDao.getEmail(loginData.email)
 ?: throw AuthenticationException()

if (user.passwordHash == null)
    throw AuthenticationException()

if (!passwordEncoder.matches(loginData.password, user.passwordHash))
    throw AuthenticationException()

val sessionId = userSessionDao.create(user.idUser, call.request.userAgent(), call.request.origin.remoteAddress)

val token = JWT.create()
    .withAudience(ApiConfig.jwtAudience)
    .withIssuer(ApiConfig.jwtIssuer)
    .withClaim(JwtClaims.JWT_SESSION_ID_CLAIM, sessionId.toString())
    .withClaim(JwtClaims.JWT_USER_ID_CLAIM, user.idUser.toString())
    .withExpiresAt(Date(System.currentTimeMillis() + (ApiConfig.sessionMaxAgeInSeconds * 1000)))
    .sign(Algorithm.HMAC256(ApiConfig.jwtSecret))

call.respond(hashMapOf("token" to token))
```

## Password forgotten

Per permettere all'utente di reimpostare la password vi sono diversi passaggi:

- Si riceve una richiesta di "password dimenticata" sulla route `/password-forgotten`, che include la email dell'account a cui ci si riferisce tramite query parameter.
- Si decodifica la mail, siccome il valore del query parameter "email" è encoded per non interferire con l'url
- Si prende dal database l'utente con tale email
- Si controlla che non siano state già mandate 7 mail per il reset della password (email inutilizzate). In tal caso si risponde con il codice **429** (Too Many Requests).
- Si salva nella tabella del database per i reset delle password una entry che include un hashed token generato randomicamente, l'id dell'utente e una timestamp per la scadenza di tale token.
- Viene inviata un'email all'utente con un link per il reset della password, tale link conterrà il token generato al passo precedente. Questo avviene mandando una richiesta HTTP con le informazioni per l'email al servizio di email transazionali di Brevo.
- Viene inviato il codice di successo **200**.

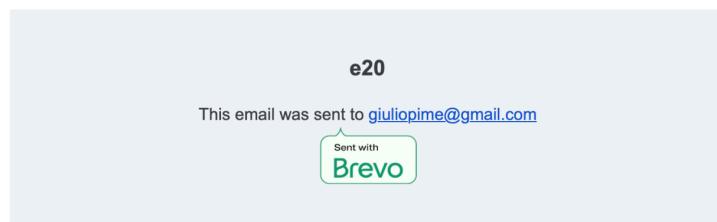
Dopo queste operazioni l'utente dovrebbe visualizzare l'email nella propria casella di posta:



## Reset your password

Open the following link to reset your password: <https://api.e-20.net/reset-password-webpage?token=HU0YK6Q4EBPWy7wxwoKfTQ==>

If you didn't request to reset your password you can ignore this email.



Il link punta direttamente ad un altro endpoint del backend.

Questo endpoint mostra un form HTML con un campo per inserire una nuova password ed un bottone per l'invio.

Al click di quest'ultimo viene mandata una nuova richiesta HTTP ad un ulteriore endpoint, questa volta `/reset-password` assieme al `token` (che transita quindi sostanzialmente per due richieste http).

L'endpoint di reset-password effettua queste operazioni:

- prende le informazioni salvate nel database riguardo quel `token` per il reset della password
- prende dal database l'utente con l'`user id` associato a quel `token`
- fa l'hashing della nuova password

- salva la nuova password dell’utente
- vengono cancellate tutte le sessioni di autenticazione dell’utente, in quanto essendo cambiata la password si considerano tutte le altre sessioni non più autenticate
- viene mandata una mail con la conferma della modifica della password
- si risponde con il codice HTTP **200**

Qui il codice della pagina web generata lato backend tramite Kotlin:

```
call.respondHtml(HttpStatusCode.OK) {
    head {
        title {
            +"E20 - Reset password"
        }
    }
    body {
        h1 {
            +"Password reset form"
        }

        p {
            +"Insert your new password in the text fields below"
        }

        form(
            action = "/reset-password?token=${it.token}",
            encType = FormEncType.applicationXWwwFormUrlEncoded,
            method = FormMethod.post,
        ) {
            p {
                +"Password:"
                passwordInput(name = "password")
            }
            p {
                submitInput { value = "Confirm password reset" }
            }
        }
    }
}
```

Qui invece il codice per la route `/reset-password`:

```
val passwordResetDto = passwordResetDao.get(request.token)
?: return@post call.respond(HttpStatusCode.NotFound)

val user = userDao.get(passwordResetDto.userId)
?: return@post call.respond(HttpStatusCode.NotFound)

val newPassword = call.receiveParameters()["password"]
?: return@post call.respond(HttpStatusCode.BadRequest)

val newPasswordHashed = passwordEncoder.encode(newPassword)

// If the user email wasn't verified before, now it can be considered verified
userDao.resetPassword(passwordResetDto.userId, newPasswordHashed)

// Invalidate all other user active sessions
userSessionDao.deleteAllSessionsOfUser(passwordResetDto.userId)

// Send notification email
brevoClient.sendPasswordResetSuccessEmail(user.email)

call.respond(HttpStatusCode.OK, "Your password has been reset")
```

## Logout

Nella route di logout si cancella semplicemente la sessione dalla cache Redis.

## Richiesta dell'utente loggato

La route `/me` serve per prendere informazioni riguardo l'utente autenticato (risponderà quindi con il codice `401` per utenti non autenticati). Per farlo prende l'id utente dalla sessione, cerca nel database l'id con l'utente corrispondente e lo restituisce nella risposta HTTP.

## Richiesta utente specifico

La route `/users/:id` si occupa di fornire le informazioni pubblicamente accessibili riguardo uno specifico utente, che includono ad esempio nome, cognome, username e immagine profilo dell'utente, assieme a tutti gli eventi da lui organizzati.

Sfruttiamo questo endpoint per spiegare anche come avvengono le query sul database.

## Interazione con PostgreSQL / Redis

L'accesso ai dati è diviso in livelli, in modo da renderlo adattabile a diverse sorgenti di dati.

Vi sono delle classi con suffisso **DAO** (Data Access Object) che si occupano di tutte le operazioni necessarie per ottenere i dati richiesti. Andranno quindi a chiamare altre classi specifiche, che nel nostro caso possono essere classi che si interfacciano con PostgreSQL (suffisso **DBI**, Database Interactor) o Redis (suffisso **CM**, Cache Manager).

Ogni classe in genere è in realtà suddivisa in un'interfaccia che ne dichiara le funzionalità, ed una classe che le implementa.

Questa è ad esempio parte dell'interfaccia per i dati sugli eventi:

```
interface EventDao {
    /**
     * Creates a new event
     *
     * @param userId id of the user that organized the event
     * @param eventCreateOrUpdaterequestData
     *
     * @return [EventData]
     */
    suspend fun create(userId: IxId<UserData>, eventCreateOrUpdaterequestData: EventData.EventCreateOrUpdaterequestData): EventData

    /**
     * Gets a single event by its [id]
     *
     * @return [EventData] or null if no event with that id exists
     */
    suspend fun get(id: IxId<EventData>): EventData?

    /**
     * Gets all the events organized by a user
     *
     * @param id organizer id
     *
     * @return a [List] of all the [EventData] organized by the user
     */
    suspend fun getOfOrganizer(id: IxId<UserData>): List<EventData>

    /**
     * Gets all events in the range of [startDate] and [endDate]
     *
     * @return the [List] of [EventData]
     */
    suspend fun getForDates(startDate: LocalDateTime, endDate: LocalDateTime): List<EventData>
```

Mentre di seguito vi è la sua implementazione:

```
class EventDaoImpl(
    private val eventDBI: EventDBI
) : EventDao {
    override suspend fun create(
        userId: IxId<UserData>,
        eventCreateOrUpdaterequestData: EventData.EventCreateOrUpdaterequestData
    ): EventData {
        val eventData = EventData(
            idEvent = newIxId(),
            name = eventCreateOrUpdaterequestData.name,
            coverImageUrl = eventCreateOrUpdaterequestData.coverImageUrl,
            idOrganizer = userId,
            description = eventCreateOrUpdaterequestData.description,
            place = eventCreateOrUpdaterequestData.place,
            openingDateTime = eventCreateOrUpdaterequestData.openingDateTime,
            doorOpeningDateTime = eventCreateOrUpdaterequestData.doorOpeningDateTime,
            type = eventCreateOrUpdaterequestData.type,
            maxParticipants = eventCreateOrUpdaterequestData.maxParticipants,
            visibility = eventCreateOrUpdaterequestData.visibility,
            availability = eventCreateOrUpdaterequestData.availability,
            paymentLink = eventCreateOrUpdaterequestData.paymentLink,
            isModified = false,
            timesShared = 0
        )
        eventDBI.create(eventData)
        return eventData
    }

    override suspend fun get(id: IxId<EventData>): EventData? {
        return eventDBI.get(id)
    }

    override suspend fun getOfOrganizer(id: IxId<UserData>): List<EventData> {
        return eventDBI.getOfOrganizer(id)
    }

    override suspend fun getForDates(startDate: LocalDateTime, endDate: LocalDateTime): List<EventData> {
        return eventDBI.getForDates(startDate, endDate)
    }
}
```

Si nota che a sua volta questa classe `EventDaoImpl`, richiede la classe `EventDBI`, che è in realtà anch'essa un'interfaccia la cui implementazione è separata.

Qui un esempio di come vengono ottenuti gli eventi organizzati da uno specifico utente nella classe che implementa `EventDBI`. La libreria exposed rende molto semplice l'interazione con il database e soprattutto tipizzata, in modo da evitare errori su nomi o tipi di dati.

```
override suspend fun getOfOrganizer(id: IxId<UserData>): List<EventData> = dbQuery {
    EventEntity.find { EventsTable.idOrganizer eq id.toEntityId(UsersTable) }
        .map { it.toData() }
}
```

### ★ Id tipizzati

Un'altra cosa importante che si nota è `IxId<UserData>`. Anche gli id delle entità sono infatti tipizzati, invece di essere dichiarati semplicemente come stringhe. Questo proviene moltissimi errori spesso comuni, e rende possibile specificare a che entità si deve riferire un id, generando un errore a compile time in caso venga passato un id non conforme.

Per poter ricevere id tipizzati dalle richieste effettuate ai vari endpoint (come `/users/:id`) e inviare gli id in formato leggibile al frontend, è stato creato un serializzatore a posta per il tipo `IxId`.

```
private class IxIdSerializer<T : IxId<*>> : KSerializer<T> {
    override val descriptor: SerialDescriptor = PrimitiveSerialDescriptor("IxIdSerializer", PrimitiveKind.STRING)

    @Suppress("UNCHECKED_CAST")
    override fun deserialize(decoder: Decoder): T =
        IxIdGenerator().create(decoder.decodeString()) as T

    override fun serialize(encoder: Encoder, value: T) {
        encoder.encodeString(value.toString())
    }
}
```

`IxId` non è l'unico tipo di Id utilizzato, vi sono infatti tre tipi di Id:

- `Id`: generico
- `IxId`: Id che **deve** essere basato su UUID (standard per universally unique identifier)
- `IxIntId`: Id che **deve** essere basato su un valore intero

Ognuno di questi tipi avrà quindi una classe simile a quella sopra per essere serializzato. Queste classi vengono raggruppate in un unico serializzatore:

```
val IdKotlinXSerializationModule: SerializersModule by lazy {
    SerializersModule {
        contextual(Id::class, IdSerializer())
        contextual(IxId::class, IxIdSerializer())
        contextual(IxIntId::class, IxIntIdSerializer())
        if (IdGenerator.defaultGenerator.idClass != IxId::class
            && IdGenerator.defaultGenerator.idClass != IxId::class) {
            @Suppress("UNCHECKED_CAST")
            contextual(
                IdGenerator.defaultGenerator.idClass as KClass<Id<*>>,
                IdSerializer()
            )
        }
    }
}
```

Questo è il serializzatore che specifichiamo infatti nella configurazione del web server, e che ci permetterà di utilizzare gli id tipizzati anche nelle route.

```
fun Application.configureSerialization() {
    install(ContentNegotiation) {
        json(Json {
            serializersModule = IdKotlinXSerializationModule
            ignoreUnknownKeys = true
            encodeDefaults = true
        })
    }
}

@Resource("/users/{id}")
class UserRoute(@Contextual val id: IxId<UserData>)
```

## Lista degli eventi

La route `/events` restituisce tutti gli eventi esistenti in un certo range di date. Si può infatti specificare una `dateStart` ed una `dateEnd` tramite query parameter. In caso non venissero specificate `dateStart = (today - 1 mese)` e `dateEnd = (today + 1 mese)`.

Abbiamo deciso di fare partire `dateStart` da un mese prima di default per semplificare il testing.

I filtri vengono quindi applicati nella query al database e si risponde alla richiesta con i dati inseriti nel body della richiesta ed il codice `200`.

## Ricerca di un singolo evento

La route `/events/:id` se chiamata con il metodo `GET`, restituisce nel corpo della risposta con stato `200` il json dell'evento con l'id passato nella path. Verrà infatti effettuata una query sul database per un evento con tale id.

In caso non esistesse viene restituito il codice `404` (Not Found).

## Creazione di un evento

La route `/events` quando chiamata con metodo `POST`, permette la creazione di un evento passando nel body del `POST` i dati necessari. Non tutti i dati di un evento possono essere scelti dagli organizzatori, nello specifico l'id dell'evento. Esso viene infatti generato dal backend, unito poi ai dati ricevuti per inserire l'evento nel database. Per questo motivo la risposta conterrà, oltre al codice `200`, l'intero json dell'evento creato con il corrispettivo id.

## Validazione dei dati

Da sottolineare che tutti i dati per la creazione dell'evento (come la maggior parte dei dati ricevuti in qualsiasi route) viene validato tramite la libreria `Konform`. In caso un json non passi la fase di validazione si risponde con il codice `400` (Bad request) in automatico. Qui l'esempio per i dati dell'evento:

```
override fun validate() = Validation {
    EventCreateOrUpdateRequestData::name {
        minLength(Validations.Event.minLength)
        maxLength(Validations.Event.maxLength)
    }
    EventCreateOrUpdateRequestData::coverImageUrl {
        minLength(Validations.minImageUrlLength)
        maxLength(Validations.maxImageUrlLength)
    }
    EventCreateOrUpdateRequestData::description {
        minLength(Validations.Event.minLength)
        maxLength(Validations.Event.maxLength)
    }
    EventCreateOrUpdateRequestData::place {
        EventPlaceData::name ifPresent {
            minLength(Validations.Event.minLength)
            maxLength(Validations.Event.maxLength)
        }
        EventPlaceData::url ifPresent {
            minLength(Validations.Event.minLength)
            maxLength(Validations.Event.maxLength)
        }
        EventPlaceData::address {
            minLength(Validations.Event.minLength)
            maxLength(Validations.Event.maxLength)
        }
    }
    EventCreateOrUpdateRequestData::maxParticipants ifPresent {
        minimum(Validations.Event.minLength)
    }
    EventCreateOrUpdateRequestData::paymentLink ifPresent {
        minLength(Validations.Event.minLength)
        maxLength(Validations.Event.maxLength)
    }
}.invoke(this)
```

I valori per la validazione sono contenuto nella classe `Validations` per semplificare la loro uniformità:

```
object Validations {
    const val minImageUrlLength = 1
    const val maxImageUrlLength = 500

    object Event {
        const val minLength = 1
        const val maxLength = 200

        const val minDescriptionLength = 1
        const val maxDescriptionLength = 1000

        const val minPlaceNameLength = 1
        const val maxPlaceNameLength = 100

        const val minPlaceUrlLength = 1
        const val maxPlaceUrlLength = 500

        const val minPlaceAddressLength = 1
        const val maxPlaceAddressLength = 200

        const val minParticipants = 1

        const val minPaymentLinkLength = 1
        const val maxPaymentLinkLength = 500
    }
}
```

## Modifica di un evento

La modifica dell'evento avviene tramite una `PUT` request alla route `/events/:id`. Si prende dal database l'evento con l'id specificato nella path, e si applicano le modifiche tramite i dati ottenuti (e validati) dal body della richiesta.

Si salvano le modifiche applicate nel database e si restituisce l'intero evento modificato con il codice di stato `200`.

Se l'evento non dovesse esistere la risposta ha codice `404`, mentre se l'organizzatore dell'evento non corrisponde all'utente autenticato, il codice di risposta sarà `403` (Unauthorized) e l'operazione non viene effettuata.

## Eliminazione di un evento

Effettuando una richiesta con metodo `DELETE` alla route `/events/:id`. Anche in questo caso se l'evento non esiste il codice di risposta è `404`, e `403` in caso l'organizzatore dell'evento non sia l'utente autenticato.

# Testing

Le principali parti dell'applicazione per cui sono stati implementati dei test sono due, quelle logiche tramite unit test e le effettive API del web server.

Tutti i test si trovano nella cartella `/src/test/`

Le parti di logica testate sono ad esempio l'hash delle password, la generazione di token, i pattern Regex utilizzati nelle validazioni e la serializzazione dei dati.

Il testing sulle API invece sono eseguiti in un ordine specifico, per simulare effettivamente le operazioni che farebbe un client.

```
@Test
@Order(1)
fun `list events no auth expect success`() {
    runBlocking {
        val getAllRes = httpClient.get(EventsRoute())
        assertEquals(getAllRes.status.value, 200)
    }
}

@Test
@Order(2)
fun `create event no auth expect unauthorized`() {
    runBlocking {
        val createEventRes = httpClient.post(EventsRoute()) {
            setBody(eventCreateData)
        }
        assertEquals(createEventRes.status.value, 401)
    }
}

@Test
@Order(3)
fun `perform login expect auth token`() {
    runBlocking {
        val res = httpClient.post(LoginRoute()) {
            setBody(LoginCredentials(ApiTestUtilities.DEFAULT_USER_EMAIL, ApiTestUtilities.DEFAULT_USER_PASSWORD))
        }

        @Serializable
        data class LoginResponse(val token: String)

        authToken = try {
            res.body<LoginResponse>().token
        } catch (e: Exception) {
            null
        }
        assertEquals(authToken != null, true)
    }
}
```

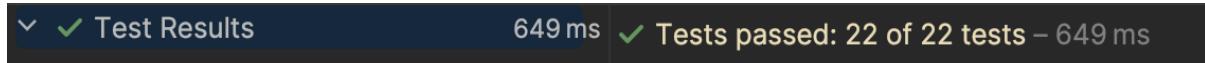
Sono stati testati tutti gli endpoint più importanti tenendo conto delle varie casistiche, quindi cercando di ottenere tutti i codici HTTP di risposta possibili.

Gli unit test sui componenti logici possono essere eseguiti senza un particolare setup, mentre tutti i test sulle api devono essere eseguiti in locale con il web server avviato, insieme all'environment necessario ottenibile con Docker compose.

Il testing non è stato implementato a livello di cicd per evitare di uscire dalle soglie dei piani gratuiti.

Il comando per eseguire i test a riga di comando è `./gradlew test`.

Si possono eseguire anche tramite un IDE come IntelliJ, con cui ci viene anche indicato il numero di test eseguiti e ci viene permesso di esportare il risultato in vari formati.



Qui i risultati dei test esportati come pagina HTML.

e20-api [test]: 22 total, 22 passed		649 ms
		<a href="#">Collapse</a>   <a href="#">Expand</a>
ApiAuthRoutesTest	perform login with invalid credentials expect unauthorized	73 ms
ApiEventRoutesTest	list events no auth expect success	passed 73 ms
	create event no auth expect unauthorized	284 ms
	perform login expect auth token	passed 20 ms
	create event expect success	passed 7 ms
	list event expect created event in list	passed 98 ms
	get created event expect success	passed 34 ms
	update event expect success	passed 30 ms
	update event with invalid data expect bad request	passed 22 ms
	delete event expect success	passed 6 ms
	create event with invalid data expect bad request	passed 15 ms
	update missing event expect not found	passed 6 ms
ApiUserRoutesTest	perform login expect auth token	passed 16 ms
	get self user expect success	154 ms
	get self user with events expect success	passed 78 ms
	get missing user with events expect not found	passed 26 ms
	perform logout expect success	passed 31 ms
PasswordEncoderTest	matches	passed 15 ms
RegexPatternsTest	testEmailPattern	4 ms
TokenGeneratorTest	hashToken	134 ms
IdKotlinXSerializationModuleKtTest	objectWithIdSerialization	0 ms
		passed 2 ms
		passed 0 ms

# Monitoring

Il monitoraggio è suddiviso in:

- monitoraggio delle performance
- monitoraggio e tracking degli errori

## Monitoring delle performance

Questo è stato ottenuto tramite la libreria Micrometer che permette di monitorare il processo Java e le informazioni relative alle richieste gestite dal web server, esposte in seguito all'endpoint <https://api.e-20.net/monitoring/metrics>.

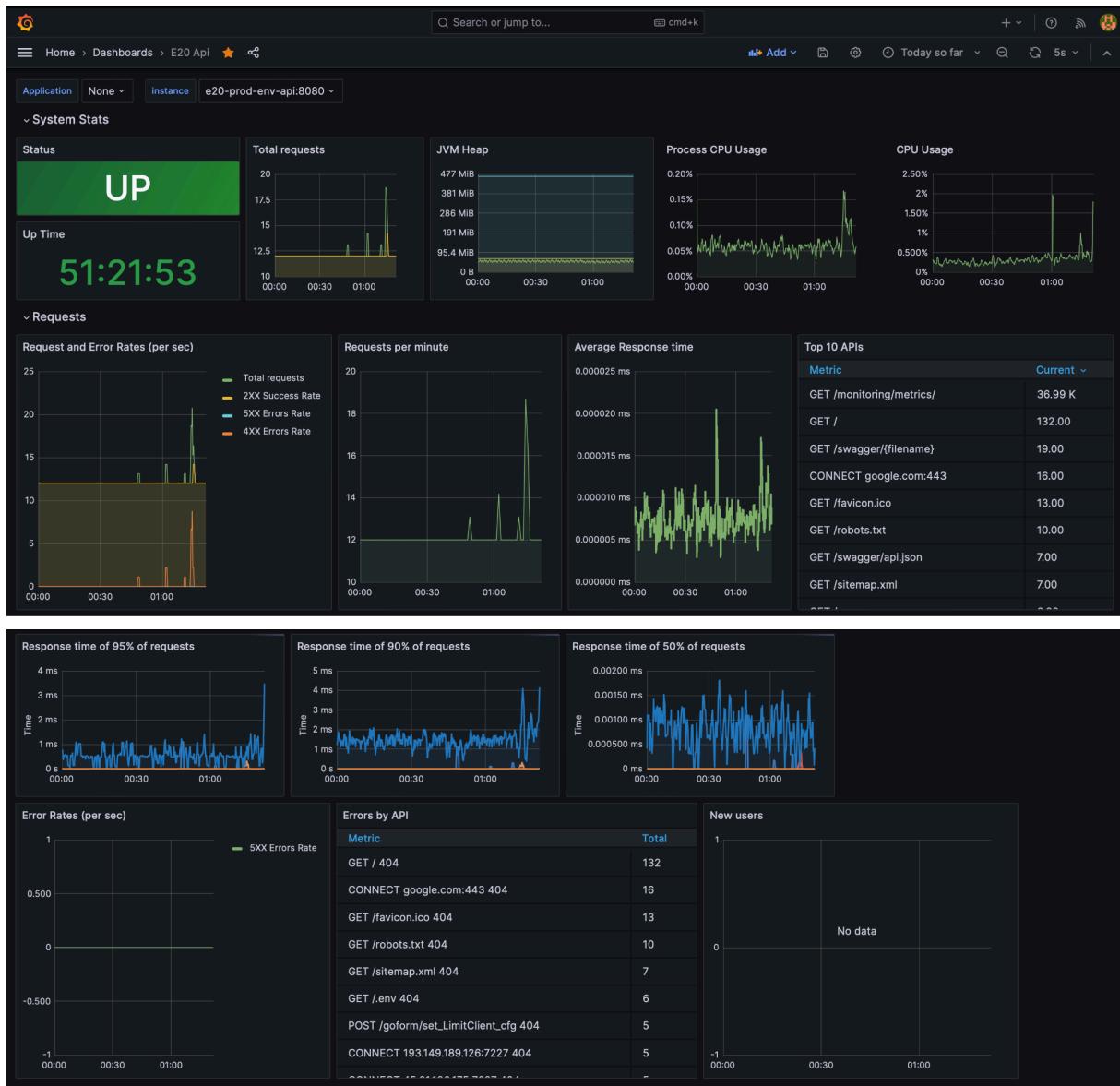
Questo endpoint viene utilizzato poi da Prometheus per elaborare le metriche. Abbiamo infatti creato un apposito file [prometheus-monitoring.yml](#) dove viene impostato l'uri e l'intervallo:

```
global:  
  scrape_interval:      5s  
  evaluation_interval:  5s  
  
scrape_configs:  
  - job_name: prometheus  
    metrics_path: /monitoring/metrics  
    static_configs:  
      - targets:  
          - host.docker.internal:8080
```

Per poter effettivamente visualizzare in modo grafico ed intuitivo le performance utilizziamo Grafana.

Abbiamo configurato Grafana aggiungendo Prometheus come data source e creato poi la seguente [dashboard](#), che ci permette di osservare:

- il numero di richieste
- l'uso della CPU e memoria
- i rate per tipologia di risposta delle richieste (200, 400, 500)
- le richieste al minuto
- il tempo di risposta medio
- i 10 endpoint più utilizzati
- la quantità di errori non gestiti e gli endpoint che generano più errori



Queste dashboard ci hanno anche permesso di visualizzare il tempo di elaborazione medio di una risposta, che è molto più basso di quello ci aspettavamo considerando che una generica richiesta attraversa le fasi di parsing, validazione, query del database e formattazione della risposta. Il tutto ci impiega **da 1 a 4 millisecondi** come illustrato dai grafici blu!

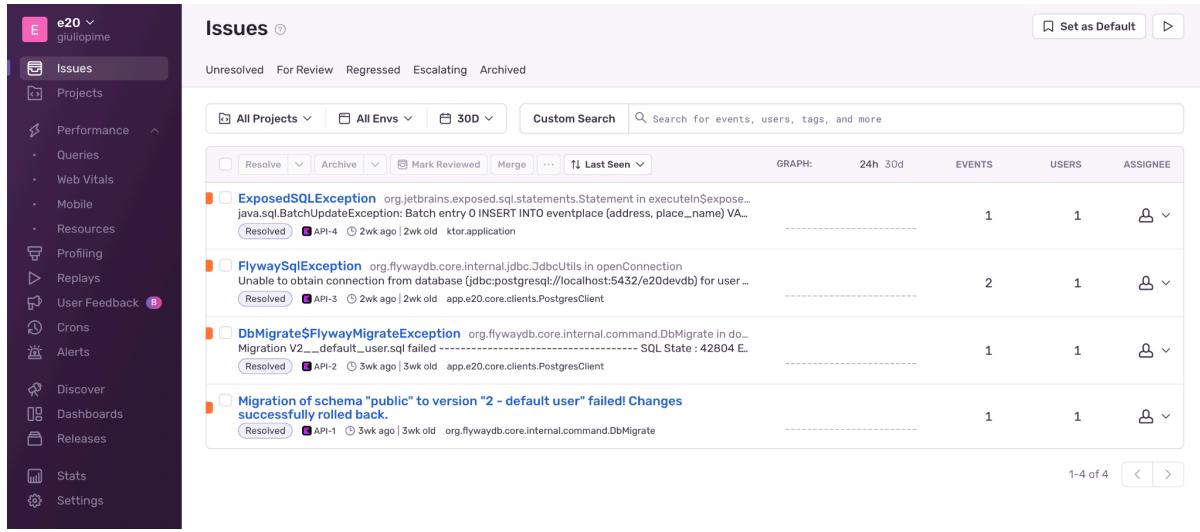
Con Grafana sarebbe stato possibile impostare anche degli avvisi per quando ad esempio viene superata una soglia per gli errori o per il tempo di risposta media, ma non l'abbiamo fatto.

La dashboard è accessibile all'indirizzo <https://grafana.e-20.net>, utilizzando come credenziali user: **admin** pwd: **e20admin**

## Monitoring e tracking degli errori

Per gli errori non gestiti viene utilizzato Sentry. Abbiamo configurato il logger della libreria logback in modo da riportare gli errori a Sentry, tramite il file [logback.xml](#).

Ottieniamo così una dashboard direttamente sul sito di Sentry con tutti gli errori riportati ed il loro contesto.



The screenshot shows the Sentry Issues dashboard. On the left is a dark sidebar with navigation links: e20 (guiliopine), Issues (selected), Projects, Performance (Queries, Web Vitals, Mobile, Resources), Profiling, Replays, User Feedback (4 notifications), Cron, Alerts, Discover, Dashboards, Releases, Stats, and Settings. The main area is titled "Issues" with a count of 4. It includes filters: All Projects (dropdown), All Envs (dropdown), 30D (dropdown), Custom Search (text input), and sorting by Last Seen (dropdown). A search bar at the top right says "Search for events, users, tags, and more". Below is a table of errors:

	Type	Message	Resolved	API	Time Ago	Events	Users	Assignee
1	ExposedSQLException	org.jetbrains.exposed.sql.statements.Statement in executeIn\$expose... java.sql.BatchUpdateException: Batch entry 0 INSERT INTO eventplace (address, place_name) VA...	Resolved	API-4	2wk ago	2wk old	ktor.application	
2	FlywaySqlException	org.flywaydb.core.internal.jdbc.JdbcUtils in openConnection Unable to obtain connection from database [jdbc:postgresql://localhost:5432/e20devdb] for user ...	Resolved	API-3	2wk ago	2wk old	app.e20.core.clients.PostgresClient	
1	DbMigrate\$FlywayMigrateException	org.flywaydb.core.internal.command.DbMigrate in do... Migration V2__default_user.sql failed ----- SQL State : 42804 E...	Resolved	API-2	3wk ago	3wk old	app.e20.core.clients.PostgresClient	
1	Migration of schema "public" to version '2 - default user' failed! Changes successfully rolled back.	(Resolved)	API-1	5wk ago	3wk old	org.flywaydb.core.internal.command.DbMigrate		

At the bottom right, it says "1-4 of 4" with navigation arrows.

# Implementazione del frontend

In questo segmento, presenteremo le interfacce utente front end sviluppate. Di seguito, un sommario delle interfacce realizzate per il progetto:

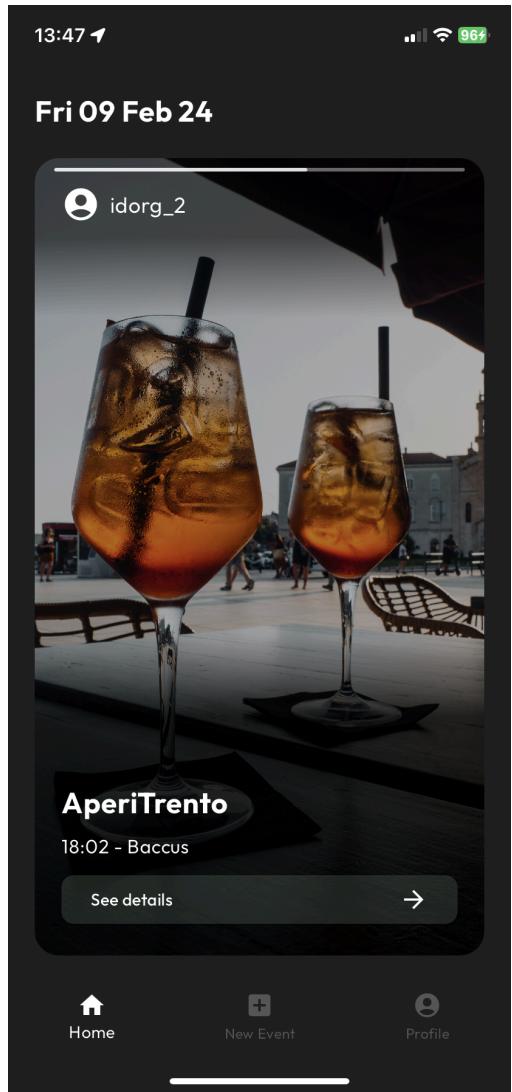
- *homepage (o bacheca)*
- *profilo utente*
- *autenticazione*
- *creazione di un evento*
- *dettagli dell'evento*

## Homepage (o bacheca)

La homepage dell'applicazione, che funge anche da bacheca per gli eventi, è progettata per mostrare gli eventi che si svolgono nelle vicinanze di un utente specifico. Questa pagina è completamente accessibile sia agli utenti che hanno effettuato l'accesso sia a quelli che non sono autenticati.

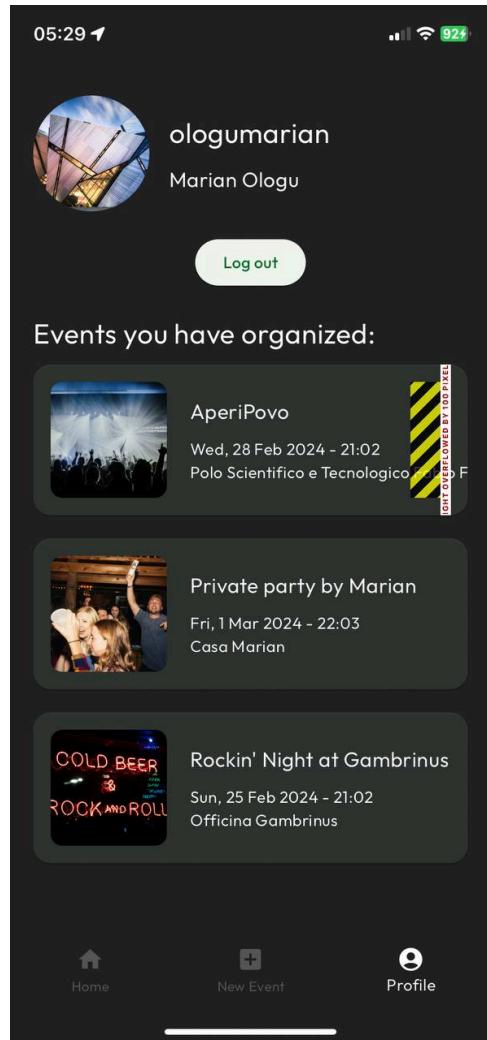
Per lo sviluppo di questa pagina e' stato rispettato lo user flow diagram precedentemente illustrato. La navigazione quindi segue le seguenti gesture:

- *scroll verticale* per navigare tra eventi di date diverse
- *click sul bordo* di un evento per navigare tra eventi della stessa data



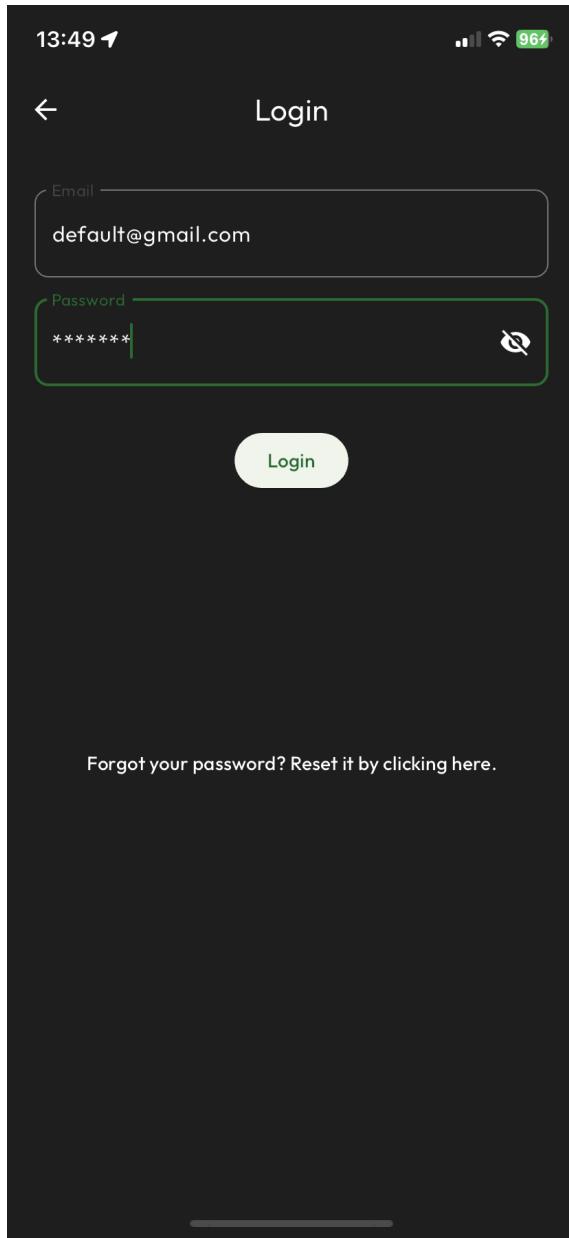
## Profilo utente

La schermata di profilo utente funge da vetrina per gli eventi che un utente ha in programma, ha partecipato e quelli che ha organizzato.



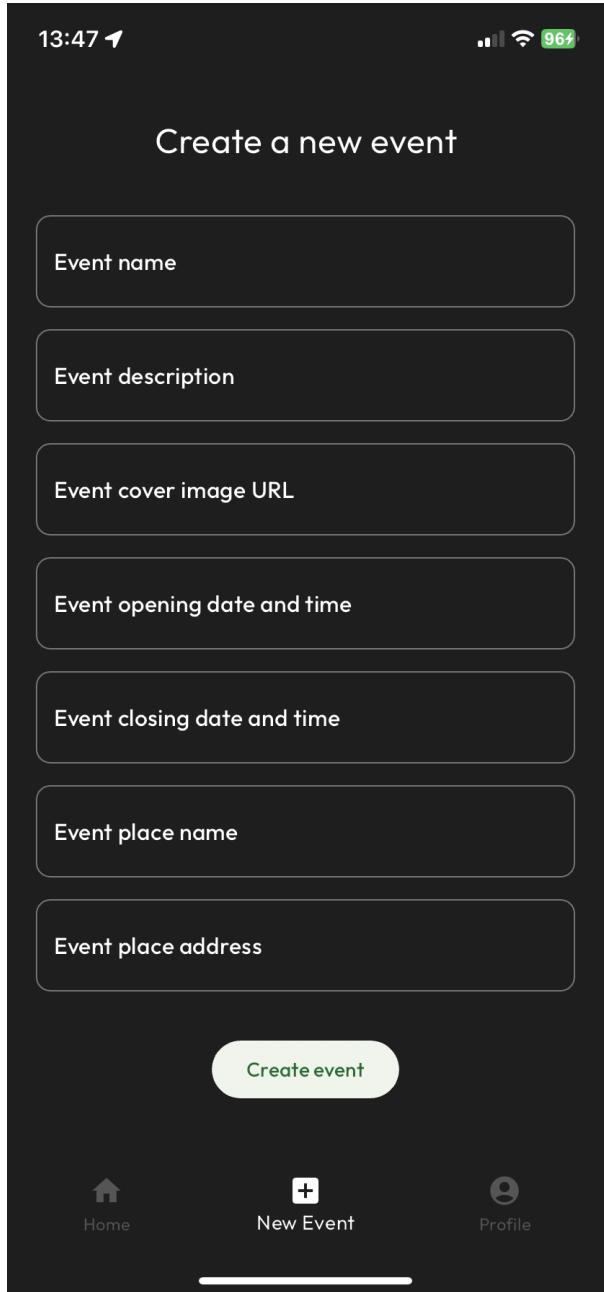
## Autenticazione

Questa interfaccia consente a un visitatore di accedere al sito per avvalersi delle funzionalità riservate agli utenti iscritti. Se l'utente dispone già di un account nel database, sarà necessario immettere l'indirizzo email / telefono e la password scelti in fase di registrazione per autenticarsi. Il sistema procederà quindi a confrontare le credenziali fornite con quelle archiviate nel database, informando l'utente sull'esito dell'operazione di accesso, sia in caso di successo che di insuccesso.



## Creazione evento

La pagina di creazione di un evento è accessibile unicamente agli utenti che hanno effettuato l'accesso, consentendo loro di organizzare un evento compilando i dati richiesti. Questi dati saranno sottoposti a verifica nel momento in cui l'utente procede con la conferma di creazione dell'evento; se risulteranno conformi, l'evento sarà pubblicato sulla homepage. In caso di creazione riuscita, l'utente sarà automaticamente rediretto alla propria pagina del profilo, dove potrà visualizzare i dettagli dell'evento appena inserito. Utilizzando *annulla*, invece, l'utente avrà la possibilità di tornare alla homepage senza generare alcun evento.



# Deployment e CICD

Il componente fondamentale per il deployment è Docker. È infatti presente un [Dockerfile](#) nella root del progetto che contiene le istruzioni per creare una Docker image del progetto.

Tramite poi CircleCI, ogni volta che viene fatto un commit sul branch main della repository di GitHub, viene eseguita una pipeline su CircleCI che va a creare la Docker image e la pubblica su Google Container Registry, un servizio di Google per lo storage delle immagini.

Questo processo è configurato tramite il file [/.circleci/config.yml](#), qui indichiamo con filters che il branch deve essere chiamato `main` per eseguire questa pipeline. In ogni caso il nome di ogni immagine Docker generata termina con il nome del branch, così da evitare ogni tipo di conflitto.

```
version: 2.1

orbs:
  gradle: circleci/gradle@3.0.0
  gcp-gcr: circleci/gcp-gcr@0.15.0

jobs:
  build-and-push-image:
    executor: gcp-gcr/default
    steps:
      - checkout
      - gcp-gcr/gcr-auth
      - gcp-gcr/build-image:
          image: 'e-20-api-<< pipeline.git.branch >>'
          no_output_timeout: 20m
          registry-url: eu.gcr.io
      - gcp-gcr/push-image:
          image: 'e-20-api-<< pipeline.git.branch >>'
          registry-url: eu.gcr.io

workflows:
  test-build-publish:
    jobs:
      - build-and-push-image:
          filters:
            branches:
              only:
                - main
```

Sulla dashboard di CircleCI possiamo quindi analizzare tutte le pipeline effettuate, con tutte le relative informazioni come il tempo impiegato:

**e20-api** [Add team members](#)

[Edit Config](#) [Trigger Pipeline](#) [Project Settings](#)

Filters [Everyone's Pipelines](#) [e20-api](#) main All days [...](#) Auto-expand

Pipeline	Status	Workflow	Trigger Event	Start	Duration	Actions
e20-api 39	<span>✓ Success</span>	test-build-publish	main Oabeaa4 Merge pull request #13 from G29-IS/develop	2d ago	2m 5s <span>7%</span>	<a href="#">CI</a> <a href="#">CI</a> <a href="#">X</a> <a href="#">...</a>
Jobs					1m 52s	
e20-api 37	<span>✓ Success</span>	test-build-publish	main 943415b Merge pull request #12 from G29-IS/develop	2d ago	1m 52s <span>17%</span>	<a href="#">CI</a> <a href="#">CI</a> <a href="#">X</a> <a href="#">...</a>
e20-api 35	<span>✓ Success</span>	test-build-publish	main 2bc2868 Merge pull request #11 from G29-IS/develop	2d ago	2m 2s <span>9%</span>	<a href="#">CI</a> <a href="#">CI</a> <a href="#">X</a> <a href="#">...</a>
e20-api 33	<span>✓ Success</span>	test-build-publish	main d66491f Merge pull request #10 from G29-IS/develop	2d ago	2m 4s <span>8%</span>	<a href="#">CI</a> <a href="#">CI</a> <a href="#">X</a> <a href="#">...</a>
e20-api 31	<span>✓ Success</span>	test-build-publish	main c277d4f Merge pull request #9 from G29-IS/develop	2d ago	2m 5s <span>7%</span>	<a href="#">CI</a> <a href="#">CI</a> <a href="#">X</a> <a href="#">...</a>
e20-api 28	<span>✓ Success</span>	test-build-publish	main f34e029 Merge pull request #7 from G29-IS/develop	4d ago	1m 57s <span>13%</span>	<a href="#">CI</a> <a href="#">CI</a> <a href="#">X</a> <a href="#">...</a>
e20-api 24	<span>✓ Success</span>	test-build-publish	main 9cf0157 Merge pull request #6 from G29-IS/develop	6d ago	2m 21s <span>5%</span>	<a href="#">CI</a> <a href="#">CI</a> <a href="#">X</a> <a href="#">...</a>

Per mettere online il backend e renderlo disponibile al pubblico, abbiamo comprato un VPS (Virtual Private Server) da Hetzner e creato un record DNS di tipo **CNAME** che indirizza il sottodomino **api** all'indirizzo IP del nostro server. Su di esso abbiamo innanzitutto installato Nginx e Docker.

Questa di seguito è la configurazione di Nginx usato come reverse proxy che permette di inoltrare tutto il traffico ottenuto da `https://api.e-20.net` alla porta 8080 del server.

Certbot è stato utilizzato per creare un certificato ssl in modo da rendere l'https disponibile e fare il redirect automatico delle richieste da http ad https.

```

server {
    server_name api.e-20.net;
    location / {
        proxy_pass http://127.0.0.1:8080;
        proxy_set_header X-Forwarded-Proto https;
        proxy_pass_request_headers      on;
    }

    listen 443 ssl; # managed by Certbot
    ssl_certificate /etc/letsencrypt/live/api.e-20.net/fullchain.pem; # managed by Certbot
    ssl_certificate_key /etc/letsencrypt/live/api.e-20.net/privkey.pem; # managed by Certbot
    include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot
}

server {
    if ($host = api.e-20.net) {
        return 301 https://$host$request_uri;
    } # managed by Certbot

    server_name api.e-20.net;

    listen 80;
    return 404; # managed by Certbot
}

```

Fatto ciò ed installato Docker abbiamo copiato il file [docker-compose-prod.yml](#) sul server e creato il file [.env](#).

Il file docker-compose crea tutto l'environment necessario all'applicazione per essere eseguita (PostgreSQL, Redis, Prometheus, Grafana) assieme a runnare l'immagine Docker dell'applicazione, presa dal Google Container Registry.

Per permettere al VPS di scaricare l'immagine dal GCR, è stato inoltre necessario creare un account di servizio ed impostare quindi l'autenticazione al registry di Google tramite tale account. Qui non descriviamo tale processo perché fuori dallo scopo del documento.

Ecco un esempio con la prima parte del file docker-compose:

```

version: "3.8"
name: "e20-prod-env"

services:
  e20:
    image: eu.gcr.io/e-20-408110/e-20-api-main:latest
    restart: unless-stopped
    container_name: ${COMPOSE_PROJECT_NAME}_api
    env_file:
      - .env
    ports:
      - "8080:${API_PORT}"
    networks:
      - e20-prod-network
  postgres:
    image: postgres
    container_name: ${COMPOSE_PROJECT_NAME}-postgres
    restart: no
    environment:
      POSTGRES_DB: e20proddb
      POSTGRES_USER: e20ProdUser
      POSTGRES_PASSWORD: e20ProdPassword
    volumes:
      - e20-prod-pgdata:/var/lib/postgresql/data
    ports:
      - "5432:5432"
    networks:
      - e20-prod-network

```

Per evitare di dover eseguire una serie di comandi ogni volta che l'applicazione veniva aggiornata, è stato creato anche un piccolo script bash che semplifica il tutto.

Fa semplicemente il pull dell'immagine da gcr e poi restarta docker compose.

```

#!/bin/bash

docker pull eu.gcr.io/e-20-408110/e-20-api-main:latest
docker compose stop
docker compose up -d

```

## Utilizzo dell'applicazione

L'applicazione è visualizzabile all'indirizzo <https://e-20.net>.

Le credenziali a disposizione per effettuare il login sono:

<b>email</b>	<b>password</b>
giuliopime@gmail.com	g29-default
default@gmail.com	g29-default

alessandro.tomasi@unitn.it	g29-default
antonio.buccharone@unitn.it	g29-default

## Esecuzione in locale

Per eseguire l'applicazione in locale bisogna innanzitutto avere Docker, Java, JDK 21 e Flutter installato e disponibile nella path.

Per isogna poi creare l'environment di test tramite docker compose eseguendo il comando `docker compose -f ./docker-compose/docker-compose-dev.yml up -d`. Fatto ciò si copia il file [/env/.env.development](#) nella root del progetto, rinominandolo `.env`.

A questo punto si può eseguire il comando `chmod +x ./gradlew` per rendere il file gradlew esequibile e `./gradlew run` per runnare il progetto.

Dovrebbero apparire dei log nella console come segue:

Per runnare il frontend basta eseguire il comando `flutter run -d chrome` nel progetto del frontend.

# Dashboard

Sono disponibili inoltre diverse dashboard, ad esempio per la visualizzazione dei dati di Redis e PostgreSQL:

<b>Service</b>	<b>username</b>	<b>password</b>	<b>web dashboard</b>
PostgreSQL	e20DevUser	e20DevPassword	localhost:8081
Redis			localhost:8082

Grafana	admin	admin	localhost:3000
---------	-------	-------	----------------

Per collegarsi ad esempio alla dashboard di PostgreSQL all'url <http://localhost:8081> si usano i parametri:

- System: PostgreSQL
- Server: e20-dev-env-postgres
- User: e20DevUser
- Password: e20DevPassword
- Database: e20devdb

Quella che si ottiene è una dashboard minimal ma funzionale per visualizzare e manipolare i dati del database.

The screenshot shows the Adminer 4.8.1 interface. At the top, it displays the URL "PostgreSQL » e20-dev-env-postgres » e20devdb » Schema: public". On the left, there's a sidebar with "Language: English" and a "Logout" button. Below that, it says "Adminer 4.8.1" and shows "DB: e20devdb" and "Schema: public". Under "Tables and views", there's a search bar and a table listing five tables: eventplace, events, flyway\_schema\_history, passwordreset, and users. The table has columns for Table, Engine, Collation, Data Length, Index Length, Data Free, Auto Increment, Rows, and Comment. At the bottom, there's a section for "Selected (0)" with buttons for Vacuum, Optimize, Truncate, and Drop, and a "Move to other database" button set to "public".

Un'altra dashboard disponibile è quella di Redis, reperibile all'url <http://localhost:8082>.

Una volta aperto l'url, si clicca su "I already have a database", poi "Connect a database" e si inseriscono:

- host: e20-dev-env-redis
- port: 6379
- name: e20 (indifferente)

Tramite la dashboard si possono vedere alcune statistiche:



Ma soprattutto i dati presenti nella cache Redis, qui ad esempio si vede una sessione di un utente autenticato:

redisinsight e20 Browser Real time view of data in your Redis database

**OVERVIEW**

**BROWSE**

- Browser
- CLI
- Streams
- RedisGraph
- RedisGears **β**
- RedisAI **β**
- RedisTimeSeries
- RedisSearch

**ANALYSE**

Keys Database: 0

\* 🔍 ⟳

**sessions:bc9e631a-5e19-4885-ac01-1a3c48ffe9d3:0ff257e1-df01-4086-a57e-3a41d2609050**

String (294) | 456 B | TTL: 604752

🔗 🗑️

```
{
  "id": "0ff257e1-df01-4086-a57e-3a41d2609050",
  "userId": "bc9e631a-5e19-4885-ac01-1a3c48ffe9d3",
  "iat": 1707096494890,
  "deviceName": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/17.1 Safari/605.1.15",
  "ip": "localhost"
}
```

**JSON**