1) Creazione dipendenze in package.json  
2) Creazione script per runnare il progetto  
3) Creazione migrations con sequelize-cli per creare le tabelle nel db Postgres in automatico  
4) Creazione utente admin

**Dipendenze del progetto e librerie varie**

**Backend**

* **Node.js + Express:** API REST, middleware di sicurezza, error handling centralizzato.
* **Auth:** JWT access (breve) in Authorization: Bearer …, JWT refresh in cookie httpOnly (SameSite=Strict, Secure in prod).
* **RBAC:** ruoli USER | ADMIN + middleware requireRole e requireOwnershipOrAdmin.
* **Password:** bcrypt con salt rounds 12 (prod) / 10 (dev) + pepper opzionale (da .env).
* **DB:** PostgreSQL con Sequelize (migrazioni CLI, query parametrizzate, tipi GEOMETRY/GEOGRAPHY).
* **Geo:** PostGIS (colonna geography(Point,4326) su Event, indice GIST, query “entro raggio” con funzioni ST\_\*).
* **Realtime:** Socket.IO (1 istanza, niente Redis per ora). Rooms per eventId e room admin.
* **Upload immagini:** S3 con Presigned URL (upload diretto dal browser), validazione mime/size server-side.
* **Doc API:** OpenAPI 3 con swagger-jsdoc + swagger-ui-express**.**
* **Sicurezza:** helmet, cors (whitelist), express-rate-limit (login e API sensibili), hpp, xss-clean, express-validator.
* **Logging:** Pino + pino-http (log JSON, redaction password/token, requestId).
* **Container:** Docker (compose per dev/prod).

**Frontend (vanilla)**

* **HTML + CSS/SCSS + JS (senza framework).**
* **UI:** Bootstrap 5 + FontAwesome + SASS personalizzato (variabili/mixins, tema leggero).
* **Build CSS:** sass (compila src/scss → public/assets/css).
* **Dev server:** lite-server (statico con live reload).
* **Routing:** pagine/ancore HTML tradizionali (niente SPA).
* **Data fetching:** Fetch API (o axios se preferisci, ma non è obbligatorio).
* **Form:** validazione HTML5 + piccole regole JS; lato server restano i controlli con express-validator.
* **Realtime:** socket.io-client autenticato con access token.
* **Sicurezza UI:** chat solo testo (niente HTML da utenti).

**Analytics / CDC**

* **Power BI in read-only su PostgreSQL** (utenza DB dedicata + view ad hoc: es. vw\_event\_registrations).

**Cosa sono (in breve):**

* **express core; helmet/cors/rate-limit/hpp/xss-clean/express-validator sicurezza e validazione**
* **jsonwebtoken + cookie-parser per JWT access/refresh**
* **bcrypt hashing password**
* **socket.io realtime**
* **swagger-jsdoc/swagger-ui-express documentazione API**
* **pino/pino-http logging JSON con redaction e requestId**
* **dayjs/uuid utility**
* **@aws-sdk/\* S3 presigned URL**
* **sequelize / sequelize-cli / pg / pg-hstore ORM + driver Postgres**

**Dev dipendenze (consigliate)**

* **npm i -D nodemon pino-pretty**

**Frontend (eventhub/client)**

**Dipendenze runtime (obbligatorie)**

* **bootstrap stile + componenti**
* **@fortawesome/fontawesome-free icone (versione per progetti senza React)**
* **socket.io-client per la chat/notifiche realtime**
* **axios se preferisci a fetch() (non è obbligatorio)**

**Dev dipendenze (obbligatorie per SASS + dev server)**

* **sass compila SCSS → CSS**
* **lite-server server statico con live reload**
* **concurrently per eseguire insieme watcher SASS + web server**

**Aggiorniamo gli scripts di package.json**{

"scripts": {

"dev:server": "NODE\_ENV=development nodemon server/src/server.js | pino-pretty",

"dev:sass": "sass --load-path=node\_modules client/src/scss/main.scss client/public/assets/css/main.css --watch",

"dev:client": "lite-server --config=client/bs-config.json",

"dev": "concurrently -n SASS,WEB,API -c cyan,magenta,blue \"npm run dev:sass\" \"npm run dev:client\" \"npm run dev:server\"",

"db:migrate": "sequelize-cli db:migrate",

"db:migrate:undo": "sequelize-cli db:migrate:undo",

"db:seed": "sequelize-cli db:seed:all",

"db:seed:undo": "sequelize-cli db:seed:undo:all"

}

}

**1) "dev:server" — avvia l’API in modalità sviluppo**

* NODE\_ENV=development → imposta l’ambiente “dev” (feature e log più verbosi, corsi dev ecc.).
* nodemon server/src/server.js → esegue il tuo server Express e **riavvia automaticamente** quando modifichi file.
* | pino-pretty → “abbellisce” i log JSON prodotti da Pino in un formato leggibile in console (solo in dev).

💡 **Nota Windows**: NODE\_ENV=development *non funziona* su Windows cmd. Per compatibilità totale usa **cross-env**:

npm i -D cross-env  
  
Che cambia lo script così:

"dev:server": "cross-env NODE\_ENV=development nodemon server/src/server.js | pino-pretty"

**2) "dev:sass" — compila SASS → CSS con watcher**

* sass --load-path=node\_modules → abilita gli @import direttamente dai pacchetti (es. @import "bootstrap/scss/bootstrap").
* client/src/scss/main.scss → **file sorgente** SASS.
* client/public/assets/css/main.css → **file CSS di output** che userà l’HTML.
* --watch → rimane in ascolto e ricompila quando salvi.

Effetto: ogni volta che modifichi SCSS, il CSS nella public/ viene aggiornato e il browser (via lite-server) ricarica.

**3) "dev:client" — serve il frontend statico con live reload**

* lite-server → server statico leggero che serve client/public/.
* --config=client/bs-config.json → file di configurazione (porta, cartella, ecc.).

Esempio **client/bs-config.json** consigliato:

{

"port": 5173,

"server": { "baseDir": "client/public" },

"files": ["client/public/\*\*/\*"]

}

Così apri il frontend su http://localhost:5173, con **auto-reload** su modifiche a HTML/CSS/JS.

**4) "dev" — avvia tutto in parallelo**

* concurrently lancia **tre processi** insieme: SASS (watch), WEB (lite-server) e API (nodemon).
* -n SASS,WEB,API → etichette per le tre pipeline in console.
* -c cyan,magenta,blue → colori per riconoscere i log (comodissimo).

Risultato: un **unico comando** (npm run dev) attiva frontend + CSS watcher + backend in parallelo.  
Per fermare tutto: **Ctrl+C** una volta (magari due se chiede conferma).

**Script DB (Sequelize CLI)**

Questi usano **sequelize-cli** e i percorsi che hai impostato in **.sequelizerc** (in root). La CLI leggerà il config da server/src/db/config/config.js. Assicurati che in quel file ci sia:

require('dotenv').config(); // così prende la .env dalla root

**5) "db:migrate"**

Esegue **tutte le migrazioni pendenti** in server/src/db/migrations/ sull’ambiente **development** (default CLI).

* Genera una migrazione:
* npx sequelize-cli migration:generate --name create-users
* Poi **la applichi**:
* npm run db:migrate

**6) "db:migrate:undo"**

**Rollback** dell’ultima migrazione applicata (una sola per volta). Utile quando hai sbagliato la migrazione.

npm run db:migrate:undo

**7) "db:seed"**

Esegue **tutti i seeder** in server/src/db/seeders/. Tipico per inserire un utente admin di test, categorie, ecc.

* Genera un seeder:
* npx sequelize-cli seed:generate --name seed-admin-user
* Applica:
* npm run db:seed

**8) "db:seed:undo"**

Rimuove **tutti** i dati inseriti dai seeders (esegue le funzioni down dei seed).  
⚠️ Attenzione in ambienti reali: di solito **non** si eseguono i seed in produzione.

**Requisiti**1. **.sequelizerc (root)** deve puntare alle cartelle sotto server/src/db/...:

const path = require('path');

module.exports = {

'config': path.resolve('server/src/db/config/config.js'),

'models-path': path.resolve('server/src/db/models'),

'seeders-path': path.resolve('server/src/db/seeders'),

'migrations-path': path.resolve('server/src/db/migrations')

};

1. **Config DB** (server/src/db/config/config.js) deve caricare .env dalla **root**:

require('dotenv').config(); // senza path -> .env nella root del progetto

E definire:  
  
module.exports = {

development: { dialect: 'postgres', url: process.env.DATABASE\_URL, /\* ... \*/ },

/\* test, production ... \*/

}

3. **DATABASE\_URL** deve essere valorizzata nella .env (root), per esempio in dev con Docker:

DATABASE\_URL=postgresql://eventhub:eventhub@localhost:5432/eventhub?schema=public

(se usi compose e fai girare il server in container, dentro il container l’host sarà postgres).

**Migrazioni delle tabelle**

* Pensa alle migrazioni come a **commit del database**.
* Ogni migrazione è un file con **due funzioni**:
  + up: *applica* una modifica (es. crea tabella/indice/colonna).
  + down: *annulla* quella modifica (rollback).
* Le migrazioni sono **ordinate** (per timestamp nel nome file) e Sequelize tiene traccia di quelle già eseguite in una tabella interna (SequelizeMeta).
* Vantaggi:
  + Tutti i dev e gli ambienti (dev/stage/prod) hanno **lo stesso schema**, riproducibile.
  + Puoi **andare avanti/indietro** in modo controllato.
  + Lo **schema è versionato** come il codice.

**Seeder**: simili alle migrazioni, ma per **dati di esempio/iniziali** (es. utente admin). Sequelize li traccia in SequelizeData.

**Modello dati (semplice e chiaro)**

Tabelle principali (campi essenziali, niente “magia”):

* **users**: id (uuid), email (unique), password\_hash, role ('user'|'admin'), created\_at.
* **events**: id, owner\_id (fk users), title, description, starts\_at, location\_lat, location\_lng, capacity, seats\_taken (default 0), category, status ('pending'|'approved'|'rejected'), created\_at.
* **registrations**: id, event\_id (fk), user\_id (fk), created\_at, **UNIQUE** (event\_id,user\_id).
* **messages**: id, event\_id (fk), user\_id (fk), body, created\_at.

Nota: seats\_taken lo aggiorniamo in transazione quando uno si iscrive/disiscrive (così evitiamo overbooking).

**Cosa fa la “Sequelize CLI” e gli script db:\***

Gli script si basano su sequelize-cli e su un file di configurazione che gli dice:

* **dove** sono migrations/, seeders/, models/
* **come** connettersi (URL del DB) in base all’**ambiente** (development, test, production)

Gli script che userai:

* **npm run db:migrate**  
  Esegue **tutte le migrazioni pendenti** (chiama up in ordine). Registra i nomi in SequelizeMeta.
* **npm run db:migrate:undo**  
  **Torna indietro di una** migrazione (chiama il down dell’ultima eseguita).  
  (Esistono anche db:migrate:undo:all e opzioni per tornare a uno specifico stato, ma per ora basta così.)
* **npm run db:seed**  
  Esegue **tutti i seeders** (funzioni up). Registra i file eseguiti in SequelizeData.
* **npm run db:seed:undo**  
  Esegue i down dei seeders (li annulla).

L’**ambiente** usato (development/production) è preso da NODE\_ENV al momento dell’esecuzione. In dev basta lanciare senza specificare: userà development.

**Creazione dei file per il database Postgres**

**Da zero: crea i file minimi per usare le migrazioni**

Hai detto che hai **solo installato le dipendenze**. Perfetto: facciamo ora i file fondamentali.

**1) Crea .env (root del progetto)**

Contiene la connessione e i segreti base (per ora basta la DB URL):

NODE\_ENV=development

# Postgres locale (porta 5432). Se usi Docker compose, host potrebbe essere "localhost".

DATABASE\_URL=postgresql://eventhub:eventhub@localhost:5432/eventhub?schema=public

# JWT (serviranno più avanti)

JWT\_ACCESS\_SECRET=dev\_access

JWT\_REFRESH\_SECRET=dev\_refresh

# S3 placeholder (più avanti)

AWS\_REGION=eu-central-1

S3\_BUCKET=eventhub-dev

**2) Crea .sequelizerc (root)**

Dice alla CLI **dove** mettere/leggere i file. Anche se il resto del progetto è ESM, questo file va in **CommonJS** (richiesto da sequelize-cli).

.sequelizerc  
  
const path = require('path');

module.exports = {

'config': path.resolve('server/src/db/config/config.js'),

'models-path': path.resolve('server/src/db/models'),

'seeders-path': path.resolve('server/src/db/seeders'),

'migrations-path': path.resolve('server/src/db/migrations')

};

**3) Inizializza la struttura della CLI**

npx sequelize-cli init  
  
Ti crea:

server/src/db/

├─ config/config.js

├─ models/index.js

├─ migrations/

└─ seeders/

**4) Configura la connessione DB con .env**

Apri **server/src/db/config/config.js** e sostituisci tutto con:

// CommonJS per compatibilità con sequelize-cli

require('dotenv').config();

const common = {

dialect: 'postgres',

logging: false,

define: {

freezeTableName: true, // il nome tabella è quello che specifichi, non pluralizzato

timestamps: true // createdAt/updatedAt di default

}

};

module.exports = {

development: {

...common,

url: process.env.DATABASE\_URL

},

test: {

...common,

url: process.env.DATABASE\_URL\_TEST || process.env.DATABASE\_URL

},

production: {

...common,

url: process.env.DATABASE\_URL,

dialectOptions: {

// ssl: { require: true, rejectUnauthorized: false } // scommenta se usi PG gestito con SSL

}

}

};

La CLI userà development quando NODE\_ENV=development (default).

**Le nostre prime migrazioni (PostGIS + creazione tabelle)**

Useremo **4 migrazioni** nell’ordine:

1. **enable-postgis**: abilita l’estensione PostGIS nel DB (necessaria per il tipo GEOGRAPHY).
2. **create-users**
3. **create-events** (con location GEOGRAPHY(Point,4326) e indice GIST)
4. **create-registrations**
5. **create-messages** (puoi farne due separate o una sola; sotto le separo per chiarezza)

Genera i file con la CLI, poi incolla il contenuto che ti metto qui.

**1) Migrazione: abilita PostGIS**

Genera:

npx sequelize-cli migration:generate --name enable-postgis

Apri il file creato in server/src/db/migrations/\*-enable-postgis.js e incolla:

'use strict';

module.exports = {

async up(queryInterface) {

// Abilita PostGIS (serve ruolo owner del DB o superuser)

await queryInterface.sequelize.query('CREATE EXTENSION IF NOT EXISTS postgis;');

},

async down(queryInterface) {

// Disabilita PostGIS (ATTENZIONE: se ci sono colonne geography/geometry, fallirà)

await queryInterface.sequelize.query('DROP EXTENSION IF EXISTS postgis;');

}

};

In pratica non farai quasi mai il down di PostGIS se hai già creato colonne geo. Va bene così per l’esercizio.

**2) Migrazione: Users**

Genera:

npx sequelize-cli migration:generate --name create-users

**Contenuto** (server/src/db/migrations/\*-create-users.js):

'use strict';

module.exports = {

async up(queryInterface, Sequelize) {

await queryInterface.createTable('Users', {

id: {

type: Sequelize.UUID,

defaultValue: Sequelize.literal('gen\_random\_uuid()'),

allowNull: false,

primaryKey: true

},

email: {

type: Sequelize.STRING(180),

allowNull: false,

unique: true

},

passwordHash: {

type: Sequelize.STRING(255),

allowNull: false

},

role: {

type: Sequelize.ENUM('USER', 'ADMIN'),

allowNull: false,

defaultValue: 'USER'

},

createdAt: { type: Sequelize.DATE, allowNull: false, defaultValue: Sequelize.literal('NOW()') },

updatedAt: { type: Sequelize.DATE, allowNull: false, defaultValue: Sequelize.literal('NOW()') }

});

// Estensione per gen\_random\_uuid su Postgres (se non esiste)

await queryInterface.sequelize.query('CREATE EXTENSION IF NOT EXISTS pgcrypto;');

},

async down(queryInterface, Sequelize) {

await queryInterface.dropTable('Users');

await queryInterface.sequelize.query('DROP TYPE IF EXISTS "enum\_Users\_role";'); // pulizia ENUM

}

};

Se pgcrypto non è disponibile, puoi usare uuid-ossp o generare UUID lato app. L’immagine Postgres recente spesso ha pgcrypto.

**3) Migrazione: Events (con GEOGRAPHY(Point,4326))**

Genera:

npx sequelize-cli migration:generate --name create-events

**Contenuto** (\*-create-events.js):

'use strict';

module.exports = {

async up(queryInterface, Sequelize) {

await queryInterface.createTable('Events', {

id: {

type: Sequelize.UUID,

defaultValue: Sequelize.literal('gen\_random\_uuid()'),

allowNull: false,

primaryKey: true

},

ownerId: {

type: Sequelize.UUID,

allowNull: false,

references: { model: 'Users', key: 'id' },

onDelete: 'CASCADE',

onUpdate: 'CASCADE'

},

title: { type: Sequelize.STRING(200), allowNull: false },

description: { type: Sequelize.TEXT, allowNull: true },

startsAt: { type: Sequelize.DATE, allowNull: false },

capacity: { type: Sequelize.INTEGER, allowNull: false },

seatsTaken: { type: Sequelize.INTEGER, allowNull: false, defaultValue: 0 },

category: { type: Sequelize.STRING(80), allowNull: true },

status: {

type: Sequelize.ENUM('PENDING', 'APPROVED', 'REJECTED'),

allowNull: false,

defaultValue: 'PENDING'

},

imageUrl: { type: Sequelize.STRING(500), allowNull: true },

// Colonna geospaziale (richiede estensione postgis attiva)

location: {

type: Sequelize.GEOGRAPHY('POINT', 4326),

allowNull: true

},

createdAt: { type: Sequelize.DATE, allowNull: false, defaultValue: Sequelize.literal('NOW()') },

updatedAt: { type: Sequelize.DATE, allowNull: false, defaultValue: Sequelize.literal('NOW()') }

});

// Indici utili

await queryInterface.addIndex('Events', ['startsAt']);

await queryInterface.addIndex('Events', ['category']);

// Indice GIST sulla colonna geografia (per ST\_DWithin)

await queryInterface.sequelize.query('CREATE INDEX "event\_location\_gix" ON "Events" USING GIST ("location");');

},

async down(queryInterface) {

await queryInterface.removeIndex('Events', ['startsAt']);

await queryInterface.removeIndex('Events', ['category']);

await queryInterface.sequelize.query('DROP INDEX IF EXISTS "event\_location\_gix";');

await queryInterface.dropTable('Events');

await queryInterface.sequelize.query('DROP TYPE IF EXISTS "enum\_Events\_status";');

}

};

4) Migrazione: Registrations

Genera:

npx sequelize-cli migration:generate --name create-registrations

Contenuto (\*-create-registrations.js):

'use strict';

module.exports = {

async up(queryInterface, Sequelize) {

await queryInterface.createTable('Registrations', {

id: {

type: Sequelize.UUID,

defaultValue: Sequelize.literal('gen\_random\_uuid()'),

allowNull: false,

primaryKey: true

},

eventId: {

type: Sequelize.UUID,

allowNull: false,

references: { model: 'Events', key: 'id' },

onDelete: 'CASCADE',

onUpdate: 'CASCADE'

},

userId: {

type: Sequelize.UUID,

allowNull: false,

references: { model: 'Users', key: 'id' },

onDelete: 'CASCADE',

onUpdate: 'CASCADE'

},

createdAt: { type: Sequelize.DATE, allowNull: false, defaultValue: Sequelize.literal('NOW()') },

updatedAt: { type: Sequelize.DATE, allowNull: false, defaultValue: Sequelize.literal('NOW()') }

});

// UNIQUE(eventId, userId) evita doppie iscrizioni allo stesso evento

await queryInterface.addConstraint('Registrations', {

fields: ['eventId', 'userId'],

type: 'unique',

name: 'uk\_registrations\_event\_user'

});

},

async down(queryInterface) {

await queryInterface.removeConstraint('Registrations', 'uk\_registrations\_event\_user');

await queryInterface.dropTable('Registrations');

}

};

5) Migrazione: Messages (chat)

Genera:

npx sequelize-cli migration:generate --name create-messages

Contenuto (\*-create-messages.js):

'use strict';

module.exports = {

async up(queryInterface, Sequelize) {

await queryInterface.createTable('Messages', {

id: {

type: Sequelize.UUID,

defaultValue: Sequelize.literal('gen\_random\_uuid()'),

allowNull: false,

primaryKey: true

},

eventId: {

type: Sequelize.UUID,

allowNull: false,

references: { model: 'Events', key: 'id' },

onDelete: 'CASCADE',

onUpdate: 'CASCADE'

},

userId: {

type: Sequelize.UUID,

allowNull: false,

references: { model: 'Users', key: 'id' },

onDelete: 'CASCADE',

onUpdate: 'CASCADE'

},

body: { type: Sequelize.TEXT, allowNull: false },

createdAt: { type: Sequelize.DATE, allowNull: false, defaultValue: Sequelize.literal('NOW()') },

updatedAt: { type: Sequelize.DATE, allowNull: false, defaultValue: Sequelize.literal('NOW()') }

});

await queryInterface.addIndex('Messages', ['eventId', 'createdAt']);

},

async down(queryInterface) {

await queryInterface.removeIndex('Messages', ['eventId', 'createdAt']);

await queryInterface.dropTable('Messages');

}

};

**Come si eseguono le migrazioni (ordine consigliato)**

1. Assicurati che Postgres sia acceso (locale o via Docker).
2. Lancia, da root del progetto: npm run db:migrate  
     
   Verranno eseguite in ordine:  
   enable-postgis → create-users → create-events → create-registrations → create-messages.
3. Verifica con pgAdmin o psql che ci siano tabelle e indici:
   * Users, Events (con colonna location), Registrations, Messages
   * indice event\_location\_gix su Events
4. (quando vorrai) crea seed di un admin: **npx sequelize-cli seed:generate --name seed-admin**

E nel file:

'use strict';

const bcrypt = require('bcrypt');

module.exports = {

async up (queryInterface) {

const hash = await bcrypt.hash('AdminPassword123!', 10);

await queryInterface.bulkInsert('Users', [{

id: null, // usiamo default gen\_random\_uuid

email: 'admin@eventhub.local',

passwordHash: hash,

role: 'ADMIN',

createdAt: new Date(),

updatedAt: new Date()

}]);

},

async down (queryInterface) {

await queryInterface.bulkDelete('Users', { email: 'admin@eventhub.local' });

}

};

Poi:

npm run db:seed

Domande frequenti (brevi)

* Se sbaglio una migrazione?  
  npm run db:migrate:undo per annullare l’ultima, correggi il file, rilancia db:migrate.
* Come forzo l’ambiente?  
  NODE\_ENV=production npm run db:migrate (su Windows usa cross-env NODE\_ENV=production).
* Se ho già tabelle e voglio ricominciare?  
  In sviluppo puoi droppare il database e rieseguire db:migrate. (In produzione: mai senza un piano di migrazione dati!)

**Flussi chiave (step-by-step “da junior”)**

1. **Auth**
   * POST /auth/register: crea utente (hash password con bcrypt).
   * POST /auth/login: genera JWT access + refresh (refresh in cookie httpOnly).
   * Middleware requireAuth legge JWT e mette req.user.
   * requireRole('admin') per rotte admin.
2. **Eventi**
   * CRUD proprietario (solo owner può modificare/cancellare).
   * Lista pubblica filtrabile per categoria, data, distanza (lat/lng semplici).
3. **Iscrizioni con capienza**
   * POST /events/:id/registrations: in **transazione**:  
     a) controlla seats\_taken < capacity  
     b) crea registration (UNIQUE evita doppioni)  
     c) seats\_taken = seats\_taken + 1
   * Disiscrizione fa l’inverso.
4. **Chat realtime**
   * Socket.IO, stanza per eventId.
   * All’evento di messaggio: salva su messages e poi io.to(eventId).emit(...).
   * Niente Redis adesso.
5. **Notifiche live**
   * Quando qualcuno si iscrive/disiscrive: emettiamo evento nella stanza dell’evento e nella stanza admin.
6. **Upload immagini**
   * multer salva file su server/src/uploads.
   * (Opzionale) sharp per resize.
7. **Doc API**
   * Swagger UI su /api-docs.