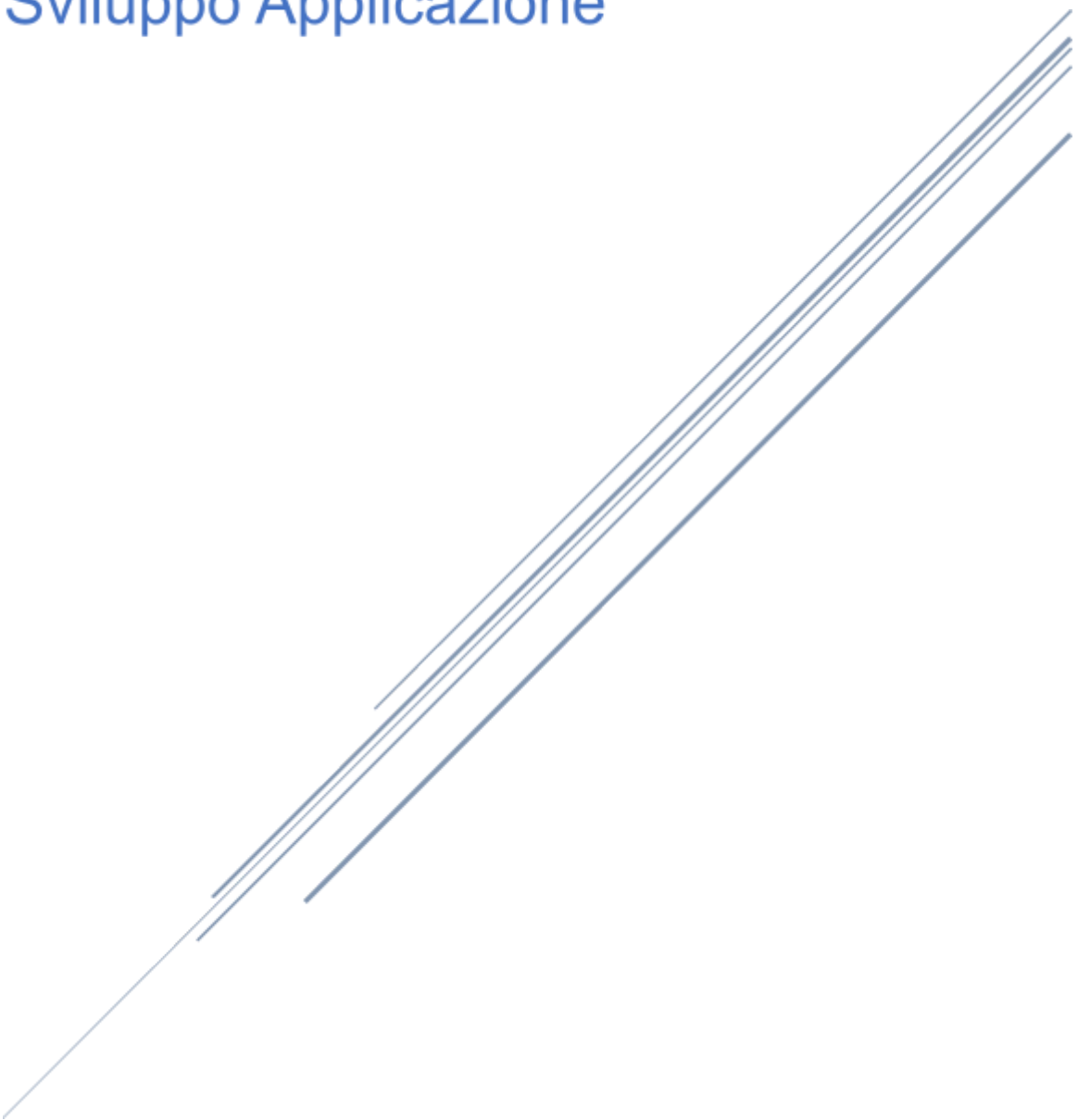
SOLVE.IT

Sviluppo Applicazione



Gruppo T053
Ingegneria del Software

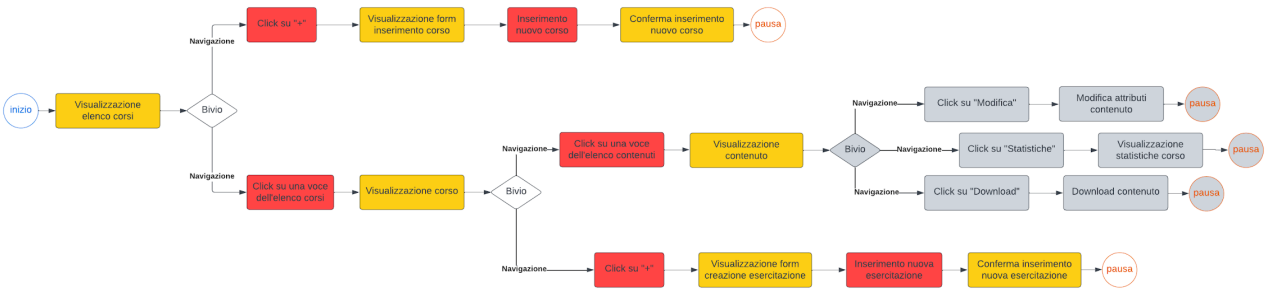
Sommario

Scopo del documento

Il presente documento riporta tutte le informazioni necessarie per lo sviluppo di una parte dell'applicazione Solve.it. In particolare, presenta tutti gli artefatti necessari per realizzare i servizi di gestione dei corsi e dei loro contenuti
Partendo dalla descrizione degli user flow legate al ruolo del docente, il documento prosegue con la presentazione delle API necessarie (tramite l'API Model e il Modello delle risorse) per poter visualizzare, inserire e modificare corsi, esercitazioni, e contenuti multimediali.
Per ogni API realizzata, oltre ad una descrizione delle funzionalità fornite, il documento presenta la sua documentazione e i test effettuati. Infine una sezione e' dedicata alle informazioni del Git Repository e il deployment dell'applicazione stessa.

1. User Flows

In questa sezione del documento di sviluppo riportiamo gli "user flows" per il ruolo del docente nella nostra applicazione.
Un nuovo corso, una nuova esercitazione, o un nuovo contenuto possono essere inseriti da un docente dopo aver riempito un Form. Questa operazione viene iniziata dal click sul bottone "+", che ha un doppio significato in base alla pagina in cui viene cliccato: crea un nuovo corso se cliccato dalla pagina dell'elenco dei corsi, crea un nuovo contenuto o esercitazione all'interno di un corso se cliccato dalla schermata della visualizzazione del corso. Il docente può anche decidere di modificare alcuni attributi di un contenuto o un'esercitazione dopo averli creati.



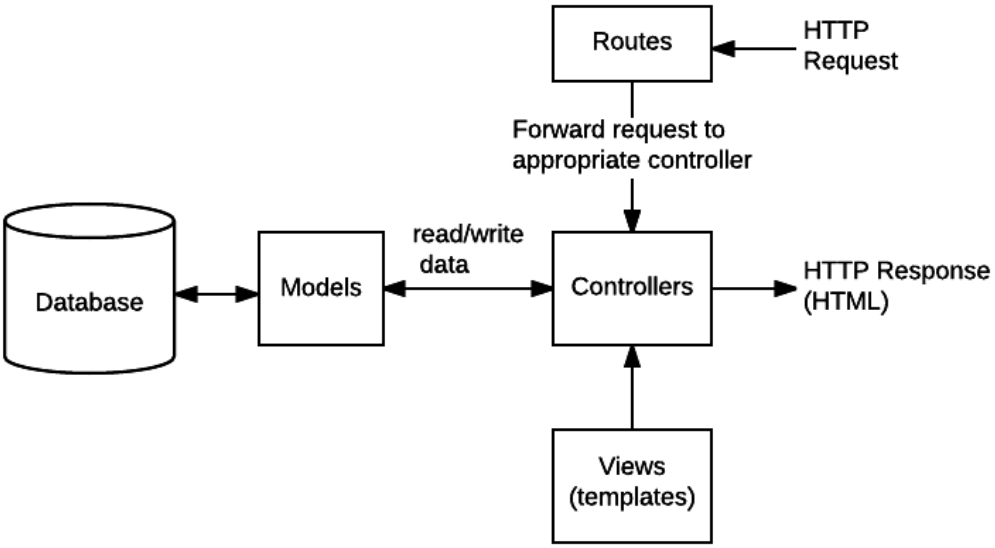
2. Implementazione e documentazione

Nelle sezioni precedenti abbiamo identificato le varie features che devono essere implementate per la nostra applicazione con un'idea di come i nostri utenti finali, docenti e studenti, possono utilizzarle nel loro flusso applicativo. L'applicazione è stata sviluppata utilizzando NodeJS ed Express. Per la gestione dei dati abbiamo

utilizzato MongoDB. L’interfaccia grafica è stata realizzata usando Bootstrap e Tailwind

2.1 Struttura del progetto

La struttura del progetto composta da una cartella routes che contiene i gestori delle rotte dell’applicazione, una cartella models che rappresenta i modelli dei dati all’interno del database e una cartella controllers, che contiene i controller, ovvero le parti dell’applicazione che gestiscono le logiche dell’applicazione.
Questa struttura è basata sul modello mvc (model view controller) rappresentata nella figura sottostante



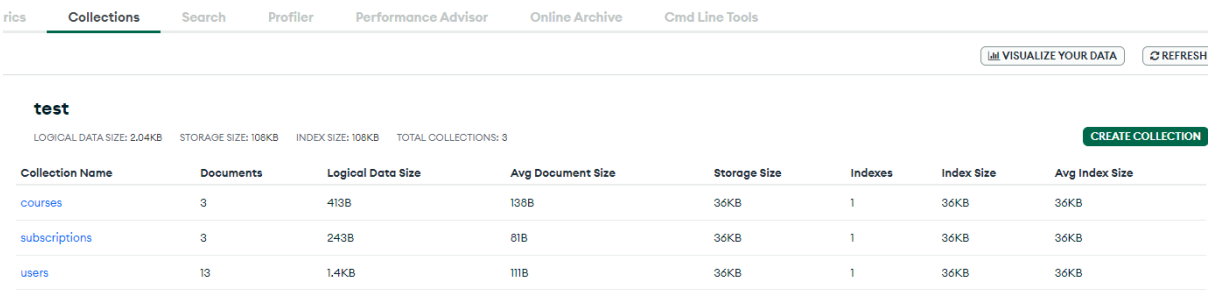
2.2 Dipendenze del progetto

I seguenti moduli Node sono stati utilizzati e aggiunti al file Package.Json

- Dotenv
- Express
- Mongoose

2.3Project Data Or DB

Per la gestione dei dati all’interno dell'applicazione abbiamo definito tre principali strutture di dati, una collection di utenti e una di corsi, ed una collection subscription che associa gli studenti ai corsi a cui sono iscritti.



Per rappresentare utenti e corsi e subscription abbiamo definito i seguenti tipi di dato:

```
_id: ObjectId('63d79d8992676e523dcde7c5')
title: "Analisi 1"
abstract: "integrali e numeri complessi"
teacher: "Barbara.Farappi"
id: "0"
```

corso, Analisi 1

```
_id: ObjectId('63d798f492676e523dcde7ba')
firstname: "Todd"
lastname: "Johns"
role: "teacher"
unitnName: "Todd.Johns"
```

studente, Todd Johns

```
_id: ObjectId('63e766e373a1fad831496173')
student: "Andrea.Rossi"
id: "63d79dcf92676e523dcde7c6"
```

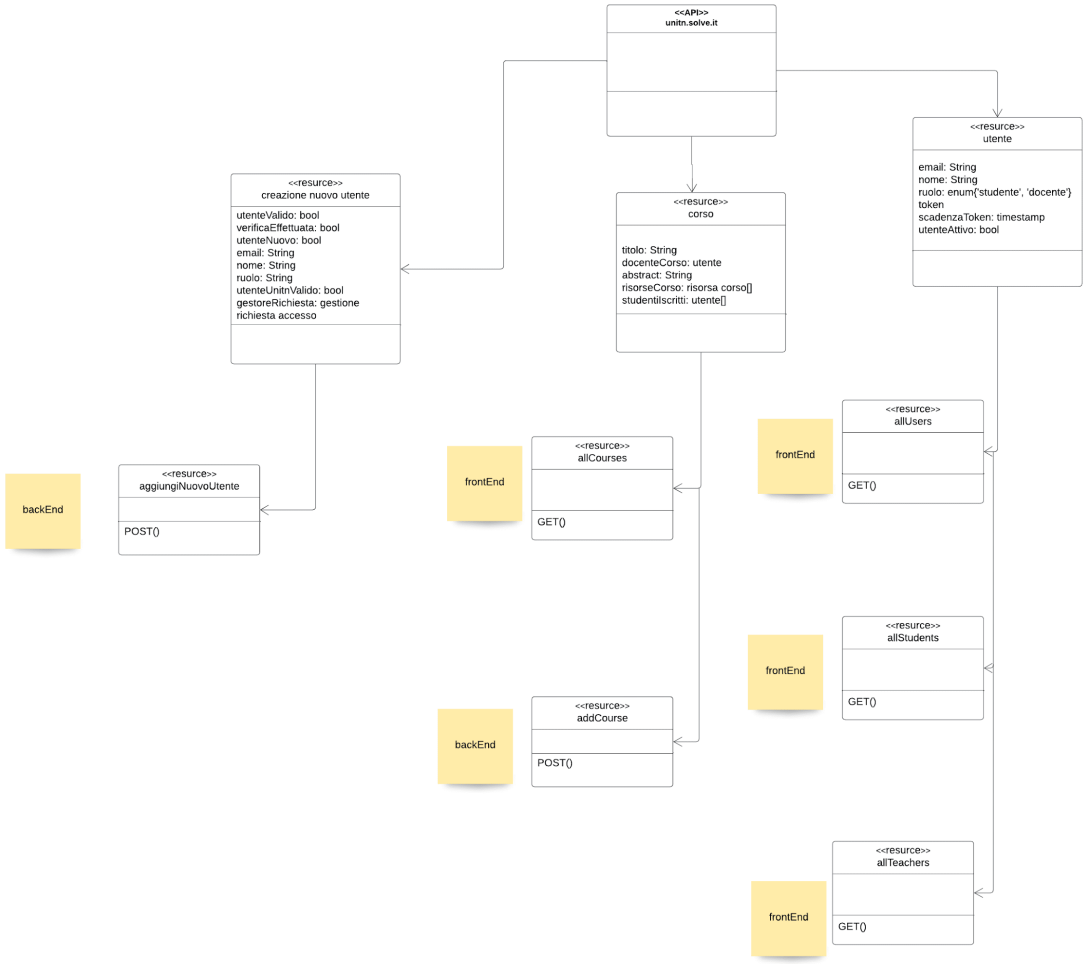
subscription di Andrea.Rossi al corso con id uguale al campo id

2.4 Project APIs

Per la gestione dei dati utili all'applicazione abbiamo definito due principali strutture dati: una per i corsi ed una per gli utenti

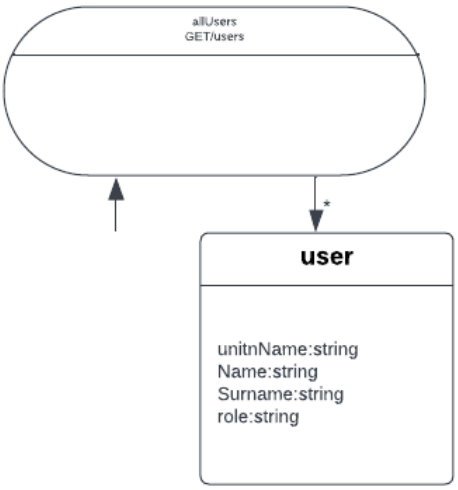
2.4.1 Risorse estratte dal class diagram

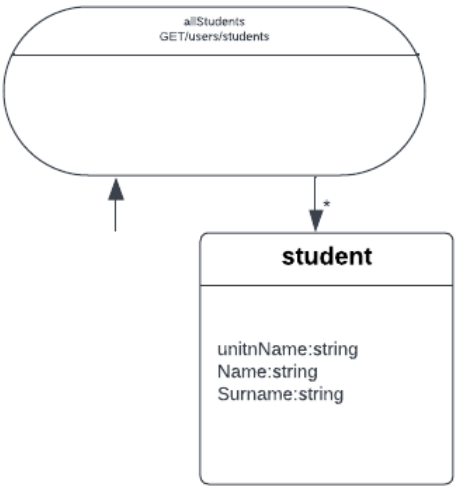
Nel diagramma sono mostrate quali risorse sono state estratte da quali classi per la definizione delle API, poiché nel diagramma delle classi si è scelto di non definire i metodi relativi al frontend, alcune delle risorse non sono direttamente collegate ad un metodo.

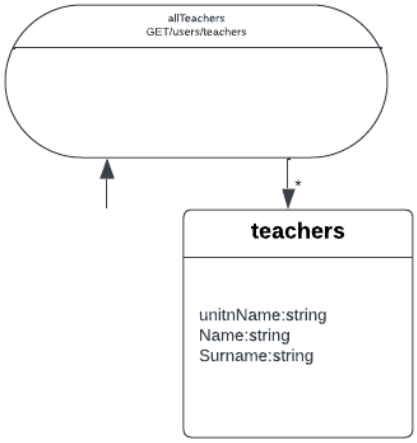


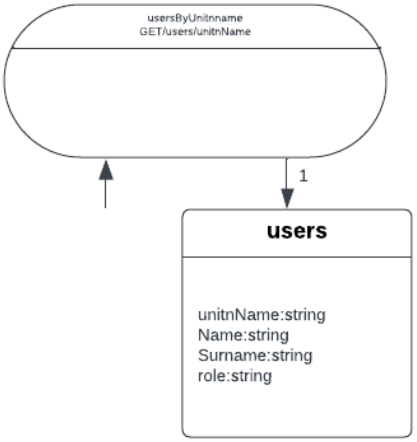
2.4.2 Resource model

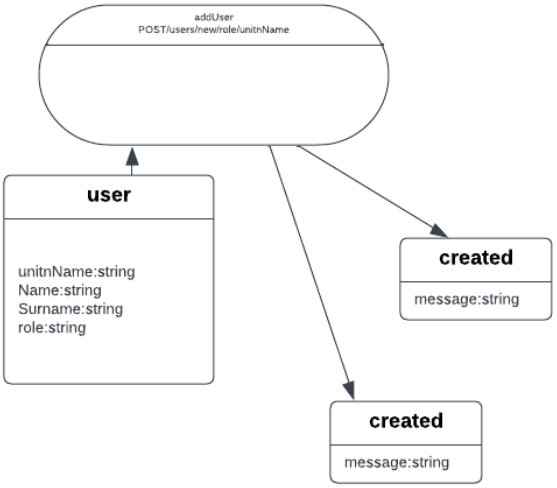
Il diagramma che segue mostra le risorse estratte dal diagramma delle classi per la definizione delle API, indicando i metodi,(GET-POST…), e i valori in entrata e in uscita per le API.

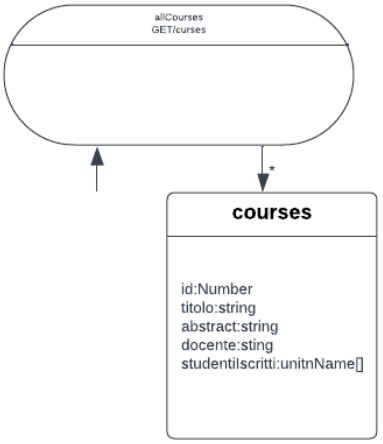

allUsers
GET/users
*
user
unitnName:string
Name:string
Surname:string
role:string

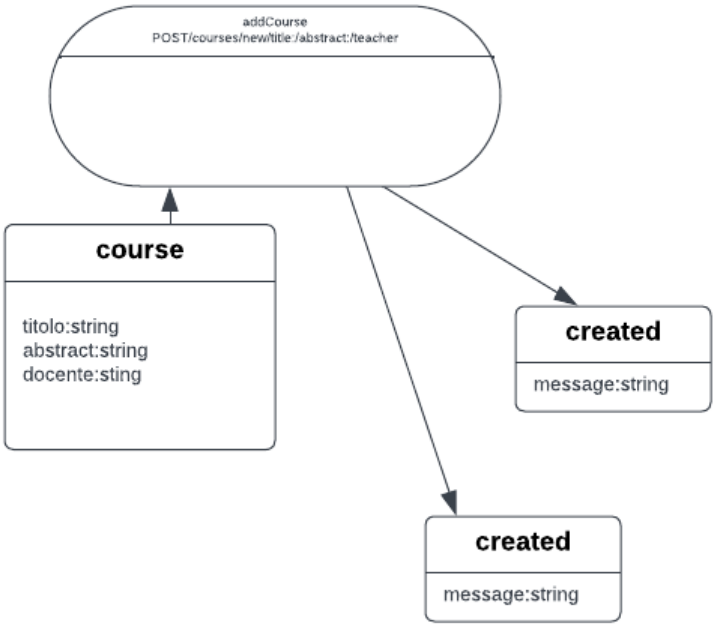

allStudents
GET/users/students
*
student
unitnName:string
Name:string
Surname:string


allTeachers
GET/users/teachers
*
teachers
unitnName:string
Name:string
Surname:string


usersByUnitnname
GET/users/unitnName
1
users
unitnName:string
Name:string
Surname:string
role:string


addUser
POST/users/new/role/unitnName
user
unitnName:string
Name:string
Surname:string
role:string
created
message:string
created
message:string


allCourses
GET/curses
*
courses
id:Number
titolo:string
abstract:string
docente:sting
studentiIscritti:unitnName[]



2.5 Sviluppo API

2.5.1 Aggiunta utente

Tramite questa API l'applicazione può aggiungere un utente al sistema.

```
router.post('/users/new/:role/:unitnname', userctr.addUser);
```

```
const addUser = async (req, res, next) => {
    result = await usermodel.addUser(req.params.unitnname)
    res.sendStatus(result == 0 ? 200 : 400);
}
```

```
async function addUser(name, role) {
    // Aggiunge l'utente con nome name e ruolo role al sistema
    // ritorna 0 in caso tutto sia andato a buon fine
    // 1 se l'utente esiste già, -1 in caso di errore

    let result = await usermodel.find({unitnName: name});

    if(result.length > 0){
        return 1;
    }
    else{
        usermodel.create({
            unitnName: name,
            firstname: name.split(".")[0],
            lastname: name.split(".")[1].split("-")[0],
            role: role
        }, function (err, user) {
            if (err){
                return -1;
            }
        })
        return 0;
    }
};
```

2.5.2 Elenco utenti

Tramite questa API l'applicazione può recuperare l'elenco degli utenti attualmente memorizzati nel sistema. Ritorna gli utenti utilizzando il metodo find({})

```
router.get('/users', userctr.allUsers);
```

```
const allUsers = async (req, res, next) => {
    result = await usermodel.allUsers()
    res.status(200);
    res.json(result);
}
```

```
async function allUsers () {
    let users = await usermodel.find({});
    users = users.map((user) => {
        return {
            unitnName: user.unitnName,
            firstname: user.firstname,
            lastname: user.lastname,
            role: user.role
        }
    });
    return users;
};
```

2.5.3 Elenco studenti

Tramite questa API l'applicazione può recuperare l'elenco degli studenti attualmente memorizzati nel sistema. Ritorna gli studenti utilizzando il metodo find({role: 'student'})

```
router.get('/users/students', userctr.allStudents);
```

```
const allStudents = async (req, res, next) => {
    result = await usermodel.allStudents()
    res.status(200);
    res.json(result);
}
```

```
async function allStudents () {
    let students = await usermodel.find({role: 'student'});
    students = students.map((student) => {
        return {
            unitnName: student.unitnName,
            firstname: student.firstname,
            lastname: student.lastname,
            // role: user.role
        }
    });
    return students;
};
```

2.5.4 Elenco docenti

Tramite questa API l'applicazione può recuperare l'elenco dei docenti attualmente memorizzati nel sistema. Ritorna i docenti utilizzando il metodo find({role: 'teacher'})

```
router.get('/users/teachers', userctr.allTeachers);
```

```
const allTeachers = async (req, res, next) => {
    result = await usermodel.allTeachers()
    res.status(200);
    res.json(result);
}
```

```
async function allTeachers () {
    let teachers = await usermodel.find({role: 'teacher'});
    teachers = teachers.map((teacher) => {
        return {
            unitnName: teacher.unitnName,
            firstname: teacher.firstname,
            lastname: teacher.lastname,
            // role: user.role
        }
    });
    return teachers;
};
```

2.5.5 Utente dato il nome

Dato il nome di un utente, restituisce l'oggetto che rappresenta quell'utente memorizzato nel sistema, se l'utente non esiste ritorna una pagina vuota con codice di stato 404

```
router.get('/users/:unitnname', userctr.usersByUnitnname);
```

```
const usersByUnitnname = async (req, res, next) => {
    result = await usermodel.usersByUnitnname(req.params.unitnname)
    if(result.length == 0){
        res.status(404);
        res.send();
    }
    else{
        res.status(200);
        res.json(result);
    }
}
```

```
async function usersByUnitnname (name) {
    let result = await usermodel.find({unitnName: name});
    result = result.map((user) => {
        return {
            unitnName: user.unitnName,
            firstname: user.firstname,
            lastname: user.lastname,
            role: user.role
        }
    });
    return result;
};
```

2.5.6 Elenco corsi

Tramite questa API l'applicazione può recuperare l'elenco dei corsi attualmente memorizzati nel sistema.

```
router.get('/courses', coursesctr.allCourses);
```

```
const allCourses = async (req, res, next) => {
    result = await coursemodel.allCourses()
    res.status(200)
    res.send(result)
}
```

```
async function allCourses () {
    let courses = await coursemodel.find({});
    courses = courses.map((course) => {
        return {
            id: course.id,
            titolo: course.title,
            abstract: course.abstract,
            docente: course.teacher,
        }
    });
    return courses;
};
```

2.5.7 Aggiunta corso

Tramite questa API l'applicazione può aggiungere un corso al sistema

```
router.post('/courses/new/:title/:abstract/:teacher', coursesctr.addCourse);
```

```
const addCourse = async (req, res, next) => {
    result = await coursemodel.addCourse(req.params.title, req.params.abstract, req.params.teacher)
    res.sendStatus(result == 0 ? 200 : 400)
}
```

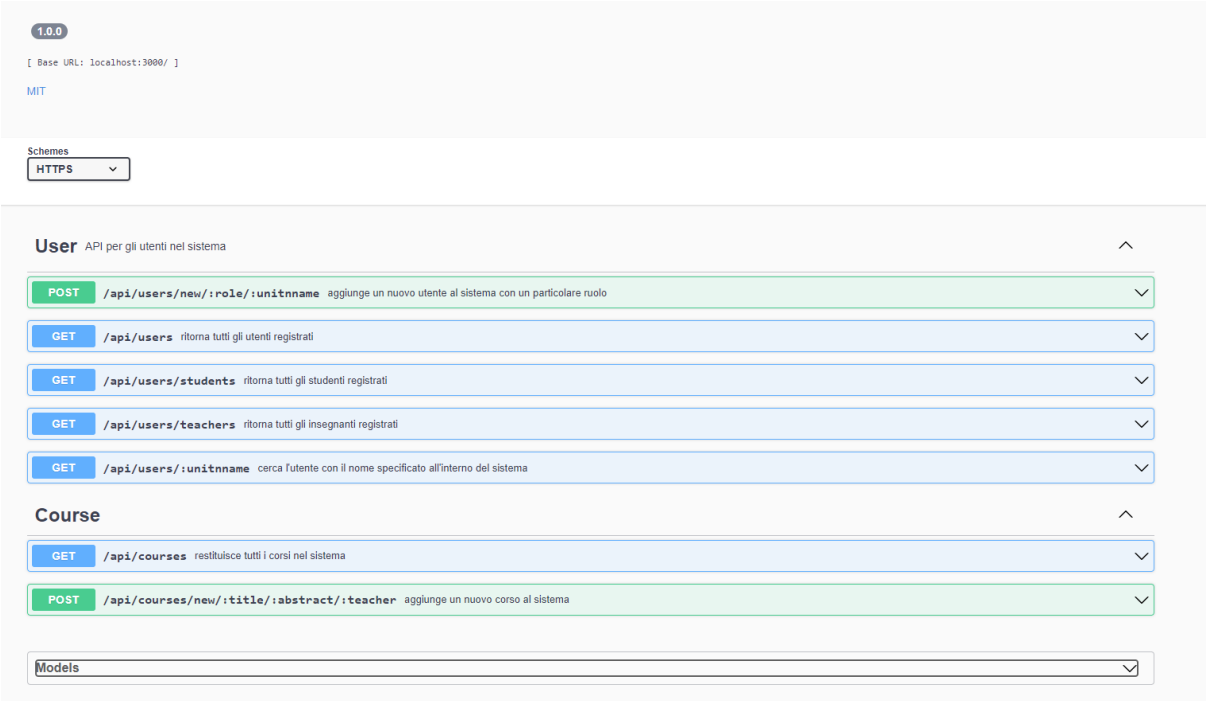
```
async function addCourse (title, abstract, teacher) {
    // Aggiunge un corso con titolo, abstract, ed insegnante specificati
    // ritorna 0 in caso l'operazione sia andata a buon fine
    // -1 in caso di errore

    const course = new coursemodel({
        title: title,
        abstract: abstract,
        teacher: teacher
    });
    course.save((err, data) => {
        if(err){
            return -1
        }
    })

    return 0;
};
```

3 Documentazione API

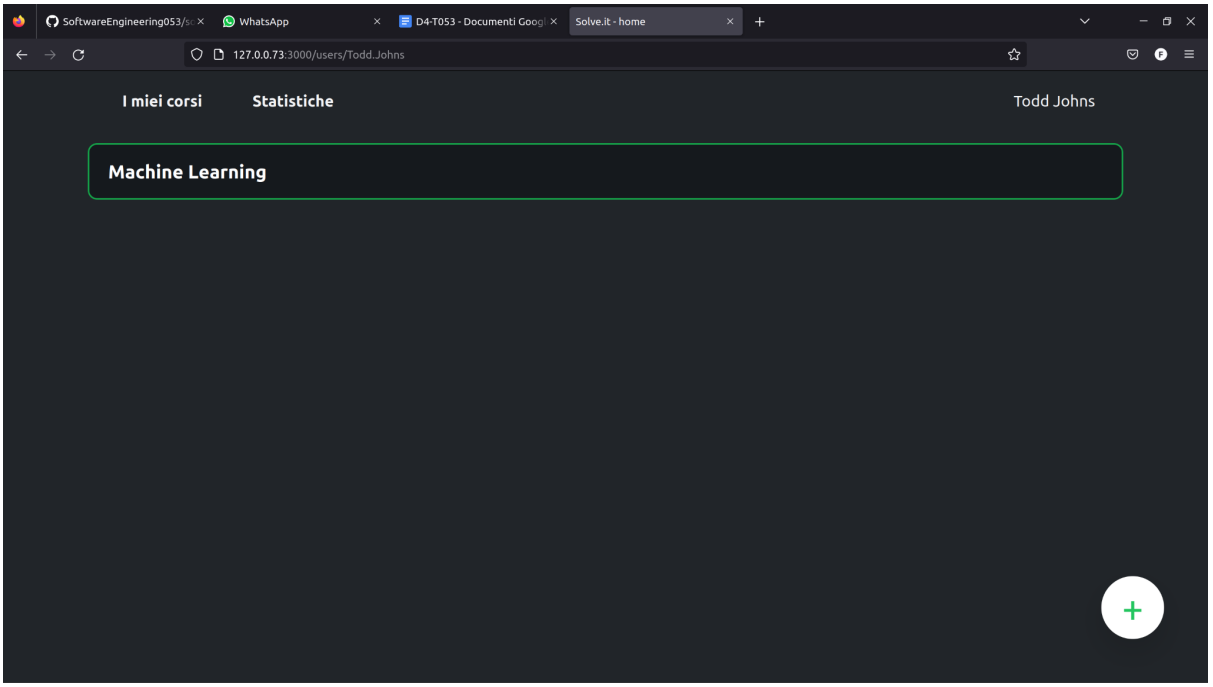
Le API fornite dall' applicazione Solve.it sono state documentate utilizzando il modulo NodeJS Swager UI Express.
Swagger UI crea una pagina web contenente la documentazione delle API, accessibile all'endpoint "http://localhost:3000/docs".
In questo modo la documentazione è direttamente disponibile a chiunque veda il codice.
In particolare sotto mostriamo la pagina web che contiene la documentazione relativa alle API del progetto.



Per user la POST permette di aggiungere un nuovo user, le successive tre GET permettono di ritornare tutti gli utenti registrati, tutti gli studenti e i professori. L'ultima GET permette di cercare un utente per unitnName.
Per Course la GET restituisce tutti i corsi del sistema, la POST permette di aggiungere un nuovo corso.

4 Implementazione FrontEnd

Il FrontEnd fornisce le funzionalità di visualizzazione di parte dei dati gestiti dall'applicazione solve.it. In particolare, l'applicazione è composta da una pagina di login, da una pagina home per studenti ed una home per i docenti.



5 Repository GitHub

La repository di github che contiene il progetto è composta da 2 sottomoduli: uno per il front-end e uno per il back-end. é possibile installare ed eseguire l'applicazione con i seguenti comandi:

git clone https://github.com/SoftwareEngineering053/solve-it.git

cd solve-it/

git submodule update --init --recursive

cd back-end/nodeProject/

npm install

echo "DEV_PORT=3000

DB_URI=mongodb+srv://IvanTonidandel:Mongolodb23@cluster0.zjvjpp7.mongodb.net/?retryWrites=true&w=majority

STATIC_PATH=../../front-end/src/" > .env

node server.js

6 Test

6.1 Test locali

Abbiamo inserito nel database alcuni utenti manualmente ed effettuato test in locale per verificare che il codice scritto funzionasse

Tipo richiesta	Percorso richiesta	Codice risposta
GET	http://localhost:3000/api/users	200 OK
GET	http://localhost:3000/api/users/students	200 OK
GET	http://localhost:3000/api/users/teachers	200 OK
GET	http://localhost:3000/api/courses	200 OK
POST	http://localhost:3000/api/users/new/student/Sara.Verdi	200 OK
POST	http://localhost:3000/api/users/new/student/Sara.Verdi	400 Bad request*

I contenuti delle risposte erano quelli attesi, da come si può notare dalle risposte non è possibile aggiungere due volte un utente con lo stesso nome