TrackER Project

Applicazioni e Servizi Web

Emanuele Dall'Ara - 0001061501
emanuele.dallara@studio.unibo.it

Nicholas Ricci - 0001036866
nicholas.ricci@studio.unibo.it

12 Settembre 2022

Indice

Capitolo 1

Introduzione

1.1 Contesto di riferimento

Il progetto nasce come una soluzione software web-based di **gestione, tracciamento e vendita dell'energia** sia dal punto di vista dei venditori (**Vendor**) sia per gli utenti consumatori (**Buildings Owner**).

L'obiettivo primario è quello di consentire a ciascun utente di poter visionare e gestire i propri edifici per quanto riguarda la loro produzione (tramite fonti rinnovabili) ed il loro consumo di risorse energetiche.

La web application si compone di due principali interfacce: una relativa al Vendor ed una al Buildings Owner raggiungibili soltanto una volta dopo essersi registrati nel sistema.

Capitolo 2

Requisiti

2.1 Requisiti Utente

Di seguito sono elencate le principali funzionalità per ogni tipo di account:

- **Vendor**:
  - Creare un'organizzazione selezionando il tipo di energie che mette a disposizione dei consumatori;
  - Controllare la mole di dati energetici prodotta e consumata;
  - Modificare alcune informazioni relative all'organizzazione;
  - Controllare i guadagni (totali e relativi ai singoli edifici);
  - Controllare i costi di produzione e delle imposte (totali e relativi ai singoli edifici);
  - Controllare lo stato di produzione dei dispositivi installati;
  - Personalizzare il proprio account.
- **Buildings Owner**:
  - Aggiungere i propri edifici sulla piattaforma;
  - Modificare alcune informazioni dei propri edifici registrati;
  - Selezionare il contratto di fornitura d'energia che si vuole sottoscrivere per ciascun edificio;
  - Monitorare i consumi dei propri edifici;

  - Installare dispositivi di energie rinnovabili;
  - Visualizzare la mole di energia prodotta da fonti rinnovabili e il corrispondente guadagno;
  - Personalizzare il proprio account.

2.2 Requisiti Funzionali

- Registrazione di un nuovo utente;
- Generazione di notifiche e/o avvisi sullo stato dei dati.

2.3 Requisiti Non Funzionali

- Rendere il sistema facile ed intuitivo;
- Rendere il sistema sicuro mediante meccanismi di autenticazione, autorizzazione e con l'ausilio della crittografia;
- Rendere il sistema reattivo;
- Rendere il sistema responsive.

Capitolo 3

Design

Il modello utilizzato è di tipo iterativo e basato su User Centered Design con utenti virtualizzati. Sono stati individuati gli utenti target dell'applicazione e su di essi si sono sviluppate le Personas che hanno contribuito a sviluppare e comprendere le funzionalità del sistema e i relativi requisiti utente. Il team ha gestito tutto il progetto tramite **Issue** su **GitHub**. Ogni issue aveva un certo grado di priorità permettendo così di organizzare lo sviluppo in vari sprint. Quindi si è seguito un approccio simile a quello **AGILE** che ha permesso, in modo incrementale, lo sviluppo delle varie funzionalità in base alla priorità. Nello svolgimento del progetto e del design delle interfacce si è cercato di rispettare il principio di **KISS** per risolvere le funzionalità nel modo più semplice possibile. Nello sviluppo delle interfacce si è sempre tenuto in considerazione la riusabilità di esse. Nello sviluppare l'applicazione sono state utilizzate tecnica di Responsive Design, in modo che le pagine web adattino automaticamente il layout per fornire una visualizzazione ottimale in funzione dell'ambiente nei quali vengono visualizzati: pc, tablet, smartphone sono i principali.

3.1 Mockup

Per la realizzazione dei mockup si è preferito svolgere disegni a penna su carta delle interfacce principali. Questi rappresentano una versione semplificata del prodotto finale.

3.1.1 Sign In/Up page

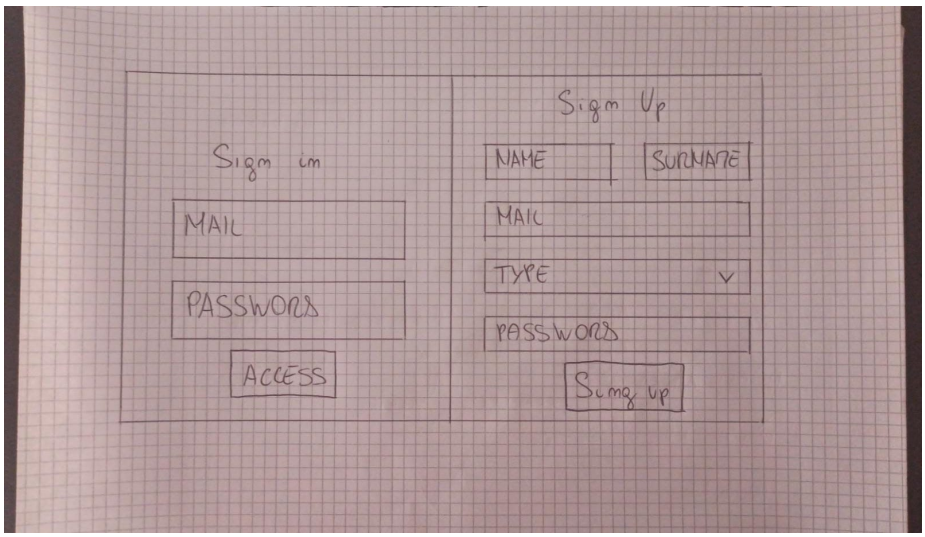

Figura 3.1: Sign In/Up page.

Per la form di Sign In/Up all'applicazione si è pensato di suddividere l'accesso e la registrazione all'applicazione in una singola schermata per non disperdere l'utente nelle varie pagine del sistema. La registrazione richiederà il tipo di utente che si vuole creare (Vendor o Buildings Owner) mentre per accedere al sistema sarà necessario effettuare il Sign In tramite Mail e Password.

3.1.2 Account page

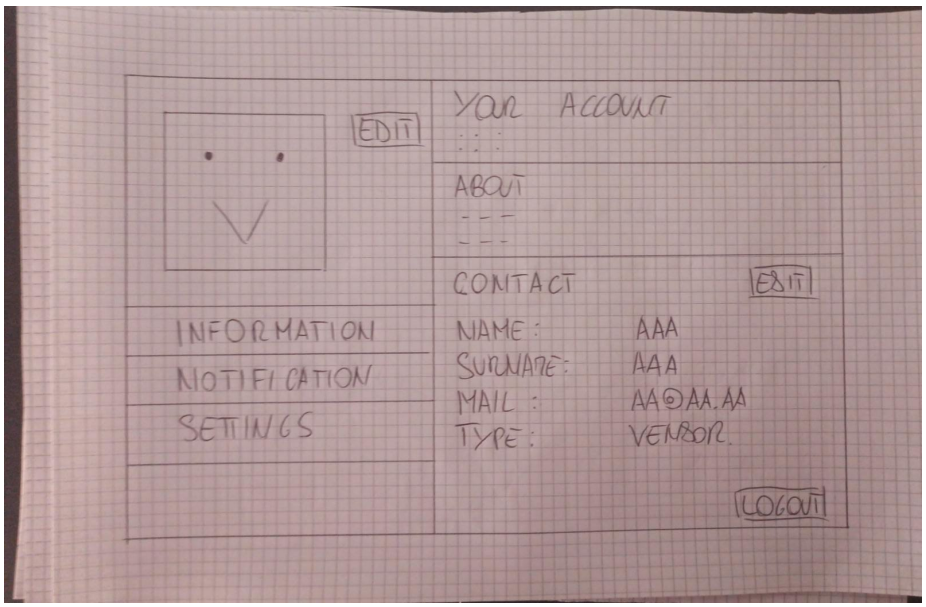

Figura 3.2: Account page.

Per la pagina Account si è pensato di creare due sezioni:

- una con un menù di navigazione sulle varie iterazioni che l'utente può fare con il suo account(parte sinistra di figura 3.2);
- una inerente alla renderizzazione dei dati provenienti dalla sezione precedente (parte destra di figura 3.2)

La differenza sostanziale tra le due tipologie di account per questa pagina sarà inerente alla personalizzazione dell'avatar in cui, un Vendor, potrà caricare il logo della sua organizzazione mentre un Buildings Owner potrà selezionare un avatar presente nel sistema.

3.1.3 Vendor Dashboard

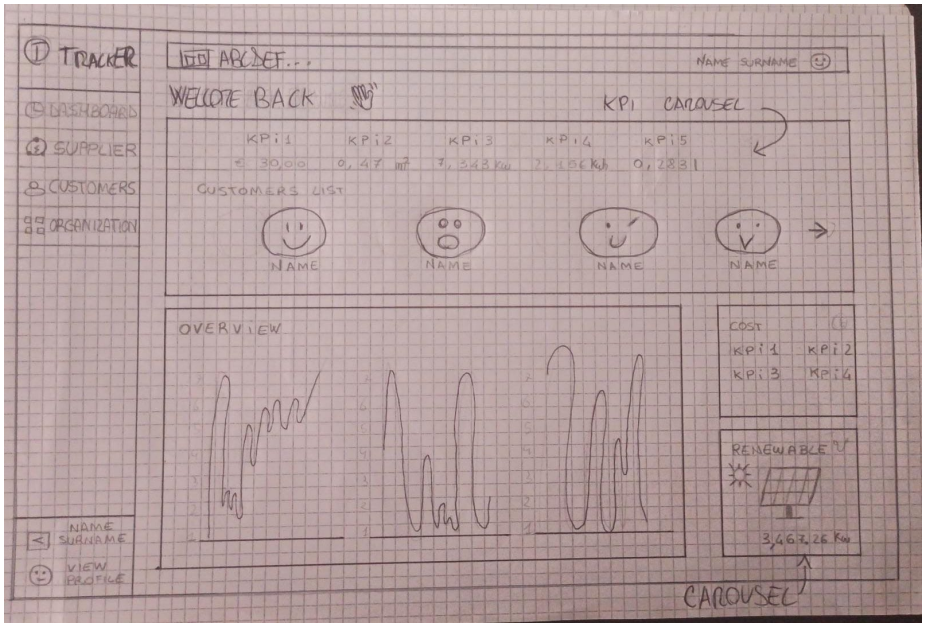

Figura 3.3: Vendor Dashboard.

La Vendor DashBoard avrà come obiettivo principale quello di dare una visione generale sui costi e guadagni di un Vendor. Questa pagina sarà composta da grafici inerenti ai consumi utente, un collegamento veloce agli edifici utente e una vasta gamma di KPI.

3.1.4 Buildings Owner Dashboard

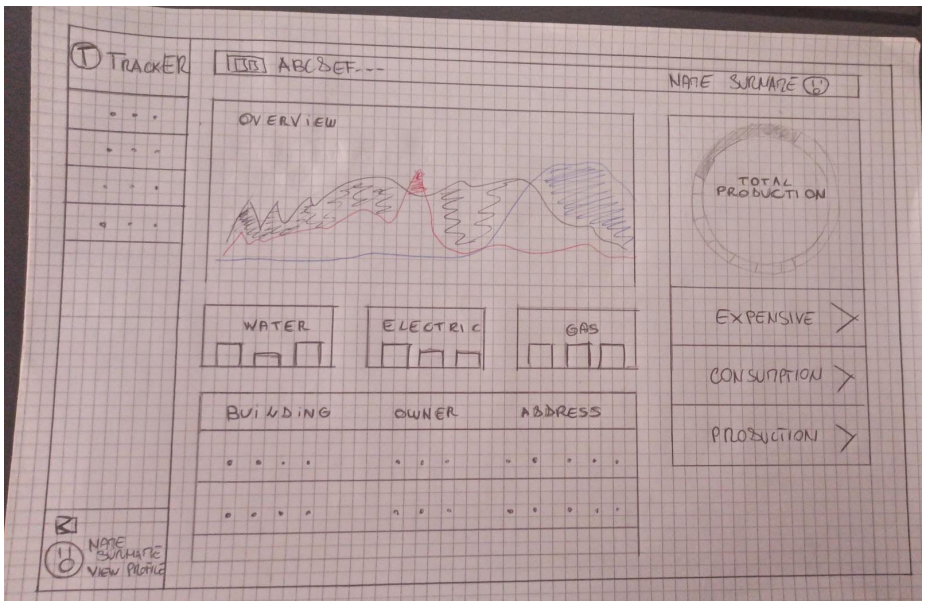

Figura 3.4: Buildings Owner Dashboard.

La Buildings Owner DashBoard sarà molto simile alla Vendor DashBoard ma il suo obiettivo principale sarà quello di dare una visione generale sui consumi e risparmi energetici di un Buildings Owner. Questa pagina avrà meno KPI rispetto a quella vendor ma conterrà più grafici inerenti ai consumi.

3.1.5 Buildings page

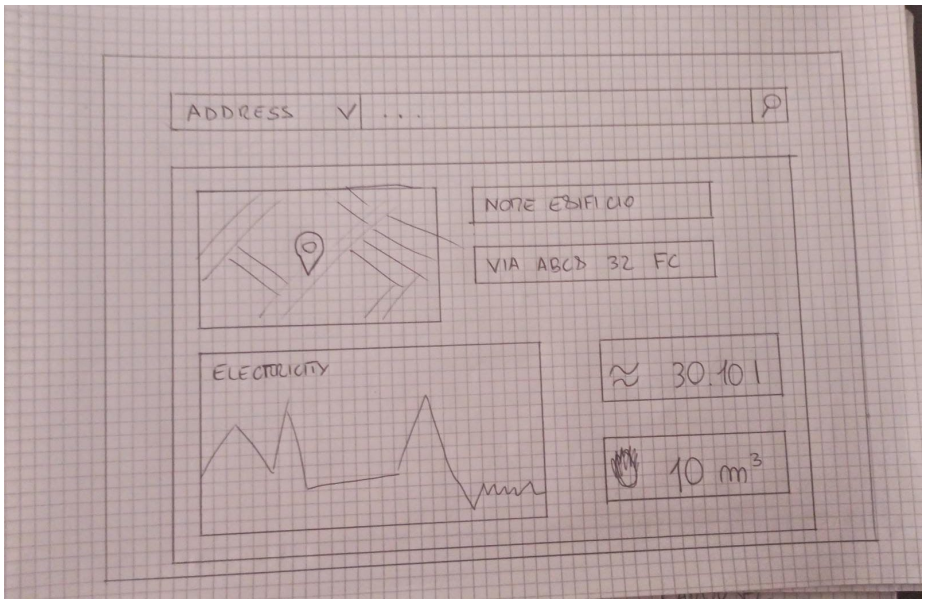

Figura 3.5: Buildings page.

La Buildings Page sarà composta da più interfacce che rappresentano un singolo edificio. Ogni edifico mostrerà la propria ubicazione tramite una mappa, le sue informazioni principali e i suoi costi totali. Si permetterà inoltre ad un utente di poter cercare i propri edifici tramite **nome edificio** e **indirizzo edificio**.

3.2 Analisi target user

Sono state costruite diverse Personas sui profili definiti in precedenza (Vendors e Buildings Owners):

- **Personas Gavin**:
  E' il proprietario di Energy Cop (che fornisce energia elettrica, gas e acqua) e vuole monitorare il numero di clienti affiliati e i suoi costi e guadagni totali.

  1. Si registra ed accede alla piattaforma come Vendor;
  2. Registra la sua organizzazione come fornitrice di energia elettrica, di gas e di acqua;

3. Accede all'applicazione e verifica i propri costi e guadagni totali.

- **Personas Tony**:
  Possiede un'impresa che fornisce dispositivi di energie rinnovabili (pannelli solari e fotovoltaici) ed ha bisogno di espandere la rete dei suoi clienti.

  1. Si registra ed accede alla piattaforma come Vendor;
  2. Registra la sua organizzazione come fornitrice di sole energie rinnovabili;
  3. Aggiunge i dispositivi che vende all'interno dell'applicazione;
  4. Accede all'applicazione e verifica il numero di clienti totali.

- **Personas Francesco**
  Possiede una vasta rete di edifici aziendali e vuole verificare i consumi dei propri edifici e possibilmente installare qualche dispositivo di energia rinnovabile per ridurre le spese totali.

  1. Si registra ed accede alla piattaforma come Buildings Owner;
  2. Registra i propri edifici sotto un'organizzazione fornitrice di energie;
  3. Controlla il consumo energetico dei propri edifici;
  4. Decide se installare fonti di energie rinnovabili per uno o più edifici.

- **Personas Jafar**
  Possiede una casa e in questo periodo vuole abbassare il più possibile i consumi elettrici e di gas con l'ausilio di fonti rinnovabili.

  1. Si registra ed accede alla piattaforma come Buildings Owner;
  2. Registra la propria casa sotto un'organizzazione fornitrice di energie;
  3. Controlla ed installa il dispositivo di energie rinnovabili più conveniente per lui;
  4. Monitora i costi delle proprie bollette.

Capitolo 4

Tecnologie

Il sistema è stato realizzato sulla base dello stack MERN che come l'acronimo descrive, comprende le seguenti tecnologie:

- MongoDB;
- Express;
- React;
- Node.js.

MERN è una variante dello stack MEAN che, utilizzando interamente **Javascript** e **JSON**, permette di costruire facilmente un'architettura a 3 livelli (frontend, backend, database).

Il progetto è stato realizzato grazie all'utilizzo di diverse tecnologie:

- Front-End:
  - **React**;
  - **Ant Design**;
  - **Redux**;
  - **Axios**;
  - **Styled-Components**;
  - **ApexCharts**;
  - **MapBoxGL**;
  - **Socket.IO**.

- Back-End:
  - **Express**;
  - **Moongose**;
  - **Bcrypt**;
  - **Nodemon**;
  - **Socket.IO**.

4.1 Front-End

4.1.1 React

React è un framework open sources scritto in JavaScript usato per sviluppare applicazioni web.
Si basa principalmente su ReactJS, ovvero una libreria open source Javascript fondata sul linguaggio JSX.

Il concetto principale su cui si basa React è il componente, ogni elemento presente in un'applicazione (Button, View, Dialog, Text, Image, ecc.) è riconducibile a un componente.
Ogni componente contribuisce alla creazione dell'interfaccia grafica e possiede un proprio stato, un insieme di funzioni, detti metodi, può accettare delle proprietà, e detiene un ciclo di vita. La caratteristica più importante dei componenti è che sono componibili: si può creare un componente complesso dall'unione di più componenti semplici e ciascun di essi sarà riutilizzabile, infatti, una volta definito, questo potrà essere chiamato e visualizzato in qualsiasi schermata dell'applicazione.

4.1.2 Ant Design

Ant Design è un sistema di progettazione composto da propri principi di progettazione, guide di stile e da una libreria di componenti scritta in TypeScript. Il design Ant basato sul popolare stack di React, Redux, React-Router. Ant Design fornisce una serie di elementi di interfaccia utente predefiniti di alta qualità per lo sviluppo e la manutenzione di applicazioni aziendali in background. Il pacchetto è implementato in AngularJS, React e Vue.js.

4.1.3 Redux

Redux è una libreria di state management creata appositamente per React. Permette l'utilizzo di uno "Store" globale che contiene idealmente l'intero stato dell'applicazione. Ogni componente può accedere a qualsiasi stato presente nello Store.

Questa libreria crea procedure e processi per interagire con lo Store in modo che i componenti non si aggiornino o leggano i vari stati presenti all'interno dello Store in modo casuale.

In breve, possiamo dire che Redux è una cache o uno storage a cui possono accedere tutti i componenti in modo strutturato.
Per accedere allo stato o modificarlo, è necessario utilizzare dei "Reducer" e delle "Action". Per Reducer, la libreria intende una pura funzione che, preso lo stato attuale dell'applicazione e l'azione da effettuare (Action), restituisce un nuovo stato all'applicazione.
Una volta creati i vari Reducer, possiamo creare uno Store, composto da questi ultimi ed ogni componente, che necessità di accedere allo stato globale, potrà quindi leggere e aggiornare quest'ultimo.

4.1.4 Axios

Axios è una libreria Javascript che permette di connettersi con le API di backend e gestire le richieste effettuate tramite il protocollo HTTP. Il principale vantaggio di queta libreria risiede nell'essere promise-based, permettendo quindi l'implementazione di codice asincrono. Il codice asincrono permetterà, in una pagina, di caricare più elementi contemporaneamente invece che in maniera sequenziale, snellendo sensibilmente i tempi di caricamento.

Il concetto di Promise, su cui si basa Axios, riguarda invece un oggetto Javascript che permette di completare delle richieste in maniera asincrona, facendole passare da tre stati (in sospeso, soddisfatta, rifiutata).

4.1.5 Styled-Components

Styled-Components è una libreria che consente di utilizzare fogli di stili a livello di componente sfruttando una tecnica chiamata CSS-in-JS che deriva dalla combinazione di JavaScript e CSS.
Questa tecnica permette di non mappare i vari componenti su fogli di stile esterni o su codice CSS inline ma bensì il codice CSS effettivo viene scritto all'interno

di una stringa delimitata dai caratteri backtick. I vantaggi di utilizzare questa libreria rispetto a creare diversi fogli di stile sono:

- Eliminazione dei problemi relativi alla duplicazione e sovrapposizione dei nomi delle classi;
- Si rende più intuibile a quale stile un componente sia legato e si riduce la scrittura di classi inutilizzate.

4.1.6 ApexCharts

ApexCharts è una moderna libreria di grafici JavaScript open source che consente di creare visualizzazioni di dati interattive con API semplici.
ApexCharts include oltre una dozzina di tipi di grafici che offrono visualizzazioni reattive.

4.1.7 MapBoxGL

Mapbox è un servizio di mappe avanzato e flessibile che può essere integrato nelle applicazioni web e mobile. Questa libreria Javascript permette a un qualsiasi utilizzatore di creare mappe estremamente personalizzabili, inoltre, permette la visualizzazione di posizioni geografiche tramite l'ausilio di marker, cluster o loghi personalizzati. Per integrare MapBox nel progetto si è utilizzata la sua libreria open source MapBoxGL, progettata appositamente per rendere le mappe visualizzabili React.

4.2 Back-End

Come già citato, il Back-End è stato realizzato con Node e Express per realizzare la business logic lato server e MongoDB per il layer di persistenza.

4.2.1 Persistenza

La persistenza dei dati è stata ottenuta tramite un database non relazionale. Per interfacciarci al database è stata utilizzata la libreria Mongoose che consente di creare modelli che rappresentano uno schema di dati che deve essere mantenuto nel database.

La documentazione completa del Back-End che espone tutte le sue API è disponibile al seguente link:
https://app.swaggerhub.com/apis-docs/Dallas99/TrackER/1.0

Capitolo 5

Codice

5.0.1 Front-End

Per la gestione dello stato globale dell'applicazione si è utilizzato Redux come detto precedentemente. Per funzionare, questo necessita di azioni e di reducer che convertono le informazioni fornite dall'azione e aggiornano lo stato delle Store.

I reducer e le azioni sono stati dichiarati all'interno dell'applicativo come segue:

```
1  import { createSlice } from '@reduxjs/toolkit'
2
3  const initialState = {
4       buildings: JSON.parse(localStorage.getItem("buildings")),
5  }
6
7  export const buildingsSlice = createSlice({
8      name: 'buildings',
9      initialState: initialState,
10     reducers: {
11         fetchBuildings: (state, action) => {
12             state.buildings = action.payload
13             localStorage.setItem("buildings", JSON.stringify(action
       .payload))
14         },
15     },
16 })
17
18 export const { fetchBuildings } = buildingsSlice.actions
19
20 export default buildingsSlice.reducer
```

Lo store infine è definito come composizione dei vari reducers:

```
import { configureStore } from '@reduxjs/toolkit'
import { allOrganizationSlice } from './reducers/allOrganization'
import { allUserSlice } from './reducers/allUsers'
import { buildingsSlice } from './reducers/buildings'
import { organizationSlice } from './reducers/organization'
import { preferenceSlice } from './reducers/preference'
import { userSlice } from './reducers/user'

export default configureStore({
  reducer: {
    user: userSlice.reducer,
    preference: preferenceSlice.reducer,
    buildings: buildingsSlice.reducer,
    organization: organizationSlice.reducer,
    allOrganization: allOrganizationSlice.reducer,
    allUser: allUserSlice.reducer
  },
})
```

Infine per richiamare un'azione sarà necessario usare la funzione useDispatch di Redux come segue:

```
const dispatch= useDispatch()
.
.
.
dispatch(fetchBuildings(data))
```

5.0.2 Back-End

Per la realizzazione del Back-End sono stati definiti una serie di model, controllers e routes.

I models definisco lo schema che dovrà assumere un oggetto e i suoi campi perciò indica come i dati dovranno apparire all'interno delle tabelle del database.

```
const mongoose = require("mongoose")
```

```
2
3  const userSchema = new mongoose.Schema({
4      name: {
5          type: String,
6          required: [true, 'Please add a name'],
7          minlength: 3,
8          maxlength: 30,
9      },
10     surname: {
11         type: String,
12         required: [true, 'Please add a surname'],
13         minlength: 3,
14         maxlength: 30,
15     },
16     email: {
17         type: String,
18         required: [true, 'Please add a email'],
19         maxlength: 255,
20         minlength: 6
21     },
22     password: {
23         type: String,
24         required: [true, 'Please add a password'],
25         maxlength: 1024,
26         minlength: 8
27     },
28     type: {
29         type: String,
30     },
31     date: {
32         type: Date,
33         default: Date.now
34     }
35 })
36
37 module.exports = mongoose.model("User", userSchema);
```

I controllers definisco i metodi da cui è possibile scrivere, leggere ed aggiornare i dati dal database. Inoltre ogni controller genera una risposta positiva o negativa da passare al client per ogni chiamata API.

```
1  const registerUser = asyncHandler(async (req, res) => {
2    const { name, surname, email, password, type } = req.body
3
```

```
4   if (!name || !surname || !email || !password) {
5     res.status(400)
6     throw new Error('Please add all fields')
7   }
8
9   // Check if user exists
10  const userExists = await User.findOne({ email })
11
12  if (userExists) {
13    res.status(400)
14    throw new Error('User already exists')
15  }
16
17  // Hash password
18  const salt = await bcrypt.genSalt(10)
19  const hashedPassword = await bcrypt.hash(password, salt)
20
21  // Create user
22  const user = await User.create({
23    name,
24    surname,
25    email,
26    password: hashedPassword,
27    type
28  })
29
30  if (user) {
31    res.status(201).json({
32      _id: user.id,
33      name: user.name,
34      surname: user.surname,
35      email: user.email,
36      password: hashedPassword,
37      type: type,
38      token: generateToken(user._id),
39    })
40  } else {
41    res.status(400)
42    throw new Error('Invalid user data')
43  }
44 })
```

Le routes indicano la tipologia di operazione API (get, post, put, delete) e quale controller deve essere utilizzato per un certo end points.

```
const express = require('express')
const router = express.Router()
const {
  registerUser,
  loginUser,
  getMe,
  getUserById,
  updateUserById,
  deleteUserById,
  updateUserPasswordById,
  getAll
} = require('../controllers/userController')
const { protect } = require('../middleware/authMiddleware')

router.post('/api/user/register', registerUser)
router.post('/api/user/login', loginUser)
router.get('/api/user/me', protect, getMe)
router.get('/api/user/all', getAll)
router.get('/api/user/:id', getUserById )
router.put('/api/user/:id', updateUserById )
router.put('/api/user/password/:id', updateUserPasswordById )
router.delete('/api/user/:id', deleteUserById )

module.exports = router
```

Infine per garantire il funzionamento del Back-End, è necessario instaurare il collegamento con il Database MongoDB tramite mongoose e infine si inseriscono tutte le route per esporre tutti gli end points.

```
const express = require("express");
const mongoose = require('mongoose')
const app = express();
const cors = require("cors");
require("dotenv").config({ path: "./.env" });
mongoose.connect(process.env.ATLAS_URI, { useNewUrlParser: true },
    () => {
  console.log("CONNECTED")
})
const port = process.env.PORT || 3000;
app.use(cors());
app.use(express.json());
app.use(require("./routes/users"));
app.use(require("./routes/buildings"));
app.use(require("./routes/activity"));
```

```
app.use(require("./routes/userPreference"));
app.use(require("./routes/mailer"));
app.use(require("./routes/organization"));
app.use(require("./routes/renewable"))
app.use(require("./routes/bills.js"))

// get driver connection
const dbo = require("./db/conn");

app.listen(port, () => {
  dbo.connectToServer(function (err) {
    if (err) console.error(err);

  });
  console.log(`Server is running on port: ${port}`);
});
```

5.0.3 Socket.IO

Per l'invio di notifiche in tempo reale si sono utilizzate le Socket.IO, lato server si è creata una socket in ascolto sulla porta 3002 come segue:

```

io.on("connection", (socket) => {
  socket.on("newUser", (id) => {
    addNewUser(id, socket.id);
  });

  socket.on("newBuilding", ({ sender, receiver }) => {
    const rec = getUser(receiver)
    if(!rec) return
    io.to(rec.socketId).emit("getNotification", {
      sender, msg: "has added a new building under your
    organization!", type: "New"
    });
  })

  socket.on("newRenewable", ({ sender, receiver }) => {
    const rec = getUser(receiver)
    if(!rec) return
    io.to(rec.socketId).emit("getNotification", {
      sender, msg: "has installed a new energy resources from your
    Organization!", type: "Renewable"
```

```
20      });
21    })
22
23    socket.on("disconnect", () => {
24      removeUser(socket.id);
25      console.log("User Disconnected", socket.id)
26    });
27  })
28  io.listen(3002)
```

Lato Front-End, ogni volta che si effettua il login, viene creata una nuova socket con id uguale a quello dell'utente registrato.

```
1   useEffect(() => {
2           setSocket(io("http://localhost:3002"))
3       }, []);
4
5       useEffect(() => {
6           if (socket === null) return
7           socket.emit("newUser", user._id);
8       }, [socket, user]);
```

Quando si deve mandare una notifica come la registrazione di un edificio per un'organizzazione, il client notifica la socket del server come segue:

```
1   socket.emit("newBuilding", { sender: user._id, receiver:
        organizationId })
```

Capitolo 6

Test

Le funzionalità del sistema sono state testate dai componenti del team sia su browser Chrome che su browser Safari con l'obiettivo di garantire portabilità inoltre è stato opportunamente testato anche la compatibilità con dispositivi mobile. I test sono serviti per verificare che le funzionalità e i task principali funzionassero allo stesso modo in tutti i browser e dispositivi garantendo che la visualizzazione fosse corretta anche su piattaforme differenti senza errori o malfunzionamenti dell'applicazione (Es. crash della pagina, blocco della pagina...).

Una volta definite le varie API lato Back-End, prima di implementarle nel codice Front-End, si è verificato il loro corretto funzionamento utilizzando l'applicativo Postman.

Il sistema è stato sottoposto sia in fase di sviluppo che durante il suo completamento all'attenzione di entrambe le tipologie di utente coninvolte in modo da valutare la correttezza dell'uso del sistema.
In questa fase tutti i componenti del team si sono immedesimati nei target user individuati in fase di analisi, testando opportunamente il sistema.
Le icone e i bottoni presenti sulla piattaforma cercano di essere il più intuitive ed autoesplicative possibile, si è cercato di mantenere lo stile di ogni pagina uniforme per tutto il sistema.

6.1 User Experience

Per massimizzare la User Experience si è utilizzato ciò che è enunciato dalle euristiche di Nielsen:

- **Controllo e libertà**: il numero di operazioni necessarie per portare l'utente al compimento di un task sono ridotte al minimo indispendabile;
- **Consistenza e standard nel sistema**: la definizione iniziale di una gamma di colori da utilizzare è resa uniforme all'interno di tutto il sistema;
- **Design ed estetica minimalista**: si basa sulle regole KISS che permettono di comporre interfacce chiare e semplici all'uso.
- **Prevenzione dell'errore**: si evitano situazioni ambigue per l'utente durante la navigazione nel sistema, che possano portarlo a commettere errori.
- **Riconoscimento più che ricordo**: i layout sono semplici e schematici. L'utente non ha bisogno di adattarsi per orientarsi;
- **Facilità di riconoscimento, diagnosi e risoluzione dalle situazioni di errore**: i messaggi di errore sono descrittivi definendo il tipo di errore commesso e le possibili soluzioni.

Capitolo 7

Deployment

7.1 Installazione ed Avvio

Per installare e avviare l'applicativo, è necessario aver installato una versione di **nodeJs** maggiore o uguale alla **v14.17.6**.
Una volta verificata la versione di nodeJs con il comando node -v, seguire i seguenti steps:

1. Clonare il seguente repository GitHub:
   https://github.com/DallasCorporation/TrackER.git;

2. Contattare emanuele.dallara@studio.unibo.it per richiedere i file `.env` contenenti le chiavi di cifratura e di accesso al Database da inserire rispettivamente all'interno delle cartelle **/TrackER/tracker/server** e **/TrackER/tracker/tracker**

3. Tramite un terminale, spostarsi all'interno della cartella appena clonata e muoversi dentro la sottocartella tracker con il comando `cd tracker`.

   Per verificare che il percorso sia corretto, eseguire `pwd` e verificare che il percorso termini con /TrackER/tracker.

   Una volta verificato che il percorso sia corretto, eseguire le seguenti operazioni **in ordine** per installare ed avviare i seguenti componenti (si consiglia di utilizzare tre differenti Bash per velocizzare le operazioni di installazione. **Tutti i comandi sono relativi al percorso di partenza**):

   (a) Back-End:

i. `cd server`;
ii. `npm install` per installare tutte le librerie utilizzate;
iii. `npm run dev` per avviare il Back-End.

(b) Front-End:

i. `cd tracker`
ii. `npm install` per installare tutte le librerie utilizzate;
iii. `npm start`. Una volta lanciato il comando, il terminale ci chiederà di utilizzare la porta 3001 poiché la 3000 (porta di default) è occupata dal Back-End. Premere `Y` per avviare il Front-End. Automaticamente si aprirà una scheda di navigazione all'URL localhost:3001.

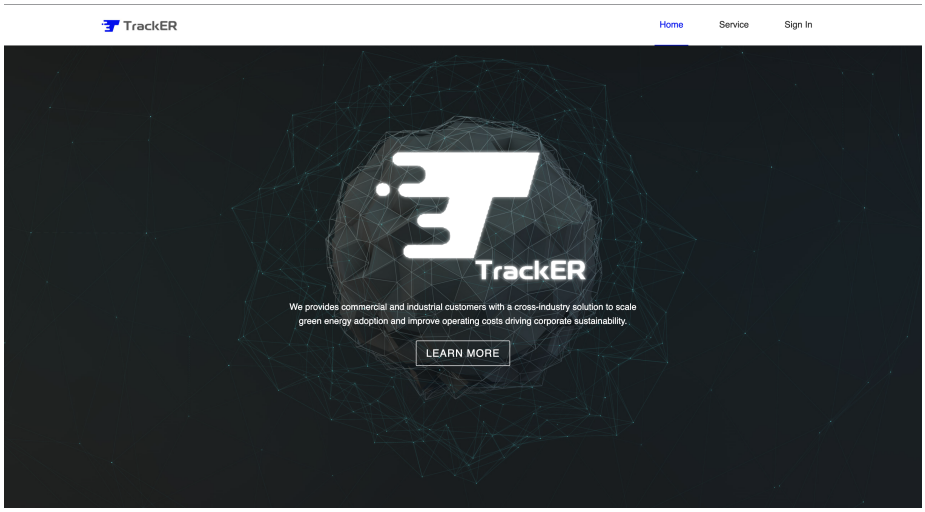

Figura 7.1: TrackER Home.

(c) Generatore di dati energetici (opzionale):

i. `cd generator`
ii. `npm install` per installare tutte le librerie utilizzate;
iii. `npm run myScript` per eseguire lo script.

Capitolo 8

Conclusioni

Il team si ritiene complessivamente soddisfatto del lavoro svolto e del risultato finale ottenuto. Gli obiettivi che si erano prefissati sono stati raggiunti anche se il carico di lavoro è risultato eccessivo poiché il progetto era stato pensato per tre persone ma poi riadattato a due.

Quello che si è raggiunto è un prodotto innovativo con una grafica accattivante ed intuitiva che potrebbe benissimo essere utilizzato da una vera azienda energetica.
Alcuni elementi che sicuramente aumenterebbero il valore di questo prodotto sono:

- Supporto email; rimosso poiché la libreria nodemailer non permette di modificare il sender predefinito. Si era pensato di mandare una mail con mittente info@tracker.com ma la creazione di questa mail richiede costi aggiuntivi;
- Un meccanismo che dia la possibilità di registrare un edificio all'interno di un'organizzazione soltanto se questo è situato entro ad una certa portata (Km) dal fornitore (es. un'organizzazione statunitense non potrà fornire un cittadino italiano per questioni logistiche);
- La possibilità di poter scegliere la fornitura da diversi fornitori senza il vincolo di selezionare quella completa (luce, gas e acqua);

Era stata presa in considerazione anche l'idea di inserire un bot telegram come supporto per le notifiche, scartata poi per la mancanza di tempo.