

# UniPG Online

---



- Realizzazione di un servizio REST per la gestione di un registro universitario online
- Accessibilità al web service tramite solamente un browser e una connessione ad internet
- I professori possono inserire voti nel database
- Gli studenti possono scegliere se accettare un voto o rifiutarlo

# Servizio REST



Il web service è stato costruito utilizzando l'ambiente server open source Node.js basato su javascript

REpresentational State Transfer (REST) è uno stile architetturale che rappresenta una trasmissione di risorse sul protocollo HTTP utilizzando file JSON e una serie di metodi (GET, PUT, POST, DELETE)

PostgreSQL



Express è il framework utilizzato al lato server per gestire queste richieste, quando interrogato l'URI corrispondente esegue la query nel database e rinvia un file JSON contenente la risposta

Il database utilizzato è PostgreSQL, un DBMS a oggetti open source relazionale moderno e semplice da usare

# Client



Per la gestione del sito web a lato client è stata utilizzata la libreria Javascript ReactJS, specializzata nella creazione di interfacce utente visualizzabili in qualsiasi browser

Con ReactJS possiamo leggere file JSON e ottenerne i dati senza l'utilizzo di librerie o di formattazioni complicate del file

Le comunicazioni con il web service sono gestite tramite la libreria per Node.js Axios, che permette di effettuare richieste HTTP



# Sicurezza



Le password usate per il login sono criptate con l'algoritmo non reversibile HS256, grazie alla libreria bcrypt

Alla registrazione la password viene criptata e il risultato è ciò che viene salvato nel database, al momento del login anche la password inserita viene criptata e solo poi confrontata con quella salvata, in modo da non avere salvataggi di informazioni in chiaro

L'accesso agli URI viene protetto con dei token utilizzati con la libreria JWT: al login viene consegnato un token criptato all'utente che viene richiesto ogni volta che quest'ultimo prova ad accedere ad una risorsa.

Avendo due tipi di utenti diversi ci sono due funzioni per proteggere le risorse in due modi diversi

```
const passport = require('passport')
require('../services/passport')

module.exports.requireProfessoreAuth = passport.authenticate('jwt-professore', { session: false })
module.exports.requireStudenteAuth = passport.authenticate('jwt-studente', { session: false })
```

# Connessioni



Il server si connette al database tramite una singola configurazione

Si mette poi in ascolto sulla porta scelta

Il client può effettuare richieste semplicemente accedendo a  
`http://indirizzo_server:porta_server/uri`

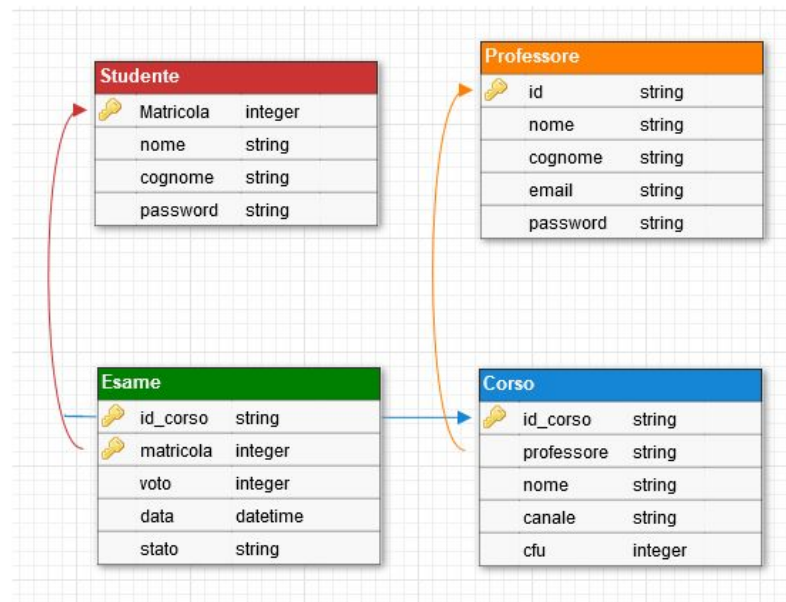
```
5  const { Pool } = require('pg')
6
7  const pool = new Pool({
8    user: 'postgres',
9    host: 'localhost',
10   database: 'Unipg',
11   password: 'postgres'
12 })
13
```

```
const app = express()
const port = 3001

app.listen(port, () => {
  console.log(`App running on port ${port}.`)
})
```

# Struttura database

La struttura del Database è molto semplice, troviamo una tabella per gli studenti e una per i Professori, una tabella Corso che contiene tutti i corsi tenuti da ciascun professore e un'ultima tabella Esame, contenente per ogni esame effettuato da ciascuno studente anche se il voto è stato accettato o è ancora in attesa



# Post



Le richieste Post vengono utilizzate per l'aggiunta di una risorsa, in questo caso andiamo ad aggiungere un nuovo studente al database come conseguenza della registrazione

Con una sola riga di codice possiamo accedere alla risorsa `/studenti/nuovo` con il metodo `post` e passargli il file JSON contenente i dati necessari presi dalla form

```
// inserisco il nuovo studente nel database
postStudente() {
  const { matricola } = this.state
  const { nome } = this.state
  const { cognome } = this.state
  const { email } = this.state
  const { password } = this.state

  Axios.post('http://localhost:3001/studenti/nuovo', { matricola, nome, cognome, email, password })
    .then(this.props.onLoginExecuted(this.state, 'studente'))
}
```

# Post



La funzione che troviamo nel server è più semplice di quello che sembra, riceve la richiesta, cripta (hash and salt) la password.

Inserisce poi i dati nel database e se tutto è andato a buon fine risponde con lo stato 201 = created

```
app.post('/studenti/nuovo', studente.postNuovoStudente)
```

```
//usato per la registrazione di un nuovo studente
const postNuovoStudente = (request, response) => {
  const { matricola, nome, cognome, email, password } = request.body
  bcrypt.genSalt(10, (error, salt) => {
    if (error) {
      return response.sendStatus(500)
    }
    bcrypt.hash(password, salt, null, (err, hash) => {
      console.log(hash.length)
      if (err) {
        return response.sendStatus(500)
      }
      pool.query(
        'INSERT INTO studente VALUES($1,$2,$3,$4,$5)',
        [ matricola, nome, cognome, email, hash ],
        (error, results) => {
          if (error) {
            throw error
          }
          response.status(201).send(`Studente added: ${matricola}`)
        }
      )
    }
  })
}
```



# Get



La funzione Get serve a leggere una risorsa presente nel server, in questo caso viene usata per ricevere la lista degli esami del professore che ha effettuato il login per poter inserire un voto su tale esame.

L'URI della risorsa è `.../corsi/:id_professore` e come vediamo il metodo invia anche il token ricevuto al login

Se non dovesse corrispondere al token inviato dal server riceverebbe un errore 401 = Unauthorized

```
// cerco la lista degli esami e degli studenti nel database per mostrarli nei menù  
getCorsi() {  
  const token = this.props.token  
  const { id } = this.props.utente  
  Axios.get('http://localhost:3001/corsi/' + id, {  
    headers: { Authorization: 'Bearer ' + token }  
  })  
    .then((res) => this.setState({ data: res.data }))  
    .catch((err) => console.log(err))  
}
```

# Get

```
app.get('/corsi/:professore', requireProfessoreAuth, corso.getCorsoFromProf)
```

```
//utilizzato per ricevere i corsi di un certo prof  
const getCorsoFromProf = (request, response) => {  
  const professore = request.params.professore  
  
  pool.query('SELECT id_corso, nome FROM corso WHERE professore = $1', [ professore ], (error, results) => {  
    if (error) {  
      throw error  
    }  
    response.status(200).json(results.rows)  
  })  
}
```

La funzione esegue una semplice query nel database e restituisce il risultato in formato JSON

La funzione Get in questo caso prende anche la funzione di riconoscimento del token come parametro, così che ne basti una per tipo di utente e non una per metodo REST

# Put



Il metodo Put viene utilizzato per modificare una risorsa nel server, nel nostro caso per modificare il campo 'statoesame' nella tabella esame per cambiarlo da Waiting ad Accettato, come conseguenza del click del bottone 'accetta' accanto all'esame

```
// modifico l'esame nel database mettendo lo stato ad 'Accettato'  
accettaEsame(corso) {  
  const { token } = this.props  
  const data = {  
    matricola: this.props.utente.matricola,  
    corso: corso  
  }  
  const config = {  
    headers: { Authorization: 'bearer ' + token }  
  }  
  Axios.put('http://localhost:3001/studenti/esami/', data, config)  
    .then((res) => this.getEsami())  
    .catch((err) => console.log(err))  
}
```

# Put



Anche qui la funzione lato server è semplice, viene effettuato l'update solo se il token ricevuto è giusto e, se non ci sono errori viene inviata una risposta con codice 200 = OK al client

```
app.put('/studenti/esami/', requireStudenteAuth, esame.accettaEsame)
```

```
// usato per accettare un esame
const accettaEsame = (request, response) => {
  const { matricola, corso } = request.body
  pool.query(
    "UPDATE esame SET statoesame = 'Accettato' WHERE matricola = $1 and id_corso = $2",
    [ matricola, corso ],
    (error) => {
      if (error) {
        throw error
      }
      response.status(200).send(`User modified: ${matricola}`)
    }
  )
}
```

# Delete



Il metodo Delete, come si può immaginare, si usa per eliminare una risorsa. In questo caso viene utilizzato per rifiutare un esame da parte di uno studente e quindi eliminarlo dal database

L'URI utilizzato è .../studenti/esami/:corso/:matricola e anche in questo caso viene mandato il token come corpo della richiesta

```
// rifiuto l'esame e quindi elimino la riga dal database
rifiutaEsame(corso) {
  const token = this.props.token
  const { matricola } = this.props.utente
  Axios.delete('http://localhost:3001/studenti/esami/' + corso + '/' + matricola, {
    headers: { Authorization: 'Bearer ' + token }
  })
  .then((res) => this.getEsami())
  .catch((err) => console.log(err))
}
```

# Delete



```
app.delete('/studenti/esami/:corso/:matricola', requireStudenteAuth, esame.rifiutaEsame)
```

```
//usato per rifiutare e quindi eliminare un esame
const rifiutaEsame = (request, response) => {
  const { matricola, corso } = request.params
  pool.query('DELETE FROM esame WHERE matricola = $1 AND id_corso = $2', [ matricola, corso ], (error, results) => {
    if (error) {
      throw error
    }
    response.status(200).send(`Esame deleted: ${matricola}, ${corso}`)
  })
}
```

# Login



Dal click del tasto login viene effettuata una richiesta Post per controllare l'esistenza dell'email inserita e se le password coincidono, in base all'utente si passa poi alla pagina specifica


# Studente



Con una richiesta get vengono visualizzati tutti gli esami registrati con la matricola dello studente. I tasti Accetta e Rifiuta servono a chiamare i metodi Put e Delete sulla stessa tabella



# Professore



Al caricamento della pagina due richieste Get richiedono i corsi dell'insegnante che ha effettuato il login e la lista degli studenti iscritti. Inserito il voto, se non esiste un duplicato viene effettuato il Put

# Registrazione



Una volta inseriti i valori nei campi obbligatori al click del tasto Registrati verrà effettuata una richiesta Post che inserirà i valori corrispondenti al nuovo utente nella tabella dei professori o degli studenti .