# Documentazione

## YuGiOh! Project

YuGiOh! Project è un'applicazione web realizzata per il corso di Tecnologie e Applicazioni dei Sistemi Distribuiti del corso di laurea magistrale Teoria e Tecnologia della comunicazione dell'Università degli Studi di Milano-Bicocca.

Questa web app fornisce all'utente la possibilità di vedere le card della prima edizione di YuGiOh, gioco di carte dell'omonimo anime giapponese uscito nel 2009 in Italia diventato famoso tra i più giovani.

## Indice


1. Struttura
2. App.js
3. Componenti
    1. Header
    2. Footer
    3. MainTemplate
    4. YugiohCard
    5. Display
        1. GridDisplay
        2. ListDisplay
4. Views
    1. YugiohCardDetails
    2. Home
    3. Cards
    4. About
5. Deploy


## 6. Note

## Struttura

Si tratta di un'applicazione che potrebbe interessare ai più nostalgici ma anche alle nuove generazioni che hanno la curiosità di vedere come fossero le carte originali.

L'applicazione è articolata nelle seguenti sezioni:

- La sezione About riporta una breve introduzione sul mondo di YuGiOh.
- La sezione Cards permette all'utente di vedere tutte le carte della prima edizione e permette di filtrarle per deck, ovvero mazzi di carte appartenenti ai personaggi principali della serie: Yugi Muto è il protagonista, Katsuya Jonouchi è il miglior amico di Yugi e Seto Kaiba è l'antagonista. Inoltre, è possibile visualizzarle a griglia oppure a lista. Ogni card se cliccata viene visualizzata nel dettaglio.
- Nella Home vengono visualizzate le tre carte più conosciute di YuGiOh che sono state fondamentali per Yugi Muto.

Il progetto è organizzato in cartelle secondo la seguente alberatura:

- nella cartella `public` si trovano i documenti di default di React in cui è stata sostituita la favicon con quella del progetto
- nella cartella `src` troviamo:
    - la cartella `assets` che contiene le seguenti due cartelle:
        - `data` che contiene un file json creato per estrarre gli id usati durante le chiamate all'API. Inoltre, questi id sono stati suddivisi per personaggio, in modo da poter aggiungere un filtro per personaggio nella pagina `Cards.js`
        - `img` che contiene le immagini statiche utilizzate nel sito come il logo dell'app, le immagini di sfondo e i loghi universitari nel footer.
    - la cartella `components` che contiene le seguenti cartelle (una per ogni componente):
        - `header` contiene il componente header con il relativo foglio di stile
        - `footer` contiene il componente footer con il relativo foglio di stile
        - `display` contiene le cartelle `listDisplay` e `gridDisplay` con le due modalità di visualizzazione differente dei dati (lista e griglia)
        - `mainTemplate` contiene il template di ogni pagina composto da Header e Footer
        - `yugiohCard` contiene il componente che visualizza la singola carta in uno dei due display (grid o list) e il relativo foglio di stile.
    - la cartella `views` contiene le viste di navigazione dell'applicazione:
        - `home`, `cards`, `about` sono le principali sezioni presenti nella navbar
        - `yugiohCardDetails` contiene la view per visualizzare i dettagli della singola carta con il rispettivo foglio di stile
    - i file `App.js` e `index.js` con i relativi fogli di stile e il file `reportWebVitals.js`

## App.js

Questo file contiene la funzione `App` che è l'entry point di React-app.

In questa funzione si settano 3 costanti contenenti i dati presenti nel footer (specifici per questo progetto) e vengono dichiarate le routes ovvero i link per navigare la web app sia nell'header che nel footer definendo una struttura ad array di oggetti.

```
const pageListItem = [
    {
        text: "Home",
        url: "/"
    },
    {
        text: "Cards",
        url: "/cards"
    },
    {
        text: "About",
        url: "/about"
    }
];
```

Viene dichiarata una costante che servirà per richiamare l'API di YuGiOh! e ottenere le carte.

```
const apiUrl = 'https://db.ygoprodeck.com/api/v7/cardinfo.php';
```

Le istruzioni per utilizzare questa API si possono trovare qui: Guida API YuGiOh

Nel `return` si utilizzano i componenti della libreria `react-router-dom` (`BrowserRouter, Route, Routes`) per far funzionare il meccanismo di routing, ossia la navigazione interna dell'applicazione attraverso le view. Inoltre, si definisce l'ordine degli elementi. Verrà renderizzato sempre il componente `mainTemplate` con il `child Route` corrispondente alla `view` corrente.

```
 return (
   <BrowserRouter>
       <MainTemplate
           lastLineText={lastLineText}
           first_logo_url={first_logo_url}
           second_logo_url={second_logo_url}
           pageListItem={pageListItem}
       >
           <Routes>
               <Route exact path="/" element={<Home apiUrl={apiUrl}/>}/>
               <Route exact path="/cards" element={<Cards apiUrl={apiUrl}/>}/>
               <Route exact path="/cards/:deck" element={<Cards apiUrl={apiUrl}/>}/>
               <Route exact path="/card/:id" element={<YugiohCardDetails apiUrl={apiUrl}/>}/>
               <Route exact path="/about" element={<About/>}/>
           </Routes>
       </MainTemplate>
   </BrowserRouter>
)
```

---

# Componenti

## Header

Il componente `Header` è un componente statefull che sfrutta l'hook `useState`.

Questo permette al componente di ricordarsi se il menù in modalità mobile è aperto oppure no.

```
 const [isToogleOpen, setIsToogleOpen] = useState(false);
```

```
<NavbarToggler onClick={() => setIsToogleOpen(!isToogleOpen)}/>
<Collapse isOpen={isToogleOpen} navbar>
    <Nav className="ml-auto container-fluid" navbar>
        {listItems}
    </Nav>
</Collapse>
```

Il componente `Header` si aspetta in input una `props`: `pageListItem` che deve essere un array di oggetti del tipo

```
[
    {
        text: "Home",
        url: "/"
    },
    {
        text: "Cards",
        url: "/cards"
    },
    {
        text: "About",
        url: "/about"
    }
]
```

Ogni oggetto dell'array identifica il testo della view da visualizzare e il relativo url. Questo array viene mappato con la seguente funzione anonima

```
const listItems = pageListItem.map((item) => {
    return (
        <NavItem>
            <NavLink to={item.url} className="nav-link header-link">
                {item.text}
            </NavLink>
        </NavItem>
    );
});
```

Questa funzione trasforma ogni singolo oggetto dell'array in un link per il menù dell'header sfruttando le routes.

Nel `return` viene renderizzato il menù sfruttando sia Reacstrap per la UI che le routes per la navigazione.

```
return (
    <>
        <Navbar dark full expand="md" fixed="top">
            <NavbarBrand>
                <NavLink to="/">
                    <img src={logo}/>
                </NavLink>
            </NavbarBrand>
            <NavbarToggler onClick={() => setIsToogleOpen(!isToogleOpen)}/>
            <Collapse isOpen={isToogleOpen} navbar>
                <Nav className="ml-auto container-fluid" navbar>
                    {listItems}
                </Nav>
            </Collapse>
        </Navbar>
        {brSpaces}
    </ >
)
```

---

## Footer

Il componente `Footer` è un componente stateless e vuole 4 `props`:

- `lastLineText` che contiene il testo da stampare in fondo al footer
- `first_logo_url` che contiene l'url del primo logo (quello a sinistra)
- `second_logo_url` che contiene l'url del secondo logo (quello a destra)
- `pageListItem` che contiene lo stesso array di oggetti illustrato nella sezione Header.

Il componente renderizza i png dei loghi dell'Università e del Dipartimento di riferimento. I nomi scelti dei png sono generici per permettere a sviluppatori futuri di cambiare file senza modificare il codice.

```
<div className="col-md-3 offset-md-8 col-sm-12 text-center d-md-flex justify-content-end"
     id="col-logos">
    <a href={first_logo_url}>
        <img src={first_logo}/>
    </a>
    <a href={second_logo_url}>
        <img src={second_logo}/>
    </a>
</div>
```

---

## MainTemplate

Il componente `mainTemplate` è stateless.

Il componente `mainTemplate` renderizza `Header` e `Footer`.

Tra questi due viene inserita la costante `pageBody` che cambia a seconda del valore della `props.children`.

La `props.children` contiene i componenti che si trovano tra il tag di apertura e quello di chiusura di `mainTemplate`.

```
return (
    <>
        <Header
            pageListItem={pageListItem}
        />
        {pageBody}
        <Footer
            lastLineText={lastLineText}
            first_logo_url={first_logo_url}
            second_logo_url={second_logo_url}
            pageListItem={pageListItem}
        />
    </>
);
```

---

## YugiohCard

Il componente `YugiohCard` è un componente statefull che vuole 3 `props`: `displayType`, `cardID`, `apiUrl`.

Questo componente usa l'hook `useEffect` per chiamare l'API tramite una `fetch` che manda una richiesta HTTP al server passandogli come parametro l'url della API con la relativa `props.cardID` che identifica la singola carta richiesta. La `fetch` converte i dati ricevuti in dati json con il metodo `.json()` e configura la risposta sia in caso di successo che di errore, come si può vedere di seguito:

```
useEffect(() => {
    fetch(apiUrl + '?id=' + cardID)
        .then(res => res.json())
        .then(
            (result) => {
                setIsLoaded(true);
                if (result.data === undefined) {
                    setError("the card does not exist");
                } else {
                    let fullCardDetails = result.data[0];
                    setCard(extractCardDetails(fullCardDetails));
                    setCardImage(fullCardDetails.card_images[0].image_url)
                }
            },
            (error) => {
                setIsLoaded(true);
                setError("The card is temporarily unavailable");
            }
        )
});
```

Il rendering di questo componente dipende dal tipo di display impostato.

---

## Display

### GridDisplay

`GridDisplay` è un componente stateless che imposta il layout delle card a griglia.

Questo componente contiene due `props` : `cardsID` e `apiUrl` .

Questo componente contiene la variabile `const arrayToMatrix` che contiene una funzione anonima che prende come parametro gli ID delle carte e trasforma l'array di stringhe in un'array innestato o matrice (composta da `N` array, ognuno di 4 elementi).

```
const arrayToMatrix = function (cardsId) {
    const matrix = [];
    let temp = [];
    for (let cardId of cardsId) {
        temp.push(cardId);
        if (temp.length === 4) {
            matrix.push(temp);
            temp = [];
        }
    }
    //This if is used to deny that, if the array length is shorter than 4, the last elements are lost.
    if (temp.length > 0) {
        matrix.push(temp);
    }
    return matrix;
}
```

Ogni array contenuto nella matrice viene trasformato in una `row` di Reactstap con la funzione `createRows` .

```
const createRows = function () {
    const rows = [];
    for (let array of matrix) {
        const cols = createCols(array);
        rows.push(
            <div className='row mb-5'>
                {cols}
            </div>
        );
    }
    return rows;
}
```

I 4 elementi di ogni singolo array contenuto nella matrice vengono trasformati in una `col` di Reactstrap

```
const createCols = function (array) {
    //
    const cols = [];
    for (let cardID of array) {
        cols.push(
            <div className='col-md col-sm-12'>
                <YugiohCard cardID={cardID} apiUrl={apiUrl} displayType='grid'/>
            </div>
        );
    }
    return cols;

}
```

---

### ListDisplay

`ListDisplay` è un componente stateless che imposta il layout delle card a elenco. Questo componente contiene due `props`: `cardsId`, `apiUrl`.

`ListDisplay` contiene una variabile `createItems` che contiene una funzione anonima che converte gli ID delle carte in un `ListGroupItem` di un `ListGroup` di Reactstrap. Nello specifico, il componente `ListGroupItem` viene renderizzato in `YugiohCard` passando alla `props displayType` il valore `list`, quindi con il layout a elemento di una lista.

```
const createItems = function () {
    const items = [];
    for (let cardID of cardsId) {
        items.push(<ListGroup>
            <YugiohCard cardID={cardID} apiUrl={apiUrl} displayType='list'/>
        </ListGroup>);
    }
    return items;
}
```

---

# Views

## YugiohCardDetails

Nella view `YugiohCardDetails` sono presenti i dettagli della singola carta di YuGiOh!

`YugiohCardDetails` contiene la `props.apiUrl` e contiene l'hook `useParams` per poter recuperare il parametro `id` nella route definita nel componente d'ingresso dell'app `App.js`.

```
<Route exact path="/card/:id" element={<YugiohCardDetails apiUrl={apiUrl}/>}/>
```

```
const cardID = useParams().id;
```

```
fetch(apiUrl + '?id=' + cardID)
```

Nel `return` di questa view vengono renderizzati più campi ritornati dalla API (`type`, `race`, `desc`, `atk`, `def`) rispetto a quelli mostrati dal componente `YugiohCard` per avere una rappresentazione di dettaglio più ricca.

```
return (
    <>
        <div className="container">
            <div className="row m-5">
                <div className="col-md-2 col-sm-12">
                    <Link to="/cards">
                        <Button className="my-btn">Back to Cards</Button>
                    </Link>
                </div>

                <div className="col-md offset-md-4 col-sm-12">
                    <h1>{card.name}</h1>
                </div>
            </div>
            <div className="row">
                <div className="col-md col-sm-12">
                    <div class="flip-card">
                        <div class="flip-card-front">
                            <img src={cardImage}/>
                        </div>
                        <div class="flip-card-back">
                            <img src={cardback}/>
                        </div>
                    </div>
                </div>
                <div className="col-md  col-sm-12">
                    <div className="row">
                        <div className="col-12">
                            <Table borderless dark>
                                <tbody>
                                    <tr>
                                        <th>Type</th>
                                        <td>{card.type}</td>
                                    </tr>
                                    <tr>
                                        <th>Race</th>
                                        <td>{card.race}</td>
                                    </tr>
                                </tbody>
                            </Table>
                        </div>
                        {atkDef}
                        <hr/>
                        <div className="col-12">
                            Description
                            <p>
                                <br/>
                                {card.desc}
                            </p>
                        </div>

                    </div>
                </div>
            </div>
        </div>
    </>
);
```

---

## Home

La view `Home` è stateless e rappresenta la sezione Home della navbar.

La `Home` viene visualizzata, all'interno della web app, come `child` componente `mainTemplate`.

Nel `return` si mostrano 3 carte molto importanti per il protagonista e rappresentative dell'intera serie in display grid.

---

## Cards

La view `Cards` è statefull e rappresenta la sezione Cards della navbar.

`Cards` contiene la `props` : `apiUrl` e contiene l'hook `useParams` per poter recuperare il parametro `deck` (se impostato) nella route definita nel componente d'ingresso dell'app `App.js` . Esistono infatti due route che portano a questa `view` una con il parametro `deck` e una senza.

Nel caso in cui il parametro deck non fosse presente il parametro viene impostato al valore `all` .

```
<Route exact path="/cards" element={<Cards apiUrl={apiUrl}/>}/>
<Route exact path="/cards/:deck" element={<Cards apiUrl={apiUrl}/>}/>
```

Questa view contiene l'hook `useState` per impostare di default la visualizzazione a griglia.

```
const [displayGrid, setDisplayGrid] = useState(true);
```

Questa view contiene due funzioni importanti per il filtraggio delle cards.

In particolare, la funzione `createButtonFilter` gestisce la creazione dei bottoni di filtraggio e la funzione `getDeck` ottiene le carte filtrate (è il reale meccanismo di filtraggio).

`createButtonFilter` crea:

- un bottone generale che rimanda alla view `Cards` con tutte le carte (tramite l'attributo `to="/cards/"` del componente Link).
- bottoni specifici (in base al numero dei proprietari di un deck salvati nel file json) che rimandano alla view `/cards/:deck` a cui si arriva accedendo al valore del campo `owner` di ogni oggetto dell'array all'interno del campo `decks` presente nel file CardsData.json.

```
    const createButtonFilter = function () {
    const allButton = <Link to="/cards">
        <Button className={clsx({active: (deck === 'all'), 'my-btn': true, 'm-1': true})}>
            All
        </Button>
    </Link>;
    let buttons = [allButton];
    for (let deckItem of CardsData.decks) {
        const url = "/cards/" + deckItem.owner
        buttons.push(
            <Link to={url}>
                <Button
                    className={clsx({active: (deck === deckItem.owner), 'my-btn': true, 'm-1': true})}>
                    {deckItem.owner}
                </Button>
            </Link>
        );
    }
    return buttons;
}
```

`getDeck` scorre tutte le cards presenti nel file json `CardsData.json` con un ciclo-for.

Se il valore di `deck` è uguale a `'all'` (e ciò significa che il deckParam dell'url è `undefined` ) allora le carte mostrate sono una concatenazione di tutte le carte di ogni proprietario di un deck.

Altrimenti, se il valore di `deck` è uguale al valore di uno dei campi `owner` allora vengono mostrate solo le carte, quindi il deck, del suo proprietario.

```
const getDeck = function () {
    let cards = [];
    for (let deckItem of CardsData.decks) {
        if (deck === 'all' || deck === deckItem.owner) {
            cards = cards.concat(deckItem.cards)
        }
    }
    return cards
}
```

`changeDisplay` è l'handler dell'evento onclick presente nei due `button` con le classi condizionali/clsx (dentro il `return` ).

La funzione `changeDisplay` cambia il tipo di display secondo l' `id` del `button`.

Se l' `id` del `button` è `isGrid`, imposta l'hook `displayGrid` a `true`. Altrimenti lo imposta a `false`. Questo permette di far funzionare l'effetto toggle dei bottoni sfruttando le classi condizionali e automaticamente cambia anche la tipologia del componente mostrato (GridDisplay o ListDisplay) che viene impostato nella funzione `getCards`.

```
 const changeDisplay = function (event) {
    const id = event.target.id;
    if (id === 'isGrid') {
        setDisplayGrid(true);
    } else {
        setDisplayGrid(false);
    }
};
```

L'effetto toggle è permesso dalla funzione `changeDisplay` che, settando lo stato, aggiunge o toglie la classe `active` al `button` cliccato (di conseguenze se la classe `active` viene aggiunta, al bottone vengono applicati gli stili di quella classe).

```
<div className='col-1'>
    <ButtonGroup>
        <Button onClick={changeDisplay} id="isGrid"
                className={clsx({active: displayGrid === true, 'my-btn': true, 'mt-1': true})}>Grid
        </Button>
        <Button onClick={changeDisplay} id="isList"
                className={clsx({active: displayGrid === false, 'my-btn': true, 'mt-1': true})}>List
        </Button>
    </ButtonGroup>
</div>
```

---

### About

La view `About` è stateless e renderizza dei paragrafi sfruttando la struttura in `rows` e `cols` di Reacstrap

---

## Deploy

La web app è stata pubblicata su GitHub Pages.

---

## NOTE

Purtroppo l'API utilizzata impone un limite di chiamate massime per arco temporale (non conosciuto).

Può, quindi, capitare che le chiamate all'API vengano bloccate.

In questo caso, è stato gestito l'errore mostrando il messaggio `The card is temporaly unavailable`.