

UniSignMiUp

A.A 2024/2025

Corso di Architetture software orientate ai servizi

By

Paolo Faltaous ~ 01828A

Giulia Floris ~ 984706

UniSignMiUp

Autore: Paolo Faltaous &

Giulia Floris



Indice:

1.0 - Introduzione:

1.1 Descrizione

1.2 Destinatari

1.3 Modello di Lavoro

1.4 Flussi di Dati

2.0 - Interfacce:

3.0 - Codice:

3.1 GET

3.2 POST

3.3 PUT

3.4 DELETE

3.5 OPTION

3.6 ROUTES, SERVER E DATABASE

3.7 CHIAMATA A SERVIZIO

3.8 DATABASE

4.0 - Conclusioni:

4.1 Considerazioni Finali

UniSignMiUp

1. INTRODUZIONE

1.1 Descrizione

UniSignMiUp è una piattaforma web progettata per semplificare il processo di iscrizione agli esami presso l'Università degli Studi di Milano (Unimi). Gli utenti possono accedere utilizzando il loro account Unimi e prenotare le strutture desiderate tramite un comodo modulo online.

Le principali sezioni dell'applicazione includono:

- Il login
- La homepage
- La pagina Dashboard
- La pagina delle Iscrizione agli esami.
- La pagina del supporto
- Pulsante Logout

Con un'interfaccia intuitiva e accessibile, l'applicazione offre a studenti e docenti un modo semplice per navigare e utilizzare le sue funzionalità.

1.2 Destinatari

UniSignMiUp è un'applicazione sviluppata per gli studenti dell'Università Statale di Milano, con l'obiettivo di semplificare la prenotazione degli appelli d'esame e agevolare la gestione del percorso accademico.

Dotata di un'interfaccia chiara e intuitiva, è progettata per essere accessibile anche a chi non possiede competenze tecniche avanzate. La piattaforma è ottimizzata per l'uso su computer, sia desktop che portatili, garantendo un'esperienza fluida e intuitiva.

Oltre a facilitare l'iscrizione agli esami, UniSignMiUp offre uno strumento organizzativo efficace, permettendo agli studenti di pianificare al meglio i propri appelli e ridurre il rischio di errori o sovrapposizioni. Il suo design e le sue funzionalità mirano a rendere il processo di prenotazione rapido, strutturato ed efficiente, rispondendo alle esigenze concrete della comunità universitaria.

1.3 Modello di lavoro

Il cuore di UniSignMiUp è la gestione efficiente delle iscrizioni agli esami per gli studenti dell'Università Statale di Milano, offrendo una piattaforma dedicata e intuitiva.

A differenza di altre soluzioni, UniSignMiUp fornisce un ambiente esclusivo e ottimizzato, eliminando la necessità di ricorrere a servizi esterni. Tutto ciò di cui hai bisogno è accessibile con un semplice clic, garantendo un'esperienza fluida e senza distrazioni.

La piattaforma consente di iscriversi agli esami in modo rapido e diretto, riducendo il tempo e lo sforzo necessari per gestire le prenotazioni. Non è necessario consultare altre fonti: UniSignMiUp offre un sistema centralizzato per organizzare al meglio il proprio percorso accademico.

Oltre a semplificare la vita universitaria, la piattaforma favorisce un senso di appartenenza, creando un ambiente digitale pensato per supportare gli studenti della Statale di Milano nella loro crescita accademica.

1.4 Flusso di dati

I contenuti di UniSignMiUp seguono uno schema uniforme per ogni dipartimento, assicurando un'esperienza coerente e strutturata in base all'affiliazione dell'utente.

Tutte le informazioni visualizzate sono generate direttamente dai dati ufficiali forniti dall'Università Statale di Milano, eliminando i costi di produzione e garantendo l'affidabilità delle comunicazioni.

Per una gestione efficiente e sicura, i dati vengono archiviati in un database **MySQL**, assicurando accessibilità immediata e una struttura ottimizzata per il recupero delle informazioni necessarie.

UniSignMiUp

2. INTERFACCE

All'accesso al sito, gli utenti vengono indirizzati alla pagina **Login.html**, dove possono autenticarsi con le proprie credenziali o registrarsi per creare un nuovo account. Una volta completata la procedura di login o registrazione, il sistema li reindirizza automaticamente alla **homepage** di UniSignMiUp (**homepage.html**).

La pagina di login è progettata con un'interfaccia moderna e intuitiva, offrendo un'esperienza all'utente fluida e accessibile. Un pratico **switch** consente di passare facilmente tra le sezioni di accesso e registrazione, semplificando la gestione dell'account e garantendo un'interazione immediata e senza complicazioni.

Nella pagina di login, qualora l'utente si fosse dimenticato la password, ha la possibilità di resettarla cliccando **"Password Dimenticata"**

Un elemento chiave per la sicurezza dell'accesso è l'implementazione del **password toggle**, che consente agli utenti di mostrare o nascondere la password durante l'inserimento, riducendo il rischio di errori e migliorando l'esperienza d'uso senza compromettere la protezione dei dati.

Dopo l'autenticazione, gli utenti vengono reindirizzati alla **homepage** (**homepage.html**), dove trovano un'interfaccia coerente con il design della pagina di login, grazie al mantenimento degli stessi colori e sfondi. Il layout è studiato per offrire un'esperienza visiva armoniosa e intuitiva.

Il fulcro della homepage è la possibilità di **isciversi agli esami**, obiettivo principale dell'applicazione, reso immediato grazie al pulsante **"Iscriviti"**, posizionato in modo strategico per facilitare l'azione da parte degli utenti.

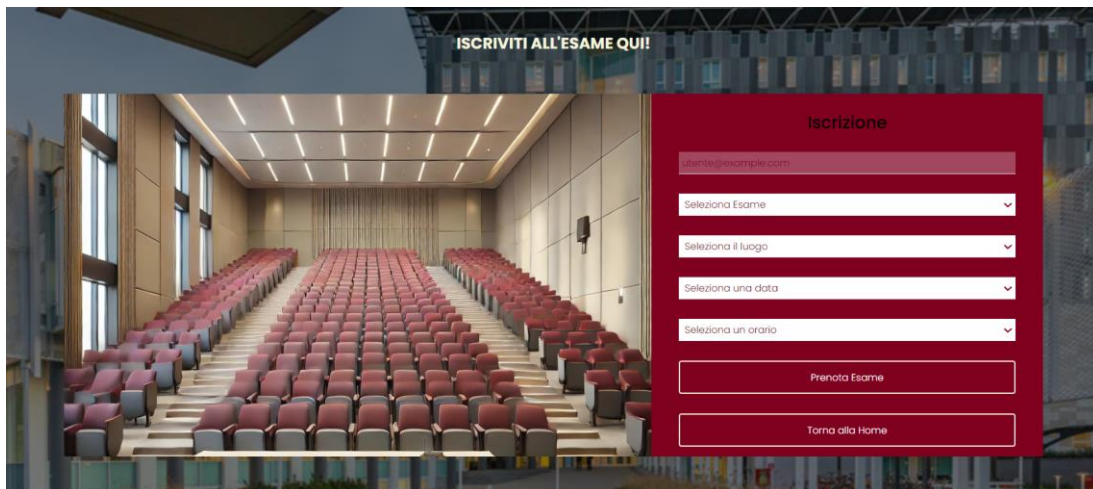
Infine, la homepage include le sezioni descritte nella **sezione**

1.1 Descrizione situate nell'angolo superiore destro della schermata. Questi elementi garantiscono un accesso rapido e strutturato alle principali funzionalità, consentendo una navigazione efficiente e intuitiva all'interno della piattaforma.



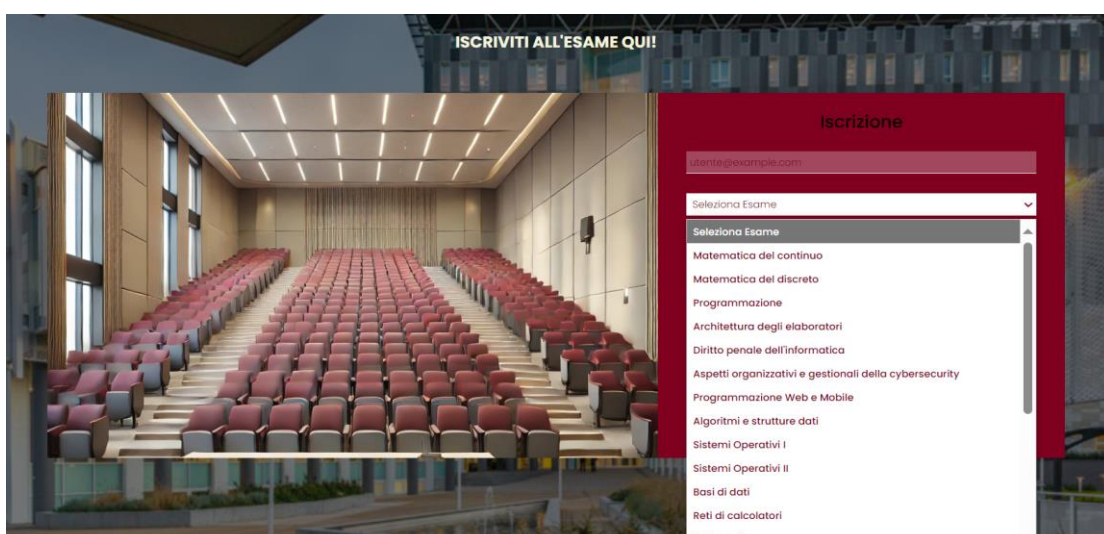
Dopo aver selezionato il pulsante **"Iscriviti"**, disponibile sia al centro della homepage che nell'angolo superiore destro, l'utente viene immediatamente reindirizzato alla pagina dedicata alla **prenotazione degli esami**.

All'interno di questa sezione, è possibile **selezionare la data, l'aula e l'edificio** in base alle disponibilità, garantendo un'esperienza di prenotazione chiara e intuitiva. L'interfaccia è progettata per offrire un accesso rapido e strutturato, semplificando la gestione delle iscrizioni agli esami e ottimizzando l'organizzazione delle sessioni di prova.

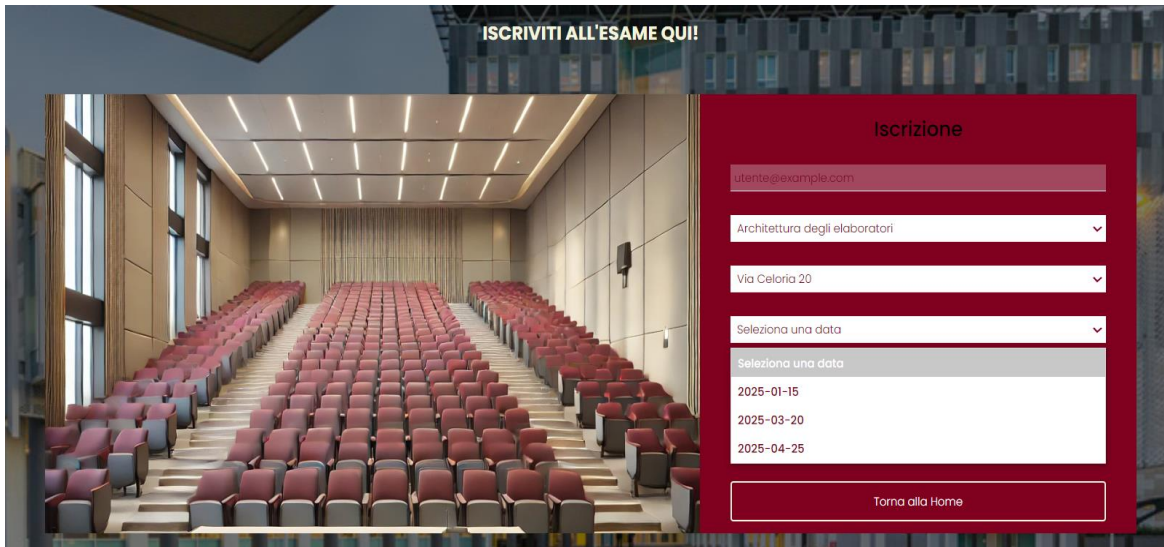


La coerenza nel design è evidente nell'uso uniforme di colori e sfondi, che si ripropongono sia nella pagina di login che nella homepage, garantendo un'esperienza visiva omogenea. In aggiunta, lo sfondo della pagina di prenotazione degli esami presenta un'immagine dell'edificio universitario situato in **Via Celoria 18**, conferendo un elemento distintivo e facilmente riconoscibile.

E' possibile quindi selezionare gli esami:



È possibile visualizzare i giorni disponibili in cui potersi iscrivere e i relativi orari per ciascun data:



ISCRIVITI ALL'ESAME QUI!

Iscrizione

utente@example.com

Architettura degli elaboratori

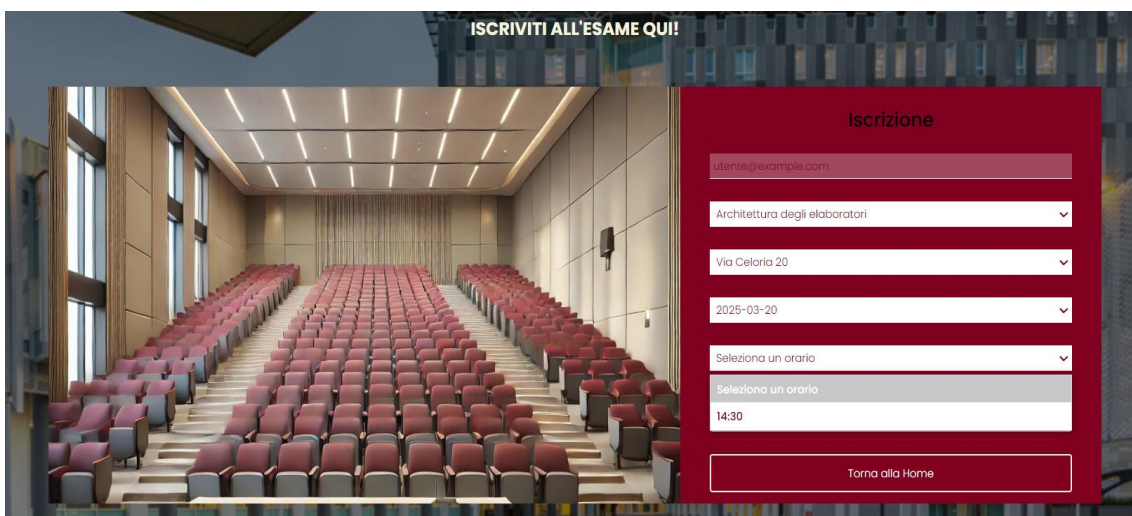
Via Celoria 20

Seleziona una data

Seleziona una data

- 2025-01-15
- 2025-03-20
- 2025-04-25

Torna alla Home



ISCRIVITI ALL'ESAME QUI!

Iscrizione

utente@example.com

Architettura degli elaboratori

Via Celoria 20

2025-03-20

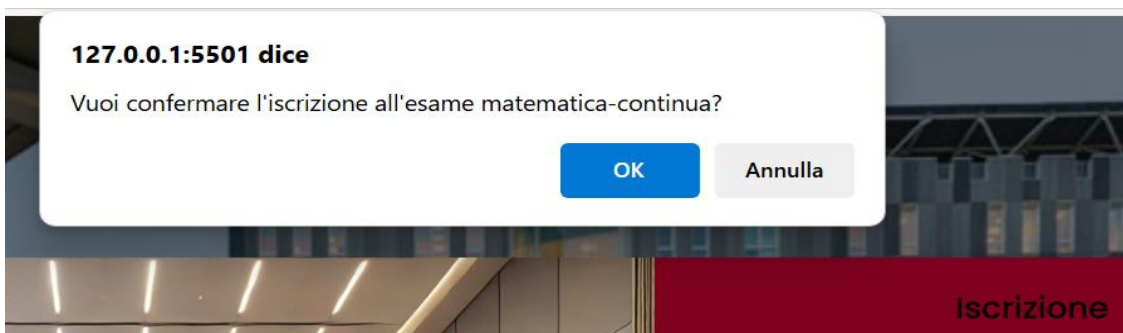
Seleziona un orario

Seleziona un orario

14:30

Torna alla Home

Dopo aver premuto il pulsante "**Prenota Esame**", comparirà un messaggio di conferma di iscrizione all'esame scelto poco prima:



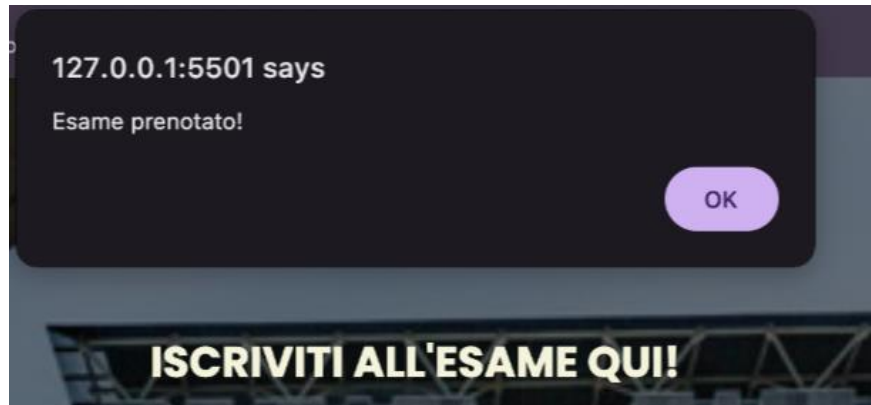
127.0.0.1:5501 dice

Vuoi confermare l'iscrizione all'esame matematica-continua?

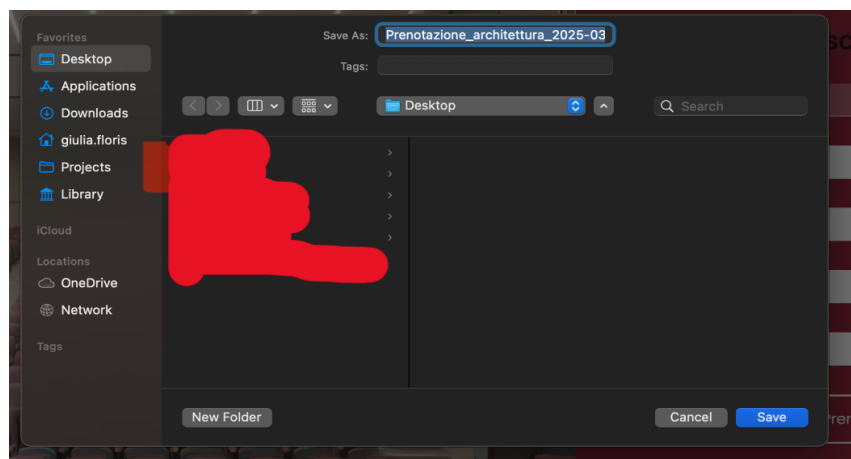
OK Annulla

Iscrizione

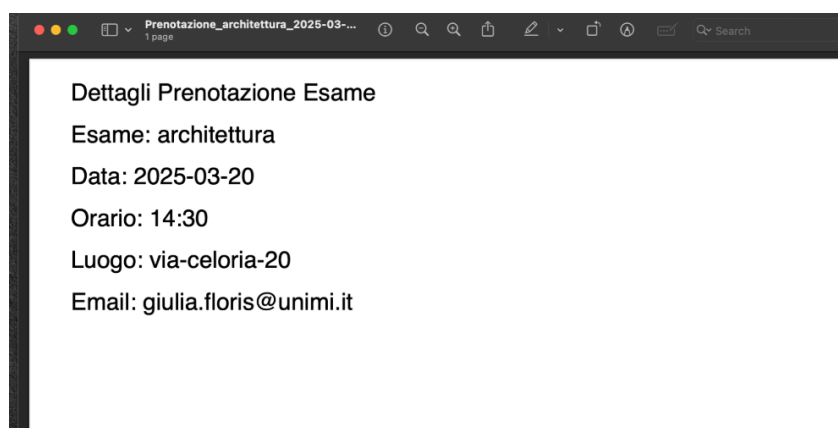
Una volta selezionato il pulsante "OK", viene visualizzato un **messaggio di conferma** che attesta l'avvenuta iscrizione all'esame scelto, fornendo all'utente una conferma chiara e immediata dell'operazione completata con successo.



Successivamente abbiamo implementato tramite JS la possibilità di avere il file pdf con il riepilogo dell'esame a cui ci si è iscritti:



Una volta scelto il luogo dove salvare il file PDF, lo si recupera e all'interno è strutturato nel seguente modo:



Tutti i dettagli della prenotazione verranno salvati nella tabella "**Exam**" del database dedicato di UniSignMiUp. Tuttavia, i dettagli su come questo processo sia implementato sono trattati nel capitolo [3.0 Codice](#)

Accedendo alla sezione "**Dashboard**", gli utenti possono visualizzare un **riassunto completo** delle proprie informazioni e delle **prenotazioni effettuate**, offrendo così una panoramica chiara e organizzata delle attività in corso.

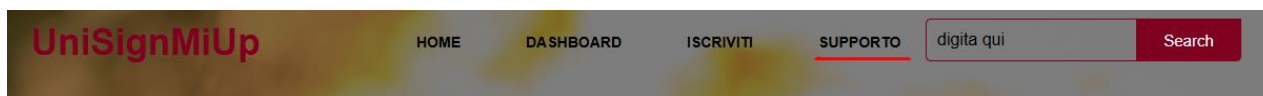


Il sito web garantisce una **coerenza visiva** in tutte le pagine, utilizzando una palette di colori uniforme, ad eccezione della barra laterale. Questa barra laterale raccoglie le diverse sezioni della homepage, semplificando la navigazione e migliorando l'efficienza nell'accesso alle varie pagine dell'applicazione.

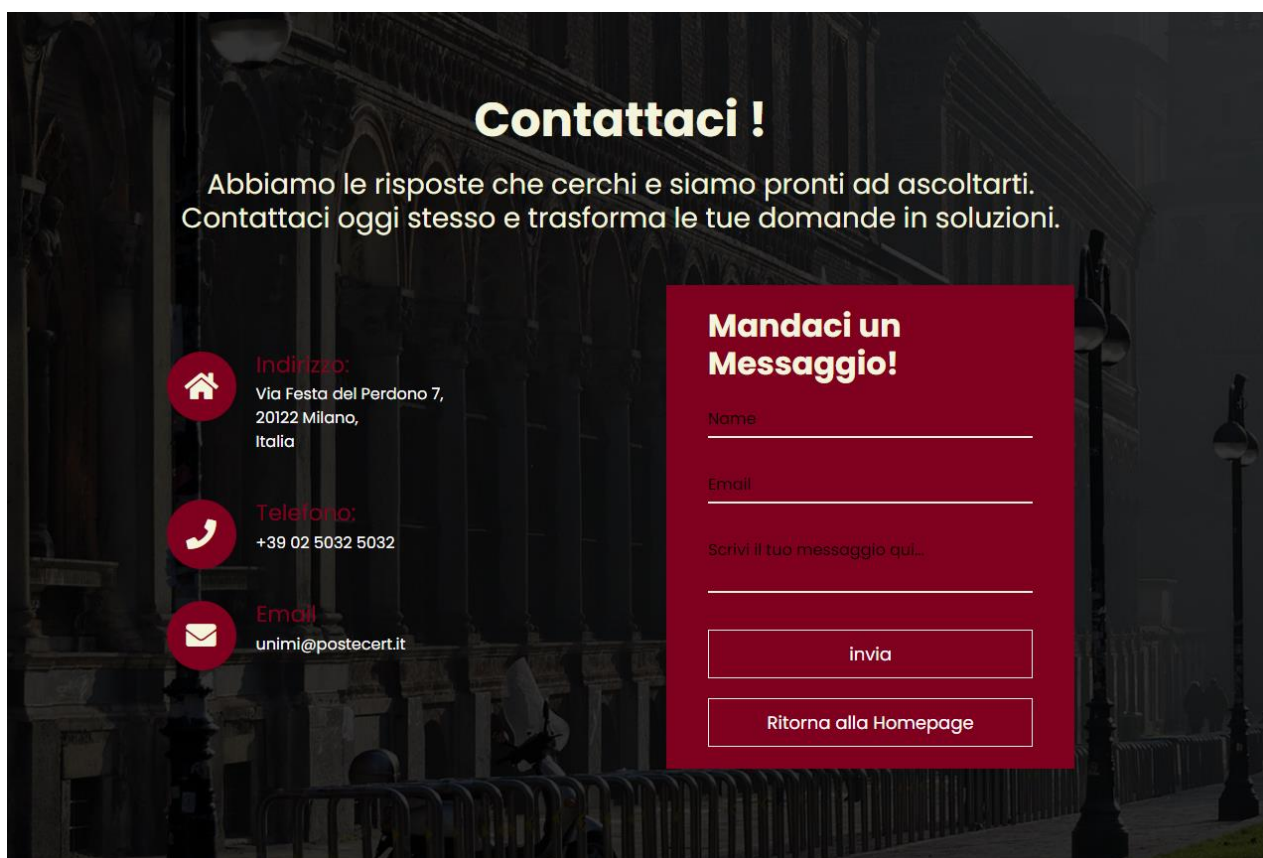
Inoltre, gli utenti hanno la possibilità di **annullare una prenotazione** tramite un apposito tasto "**Cancella Prenotazione**", offrendo così una gestione semplice e diretta delle proprie iscrizioni.

Da questa schermata di Dashboard è possibile quindi tornare alla Homepage, iscriversi ad ulteriori esami ed infine effettuare direttamente il Logout che indirizza alla schermata iniziale di Login.

Tornando alla **Homepage**, gli utenti possono accedere alla sezione **"Supporto"**, dove sono disponibili i dettagli di contatto dell'Università Statale di Milano.



In questa area, è possibile inviare un messaggio compilando un modulo apposito, attraverso il quale gli utenti possono segnalare eventuali problematiche o ricevere risposte ai loro quesiti, garantendo un supporto completo e tempestivo.



Anche in questa sezione, si può notare uno sfondo distintivo che rappresenta la sede principale dell'Università Statale di Milano, creando un legame visivo diretto con l'istituzione e arricchendo l'esperienza dell'utente con un elemento iconico e riconoscibile.

UniSignMiUp

3. CODICE

3.1 GET

In questo contesto, analizziamo l'uso del verbo "GET", prendendo come esempio la funzione **getExams**, che rappresenta la chiamata utilizzata per recuperare l'elenco degli esami associati a un'utenza.

Model:

```
exports.getExams = async (examData) => {

  console.log("[Console] getExams started ...");

  const studentId = examData.studentId;
  const sql = 'SELECT * FROM exam WHERE studentId = ?';

  const [rows] = await db.query(sql, [studentId]);

  if (!rows || rows.affectedRows === 0) {
    return { success: false };
  } else {

    this.examStruct = rows;
    return { success: true };
  }
};
```

Analizziamo il model, il quale interagisce direttamente con il database. Il valore **studentId** viene ricavato dall'oggetto **examData** passato in input e utilizzato per costruire una query SQL parametrizzata. Questa query recupera tutti gli esami associati a quello specifico **studentId** dal database. L'operazione viene eseguita in modo asincrono tramite **db.query**, e il risultato viene memorizzato nella variabile **rows**.

Se la query non restituisce dati validi o non ha prodotto risultati, il metodo ritorna un oggetto con **success: false**.

In caso contrario, i dati degli esami vengono assegnati a **examStruct**, e il metodo segnala il successo dell'operazione con **success: true**.

Controller:

```
exports.getExams = async (req, res) => {
  try {
    console.log("[Console] Received data:", req.query); // Log to check request body
    const result = await examModel.getExams(req.query);
    console.log("[Console] result", result);

    if (result.success) {
      // Send success response with a message
      console.log("⚡ [Console] getExams successfull");
      const examData = examModel.getExamsData();
      console.log("NOME", examData);

      return res.status(200).json({ examData: examData });
    } else {
      console.log("💀 [Console] getExams failed");
      return res.status(400).json({ message: 'ERROR' });
    }
  } catch (err) {
    console.log("💀 [Console] getExams failed");
    console.log(err.message);
    res.status(500).json({ error: err.message });
  }
};
```

Il **controller** funge da intermediario tra il client e il database. La funzione **getExams** richiama il metodo corrispondente del **model** per recuperare le informazioni sugli esami in base ai parametri ricevuti nella richiesta (**req.query**).

Il risultato viene poi valutato:

se l'operazione ha avuto successo, viene restituito un **200 OK** con i dati degli esami.

Se la richiesta non è andata a buon fine, viene inviato un **400 Bad Request** con un messaggio di errore generico.

In caso di eccezioni o problemi imprevisti, il flusso entra nel **catch**, restituendo un **500 Internal Server Error**, registrando i dettagli dell'errore nei log per il debug.

3.2 POST

Esaminiamo il verbo "POST", prendendo come esempio la funzione **createExam**, che esegue una chiamata per salvare un nuovo record nella tabella "exam" del database.

Model:

```
// POST a new exam
exports.createExam = async (examData) => {

  const { idexam, studentId, examName, location, dateTime, course } = examData;

  const sql = 'INSERT INTO exam (ideam, studentId, examName, location, dateTime, course) VALUES (?, ?, ?, ?, ?, ?)';

  console.log("[Console] createExam started ...")
  const [result] = await db.query(sql, [ideam, studentId, examName, location, dateTime, course]);

  if (!result || result.affectedRows === 0) {
    return { success: false };
  } else {
    return { success: true };
  }
};
```

Il modello analizzato gestisce l'interazione con il database per inserire un nuovo esame nella tabella **exam**. I valori necessari, come **ideam**, **studentId**, **examName**, **location**, **dateTime** e **course**, vengono estratti dall'oggetto **examData** ricevuto in input e utilizzati per costruire la query SQL parametrizzata.

Dopo l'esecuzione dell'operazione, il risultato viene verificato: se l'inserimento è andato a buon fine e ha modificato almeno una riga, viene restituito **success: true**. In caso contrario, o se si verifica un errore nell'operazione, il metodo ritorna **success: false**, segnalando il fallimento dell'inserimento.

Controller:

```
// POST exam
exports.createExam = async (req, res) => {
  try {
    console.log("[Console] Received data:", req.body); // Log to check request body
    const result = await examModel.createExam(req.body);

    if (result.success) {
      // Send success response with a message
      console.log("[Console] createExam successfull");
      return res.status(200).json({ message: 'createExam successful' });
    } else {
      // Send failure response if credentials are invalid
      console.log("[Console] createExam failed");
      return res.status(400).json({ message: 'ERROR' });
    }
  } catch (err) {
    console.log("[Console] createExam failed");
    console.log(err.message);
    res.status(500).json({ error: err.message });
  }
};
```

Analizziamo il **controller** gestendo le richieste HTTP e orchestrando le operazioni sui dati. Nel caso della funzione **createExam**, il controller riceve una richiesta **POST**, estrae i dati dall'oggetto **req.body** e li passa al metodo **createExam** del modello, che si occupa dell'inserimento nel database.

Il risultato dell'operazione viene poi analizzato per restituire una risposta adeguata al client:

- Se l'inserimento è andato a buon fine, il server risponde con **200 OK** e un messaggio di conferma.
- Se l'operazione fallisce per dati non validi, viene restituito un **400 Bad Request**.
- In caso di errore inaspettato durante l'elaborazione, l'eccezione viene catturata nel **catch**, e il server risponde con un **500 Internal Server Error**, registrando il dettaglio dell'errore nei log per facilitare il debugging.

3.3 PUT

Esaminiamo il verbo "PUT", prendendo come esempio la funzione **changePassword**, che esegue una chiamata per aggiornare il valore del campo "password" nel record dell'utente specifico all'interno del database.

Model:

```
exports.changePassword = async (userData) => {

  console.log("[Console] changePassword started ...");
  const email = userData.email
  const newPassword = userData.password;

  const sql = `UPDATE user
    SET password = ?
    WHERE email = ?`;

  const [result] = await db.query(sql, [newPassword, email]);

  console.log("[Console] changePassword result ...", result);

  if (!result || result.affectedRows === 0) {
    return { success: false };
  } else {
    return { success: true };
  }
};
```

In questo caso, la funzione **changePassword** riceve un oggetto **userData**, da cui estrae **email** e **newPassword** per costruire una query SQL parametrizzata.

La query **UPDATE** viene quindi eseguita per modificare la password dell'utente corrispondente all'email fornita. Dopo l'esecuzione, il sistema verifica il risultato:

- Se almeno una riga è stata modificata con successo, viene restituito **success: true**.
- Se nessuna riga è stata aggiornata (ad esempio, perché l'email non esiste nel database), il metodo restituisce **success: false**, indicando che l'operazione non è andata a buon fine.

I log di sistema registrano l'inizio e l'esito del processo per facilitare il debugging e il monitoraggio.

Controller:

```
// PUT new password
exports.changePassword = async (req, res) => {
  try {
    console.log("[Console] Received data:", req.body); // Log to check request body
    const result = await userModel.changePassword(req.body);

    if (result.success) {
      // Send success response with a message
      console.log("[Console] changePassword successful");
      return res.status(200).json({ message: 'changePassword successful' });
    } else {
      // Send failure response if credentials are invalid
      console.log("[Console] changePassword failed");
      return res.status(400).json({ message: 'ERROR' });
    }
  } catch (err) {
    console.log("[Console] changePassword failed");
    console.log(err.message);
    res.status(500).json({ error: err.message });
  }
};
```

Il **controller** ha il compito di gestire la richiesta di aggiornamento della password e di restituire una risposta adeguata al client in base all'esito dell'operazione.

La funzione **changePassword** riceve una richiesta **PUT**, estrae i dati dal **req.body** e li passa al metodo omonimo del **model**, che si occupa dell'aggiornamento nel database.

Dopo l'esecuzione, il controller analizza il risultato:

- Se l'operazione è andata a buon fine, viene restituito un **200 OK** con un messaggio di conferma.
- Se l'aggiornamento non è stato effettuato (ad esempio, per dati non validi o email inesistente), il server risponde con **400 Bad Request**.
- In caso di errore imprevisto durante l'interazione con il database, l'eccezione viene catturata nel **catch**, restituendo un **500 Internal Server Error** e registrando l'errore nei log per facilitare il debugging.

3.4 DELETE

Esaminiamo il verbo "DELETE", prendendo come esempio la funzione **deleteExam**, che esegue una chiamata per rimuovere un record specifico dalla tabella "exam" nel database.

Model:

```
// DELETE exam
exports.deleteExam = async (examData) => {

  console.log("[Console] deleteExam started ...")

  const idexam = examData.idexam;

  const sql = `
DELETE FROM exam
WHERE idexam = ?
`;

  const [result] = await db.query(sql, [idexam]);

  console.log("[Console] deleteExam result ...", result)

  if (!result || result.affectedRows === 0) {
    return { success: false };
  } else {
    return { success: true };
  }
};
```

Analizziamo il **model**, In questo contesto, l'eliminazione del record avviene identificando l'esame corrispondente tramite l'ID estratto dall'input **examData**. Al termine dell'operazione, il sistema restituisce un esito coerente con l'esecuzione della query:

- **success: true**, se la cancellazione è avvenuta con successo.
- **success: false**, in caso di errore durante l'elaborazione della richiesta.

Controller:

```
// DELETE exam
exports.deleteExam = async (req, res) => {
  try {
    console.log("[Console] Received data:", req.query); // Log to check request body
    const result = await examModel.deleteExam(req.query);

    if (result.success) {
      // Send success response with a message
      console.log("++ [Console] deleteExam successfull");
      return res.status(200).json({ message: "OK" });
    } else {
      // Send failure response if credentials are invalid
      console.log("💩 [Console] deleteExam failed");
      return res.status(404).json({ errorType: "003" });
    }
  } catch (err) {
    console.log("💩 [Console] deleteExam failed");
    console.log(err.message);
    res.status(500).json({ error: err.message });
  }
};
```

Analizziamo il **controller**, che gestisce la richiesta di eliminazione di un esame.

In primo luogo, viene invocata la funzione **deleteExam** del **model** per elaborare la richiesta e ottenere le informazioni necessarie.

Il codice di stato della risposta dipende dall'esito dell'operazione:

- **200 OK**, se l'eliminazione è andata a buon fine.
- **404 Not Found**, se l'esame da eliminare non è stato trovato.
- **500 Internal Server Error**, se si verifica un errore imprevisto durante l'interazione con il database.

3.5 OPTIONS

```
app.options('/UniSignMeUp/v1/debugService', cors());
```

Abbiamo utilizzato il verbo **OPTIONS** per implementare una chiamata di debug semplice.

Questa richiesta non prevede l'invio di un corpo (body) al client, e di conseguenza il solo codice di risposta restituito sarà **204 No Content**.

Il servizio è concepito esclusivamente per scopi di debug e, pertanto, non fornirà alcun contenuto nel corpo della risposta.

Tuttavia, negli **headers** della risposta verranno inclusi tutti i dettagli relativi alla configurazione del server, fornendo informazioni utili per diagnosticare e verificare lo stato del sistema.

```
app.use(cors({
  origin: 'http://127.0.0.1:5501', // Allow only this origin
  methods: ['GET', 'OPTIONS', 'POST', 'DELETE', 'PUT'], // Allowed methods
  allowedHeaders: ['Content-Type', 'Authorization', 'Access-Control-Allow-Methods', 'Access-Control-Expose-Headers'], // Allowed headers
  exposedHeaders: ['Content-Type', 'Authorization']
}));
```

Le impostazioni configurate tramite **CORS** (Cross-Origin Resource Sharing) consentono di gestire le richieste effettuate da un client front-end verso un server back-end che appartiene a un dominio diverso.

CORS è un meccanismo di sicurezza che permette di specificare quali origini (domini) sono autorizzate a interagire con le risorse del server. In altre parole, quando un'applicazione web tenta di fare una richiesta HTTP a un server situato su un dominio differente rispetto a quello da cui è originata, CORS definisce le politiche per consentire o bloccare tale operazione.

Se correttamente configurato, CORS consente al server di specificare le origini consentite, i metodi HTTP supportati, gli headers autorizzati e altre regole che il browser deve rispettare durante il processo di richiesta e risposta. Questo meccanismo aiuta a prevenire attacchi di tipo cross-site request forgery (CSRF) e a garantire che solo le applicazioni web previste possano interagire con le risorse del server.

3.6 ROUTES, SERVER e DATABASE

Routes:

```
// USERS

// GET all users
router.get('/UniSignMeUp/v1/getAllUsers', userController.getAllUsers);

// POST a new user
router.post('/UniSignMeUp/v1/createUser', userController.createUser);

// GET single user
router.get('/UniSignMeUp/v1/getUser', userController.getUser);

// PUT change user pw
router.put('/UniSignMeUp/v1/changePassword', userController.changePassword);

// DELETE user
// router.delete('/UniSignMeUp/v1/deleteUser', userController.deleteUser);

//EXAM

// GET all exams
router.get('/UniSignMeUp/v1/getAllExams', examController.getAllExams);

// GET all exams for single user
router.get('/UniSignMeUp/v2/getExams', examController.getExams);

// POST a new exam
router.post('/UniSignMeUp/v1/createExam', examController.createExam);

// DELETE exam
router.delete('/UniSignMeUp/v1/deleteExam', examController.deleteExam);

module.exports = router;
```

Dopo aver analizzato il **model** e il **controller**, che costituiscono le parti attive del servizio, è opportuno esaminare le componenti "passive", ovvero quelle che inizializzano il servizio **UniSignMiUp**.

Il file **routes.js** definisce ed esporta tutti i servizi disponibili. In questa sezione vengono specificati il tipo di richiesta (che può essere "GET", "DELETE", ecc.) e l'URL da utilizzare dal client per invocare il servizio desiderato.

Server:

```
// Start the server
const port = 2024; // http://localhost:2024/UniSignMeUp/v1/...
// Start the server and test the database connection
app.listen(port, async () => {
  console.log(`Server running at http://localhost:${port}`);
  try {
    // Execute a simple query to check the connection
    const [rows] = await database.query('SELECT 1'); // 'SELECT 1' is a simple query to test connection
    console.log('Database connection is successful');
  } catch (error) {
    console.error('Database connection failed:', error.message);
  }
});
```

Il **server**, pur essendo una componente semplice, rappresenta il cuore centrale del servizio. Quando il servizio viene **"attivato"**, il file da eseguire è proprio questo. All'interno del server viene avviato il processo di ascolto per eventuali richieste in ingresso, generalmente sulla porta **localhost:2024**. Inoltre, viene eseguita una verifica preliminare per assicurarsi che il database sia correttamente in ascolto e funzionante, garantendo così la corretta operatività del sistema.

Database:

```
const mysql = require('mysql2/promise'); // async \await methods

var con = mysql.createPool({
  host: "127.0.0.1",
  user: "root",
  password: "statale2024",
  database: "unisignmiup",
  port: "3306"
});

// Export the pool for use in other files
module.exports = con
```

Infine, nel file dedicato al **database** vengono inizializzati i parametri di connessione, che consentono l'interazione con il database tramite il server. Questi parametri configurano la connessione necessaria per eseguire le operazioni di lettura, scrittura e aggiornamento delle informazioni nel database.

3.7 CHIAMATA A SERVIZIO

Si giunge, quindi, alla chiamata effettiva del servizio: in questa fase, il client invia una richiesta al server, che a sua volta gestisce la connessione con il database e restituisce la risposta appropriata in base all'operazione

```
const options = {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify(formData)
};

fetch("http://localhost:2024/UniSignMeUp/v1/createExam", options)
  .then(response => {
    if (response.status === 200) {
      alert("Esame prenotato!");

      // Genera il PDF con i dettagli dell'esame prenotato
      generatePDF(selectedExam, selectedDate, selectedTime, location, userEmail);

      // Reset campi
      $('#exam-name').val('');
      $('#location').val('Seleziona il luogo');
      $('#date-select').empty().append('<option value="" disabled selected>Seleziona una data</option>');
      $('#time-select').empty().append('<option value="" disabled selected>Seleziona un orario</option>');
    }
  })
  .catch(error => {
    uniLog("GENRICO")
    uniErrorType(error.message);
  });
```

Analizziamo il processo di iscrizione a un esame:

Inizialmente, vengono definite le opzioni per la chiamata, impostando il metodo su **POST**. L'unico header necessario è la dichiarazione del tipo di contenuto, che in questo caso è **application/json**. Successivamente, viene creato il **JSON** contenente i dati da inviare al servizio.

La chiamata vera e propria viene effettuata utilizzando la funzione nativa di JavaScript **fetch**, alla quale vengono passati l'URL e le opzioni di configurazione. Il flusso di esecuzione si articola tramite il blocco **then - catch**: il flusso entra nel blocco **catch** in caso di errore del server, mentre entra nel blocco principale **then** quando il server risponde correttamente.

In quest'ultimo caso, viene verificato se lo status della risposta sia **200 OK**.

Se la risposta è positiva, si procede con la creazione del database e il reset dei campi del modulo.

Quando viene effettuata una chiamata al servizio, il flusso che il server segue è il seguente:

server → **router** → **controller** → **model**.

3.8 DATABASE

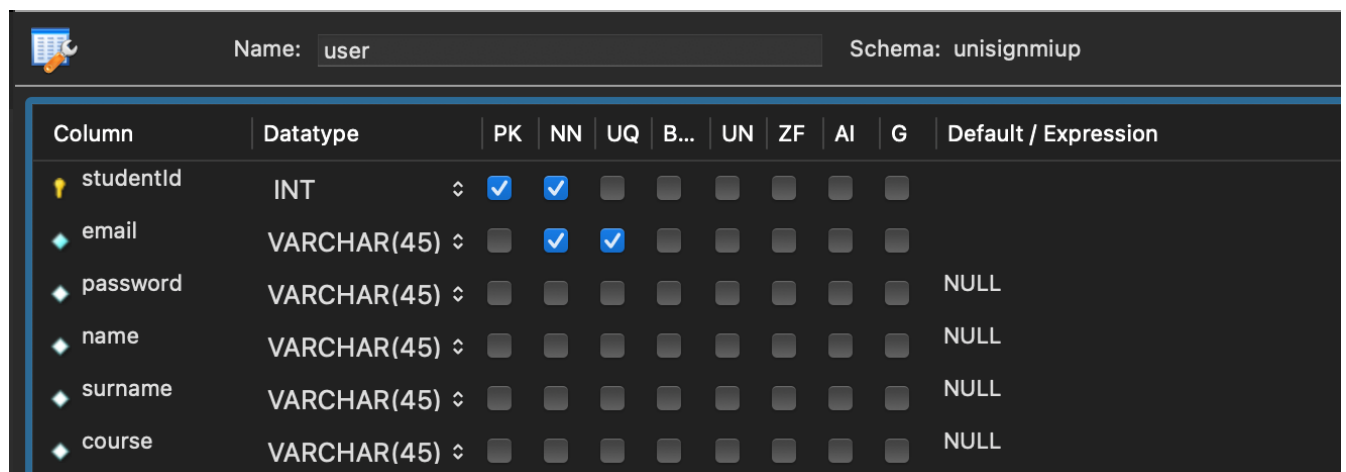
Il **database** è stato sviluppato utilizzando **MySQL** e comprende due tabelle principali: **"User"** e **"Exam"**, che sono essenziali per le operazioni di recupero e inserimento dei dati.

La tabella **"User"** gestisce le informazioni relative agli utenti registrati nel sistema. Viene utilizzata principalmente per verificare le credenziali durante il processo di login, assicurando che le informazioni inserite corrispondano a quelle registrate nel database.

La tabella **"Exam"**, invece, contiene i dettagli sugli esami a cui gli studenti universitari sono iscritti. Include le informazioni necessarie per la registrazione e l'accesso alla piattaforma, come date, orari e materie.

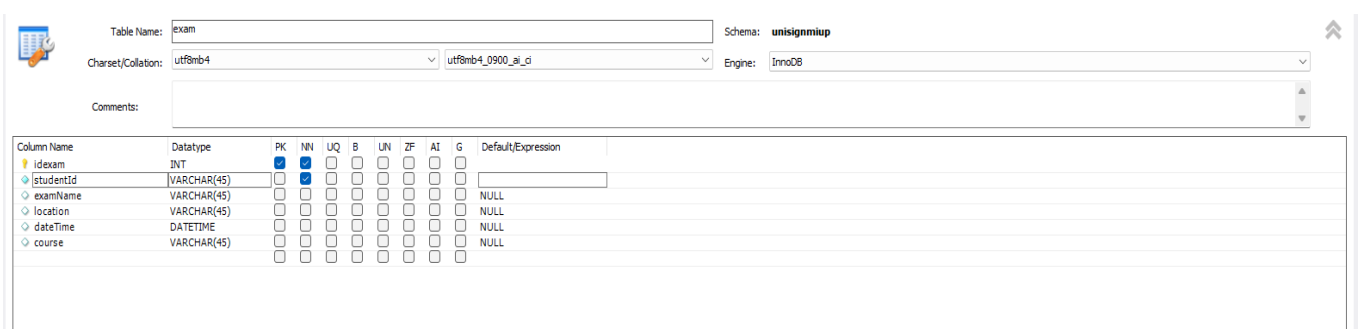
L'obiettivo principale nello sviluppo del database è stato quello di rendere la gestione dei dati il più intuitiva possibile, facilitando sia la creazione che la consultazione delle informazioni relative alle prenotazioni degli esami e agli utenti del sito.

Tabelle User:



Column	Datatype	PK	NN	UQ	B...	UN	ZF	AI	G	Default / Expression
studentId	INT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
email	VARCHAR(45)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
password	VARCHAR(45)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
name	VARCHAR(45)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
surname	VARCHAR(45)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
course	VARCHAR(45)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL

Tabelle Exam:



Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
idexam	INT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
studentId	VARCHAR(45)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
examName	VARCHAR(45)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
location	VARCHAR(45)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
dateTime	DATETIME	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
course	VARCHAR(45)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL

UniSignMiUp

4. CONCLUSIONI

4.1 CONSIDERAZIONI FINALI

Il progetto ha confermato la possibilità di sviluppare un'applicazione web dedicata alla **gestione delle prenotazioni degli esami**, sfruttando le tecnologie moderne descritte in precedenza.

Grazie a un'attenta analisi di fattibilità e a una continua attività di ricerca, è stato possibile identificare le soluzioni più adeguate a rispondere alle esigenze degli studenti e dell'Università Statale di Milano.

L'applicazione offre un sistema pratico e facilmente accessibile per la prenotazione degli esami. Il design e la struttura del sito sono stati progettati per semplificare al massimo l'esperienza dell'utente, riducendo al minimo il rischio di errori e migliorando l'efficienza nell'intero processo di iscrizione.