

UNIVERSITY
OF TRENTO

**Docu-
ment:**

Partfinder_Sviluppo

0.0

**Dipartimento di
Ingegneria e
Scienza
dell'Informazione**

Progetto:

PartFinder

Titolo del documento:

Sviluppo Applicazione

Document Info

Doc. Name	<i>D4-PartFinder_Sviluppo</i>	Doc. Number	D4
Description	Documento di sviluppo dell'applicazione		

INDICE

Scopo del documento	4
1. User Flows	4
2. Application Implementation and Documentation	5
2.1. Project Structure	5
2.2. Project Dependencies	7
2.3. Project Data or DB	8
2.4. Project APIs	10
2.4.1. Resources Extraction from the Class Diagram	10
2.4.2. Resources Models	11
2.5. Sviluppo API	13
2.5.1. Elenco gruppi	13
2.5.2. Creazione di un gruppo	15
2.5.3. Cancellazione di un gruppo	16
2.5.4. Modifica di un gruppo	17
2.5.5. Richiesta di unione a un gruppo	18
2.5.6. Rifiuto richiesta di unione	19
2.5.7. Accettazione richiesta di unione	19
2.5.8. Rimozione giocatore	20
2.5.9. Invio di un messaggio in chat	21
2.5.10. Recupero dei messaggi della chat	22
2.5.11. Lancio di un dado	23
2.5.12. Simulatore per lancio di dadi	24
2.5.13. Registrazione	25
2.5.14. Login	26
2.5.15. Visualizzazione profilo utente	27
2.5.16. Modifica profilo utente	28
2.5.17. Like a un profilo utente	29
2.5.18. Dislike a un profilo utente	30
2.5.19. Creazione di un personaggio	31

2.5.20. Visualizzazione di un personaggio	32
2.5.21. Modifica di un personaggio	33
2.5.22. Eliminazione di un personaggio	34
3. API documentation	35
4. FrontEnd Implementation	36
5. GitHub Repository and Deployment Info	41
6. Testing	41

Scopo del documento

Il presente documento riporta tutte le informazioni sullo sviluppo di una parte dell'applicazione PartFinder. In particolare, vengono presentati gli elementi necessari per la gestione dei gruppi creati da utenti. Partendo dalla descrizione dello user flow, il documento prosegue con la presentazione delle API necessarie (tramite l'API Model e il Modello delle risorse) per poter gestire i gruppi e l'interazione tra i giocatori, registrare un nuovo profilo utente e fare il login con le proprie credenziali, gestire i personaggi creati dagli utenti.

Per ogni API realizzata, oltre ad una descrizione delle funzionalità fornite, il documento presenta la sua documentazione e i test effettuati. Infine una sezione è 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 dell'utente dell'applicazione. Figura 2 descrive lo user flow relativo alle azioni che un utente registrato può compiere utilizzando il sistema PartFinder. L'utente registrato può navigare tra le 4 parti dell'applicazione(home,personaggi,campagne,profilo).

Nella sezione 'home' l'utente visualizza tutti i gruppi creati dagli altri giocatori e può decidere di unirsi.

Nella sezione 'personaggi' l'utente può creare un nuovo personaggio, visualizzare i suoi personaggi, modificare un personaggio esistente ed eliminare un suo personaggio.

Nella sezione 'campagne' l'utente può creare un nuovo gruppo, visualizzare i suoi gruppi, modificare un gruppo esistente, gestire le richieste degli altri giocatori, unirsi alla chat di gruppo.

Nella sezione 'profilo' l'utente può modificare le sue informazioni.

Nella sezione 'profilo' di un altro utente è possibile mettere Like o Dislike al profilo.

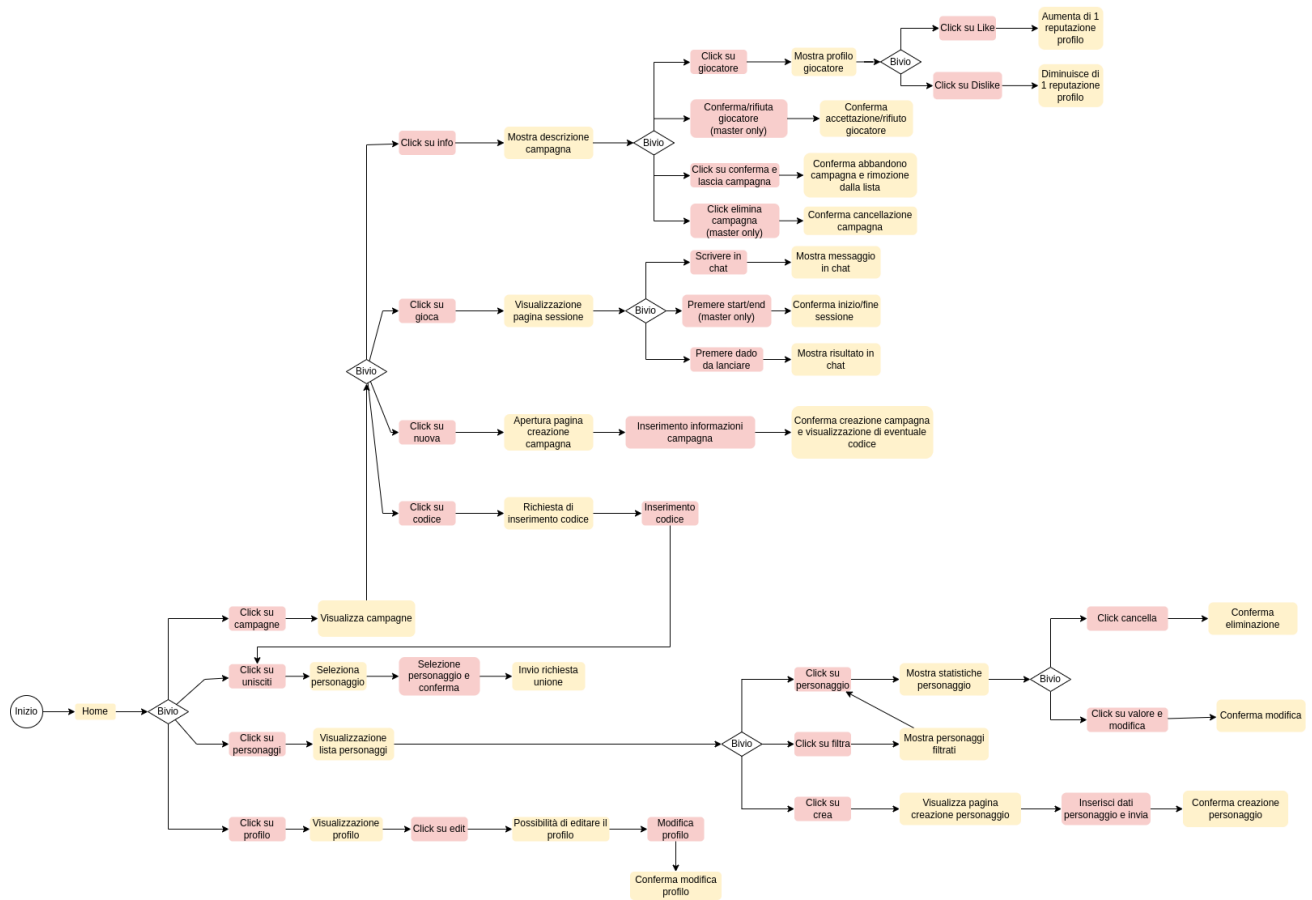


Fig.2: User Flow per l'utente registrato.

2. Application Implementation and Documentation

Nelle sezioni precedenti abbiamo identificato le varie features che devono essere implementate per la nostra applicazione.

L'applicazione è stata sviluppata utilizzando NodeJS. Per la gestione dei dati abbiamo utilizzato MongoDB.

2.1. Project Structure

La struttura del progetto è presentata in Figura 3. Il progetto è composto da 5 sottocartelle: nella cartella **controllers** sono contenute le implementazioni delle API, nella cartella **models** sono contenuti i modelli dati di MongoDB, nella cartella **routes** sono contenute le definizioni degli endpoint delle API, nella cartella **misc** è contenuto il file .env e nella cartella **node_modules** sono contenuti i file dei package importati nel progetto.

Nella cartella root del progetto sono contenuti i file **app.js**, **server.js**, **swagger.js** per la

**Docume
nt:**

PartFinder_Sviluppo
0.0

prima generazione del file swagger.json e il file **tokenChecker.js** usato come middleware per il controllo del token di autenticazione.

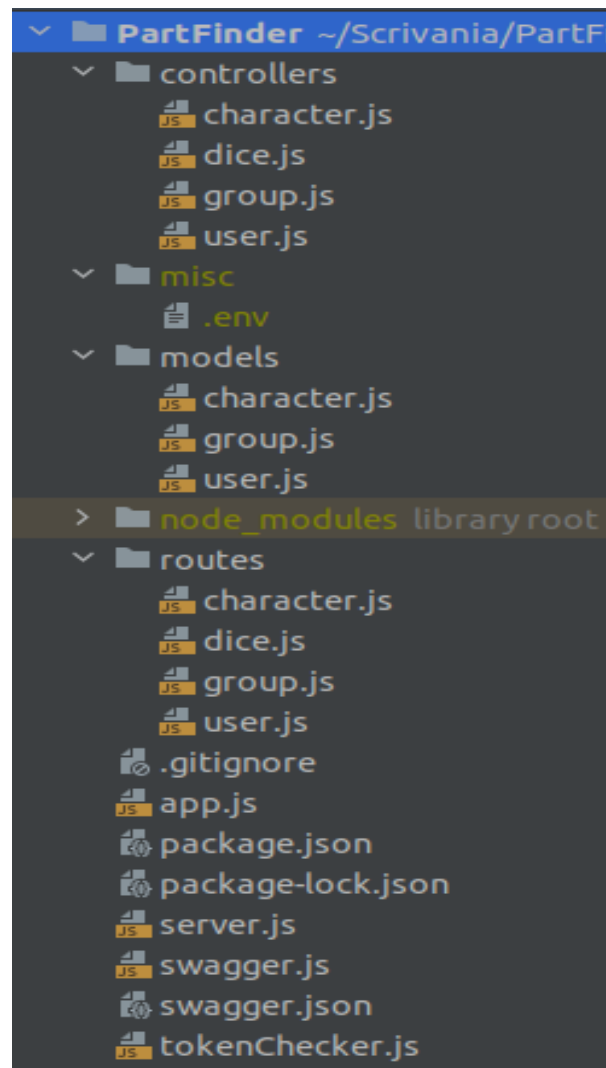


Fig.3: Struttura Codice Sorgente.

2.2. Project Dependencies

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

- Express
- Express-file upload
- MongoDB
- Mongoose
- JSONWebToken
- dotenv
- cookie-parser
- axios
- bcrypt
- lodash

- swagger-ui-express
- swagger

2.3. Project Data or DB

Per la gestione dei dati dell'applicazione abbiamo definito tre strutture dati come illustrato in Figura 4. Una collezione di "users", una collezione di "characters" e una collezione di "groups".

Collection Name	Documents	Logical Data Size	Avg Document Size	Storage Size	Indexes	Index Size	Avg Index Size
characters	3	1.16KB	398B	36KB	1	36KB	36KB
groups	2	524B	262B	36KB	2	72KB	36KB
users	3	768B	256B	36KB	3	108KB	36KB

Fig.4: Collezione Dati usati nell'applicazione.

Per rappresentare le collezioni descritte sopra vengono definiti i seguenti tipi di dati (vedi Figure 5,6 e 7).

```
_id: ObjectId('63a739f374971a2150b2c6f2')
username: "master"
email: "master@example.com"
password: "$2b$10$j1oLo/574FZGrfwW/reyG.Prq0z9lTMS67qwST5Qrujck50VlGnkC"
image: "image.png"
description: "top"
reputation: 0
isVerified: false
> upvotes: Array
> downvotes: Array
__v: 0
```

Fig.5: Tipo di Dato "users"

```
_id: ObjectId('63a9ae1e99a3c59627b76509')
code: "ADEXB"
master: "63a739f374971a2150b2c6f2"
name: "group2"
description: "testing cookie"
size: 5
> characters: Array
> requests: Array
> messages: Array
__v: 0
```

Fig.5: Tipo di Dato "groups"


```
_id: ObjectId('63a73a5974971a2150b2c6fb')
user: "63a739fe74971a2150b2c6f4"
image: "img1"
name: "char1modified"
✓ stats: Array
  ✓ 0: Object
    stat: "strength"
    value: 10
    _id: ObjectId('63a73a7574971a2150b2c70b')
  > 1: Object
  > 2: Object
  > 3: Object
✓ inventory: Array
  0: "dead rat"
  1: "bacon"
  2: "another dead rat"
  3: "18"
class: "warrior1"
__v: 0
```

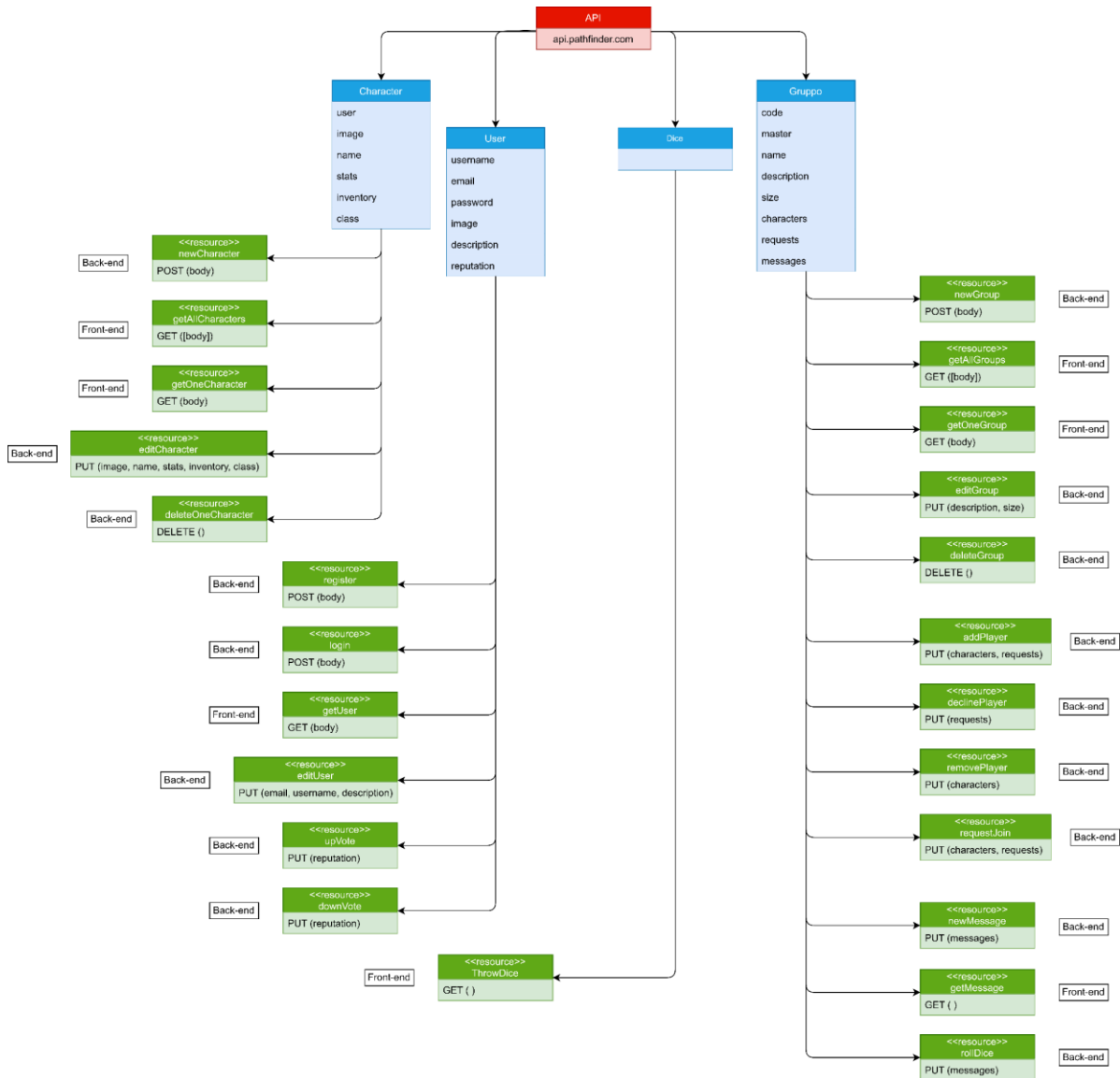
Fig.5: Tipo di Dato "characters"

2.4. Project APIs

2.4.1. Resources Extraction from the Class Diagram

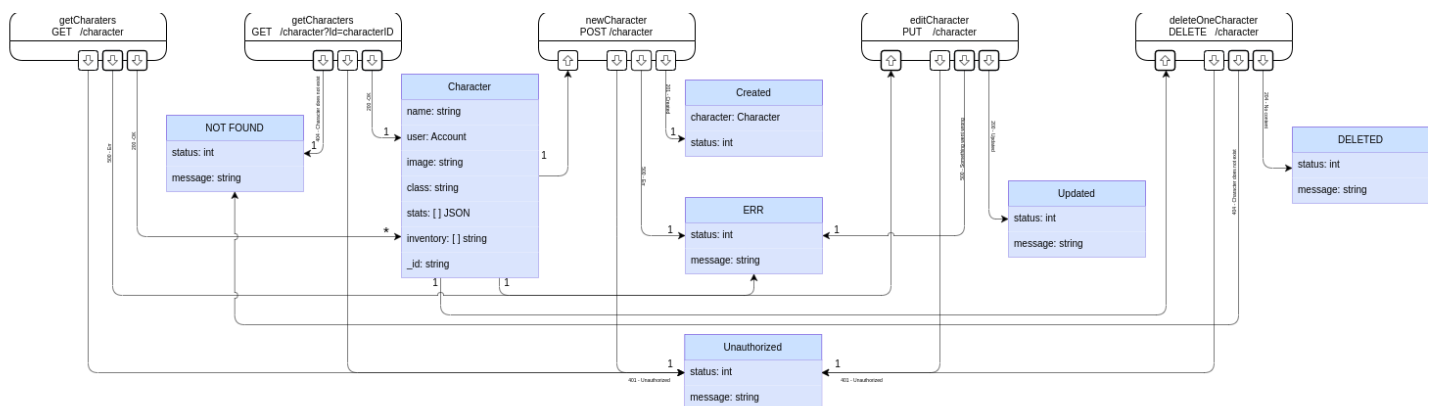
Questa parte del deliverable mostra il diagramma delle API estratto dal diagramma delle classi. All'interno possiamo trovare tutte le tre API sviluppate con le loro diverse chiamate. In particolare possiamo trovare la API Character, che gestisce la creazione, visualizzazione e modifica dei personaggi di un utente. Successivamente abbiamo API User per la registrazione, il login, la visualizzazione e modifica del profilo e la visualizzazione e modifica della reputazione. L'API Group per mostrare i gruppi, gestire le richieste di unione degli utenti e permettere l'interazione tra gli utenti tramite chat. Infine abbiamo l'API Dice che serve per simulare un lancio di dadi.

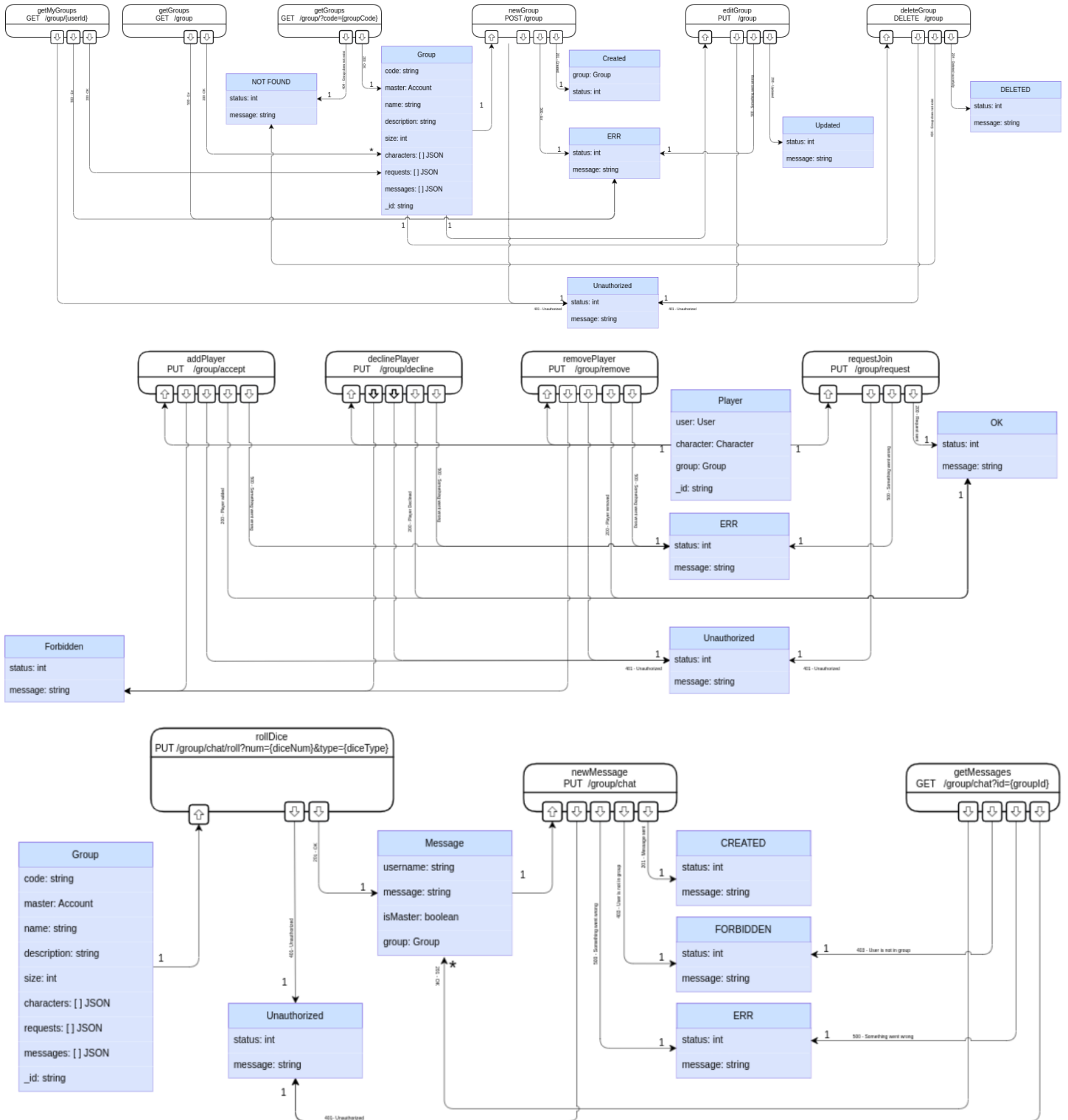
Tutto ciò viene fatto specificando il metodo HTTP di ogni chiamata, i valori che vengono richiesti e se la chiamata viene gestita dal front-end o dal back-end.

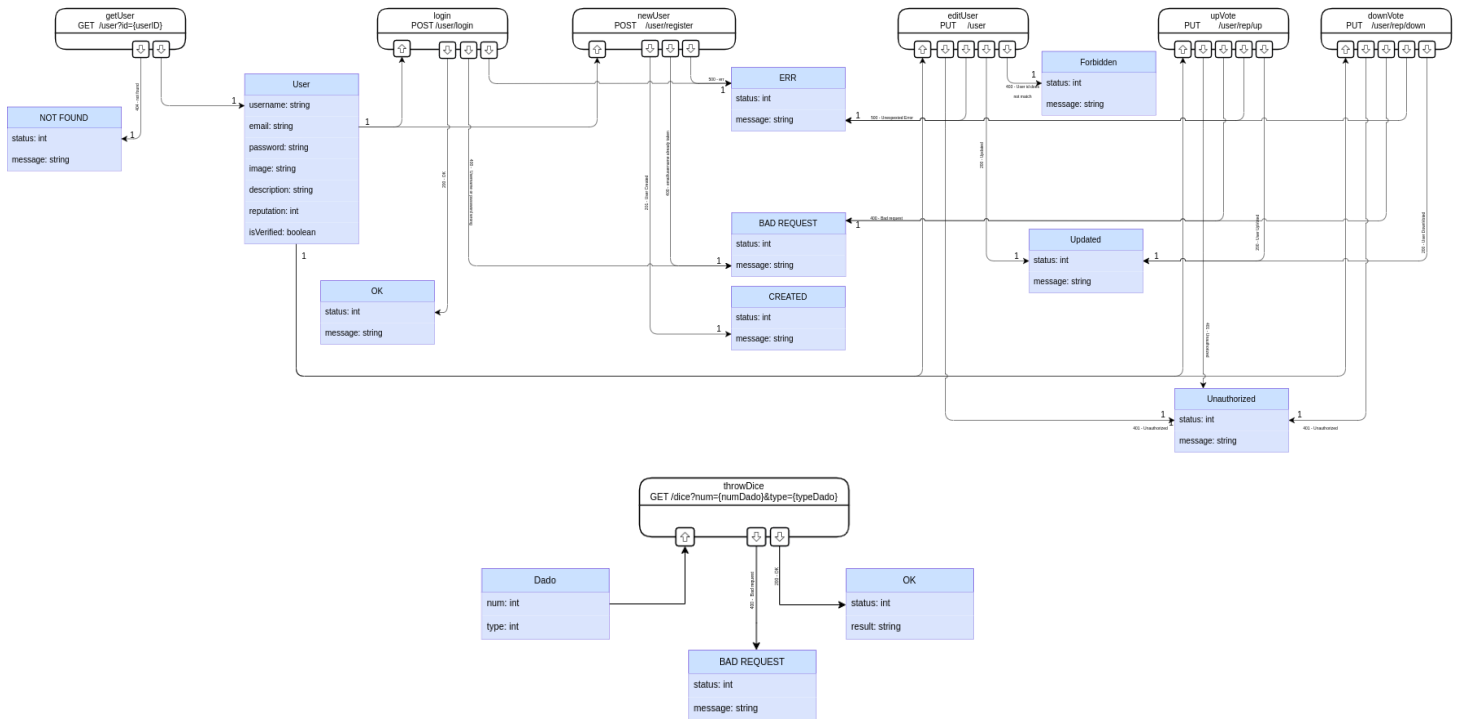


2.4.2. Resources Models

Il diagramma delle risorse mostra più in dettaglio alcuni aspetti del diagramma delle API. In particolare mostra gli input e output di alcune chiamate, i vari errori che possono essere generati e le path di ogni singola chiamata







2.5. Sviluppo API

A partire dal Resource Model sono state sviluppate le seguenti API:

- **group:** per la gestione dei gruppi e l'interazione tra i giocatori.
- **user:** per la registrazione e login dell'utente, per la visualizzare il profilo di un altro giocatore.
- **character:** per la creazione di un personaggio da parte di un utente.
- **dice:** simulatore di lancio di dadi.

2.5.1. Elenco gruppi

Tramite una chiamata GET all'API group, questa restituisce i gruppi presenti nel database. Nel caso in cui si passasse come parametro il codice univoco del gruppo, l'API restituirà il gruppo con il codice corrispondente. Nel caso in cui si passasse come parametro l'id dell'utente, l'API restituirà tutti i gruppi di cui l'utente fa parte.

In quest'ultimo caso l'API viene eseguita solo dopo aver eseguito il middleware **tokenChecker.js** per il controllo del token di autenticazione.

```
router.get(path: '/group', groupController.getGroups);
router.get(path: '/group/:user', tokenChecker, groupController.getMyGroups);
```

```
const getGroups=(req,res)=>{
  let code=req.query.code;
  if(!code) {
    Group.find({}, (err, data) => {
      if (err) return res.status(500).json({Error: err, status: 500});
      return res.status(200).json({data: data, status: 200});
    })
  }
  else {
    Group.findOne({code: code}, (err, data) => {
      if (err || !data) return res.status(404).json({message: "Group does not exist", status: 404});
      return res.status(200).json({data: data, status: 200});
    })
  }
}
```

```
const getMyGroups=(req,res)=>{
  let userInfo = req.userInfo;
  let user = req.params.user;
  if(user === userInfo.id)
    Group.find({$or:[{characters:{user:userInfo.id}},{master:userInfo.id}]}, (err, data) => {
      if (err || !data) return res.status(404).json({message: "Group does not exist", status: 404});
      else return res.status(200).json({data: data, status: 200});
    })
  else return res.status(403).send();
}
```

2.5.2. Creazione di un gruppo

Con una chiamata POST all'API group viene creato un nuovo gruppo. Per prima cosa viene chiamata la funzione **generateCode()** che genera un codice casuale da 5 lettere. La funzione continua a essere richiamata finché il codice non è univoco.

```
router.post(path: '/group', tokenChecker, groupController.newGroup);

const newGroup = async (req, res) => {
  let userInfo = req.userInfo;
  let code;
  do {
    code = generateCode();
  } while (await Group.exists({code: code}).exec());
  let newGroup = new Group( doc: {
    name: req.body.name,
    code: code,
    master: userInfo.id,
    description: req.body.description,
    size: req.body.size,
    characters: [],
    requests: [],
    messages: []
  })
  newGroup.save((err, data) => {
    if (err) return res.status(500).json({Error: err, status: 500});
    return res.status(201).json({data: data, status: 201});
  })
};
```

2.5.3. Cancellazione di un gruppo

Con una chiamata DELETE avviene la cancellazione di un gruppo. All'API viene passato l'ID del gruppo e, dopo aver controllato che l'utente che ha effettuato la chiamata sia il creatore del gruppo, lo elimina.

```
router.delete(path: '/group', tokenChecker, groupController.deleteGroup);

const deleteGroup=(req, res) => {
  let id = req.body.id;
  let userInfo=req.userInfo;
  let master=userInfo.id;
  Group.findById(id, (err, data)=>{
    if (err) return res.status(500).json({message: "Unexpected error", status: 500});
    else if (!data) return res.status(404).json({message: "Group does not exist", status: 404});
    else if(data.master!==master) return res.status(403).send();
    else
      Group.findByIdAndDelete(id, options: (err)=>{
        if(err) return res.status(500).json({message:'Unexpected error',status: 500});
        else return res.status(204).send();
      })
  })
}
```


2.5.4. Modifica di un gruppo

Questa API viene utilizzata per la modifica di un gruppo. Prima di effettuare la modifica, l'API controlla che l'utente sia il creatore del gruppo e che la dimensione del gruppo rispetti il limite imposto.

```
router.put( path: '/group', tokenChecker, groupController.editGroup );
```

```
const editGroup=(req,res)=>{
  let userInfo=req.userInfo;
  let id=req.body.id;
  let description=req.body.description;
  let size=req.body.size;
  let master=userInfo.id;
  Group.findById(id,(err,data)=> {
    if(err) return res.status(500).json({message:'Something went wrong',status:500});
    else if(size > 5) return res.status(400).json({message: 'Size too big', status: 400});
    else if(data.master !== master) return res.status(403).json({message:'User is not master',status:403});
    else {
      Group.findByIdAndUpdate(id, {
        description: description,
        size: size
      }, (err) => {
        if (err) return res.status(500).json({message: 'Something went wrong', status: 500});
        return res.status(200).json({message: 'Updated', status: 200});
      })
    }
  })
}
```

2.5.5. Richiesta di unione a un gruppo

Questa API viene utilizzata da un utente quando vuole chiedere di unirsi a un gruppo. Prima di inviare la richiesta, l'API controlla che l'utente non sia già inserito nella lista dei giocatori e che non abbia già inviato una richiesta in attesa di risposta.

```
router.put( path: '/group/request', tokenChecker, groupController.requestJoin);
```

```
const requestJoin=(req,res)=>{
  let userInfo=req.userInfo;
  let player={user:userInfo.id,character:req.body.character};
  let id=req.body.id;
  Character.findById(player.character,(err,data)=> {
    if (err) return res.status(500).json({message: 'Something went wrong', status: 500});
    else if (!data) return res.status(404).json({message: 'Group does not exist', status: 404});
    else if(data.user !== player.user) return res.status(400).json({message:'Character not in user list',status:400});
    else {
      Group.findById(id, (err, data) => {
        if (err) return res.status(500).json({message: 'Something went wrong', status: 500});
        else if (!data) return res.status(404).json({message: 'Group does not exist', status: 404});
        else if (data.master === player.user) return res.status(400).json({
          message: 'User is master',
          status: 400
        });
        else {
          let requestArray = data.requests;
          let charactersArray = data.characters;
          if (_.some(charactersArray, predicate: {user: player.user}) || _.some(requestArray, predicate: {user: player.user}))
            return res.status(400).json({message: "Cannot send more than one request"});
          //group is full
          else if (charactersArray.length === data.size) return res.status(400).json({
            message: "Group is full",
            status: 400
          });
          else {
            requestArray.push(player);
            Group.findByIdAndUpdate(id, {
              requests: requestArray
            }, (err) => {
              if (err) return res.status(500).json({message: 'Something went wrong', status: 500});
              else return res.status(200).json({message: 'Request sent', status: 200});
            })
          }
        }
      })
    }
  })
}
```

2.5.6. Rifiuto richiesta di unione

Questa API viene utilizzata quando il creatore di un gruppo rifiuta una richiesta di unione al proprio gruppo. Quando richiamata, l'API rimuove la richiesta dall'array di richieste in base al suo ID, controllando prima che la chiamata sia stata effettuata dal creatore del gruppo.

```
router.put( path: '/group/decline', tokenChecker, groupController.declinePlayer);
```

```
const declinePlayer=(req,res)=>{
  let id=req.body.id;
  let requestID=req.body.request;
  let userInfo=req.userInfo;
  let master=userInfo.id;
  Group.findById(id,(err,data)=>{
    if(err) return res.status(500).json({message:'Something went wrong',status:500});
    else if(!data) return res.status(404).json({message:'Group does not exist',status:404});
    else if(data.master!==master) return res.status(403).json({message:'Not master',status:403});
    else{
      let requestsArray=data.requests;
      if(_.some(requestsArray, predicate: {_id:ObjectId(requestID)})) {
        _.remove(requestsArray, predicate: {_id: ObjectId(requestID)});
        Group.findByIdAndUpdate(id, {
          requests: requestsArray
        }, (err) => {
          if (err) return res.status(500).json({message: 'Something went wrong', status: 500});
          else return res.status(200).json({message: 'Player declined', status: 200});
        })
      }
      else return res.status(400).json({message:'Request not in list',status:400});
    }
  })
}
```

2.5.7. Accettazione richiesta di unione

Questa API viene utilizzata quando il creatore di un gruppo accetta una richiesta di unione al proprio gruppo. Quando richiamata, l'API rimuove la richiesta dall'array di richieste in base al suo ID, controllando prima che la chiamata sia stata effettuata dal creatore del gruppo, e aggiunge il giocatore all'array di giocatori.

```
router.put( path: '/group/accept', tokenChecker, groupController.addPlayer);
```

```
const addPlayer=(req,res)=>{
  let id=req.body.id;
  let player={user:req.body.user,character:req.body.character};
  let userInfo=req.userInfo;
  let master=userInfo.id;
  let requestID=req.body.request;

  Group.findById(id,async (err, data) => {
    if (err) return res.status(500).json({message: 'Something went wrong', status: 500});
    else if (!data) return res.status(404).json({message: 'Group does not exist', status: 404});
    else if (data.master !== master) return res.status(403).send();
    else {
      let charactersArray = data.characters;
      let requestsArray = data.requests;
      if (charactersArray.length === data.size)
        return res.status(400).json({message: 'Group is full', status: 400});
      else if (_.some(charactersArray, predicate: {user:player.user}))
        return res.status(400).json({message: 'User is already in group', status: 400});
      else if (!_.some(requestsArray, predicate: {_id: ObjectId(requestID)}))
        return res.status(400).json({message: 'Request not in list', status: 400});

      else {
        if (await Character.exists({_id: player.character}).exec()) {
          charactersArray.push(player);
          _.remove(requestsArray, predicate: {_id: ObjectId(requestID)});
          Group.findByIdAndUpdate(id, {
            characters: charactersArray,
            requests: requestsArray
          }, (err) => {
            if (err) return res.status(500).json({message: 'Something went wrong', status: 500});
            else return res.status(200).json({message: 'Character added', status: 200});
          })
        } else return res.status(400).json({message: 'User or Character do not exist', status: 400});
      }
    }
  })
}
```

2.5.8. Rimozione giocatore

Questa API viene utilizzata quando il creatore di un gruppo vuole rimuovere un giocatore

presente nella lista giocatori. Quando richiamata, l'API rimuove il giocatore dall'array di giocatori in base al suo ID, controllando prima che la chiamata sia stata effettuata dal creatore del gruppo.

```
router.put( path: '/group/remove', tokenChecker, groupController.removePlayer);
```

```
const removePlayer=(req,res)=>{
  let id=req.body.id;
  let playerId=req.body.playerid;
  let userInfo=req.userInfo;
  let master=userInfo.id;
  Group.findById(id,(err,data)=>{
    if(err) return res.status(500).json({message:'Something went wrong',status:500});
    else if(!data) return res.status(404).json({message:'Group does not exist',status:404});
    else if(data.master!==master) return res.status(403).send();
    else{
      let charactersArray=data.characters;
      if(_.some(charactersArray, predicate: {_id:ObjectId(playerID)})) {
        _.remove(charactersArray, predicate: {_id: ObjectId(playerID)});
        Group.findByIdAndUpdate(id, {
          characters: charactersArray
        }, (err) => {
          if (err) return res.status(500).json({message: 'Something went wrong', status: 500});
          else return res.status(200).json({message: 'Player removed', status: 200});
        })
      }
      else return res.status(400).json({message:'Player not in list',status:400});
    }
  })
}
```

2.5.9. Invio di un messaggio in chat

Questa API viene utilizzata quando un giocatore vuole inviare un messaggio nella chat del

gruppo. Prima di inviare il messaggio, l'API controlla che l'utente faccia parte del gruppo.

```
router.put( path: '/group/chat', tokenChecker, groupController.newMessage );
```

```
const newMessage=(req,res)=>{
  let id=req.body.id;
  let userInfo=req.userInfo;
  let user=userInfo.id;
  let username=userInfo.username;
  let isMaster=false;
  let message=req.body.message;
  Group.findById(id,(err,data)=>{
    if(err) return res.status(500).json({message:'Something went wrong',status:500});
    else if(!data) return res.status(404).json({message:'Group does not exist',status:404});
    else {
      let charactersArray=data.characters;
      let messages=data.messages
      //if user is not in group
      if(user !== data.master && !_some(charactersArray, predicate: {user:user})){
        return res.status(403).json({message:'User in not in group',status:403});
      }
      else {
        if (user === data.master) isMaster = true;
        messages.push({username: username, message: message, isMaster: isMaster});
        Group.findByIdAndUpdate(id, {
          messages: messages
        }, (err) => {
          if (err) return res.status(500).json({message: 'Something went wrong', status: 500});
          return res.status(201).json({message: 'Message sent', status: 201})
        })
      }
    }
  })
}
```

2.5.10. Recupero dei messaggi della chat

Questa API viene utilizzata quando un giocatore vuole leggere i messaggi della chat di gruppo.

Prima di mostrare i messaggi, l'API controlla che l'utente faccia parte del gruppo.

```
router.get( path: '/group/chat', tokenChecker, groupController.getMessage);
```

```
const getMessage=(req,res)=>{
  let id=req.query.id;
  let userInfo=req.userInfo;
  let user=userInfo.id;
  Group.findById(id,(err,data)=>{
    if(err) return res.status(500).json({message:'Something went wrong',status:500});
    else if(!data) return res.status(404).json({message:'Group does not exist',status:404});
    else{
      let messages=data.messages;
      let charactersArray=data.characters;
      if(!_some(charactersArray, predicate: {user:user})) return res.status(403).json({message:'User is not in group',status:403});
      else return res.status(200).json(messages);
    }
  })
}
```

2.5.11.Lancio di un dado

Questa API viene utilizzata quando un giocatore vuole simulare il lancio di un dado e mostrare il risultato nella chat di gruppo. L'API fa una chiamata all'API dice, che simula il lancio del dado e restituisce il risultato. Una volta ottenuto il risultato, l'API richiama l'API per l'invio del

messaggio, passando come parametri l'id del gruppo e il risultato del lancio come corpo del messaggio.

```
router.put( path: '/group/chat/roll', tokenChecker, groupController.rollDice);
```

```
const rollDice=(req,res)=>{
  let numDadi=req.query.num;
  let dado=req.query.type;
  let id=req.body.id;
  let url='http://localhost:8080/dice?num='+numDadi+'&type='+dado;
  axios.get(url)
    .then((response : AxiosResponse<any> )=>{
      let message=response.data.result;
      url='http://localhost:8080/group/chat';
      axios.put(url, data: {id:id,message:message}, config: {headers: {cookie: req.headers.cookie}})
        .then((response : AxiosResponse<any> )=>{
          return res.status(201).send();
        }) Promise<any>
        .catch((error)=>{
          return res.status(500).json({Error:error});
        })
    })
    .catch((error)=>{
      return res.status(500).json({Error:error,status:500});
    })
}
```

2.5.12.Simulatore per lancio di dadi

Questa API viene utilizzata quando si vuole simulare il lancio di un dado. L'API prende come parametri il numero di dadi da lanciare e il tipo. Nel caso in cui si voglia simulare il lancio di dadi a 20 facce il risultato sarà una stringa contenente il risultato di ogni lancio, altrimenti il

risultato sarà il totale ottenuto dai vari lanci. Prima di generare il risultato, l'API controlla che i parametri rispettino i limiti imposti (il numero di dadi deve essere inferiore a 100, il dado può avere 4,6,8 o 20 facce).

```
router.get( path: '/dice', diceController.throwDice);
```

```
const throwDice=(req,res)=>{
  let numDadi=parseInt(req.query.num);
  let dado=parseInt(req.query.type);
  let result=0;
  let dadi=[4,6,8,20];
  if(!dadi.includes(dado) || numDadi <= 0 || numDadi > 100){
    return res.status(400).json({message:'Bad request',status:400});
  }
  else{
    if(dado !== 20){
      for(let i=0;i<numDadi;i++){
        result += (1+Math.floor( x: Math.random()*dado));
      }
      return res.status(200).json({result:result,status:200});
    }
    else{
      let string = "";
      for(let i=0;i<numDadi;i++){
        let calc=(1+Math.floor( x: Math.random()*dado))
        result += calc;
        string += calc.toString()+" ";
      }
      string += "Total: "+result.toString();
      return res.status(200).json({result:string,status:200});
    }
  }
}
```

2.5.13.Registrazione

Questa API viene utilizzata quando un utente vuole creare un nuovo account. L'API prima genera l'hash della password utilizzando un metodo del modulo **bcrypt**, successivamente,

dopo aver inserito il nuovo utente all'interno del database, genera il token di autenticazione **jwt** e lo salva in un cookie. Nel caso in cui l'email o lo username siano già in uso l'API restituirà un messaggio di errore.

```
router.post( path: '/user/register',UserController.newUser);
```

```
const newUser = async(req,res) => {
  let hashedPassword = bcrypt.hashSync(req.body.password, bcrypt.genSaltSync( rounds: 10));
  const newUser=new User( doc: {
    username:req.body.username,
    email:req.body.email,
    password:hashedPassword,
    image:req.body.image,
    description:req.body.description,
    reputation:0,
    isVerified:false,
    upvotes:[],
    downvotes:[]
  })
  newUser.save((err: CallbackError, data :any)=>{
    if(err){
      if(err.code!==11000)return res.status(500).json({Error:err,status:500});
      else return res.status(400).json({message:"Username and/or email is already registered",status:400});
    }
    else{
      let payload = {username: data.username, id: data._id};
      let options = {expiresIn: 21600};
      let token = jwt.sign(payload, process.env.SUPER_SECRET, options);
      let save = {
        success: true,
        message: "validation token",
        token: token,
        username: data.username,
        id: data._id,
      }
      return res.cookie('tk',save).status(201).send();
    }
  })
}
```

2.5.14.Login

Questa API viene utilizzata quando un utente vuole fare il login nel proprio account. Dopo aver controllato che nel database esista l'email(o username) e la password specificati, l'API salva il

token di autenticazione in un cookie.

```
router.post( path: '/user/login', userController.login);

const login = (req, res) => {
  User.findOne({$or:[{email:req.body.input},{username:req.body.input}]}, (err, data)=>{
    if(err) return res.status(500).json({message:"Unexpected error",status:500});
    else if(!data) return res.status(400).json({message:"Username or password is wrong",status:400});
    else {
      bcrypt.compare(req.body.password, data.password, cb: (err, result) => {
        if (err) {
          return res.status(500).json({message:"Unexpected error", status: 500});
        } else if (result) {
          let payload = {username: data.username, id: data._id};
          let options = {expiresIn: 21600};
          let token = jwt.sign(payload, process.env.SUPER_SECRET, options);
          let save = {
            success: true,
            message: "validation token",
            token: token
          }
          return res.cookie('tk', save).status(200).send();
        } else {
          return res.status(400).json({Result:result,message: "Username or password is wrong", status: 400});
        }
      })
    }
  })
}
```

2.5.15. Visualizzazione profilo utente

Questa API viene utilizzata quando un utente vuole visualizzare il proprio profilo o quello di un altro utente. Se viene passato l'id l'API restituirà solo le informazioni relative all'utente

specificato, altrimenti restituirà un elenco di tutti gli utenti.

```
router.get( path: '/user',UserController.getUser);

const getUser=(req,res)=>{
  let id=req.query.id;
  if(!id)
    User.find({},'username image description reputation',(err,data)=>{
      if(err) return res.status(500).json({message:'Unexpected error',status:500});
      else return res.status(200).json(data);
    })
  else
    User.findById(id,'username image description reputation',(err,data)=>{
      if(err) return res.status(500).json({message:'Unexpected error',status:500});
      else if(!data) return res.status(404).json({message:'User does not exist',status:404});
      else return res.status(200).json(data);
    })
}
```

2.5.16.Modifica profilo utente

Questa API viene utilizzata quando un utente vuole modificare il proprio username, immagine o descrizione. Se l'id dell'utente che ha effettuato la chiamata corrisponde con l'id dell'utente di cui si vuole modificare il profilo allora l'API modificherà le informazioni nel database con quelle

passate dall'utente nella chiamata. Altrimenti ritornerà errore.

```
router.put( path: '/user', tokenChecker, userController.editUser);
```

```
const editUser=(req,res)=>{
  let id=req.body.id;
  let userInfo = req.userInfo;
  let myId=userInfo.id;
  if(id === myId){
    let username=req.body.username;
    let image=req.body.image;
    let description=req.body.description;
    User.findByIdAndUpdate(myId,{
      username:username,
      description:description,
      image:image
    },(err)=>{
      if(err) return res.status(500).json({message:'Unexpected error',status:500});
      else return res.status(200).json({message:'Updated',status:200});
    })
  }
  else return res.status(403).json({message:'User id does not match',status:403});
}
```

2.5.17. Like a un profilo utente

Questa API viene utilizzata quando un utente vuole mettere Like al profilo di un altro utente, modificando la reputazione. Dato che un utente non può mettere Like al suo profilo l'API farà prima un controllo sull'id dell'utente. Successivamente l'API controllerà che l'utente non abbia già messo Like al profilo. Una volta passati i controlli, l'API incrementa di 1 la reputazione, e

aggiunge l'id dell'utente all'array upvotes(così da poter tenere traccia di chi abbia già messo Like) e rimuove, se esiste, l'id dall'array downvotes(dato che il Like a un profilo esclude il Dislike).

```
router.put( path: '/user/rep/up', tokenChecker, userController.upVote);
```

```
const upVote=(req,res)=>{
  let id=req.body.id;
  let userInfo = req.userInfo;
  let myId=userInfo.id;
  if(myId !== id) {
    User.findById(id,(err, data) => {
      if (err) return res.status(500).json({message: 'Unexpected error', status: 500});
      else if (data.upvotes.includes(myId)) return res.status(400).json({
        message: 'Cannot upvote twice',
        status: 400
      });
      else{
        User.findByIdAndUpdate(id,{
          $inc:{reputation:1},
          $push:{upvotes:myId},
          $pull:{downvotes:myId}
        },(err)=>{
          if (err) return res.status(500).json({message: 'Unexpected error', status: 500});
          else return res.status(200).json({message:'User upvoted',status:200});
        })
      }
    })
  }
  else return res.status(400).json({message:'Bad request',status:400});
}
```

2.5.18.Dislike a un profilo utente

Questa API viene utilizzata quando un utente vuole mettere Dislike al profilo di un altro utente, modificando la reputazione. Dato che un utente non può mettere Dislike al suo profilo l'API farà prima un controllo sull'id dell'utente. Successivamente l'API controllerà che l'utente non abbia

già messo Dislike al profilo. Una volta passati i controlli, l'API decrementa di 1 la reputazione, e aggiunge l'id dell'utente all'array downvotes(così da poter tenere traccia di chi abbia già messo Dislike) e rimuove, se esiste, l'id dall'array upvotes.

```
router.put( path: '/user/rep/down', tokenChecker, userController.downVote);
```

```
const downVote=(req,res)=>{
  let id=req.body.id;
  let userInfo = req.userInfo;
  let myId=userInfo.id;
  if(myId !== id) {
    User.findById(id,(err, data) => {
      if (err) return res.status(500).json({message: 'Unexpected error', status: 500});
      else if (data.downvotes.includes(myId)) return res.status(400).json({
        message: 'Cannot downvote twice',
        status: 400
      });
      else{
        User.findByIdAndUpdate(id,{
          $inc:{reputation:-1},
          $push:{downvotes:myId},
          $pull:{upvotes:myId}
        },(err)=>{
          if (err) return res.status(500).json({message: 'Unexpected error', status: 500});
          else return res.status(200).json({message:'User downvoted',status:200});
        })
      }
    })
  }
  else return res.status(400).json({message:'Bad request',status:400});
}
```

2.5.19.Creazione di un personaggio

Questa API viene utilizzata quando un utente vuole creare un nuovo personaggio. L'API usa le informazioni passate dall'utente per creare un nuovo oggetto Character che verrà poi salvato nel database.

```
router.post( path: '/character', tokenChecker, characterController.newCharacter);
```

```
const newCharacter = (req, res) => {  
  let stats = req.body.stats;  
  let userInfo = req.userInfo;  
  let user = userInfo.id;  
  if (stats.length === 4) {  
    const newCharacter = new Character( doc: {  
      name: req.body.name,  
      user: user,  
      image: req.body.image,  
      class: req.body.class,  
      stats: stats,  
      inventory: [],  
    })  
    newCharacter.save((err : CallbackError, data : ... ) => {  
      if (err) return res.status(500).json({Error: err, status: 500});  
      return res.status(201).json({data: data, status: 201});  
    })  
  }  
  else return res.status(400).json({message: "Stats array size is too big", status: 400});  
};
```

2.5.20. Visualizzazione di un personaggio

Questa API viene utilizzata quando un utente vuole visualizzare i personaggi. Nel caso in cui venisse passato l'id, l'API restituirà tutte le informazioni relative a quel personaggio, altrimenti l'API restituirà tutti i personaggi dell'utente.


```
router.get( path: '/character', tokenChecker, characterController.getCharacters);
```

```
const getCharacters=(req,res)=>{  
  let userInfo=req.userInfo;  
  let user=userInfo.id;  
  let id=req.query.id;  
  if(!id)  
    Character.find({user: user}, (err, data) => {  
      if (err) return res.status(500).json({Error: err, status: 500});  
      else return res.status(200).json({data: data, status: 200});  
    })  
  else  
    Character.findById(id, (err, data)=>{  
      if (err) return res.status(500).json({Error: err, status: 500});  
      else return res.status(200).json({data: data, status: 200});  
    })  
}
```

2.5.21.Modifica di un personaggio

Questa API viene utilizzata quando un utente vuole modificare un personaggio. L'API prima controlla che l'utente sia il creatore del personaggio. Successivamente modifica le informazioni nel database con le nuove informazioni passate dall'utente.

```
router.put( path: '/character', tokenChecker, characterController.editCharacter);
```

```
const editCharacter=(req,res)=>{
  let userInfo=req.userInfo;
  let user=userInfo.id;
  let id=req.body.id;
  Character.findById(id,(err,data)=>{
    if(err) return res.status(500).json({message:'Something went wrong', status:500});
    else if(!data) return res.status(404).json({message:'Character does not exist',status:404});
    else if(data.user!==user) return res.status(403).send();
    else
      Character.findByIdAndUpdate(id,{
        name:req.body.name,
        image:req.body.image,
        class:req.body.class,
        stats:req.body.stats,
        inventory:req.body.inventory,
      },(err)=>{
        if(err) return res.status(500).json({message:'Something went wrong', status:500});
        return res.status(200).json({message:'Updated',status:200});
      })
  })
}
```

2.5.22. Eliminazione di un personaggio

Questa API viene utilizzata quando un utente vuole eliminare un suo personaggio. L'API controlla che l'utente sia il creatore del personaggio, e successivamente lo elimina dal database.

```
router.delete( path: '/character',tokenChecker,characterController.deleteOneCharacter);
```

```
const deleteOneCharacter=(req,res)=>{
  let id=req.body.id;
  let userInfo=req.userInfo;
  let user=userInfo.id;
  Character.findById(id,(err,data)=>{
    if(err) return res.status(500).json({message:"Unexpected error",status:500});
    else if(!data) return res.status(404).json({message:"Character does not exist",status:404});
    else if(data.user!==user) return res.status(403).send();
    else
      Character.findByIdAndDelete(id, options: {err}=>{
        if(err) return res.status(500).json({message:"Unexpected error",status:500});
        else return res.status(204).send();
      })
  })
}
```

3. API documentation

Le API Locali fornite dall'applicazione PartFinder descritte nella sezione precedente sono state documentate utilizzando il modulo NodeJS chiamato **Swagger UI Express** e **Swagger-Autogen**. In questo modo la documentazione relativa alle API è direttamente disponibile a chiunque veda il codice sorgente. Per generare la prima versione del file swagger.json è stato utilizzato il modulo Swagger-Autogen.

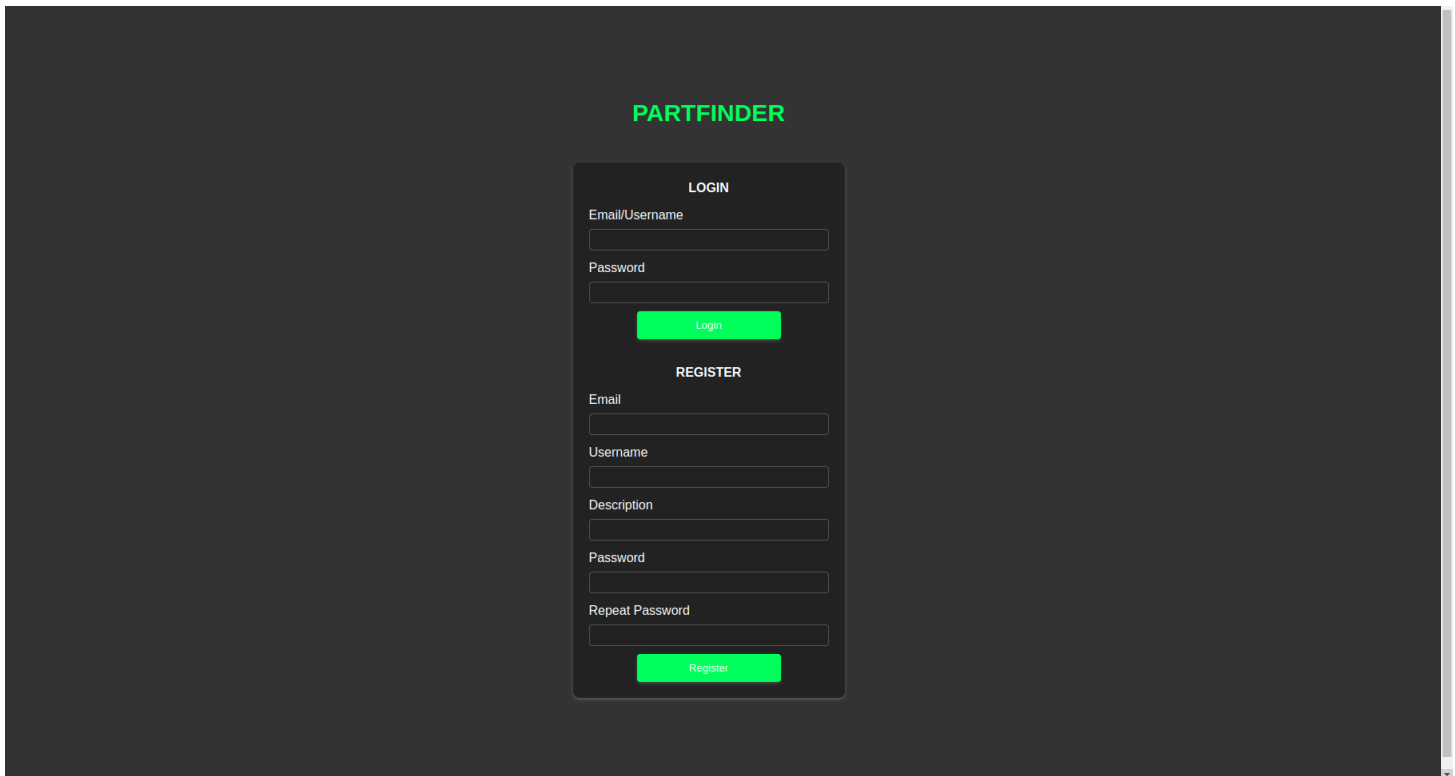
L'endpoint da invocare per raggiungere la seguente documentazione è:
<http://localhost:8080/api-docs>



Fig.6: Documentazione delle API di PartFinder.

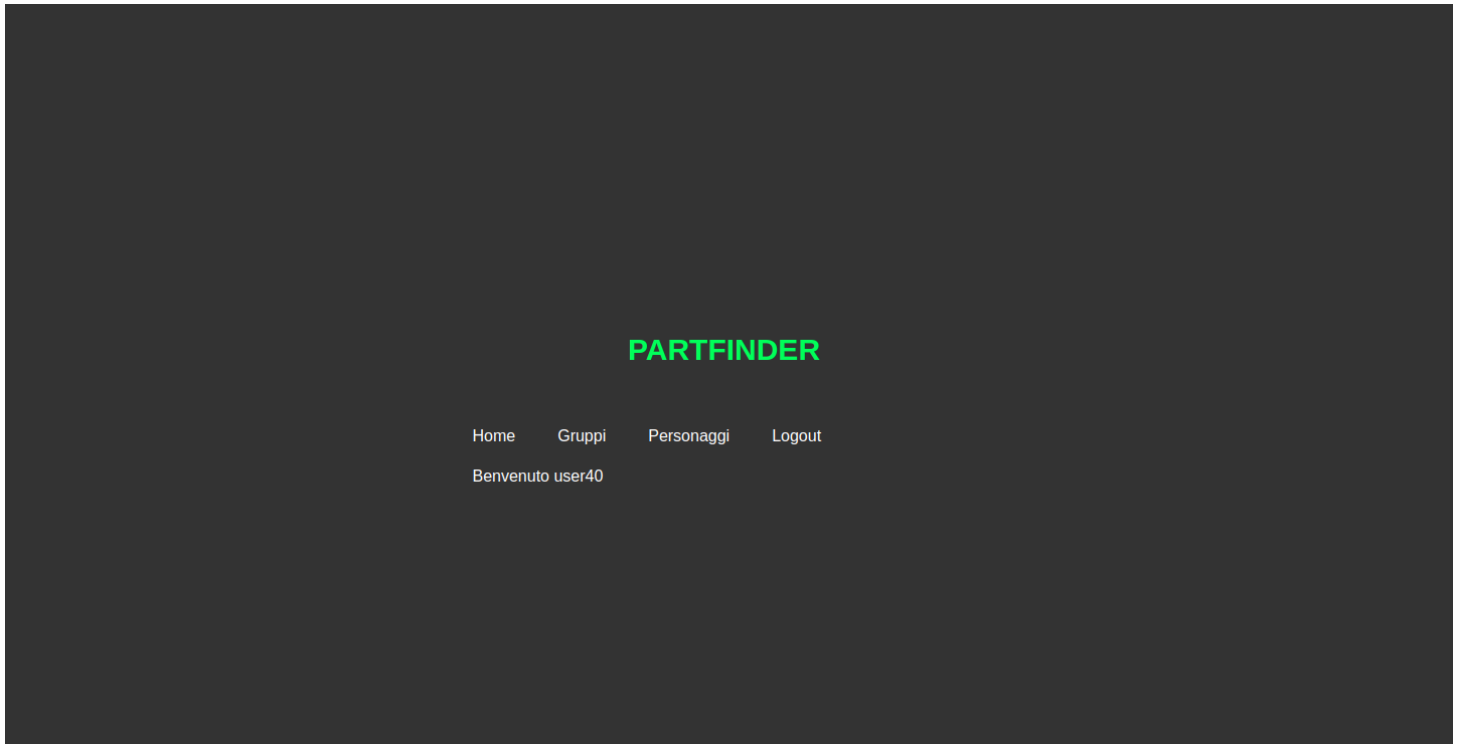
4. FrontEnd Implementation

Il FrontEnd fornisce le funzionalità di visualizzazione, inserimento e cancellazione dei dati dell'applicazione PartFinder. In particolare, l'applicazione è composta da una Homepage, da una sezione per login e registrazione, da una sezione per visualizzazione dei personaggi e da una sezione per visualizzazione dei gruppi.

The image shows a dark-themed web interface for 'PARTFINDER'. At the top center, the word 'PARTFINDER' is written in a bright green, sans-serif font. Below it, there are two distinct sections: 'LOGIN' and 'REGISTER'. The 'LOGIN' section includes a label 'Email/Username' above a text input field, a label 'Password' above another text input field, and a green button labeled 'Login'. The 'REGISTER' section includes a label 'Email' above a text input field, a label 'Username' above a text input field, a label 'Description' above a text input field, a label 'Password' above a text input field, and a label 'Repeat Password' above a text input field. At the bottom of the registration section is a green button labeled 'Register'. The entire form is centered on a dark gray background.

Sezione login dell'applicazione.

Nella sezione login l'utente può effettuare il login nel suo account, oppure registrarsi all'applicazione.



Homepage

Dalla Homepage l'utente registrato può navigare tra le varie sezioni dell'applicazione tramite i pulsanti a schermo

CreaCercaMostra TuttiGruppi Personali

Nome	Descrizione	Codice	Master	Player1	Player2	Player3	Player4	Player5	Richiesta
testingRequests_delete	testing	MOFJL	user6	Prenotato	Prenotato	X	X	X	INVIA RICHIESTA Seleziona personaggio per richiesta ▼
asd	asd	SJHXS	Ciaone1234!	X	X	X	X	X	INVIA RICHIESTA Seleziona personaggio per richiesta ▼
asd	asd	MNJRF	Ciaone1234!	X	X	X	X	X	INVIA RICHIESTA Seleziona personaggio per richiesta ▼
CiaoneGroup	CiaoneDesc	VDFRH	Ciaone1234!	X	X	X	X	X	INVIA RICHIESTA Seleziona personaggio per richiesta ▼
gruppo1	descrizione gruppo 1 di prova :))	GLOCW	user40	X	X	X	X	X	RICHIESTA NON DISPONIBILE

PARTFINDER

HomeGruppiPersonaggiLogout

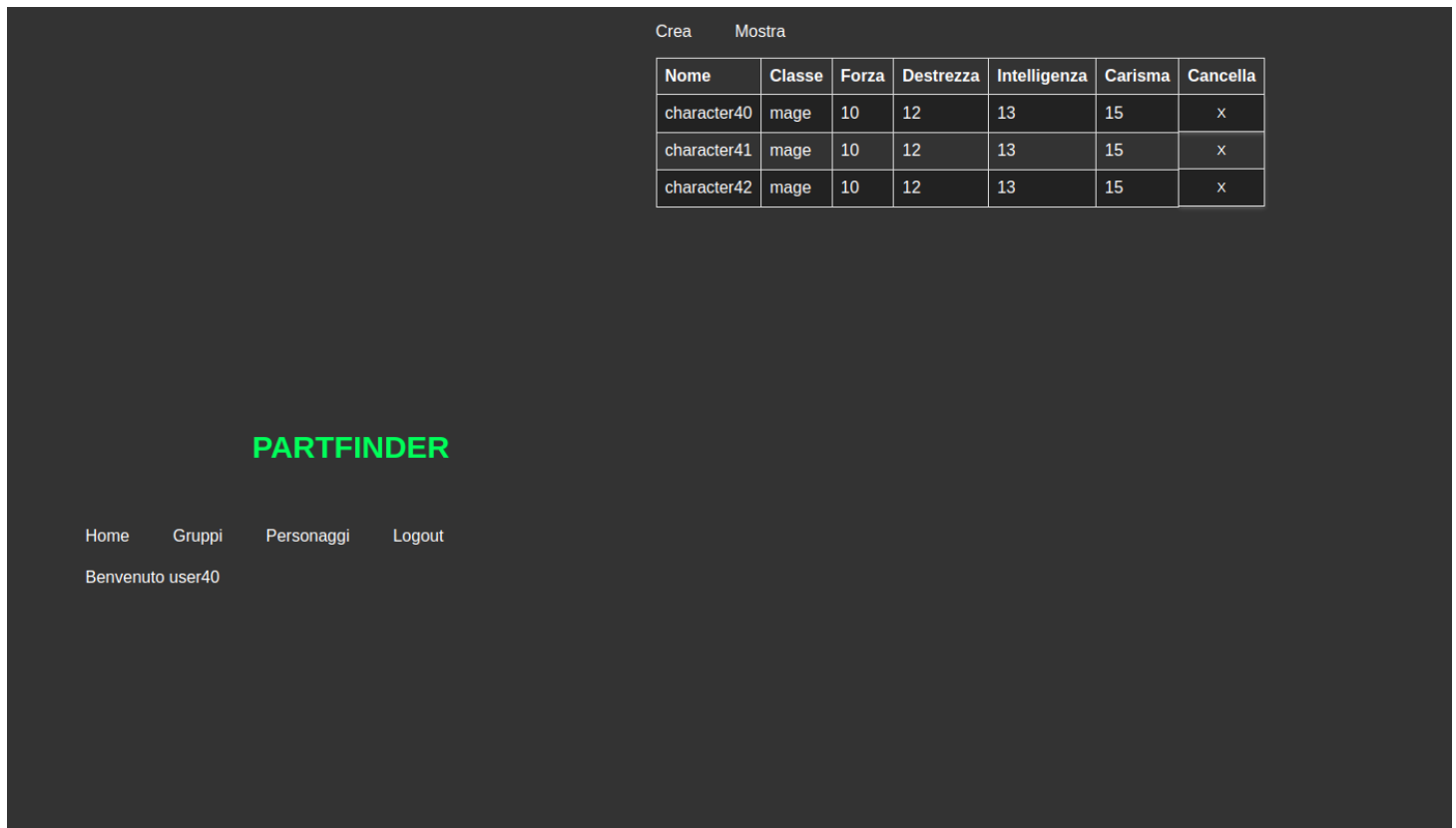
Benvenuto user40

Sezione Gruppi

Cliccando sul pulsante 'Gruppi', l'utente può navigare tra le diverse sottosezioni.

In particolare:

- Nella sottosezione 'Crea' l'utente può creare un nuovo gruppo specificando nome, descrizione e limite di giocatori.
- Nella sottosezione 'Cerca' l'utente può inserire il codice univoco del gruppo che intende cercare. Se il gruppo viene trovato allora può chiedere di unirsi.
- Nella sottosezione 'Mostra tutti' l'utente può visualizzare tutti i gruppi presenti nell'applicazione e può chiedere di unirsi ad essi.
- Nella sezione 'Gruppi Personali' l'utente può visualizzare tutti i gruppi che ha creato o ai quali partecipa.



Sezione Personaggi

Cliccando sul pulsante 'Personaggi', l'utente può navigare tra le diverse sottosezioni.

In particolare:

- Nella sottosezione 'Crea' l'utente può creare un nuovo personaggio specificando nome, classe e i valori delle sue caratteristiche.
- Nella sottosezione 'Mostra' l'utente può visualizzare tutti i suoi personaggi che ha creato. Se l'utente desidera, può cancellare un personaggio dalla lista.

5. Deployment Info

La repository di Github è strutturata in modo da avere il BackEnd nella cartella "." e il FrontEnd nella cartella "./static". All'interno della repo è anche presente un README che spiega come far partire il sito in locale

<< Info sul deployment e sul link da utilizzare per eseguire l'applicazione su Heroku >>

6. Testing

Per effettuare il testing delle API durante in fase di sviluppo è stato utilizzato il software Postman.

In seguito per il testing di tutte le API è stato usato il modulo Jest e SuperTest.

I casi di test con relativo esito sono rappresentati in figura:

server.test.js: 64 total, 64 passed			5.38 s
			Collapse Expand
server.test.js			5.38 s
■ User API test			3.07 s
tests POST request /user/register	passed		1.94 s
tests POST request /user/register with existing username	passed		189 ms
tests POST request /user/register with existing email	passed		354 ms
tests POST request /user/login	passed		264 ms
tests POST request /user/login with wrong credentials	passed		133 ms
tests POST request /user/login with non existing user	passed		39 ms
tests GET request /user	passed		51 ms
tests GET request /user?id={userId}	passed		49 ms
tests GET request /user?id={userId} with non existing userId	passed		45 ms
■ Group API test			1.92 s
tests GET request /group	passed		51 ms
tests GET request /group/:user without logging in	passed		5 ms
tests GET request /group/:user with login	passed		51 ms
tests GET request /group/:user with another user id	passed		7 ms
tests GET request /group?code={groupCode}	passed		46 ms
tests GET request /group?code={groupCode} with non existing groupcode	passed		40 ms
tests POST request /group	passed		100 ms
tests POST request /group without logging in	passed		7 ms
tests POST request /group with size bigger than 5	passed		7 ms
tests POST request /group with size smaller than 1	passed		9 ms
tests PUT request /group	passed		108 ms
tests PUT request /group without logging in	passed		6 ms
tests PUT request /group with another user group	passed		42 ms
tests PUT request /group with non existing group	passed		43 ms
tests PUT request /group with invalid size	passed		61 ms

server.test.js: 64 total, 64 passed		5.38 s
		Collapse Expand
tests PUT request /group with invalid size	passed	61 ms
tests DELETE request /group	passed	48 ms
tests DELETE request /group with another user group	passed	41 ms
tests DELETE request /group with non existing group	passed	44 ms
tests PUT request /group/request	passed	46 ms
tests PUT request /group/request with request already in list	passed	77 ms
tests PUT request /group/request with a character not in user list	passed	50 ms
tests PUT request /group/request with non existing group	passed	92 ms
tests PUT request /group/request with user being the master	passed	78 ms
tests PUT request /group/request with group being full	passed	90 ms
tests PUT request /group/accept	passed	49 ms
tests PUT request /group/accept without login in	passed	6 ms
tests PUT request /group/accept with another user group	passed	37 ms
tests PUT request /group/decline	passed	43 ms
tests PUT request /group/decline without login in	passed	4 ms
tests PUT request /group/decline with another user group	passed	36 ms
tests PUT request /group/remove	passed	50 ms
tests PUT request /group/remove with player not in list	passed	46 ms
tests PUT request /group/remove without logging in	passed	5 ms
tests PUT request /group/remove with user not being master	passed	41 ms
tests GET request /group/chat	passed	44 ms
tests GET request /group/chat without logging in	passed	5 ms
tests GET request /group/chat with user not being in group	passed	41 ms
tests PUT request /group/chat	passed	92 ms
tests PUT request /group/chat without logging in	passed	5 ms
tests PUT request /group/chat without user not being in group	passed	40 ms
tests PUT request /group/chat/roll	passed	117 ms
tests PUT request /group/chat/roll with invalid dice type	passed	95 ms
tests PUT request /group/chat/roll with invalid dice number	passed	14 ms
Character API test		389 ms
tests POST request /character	passed	56 ms
tests POST request /character with invalid stats array	passed	8 ms
tests POST request /character without logging in	passed	19 ms
tests GET request /character	passed	47 ms
tests GET request /character?id={characterId}	passed	47 ms
tests GET request /character?id={characterId} with invalid id	passed	39 ms
tests PUT request /character	passed	40 ms
tests PUT request /character with another user character	passed	41 ms
tests PUT request /character without logging in	passed	6 ms
tests DELETE request /character	passed	39 ms
tests DELETE request /character with another user character	passed	41 ms
tests DELETE request /character without logging in	passed	6 ms