

Applicazione CRUD in React e PHP

RELAZIONE TECNICA

SAMUELE CAMMARATA

Sommario

Indice delle figure	3
Abstract	4
Analisi descrittiva del progetto.....	5
Front end	11
Back end	18
DbConnect.php.....	18
Index.php.....	19
Database.....	24

Indice delle figure

Figura 1 - Logo React, PHP, MySQL.....	5
Figura 2 - Modem-Router Vodafone Station	5
Figura 3 - Orange Pi 3 LTS.....	5
Figura 4 – Dashboard di Portainer.....	6
Figura 5 - Template server Httpd	6
Figura 6 - Installazione del driver PDO	7
Figura 7 - Configurazione file .htaccess	7
Figura 8 - Configurazione NGINX	8
Figura 9 - Configurazione porta 80 NGINX	9
Figura 10 - Configurazione porta 443 NGINX	9
Figura 11 - Configurazione delle location NGINX	10
Figura 12 - ListUser pt.1.....	11
Figura 13 - ListUser pt.2.....	12
Figura 14 - EditUser pt.1	13
Figura 15 - EditUser pt.2	14
Figura 16 – CreateUser	15
Figura 17 - CustNavbar	16
Figura 18 - App.....	17
Figura 19 - DbConnect	18
Figura 20 - connessione al db	19
Figura 21 - method switch	19
Figura 22 – GET	20
Figura 23 – POST	21
Figura 24 - PUT	22
Figura 25 - DELETE	22
Figura 26 - Index.php - completo	23
Figura 27 - tabella users	24

Abstract

Il mio lavoro è stato quello di progettare e sviluppare una applicazione web in grado di poter effettuare operazioni CRUD che potesse essere eseguita su container Docker all'interno di un server personalizzato.

Nonostante l'applicazione fosse principalmente incentrata sullo sviluppo front-end React, ho voluto integrare una parte di codice back-end PHP ed un database MySQL per poter creare un progetto che toccasse tutti gli aspetti della programmazione web.

Da appassionato di reti e sistemi IT, questa volta ho voluto sperimentare di più con un *SBC* (Single Board Computer) che ho sempre utilizzato come server VPN e DNS casalingo, per poter rendere questa applicazione accessibile da internet.

La fase di progettazione inizia con la scelta e l'attivazione di un dominio DDNS gratuito da associare al mio router di casa, la scelta dei due webserver che ospiteranno, separatamente, gli script PHP e React, la progettazione della tabella presente nel database ed, infine, la progettazione degli schemi di *redirect* di un server NGINX.

Lo sviluppo inizia con la creazione dei componenti grafici in React, per poi passare alla creazione della tabella del database e la programmazione del server PHP. Affinchè i due componenti front e back end riuscissero a comunicare, è stato necessario eseguire del debugging attraverso il quale ho potuto approfondire alcuni aspetti degli header HTTP e del Rewrite Engine del webserver Apache.

Attualmente, l'applicazione è accessibile al seguente indirizzo: <https://thiccnugget.ddns.net>

Analisi descrittiva del progetto

Questo progetto prevede la progettazione e lo sviluppo di una web app composta da:

- Front-end React
- Back-end PHP
- Database MySQL



Figura 1 - Logo React, PHP, MySQL

Per ospitare i vari servizi, sono stati usati i seguenti componenti hardware:

- Modem-Router Vodafone (banda max 100 Mb/s)
- SBC Orange Pi 3 LTS
- 1x Cavo Ethernet RJ45



*Figura 2 - Modem-Router
Vodafone Station*



Figura 3 - Orange Pi 3 LTS

La scelta dell'hardware è stata influenzata dal tipo di software da eseguire su di esso: l'esecuzione di svariati container Docker richiede un modesto quantitativo di risorse; l'Orange Pi 3 LTS è sufficiente per gestire l'esecuzione di tutti i servizi contemporaneamente perché dotato di 2GB di RAM ed un processore quad-core ad 1.8GHz.

La prima fase del progetto consiste nella creazione e deploy dei container Docker per ospitare i vari servizi. Per semplificare questa fase ho deciso di utilizzare [Portainer](#), una ottima interfaccia grafica accessibile via browser che permette di creare e gestire container sul proprio server in pochissimi step.

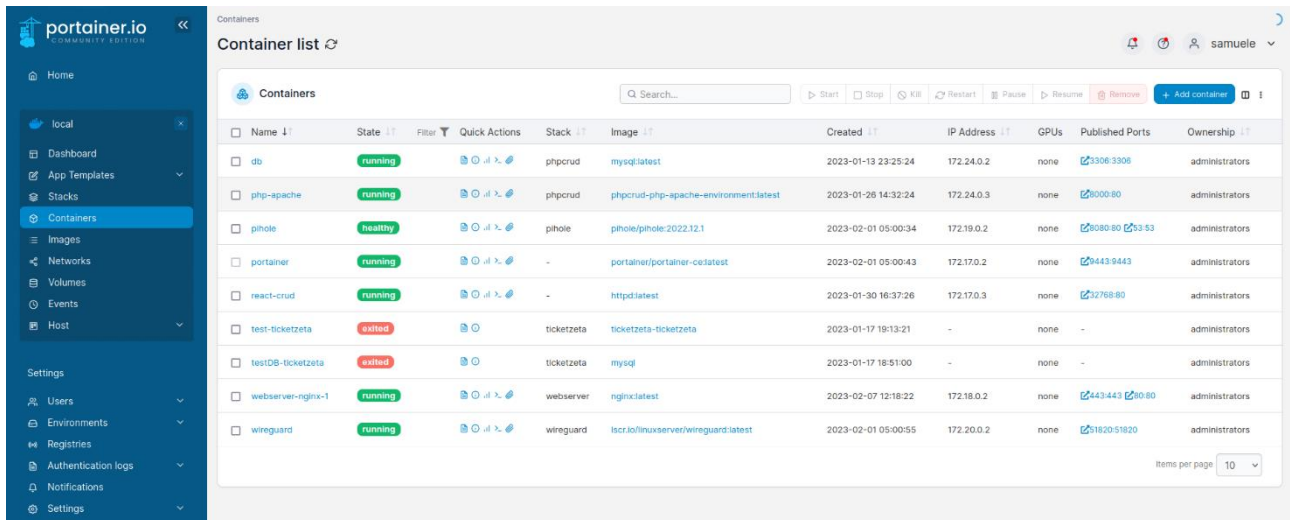


Figura 4 – Dashboard di Portainer

Dalla sezione *App templates*, ho scaricato l'immagine del server Apache ed avviati due container identici.

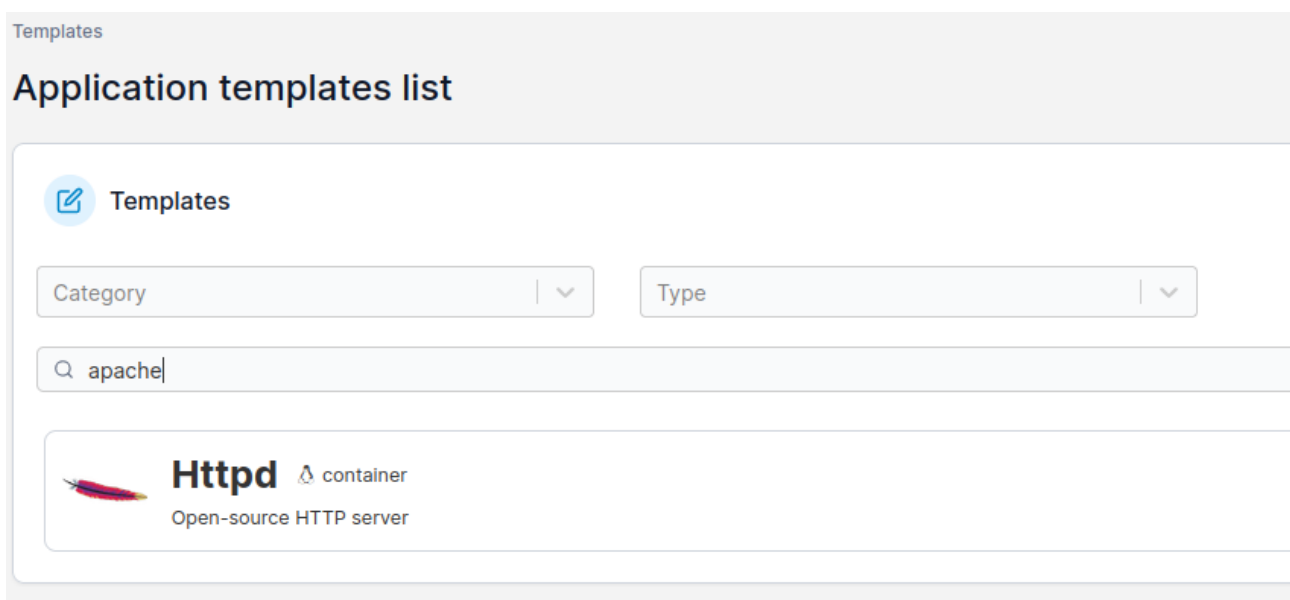


Figura 5 - Template server Httpd

Una volta avviati, ho eseguito l'accesso al terminale del container per PHP ed eseguito i comandi per attivare il rewrite dell'URL (tramite file .htaccess) e per installare il driver PDO per interfacciarsi con il database.

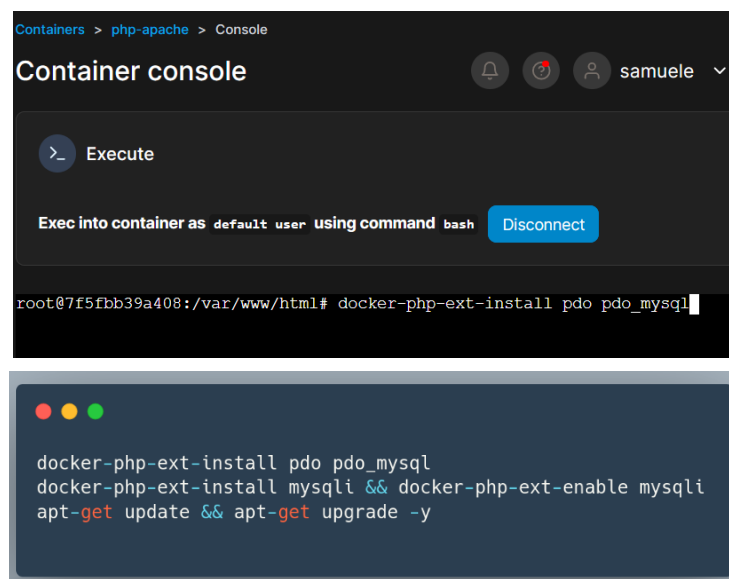


Figura 6 - Installazione del driver PDO



Figura 7 - Configurazione file .htaccess

Il file .htaccess, nella sua configurazione attuale, permette di accedere al file **/api/index.php** senza la necessità di scrivere "index.php" nell'URL del browser o delle richieste effettuate all'interno dell'app React.

Il file **index.php** sarà quindi accessibile al seguente indirizzo: <https://thiccnugget.ddns.net/api> .

L'istruzione **RewriteRule ^(.*)\$ /api/index.php/\$1 [L]** permette, inoltre, di nascondere la dicitura **index.php** anche nel caso in cui ci siano altri parametri nell'URL separati da slash (/).

Esempio:

<https://thiccnugget.ddns.net/api/user/3> è uguale a -> <https://thiccnugget.ddns.net/api/index.php/user/3>

Nel caso in cui risulti impossibile accedere all'API tramite l'URL abbreviato, sarà necessario eseguire nel terminale del webserver il seguente comando, seguito da un riavvio del container: **a2enmod rewrite** .

Da questo momento, il server che ospiterà PHP è pronto ed in ascolto sulla porta 8000. Il server che ospiterà i file React non ha bisogno di alcun'altra configurazione ed è già pronto ed in ascolto sulla porta 32768, scelta in automatico da Docker.

Ultimo step della configurazione di Docker: setup del server NGINX.

NGINX è un webserver in grado di agire da reverse proxy, un sistema che evita la comunicazione diretta tra client e server ponendo tra i due un server intermedio il cui scopo è quello di smistare le varie richieste e di agire, se necessario, sugli header HTTP, avviare e chiudere sessioni HTTPS.

```
server {
    listen 80;
    listen [::]:80;

    server_name thicc nugget.ddns.net;

    location /.well-known/acme-challenge/ {
        root /var/www/certbot;
    }

    location / {
        return 301 https://thicc nugget.ddns.net$request_uri;
    }
}

server {
    listen 443 default_server ssl http2;
    listen [::]:443 ssl http2;

    server_name thicc nugget.ddns.net;

    ssl_certificate /etc/nginx/ssl/live/thicc nugget.ddns.net/fullchain.pem;
    ssl_certificate_key /etc/nginx/ssl/live/thicc nugget.ddns.net/privkey.pem;

    location / {
        proxy_pass http://172.17.0.1:32768;
    }

    location /api {
        add_header 'Access-Control-Allow-Origin' '*';
        add_header 'Access-Control-Allow-Headers' '*';
        add_header 'Access-Control-Allow-Methods' '*';
        proxy_pass http://172.17.0.1:8000;
    }
}
```

Figura 8 - Configurazione NGINX

Questo file di configurazione abilita diverse funzioni:

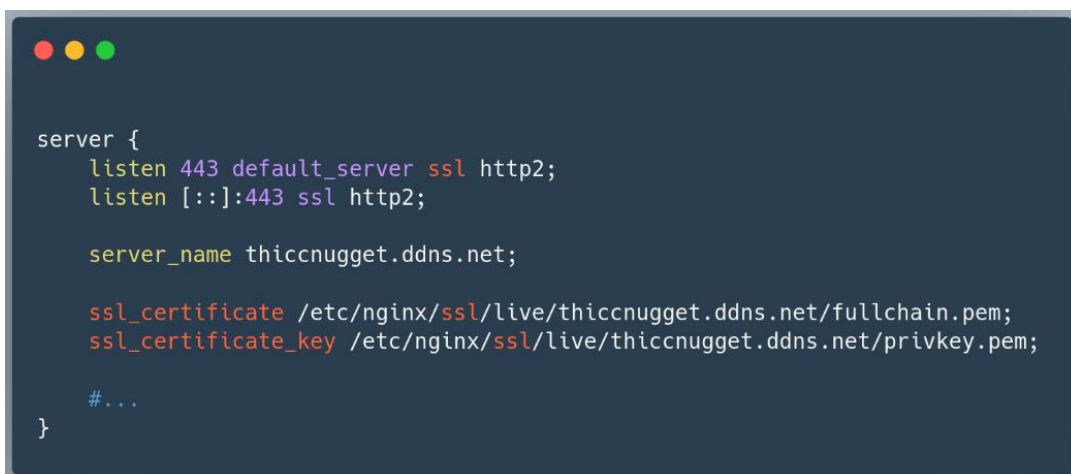
- 1) Redirect automatico alla porta 443, per non stabilire mai una connessione prolungata attraverso il protocollo http

A screenshot of a terminal window with a dark blue background and three colored window control buttons (red, yellow, green) in the top left corner. The terminal displays the following NGINX configuration code:

```
server {  
    listen 80;  
    listen [::]:80;  
  
    #....  
  
    location / {  
        return 301 https://thiccnugget.ddns.net$request_uri;  
    }  
}
```

Figura 9 - Configurazione porta 80 NGINX

- 2) Gestione della connessione sulla porta 443 per utilizzare il protocollo HTTPS. Per questo tipo di configurazione è necessario possedere un certificato SSL, in questo caso generato gratuitamente da **LetsEncrypt** attraverso la loro applicazione **Certbot**.

A screenshot of a terminal window with a dark blue background and three colored window control buttons (red, yellow, green) in the top left corner. The terminal displays the following NGINX configuration code:

```
server {  
    listen 443 default_server ssl http2;  
    listen [::]:443 ssl http2;  
  
    server_name thiccnugget.ddns.net;  
  
    ssl_certificate /etc/nginx/ssl/live/thiccnugget.ddns.net/fullchain.pem;  
    ssl_certificate_key /etc/nginx/ssl/live/thiccnugget.ddns.net/privkey.pem;  
  
    #...  
}
```

Figura 10 - Configurazione porta 443 NGINX

3) Funzione **reverse proxy** per poter comunicare con le due app PHP e React attraverso HTTPS.

A screenshot of a terminal window with a dark blue background and light-colored text. The terminal shows the configuration for two NGINX locations. The first location is a root slash, and the second is /api. Both use proxy_pass to forward requests to different ports on the same host. The /api location also includes three add_header directives for CORS. The terminal window has three colored window control buttons (red, yellow, green) in the top-left corner.

```
location / {  
    proxy_pass http://172.17.0.1:32768;  
}  
  
location /api {  
    add_header 'Access-Control-Allow-Origin' '*';  
    add_header 'Access-Control-Allow-Headers' '*';  
    add_header 'Access-Control-Allow-Methods' '*';  
    proxy_pass http://172.17.0.1:8000;  
}
```

Figura 11 - Configurazione delle location NGINX

La particolarità di questa configurazione sta nel fatto che all'indirizzo <https://thiccugget.ddns.net> sarà mostrata l'app React di default, grazie all'istruzione **proxy_pass** che punta alla porta 32768 dell'host.

Contrariamente, tutte le richieste all'indirizzo <https://thiccugget.ddns.net/api/...> saranno reindirizzate al container PHP grazie alla stessa istruzione **proxy_pass**, che punta però alla porta 8000 dello stesso host.

Front end

Il front end di questa applicazione è sviluppato in React, una libreria opensource di Javascript mantenuta da Meta e dalla community.

Il suo utilizzo non è stato immediato, ma semplice dopo aver capito le sue funzioni base. Di seguito i vari componenti dell'app:

```
import axios from "axios"
import { useEffect, useState } from "react";
import { Link } from "react-router-dom";
import { Container, Button, Table } from "react-bootstrap";

export default function ListUser() {

  //Funzione getUsers() -> richiesta GET al server per prendere i dati degli utenti dal DB
  function getUsers() {
    axios.get('https://thiccugget.ddns.net/api/').then(function(response) {
      setUsers(response.data);
    });
  }

  //Funzione deleteUser() -> richiesta
  const deleteUser = (id) => {
    axios.delete(`https://thiccugget.ddns.net/api/${id}/`).then(function(response){
      getUsers();
    });
  }

  //Funzione per eseguire il set degli utenti
  const [users, setUsers] = useState([]);

  useEffect(() => {
    getUsers();
  }, []);
}
```

Figura 12 - ListUser pt.1

Il componente ListUser è utilizzato per richiedere al backend la lista di tutti gli utenti nel database.

Sono state importate funzionalità di altri componenti, come la grafica di Bootstrap, l'Hook useState, useNavigate per navigare tra i vari componenti senza aggiornare la scheda del browser, la libreria Axios per eseguire le varie richieste http.

Le funzionalità più importanti di questo componente sono:

- la funzione *getUsers* che richiede la lista di tutti gli utenti nel database
- la funzione *deleteUser*, che chiede di eliminare un record dal database

entrambe le funzioni si basano sulla libreria axios, che permette di cambiare il tipo di richiesta da inviare al server back end. La *response.data* presente all'interno delle funzioni axios si riferisce ai dati restituiti dal server backend, in questo caso dati JSON.

```
return (
  <div className="mt-5 text-center">
    <h1 className="mb-3">Lista utenti</h1>
    <Container className="mainContainer">
      <Table className="p-4" bordered hover>
        <thead>
          <tr>
            <th>Nome</th>
            <th>Email</th>
            <th>Telefono</th>
            <th>Azioni</th>
          </tr>
        </thead>
        <tbody>
          {
            users.map(
              (user, key) =>
                <tr key={key}>
                  <td>{user.name}</td>
                  <td>{user.email}</td>
                  <td>{user.mobile}</td>
                  <td>
                    <Link to={`user/${user.id}/edit`} ><Button variant="primary"
                      className="mx-3">Modifica</Button></Link>

                    <Button variant="danger"
                      onClick={() => deleteUser(user.id)}>Elimina</Button>
                  </td>
                </tr>
              )
            )
          }
        </tbody>
      </Table>
    </Container>
  </div>
)
```

Figura 13 - ListUser pt.2

Il componente restituisce una tabella con i dati degli utenti restituiti dal back end sotto forma di dati JSON.

Attraverso la funzione *users.map* si mappano gli attributi di ogni utente e li si stampano nella corrispondente colonna della tabella. Nell'ultima colonna sono presenti due pulsanti: il primo serve a modificare l'utente e reindirizza al prossimo componente (EditUser); il secondo serve ad eliminare l'utente attraverso la funzione *deleteUser*.

```

import { useState, useEffect } from "react";
import axios from "axios";
import { useNavigate, useParams } from "react-router-dom";
import { Container, Form, Button } from "react-bootstrap";

export default function EditUser() {
  const navigate = useNavigate();
  const [inputs, setInputs] = useState([]);
  const {id} = useParams();

  useEffect(() => {
    getUser();
  }, []);

  function getUser() {
    axios.get(`https://thiccugget.ddns.net/api/${id}/`).then(function(response) {
      setInputs(response.data);
    });
  }

  const handleChange = (event) => {
    const name = event.target.name;
    const value = event.target.value;
    setInputs(values => ({...values, [name]: value}));
  }

  const handleSubmit = (event) => {
    //Blocca il submit in caso di errori
    event.preventDefault();
    //evita il submit se i campi sono vuoti
    if (inputs.name.replaceAll(' ','') !== "" && inputs.email.replaceAll(' ','') !== ""
        && inputs.mobile.replaceAll(' ','') !== "") {
      //Richiesta PUT -> modifica di un record nel DB
      axios.put(`https://thiccugget.ddns.net/api/${id}/`, inputs).then(function(response){
        if(response.data === 1){
          alert("Modifica effettuata con successo!");
        }
        //Reindirizza alla home page
        navigate('/');
      });
    }
  }
}

```

Figura 14 - EditUser pt.1

Il componente EditUser mette a disposizione una tabella nella quale inserire i dati con cui modificare un utente. Prima di tutto, il componente esegue la funzione *getUser()* per prelevare i dati dell'utente e inserirli automaticamente nella tabella grazie alla funzione *setInputs*.

Ad ogni cambiamento degli input, il loro valore viene salvato in una coppia di variabili *name*, *value* attraverso la funzione *handleChange*. I campi di input non possono essere vuoti per avviare il submit.

La funzione *handleSubmit* esegue la richiesta PUT al server backend per modificare i dati relativi all'utente con id selezionato. Se la richiesta va a buon fine e restituisce il valore 1, appare una alert con il messaggio "Modifica effettuata con successo!", successivamente si sarà reindirizzati alla pagina di index (ListUser).

```

return (
  <Container className="container">
    <h1 className="m-5 text-center">Nuovo utente</h1>
    <Form onSubmit={handleSubmit}>
      <Form.Group className="mb-4">
        <Form.Label>Nome</Form.Label>
        <Form.Control name="name" type="text" placeholder="Nome" value={inputs.name || ""}
          onChange={handleChange} />
      </Form.Group>

      <Form.Group className="mb-4">
        <Form.Label>Email</Form.Label>
        <Form.Control name="email" type="email" placeholder="Email" value={inputs.email || ""}
          onChange={handleChange} />
      </Form.Group>

      <Form.Group className="mb-5">
        <Form.Label>Telefono</Form.Label>
        <Form.Control name="mobile" type="number" placeholder="Telefono" value={inputs.mobile || ""}
          onChange={handleChange} />
      </Form.Group>
      <Button variant="primary" type="submit" className="submit">
        Crea
      </Button>
    </Form>
  </Container>
)
}

```

Figura 15 - EditUser pt.2

Il form è composto da 3 campi: Nome, Email, Telefono.

Per semplificare la modifica, i campi sono automaticamente riempiti con i dati dell'utente in questione.

Il submit è eseguibile cliccando sul tasto "Crea" o premendo Invio da tastiera.

Il form è uguale per entrambi i componenti EditUser e CreateUser.

```

import { useState } from "react";
import axios from "axios";
import { useNavigate } from "react-router-dom";
import Button from 'react-bootstrap/Button';
import Container from 'react-bootstrap/Container'
import Form from 'react-bootstrap/Form'

export default function CreateUser() {
  const navigate = useNavigate();

  const [inputs, setInputs] = useState([]);

  const handleChange = (event) => {
    const name = event.target.name;
    const value = event.target.value;
    setInputs(values => ({...values, [name]: value}));
  }

  const handleSubmit = (event) => {
    //Blocca il submit in caso di errori
    event.preventDefault();
    if (inputs.name.replaceAll(' ','') !== "" && inputs.email.replaceAll(' ','') !== ""
        && inputs.mobile.replaceAll(' ','') !== "") {
      //Richiesta HTTP POST -> inserimento dati in input nel DB
      axios.post('https://thiccugget.ddns.net/api/', inputs).then(function(response){
        if(response.data === 1){
          alert("Inserimento avvenuto con successo!");
        }
        //Reindirizza alla home page
        navigate('/');
      });
    }
  }
}

```

Figura 16 – CreateUser

Il componente CreateUser permette la creazione di nuovi utenti attraverso la compilazione di un form.

È sufficiente riempire il form e cliccare il pulsante “Crea” o il tasto Invio della tastiera.

Il submit controlla se i campi di input non sono vuoti, dopodichè invia, tramite richiesta POST al server back end, il loro contenuto per essere salvato nel database. Se il submit va a buon fine e restituisce il valore 1, appare una alert con il messaggio “Inserimento avvenuto con successo!” e reindirizzerà automaticamente alla pagina index.


```

import Container from 'react-bootstrap/Container';
import Nav from 'react-bootstrap/Nav';
import Navbar from 'react-bootstrap/Navbar';
import { Link } from "react-router-dom";

function CustNavbar() {
  return (
    <Navbar bg="light" expand="lg">
      <Container>
        <Navbar.Brand href="/">App CRUD</Navbar.Brand>
        <Navbar.Toggle aria-controls="nav" />
        <Navbar.Collapse id="nav">
          <Nav className="me-auto">
            <Link to="/" className='navLink'>Lista utenti</Link>
            <Link to="/user/create" className='navLink'>Nuovo utente</Link>
          </Nav>
        </Navbar.Collapse>
      </Container>
    </Navbar>
  );
}

export default CustNavbar;

```

Figura 17 - CustNavbar

Questa è una semplice navbar di Bootstrap. Contiene due link per:

- tornare alla home page -> Lista utenti
- creare un nuovo utente -> Nuovo utente

```

import {BrowserRouter, Routes, Route} from 'react-router-dom';
import CustNavbar from './components/CustNavbar';
import CreateUser from './components/CreateUser';
import EditUser from './components/EditUser';
import ListUser from './components/ListUser';
import './App.css';

function App() {
  return (
    <div className="App">
      <BrowserRouter>
        <CustNavbar />
        <Routes>
          <Route index element={<ListUser />} />
          <Route path="user/create" element={<CreateUser />} />
          <Route path="user/:id/edit" element={<EditUser />} />
        </Routes>
      </BrowserRouter>
    </div>
  );
}

export default App

```

Figura 18 - App

Infine, il componente App (App.js) che costituisce la vera app da renderizzare, si occupa dei reindirizzamenti tra i vari componenti. Il componente da mostrare come pagina index / homepage è ListUser.

Il primo componente da renderizzare è la navbar, sempre posizionata in alto, identificata dal componente **<CustNavbar />**

Grazie al componente **<BrowserRouter>** è possibile navigare molto rapidamente tra i vari componenti ed è anche possibile assegnare un path diverso ad ognuno di essi.

Al path **user/create** è possibile accedere al form di creazione, mentre al path **user/idUtente/edit** sarà possibile accedere al form di modifica. La rimozione dell'utente non è presente in questo componente perché è già implementata come una funzione del componente ListUser.

Back end

Il codice gestito dal server Back end è semplice; svolge due funzioni vitali:

- gestione della connessione al database.
- gestione delle richieste da parte del Front end.

La connessione al database è gestita dal driver PDO (PHP Data Objects), installato come estensione di PHP non appena creato il proprio container. PDO è un driver, a mio parere, più versatile del competitor Mysqli (connector appositamente sviluppato per database MySQL) perché rende più veloce la migrazione ad un nuovo tipo di database e consente anche l'inserimento di variabili nelle queries senza che queste siano trattate necessariamente come stringhe.

DbConnect.php

Di seguito il codice che inizializza la connessione con il database:



```
<?php
class DbConnect {
    private $server = '172.24.0.2';
    private $dbname = 'MYSQL_DATABASE';
    private $user = 'MYSQL_USER';
    private $pass = 'MYSQL_PASSWORD';

    public function connect() {
        try {
            $conn = new PDO('mysql:host=' . $this->server . ';dbname=' . $this->dbname, $this->user, $this->pass);
            $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
            return $conn;
        } catch (\Exception $e) {
            echo "Database Error: " . $e->getMessage();
        }
    }
}
```

Figura 19 - DbConnect


La connessione al database è inizializzata dalla classe DbConnect, nella quale vanno inseriti i dati di accesso:

- \$server -> indirizzo IP del server database
- \$dbname -> nome del database da usare
- \$user -> utente da usare nell'interazione con il database
- \$pass -> password dell'utente \$user

La funzione connect() restituisce (se creato con successo), un oggetto \$conn che consente di interagire con il database descritto dai dati di accesso. In caso di fallimento, il server lancerà un messaggio di errore e fermerà l'esecuzione del codice.

Index.php

Di seguito i vari pezzi del file index.php:




```
<?php
require_once 'DbConnect.php';
$objDb = new DbConnect;
$conn = $objDb->connect();
date_default_timezone_set('Europe/Rome');
#...
```

Figura 20 - connessione al db

Il file index.php non è utilizzabile senza la creazione di una connessione tra PHP ed il database, è quindi necessario includere il file “DbConnect.php” attraverso l’istruzione *require_once*, che blocca l’esecuzione del codice nel caso in cui l’inclusione del file fallisca.

È quindi necessario creare un oggetto \$objDb, istanza della classe DbConnect, e passare i dati della connessione tra il driver PDO ad un altro oggetto \$conn .

L’istruzione *date_default_timezone* cambia l’orario locale all’orario della timezone “Europe/Rome”, per caricare date nel database secondo l’orario italiano.



```
$method = $_SERVER['REQUEST_METHOD'];
switch($method) {
#...
```

Figura 21 - method switch

Alla variabile \$method va assegnato il valore della cella “REQUEST_METHOD” dell’array \$_SERVER, che contiene informazioni sul server come header, path ed impostazioni del webserver.

I metodi di richiesta accettati sono:

- GET -> usato per leggere/prelevare dati.
- POST -> per creare una nuova risorsa, in questo caso un record nel database.
- PUT -> per modificare una risorsa.
- DELETE -> per eliminare una risorsa.

Una volta ricevuta una richiesta dal front end e stabilita la sua modalità, in una struttura switch-case sono definite tutte le operazioni da eseguire in base al tipo di richiesta. Di seguito i vari casi:

```
case "GET":
    $sql = "SELECT * FROM users";
    $path = explode('/', $_SERVER['REQUEST_URI']);
    if(isset($path[2]) && is_numeric($path[2])) {
        $sql .= " WHERE id = :id";
        $stmt = $conn->prepare($sql);
        $stmt->bindParam(':id', $path[2]);
        $stmt->execute();
        $users = $stmt->fetch(PDO::FETCH_ASSOC);
    } else {
        $stmt = $conn->prepare($sql);
        $stmt->execute();
        $users = $stmt->fetchAll(PDO::FETCH_ASSOC);
    }

    echo json_encode($users);
    break;
```

Figura 22 – GET

Il metodo GET serve a restituire al front end tutti gli utenti presenti nel database sotto forma di dati JSON con l'ultima istruzione "echo json_encode(\$users)". Viene eseguita la query "SELECT * FROM users" per prelevare tutti i record della tabella e salvarli nell'array associativo \$users attraverso la funzione "\$users = \$stmt->fetch(PDO::FETCH_ASSOC)".

Nel caso in cui nell'URL sia specificato l'id di un utente in particolare, saranno restituiti soltanto i dati ad esso relativi, aggiungendo al testo della query la clausola *WHERE* seguita dall'id da ricercare. Questa funzione è resa possibile dal controllo effettuato sull'URL con l'istruzione *explode('/', ...)*; separa tutti i dati dopo ogni slash e li salva in un array. L'id dell'utente va inserito in posizione 2 dell'array \$path, così composto:

[0] -> thiccnugget.ddns.net

[1] -> api

[2] -> idUtente

```
case "POST":
    $user = json_decode( file_get_contents('php://input') );
    $sql = "INSERT INTO users(id, name, email, mobile, created_at)
           VALUES(null, :name, :email, :mobile, :created_at)";
    $stmt = $conn->prepare($sql);
    $created_at = date('Y-m-d H-i-s');
    $stmt->bindParam(':name', $user->name);
    $stmt->bindParam(':email', $user->email);
    $stmt->bindParam(':mobile', $user->mobile);
    $stmt->bindParam(':created_at', $created_at);

    if($stmt->execute()) {
        $response = 1;
    } else {
        $response = 0;
    }
    echo $response;
    break;

#...
```

Figura 23 – POST

Il metodo POST indica che si vuole creare una nuova risorsa sul server, in questo caso un nuovo utente.

I dati del nuovo utente sono passati come file JSON nel corpo della richiesta; vanno quindi decodificati e passati come parametri della query di creazione. In base al risultato della query, il server restituisce una risposta con cui gestire una *alert* nel front end:

- 1 -> query eseguita con successo
- 0 -> query non eseguita a causa di un errore

```

case "PUT":
    $user = json_decode( file_get_contents('php://input') );
    $sql = " UPDATE users SET name= :name, email = :email, mobile = :mobile, updated_at = :updated_at
            WHERE id = :id ";
    $stmt = $conn->prepare($sql);
    $updated_at = date('Y-m-d H-i-s');
    $stmt->bindParam(':id', $user->id);
    $stmt->bindParam(':name', $user->name);
    $stmt->bindParam(':email', $user->email);
    $stmt->bindParam(':mobile', $user->mobile);
    $stmt->bindParam(':updated_at', $updated_at);

    if($stmt->execute()) {
        $response = 1;
    } else {
        $response = 0;
    }
    echo $response;
    break;
#...

```

Figura 24 - PUT

Il metodo PUT indica che si vuole modificare una risorsa. Come il metodo POST, prende in input i dati da inserire e li usa per aggiornare il record con l'id dell'utente specificato. Inoltre, aggiorna la data di modifica ogni volta che questa va a buon fine.

```

case "DELETE":
    $sql = "DELETE FROM users WHERE id = :id";
    $path = explode('/', $_SERVER['REQUEST_URI']);

    $stmt = $conn->prepare($sql);
    $stmt->bindParam(':id', $path[2]);

    if($stmt->execute()) {
        $response = 1;
    } else {
        $response = 0;
    }
    echo $response;
    break;
}

```

Figura 25 - DELETE

L'ultimo caso, il metodo DELETE, definisce quale record eliminare dal database prendendo in input l'id dell'utente da rimuovere. Anche questa funzione restituisce una risposta 1 o 0 per lanciare delle alert nel front end.

```

<?php
require_once 'DbConnect.php';
$objDb = new DbConnect;
$conn = $objDb->connect();
date_default_timezone_set('Europe/Rome');

$method = $_SERVER['REQUEST_METHOD'];
switch($method) {
    case "GET":
        $sql = "SELECT * FROM users";
        $path = explode('/', $_SERVER['REQUEST_URI']);
        if(isset($path[3]) && is_numeric($path[3])) {
            $sql .= " WHERE id = :id";
            $stmt = $conn->prepare($sql);
            $stmt->bindParam(':id', $path[3]);
            $stmt->execute();
            $users = $stmt->fetch(PDO::FETCH_ASSOC);
        } else {
            $stmt = $conn->prepare($sql);
            $stmt->execute();
            $users = $stmt->fetchAll(PDO::FETCH_ASSOC);
        }

        echo json_encode($users);
        break;

    case "POST":
        $user = json_decode( file_get_contents('php://input') );
        $sql = "INSERT INTO users(id, name, email, mobile, created_at)
                VALUES(null, :name, :email, :mobile, :created_at)";
        $stmt = $conn->prepare($sql);
        $created_at = date('Y-m-d H-i-s');
        $stmt->bindParam(':name', $user->name);
        $stmt->bindParam(':email', $user->email);
        $stmt->bindParam(':mobile', $user->mobile);
        $stmt->bindParam(':created_at', $created_at);

        if($stmt->execute()) {
            $response = 1;
        } else {
            $response = 0;
        }
        echo $response;
        break;

    case "PUT":
        $user = json_decode( file_get_contents('php://input') );
        $sql = " UPDATE users SET name= :name, email = :email, mobile = :mobile, updated_at = :updated_at
                WHERE id = :id ";
        $stmt = $conn->prepare($sql);
        $updated_at = date('Y-m-d H-i-s');
        $stmt->bindParam(':id', $user->id);
        $stmt->bindParam(':name', $user->name);
        $stmt->bindParam(':email', $user->email);
        $stmt->bindParam(':mobile', $user->mobile);
        $stmt->bindParam(':updated_at', $updated_at);

        if($stmt->execute()) {
            $response = 1;
        } else {
            $response = 0;
        }
        echo $response;
        break;

    case "DELETE":
        $sql = "DELETE FROM users WHERE id = :id";
        $path = explode('/', $_SERVER['REQUEST_URI']);

        $stmt = $conn->prepare($sql);
        $stmt->bindParam(':id', $path[3]);

        if($stmt->execute()) {
            $response = 1;
        } else {
            $response = 0;
        }
        echo $response;
        break;
}
?>

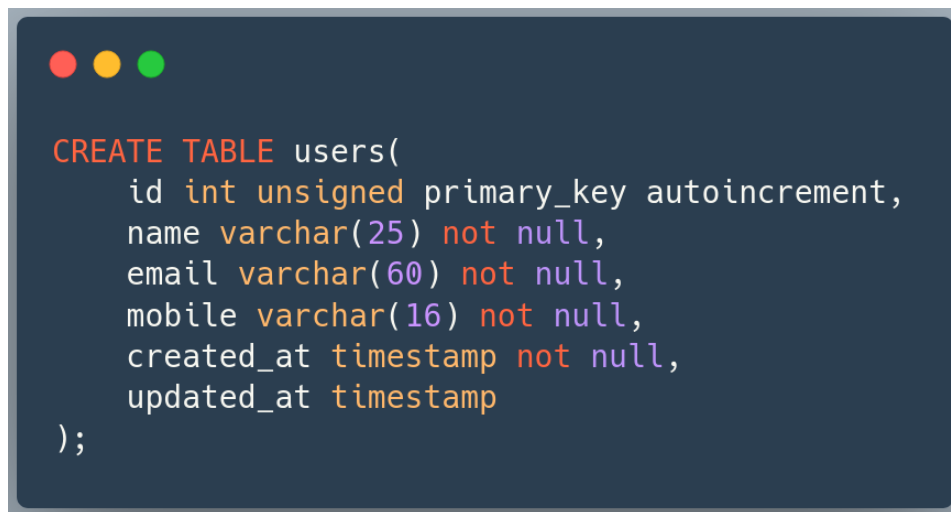
```

Figura 26 - Index.php - completo

Database

La struttura ed il funzionamento del database è molto semplice, in quanto composto da una sola tabella che contiene i dati di ogni utente creato/modificato dall'app. Il database in questione è chiamato "MYSQL_DATABASE" ed è in esecuzione su un container Docker MySQL dedicato.

Di seguito è mostrata la query di creazione:



```
CREATE TABLE users(  
  id int unsigned primary_key autoincrement,  
  name varchar(25) not null,  
  email varchar(60) not null,  
  mobile varchar(16) not null,  
  created_at timestamp not null,  
  updated_at timestamp  
);
```

Figura 27 - tabella users

Descrizione dei campi:

- Id -> identifica univocamente ogni utente; incrementa automaticamente ad ogni inserimento.
- name -> nome dell'utente, lungo fino a 25 caratteri, obbligatorio.
- email -> email dell'utente, lungo fino a 60 caratteri, obbligatorio.
- mobile -> numero di telefono, fino a 16 caratteri per includere il prefisso, obbligatorio.
- created_at -> data e ora di creazione dell'utente, mai NULL.
- updated_at -> data e ora di modifica, aggiornato ad ogni modifica, può essere NULL.

Sarebbe corretto creare un indice sul campo email e renderlo univoco per evitare conflitti tra email uguali, ma non ho ancora implementato tale funzione al fine di produrre un primo codice più semplice e funzionale.