Documentazione

Star Wars Characters App

Star Wars Characters App è un'applicazione web responsive sviluppata per il corso di “Tecnologie e Applicazioni dei Sistemi Distribuiti” (2021/2022) del corso di laurea magistrale “Teoria e Tecnologia della Comunicazione” dell'Università degli Studi di Milano-Bicocca.

La web app nasce dalla passione per i film e l’universo di Star Wars, saga cinematografica creata da George Lucas e diventata presto un fenomeno della cultura pop a livello mondiale. Essa è stata sviluppata utilizzando Create React App e fornisce all’utente la possibilità di scoprire ed esplorare tutti i personaggi dei film di Star Wars della triologia originale e della triologia dei prequel.

Indice

1. Struttura della web app

L'applicazione web è strutturata in 3 sezioni principali:

- Homepage. In questa sezione è introdotta brevemente l'app.
- Characters. Questa sezione mostra tutti i personaggi dei film di Star Wars, con la possibilità di visualizzarli in griglia o in tabella, di filtrarli in base alla loro specie e di ordinarli in base al loro id. Inoltre, cliccando sui singoli personaggi è possibile visualizzare la loro specifica scheda.
- Info. In questa sezione è presente una breve descrizione della serie cinematografica.

Il progetto è organizzato in 3 cartelle principali:

- *node_modules* contiene tutti i moduli necessari per il funzionamento della web app, sia quelli di base sia quelli aggiuntivi.
- *public* contiene i file pubblici. In questa cartella è inserita la favicon dell'applicazione e la pagina *index.html*.
- *src* contiene i file sorgente della web app.

In particolare, la cartella *src* è organizzata con la seguente struttura:

- Cartella *assets* che contiene i file statici ed in particolare include:
    - Cartella *data* con il file json *starWarsCharacters.json*. Questo contiene un array di oggetti che rappresentano i personaggi dei film con le relative proprietà: id, name, image e species.
    - Cartella images con le immagini statiche utilizzate nell'applicazione.
- Cartella *components* che contiene una cartella per ciascuno dei componenti utilizzati nella web app ed in particolare:
    - *CardApp* contiene il componente che renderizza le singole card dei personaggi ed il relativo foglio di stile.
    - *CardGridApp* contiene il componente che mostra le card dei personaggi (*CardApp*) nel layout a griglia ed il relativo foglio di stile.
    - *FooterApp* contiene il componente che renderizza il footer dell'app ed il relativo foglio di stile.
    - *NavbarApp* contiene il componente che renderizza il menù dell'app ed il relativo foglio di stile.
    - *TableApp* contiene il componente che mostra i personaggi dei film nel layout a tabella ed il relativo foglio di stile.

    - *TemplateApp* contiene il componente che renderizza il template base delle pagine, costituito dal menù (*NavbarApp*) ed il footer (*FooterApp*), ed il relativo foglio di stile.
- Cartella *utility* che contiene il file *utility.js*. Questo è stato creato per poter definire ed esportare delle funzioni che vengono utilizzate in più componenti diversi. In particolare, contiene una funzione che, in caso di errore nel caricamento di un'immagine, sostituisce il src di quell'immagine con l'immagine statica del logo dell'applicazione.

```
const starWarsDefaultImg = (onErrorEvent) => onErrorEvent.target.src =
StarWarsLogo;
```

- Cartella *views* che contiene le viste di navigazione della web app in cui i diversi componenti vengono assemblati. In particolare:
    - *App* contiene il componente *App* che viene importato all'interno di *index.js* e che definisce il routing.
    - *CharacterDetails* contiene la view che visualizza i dettagli dei singoli personaggi ed il rispettivo foglio di stile.
    - *Characters* contiene la view che visualizza i personaggi dei film nel layout a griglia (*CardGridApp*) o tabella (*TableApp*). In essa sono anche presenti dei pulsanti che permettono di filtrare i personaggi mostrati in base alla loro specie ed ordinarli in modo crescente o decrescente rispetto al loro id.
    - *Home* contiene la view che introduce brevemente l'applicazione ed il rispettivo foglio di stile.
    - *Info* contiene la view che illustra brevemente la serie cinematografica.
    - *NoMatch* contiene la view che viene visualizzata quando l'utente scrive un URL diverso da quelli definiti negli altri componenti Routes in App.
- *index.js* ha lo scopo di renderizzare il componente *App* all'interno del div #root di *index.html*.

Il visual della web app è stato definito utilizzando diverse metodologie:

- Foglio di stile esterno *index.css*, utilizzato per definire delle regole di stile globali e generiche.
- CSS modules specifici per i componenti.
- Bootstrap.
- Reactstrap ovvero una libreria che comprende i componenti di Bootstrap riscritti come componenti React.
- Il pacchetto clsx, utilizzato per gestire lo stile del pulsante di switch per il layout griglia-tabella in *Characters*.

2. View App

App è un componente funzione stateless ed è il componente più generico dell'app. Esso viene importato all'interno di *index.js* ed è dunque l'entry point di React-app.

Il componente *App* ha la funzione principale di definire il routing ovvero la capacità di spostarsi tra le diverse views della web app quando l'utente inserisce un URL o clicca su un elemento (per esempio un link, un pulsante o un'icona), garantendo quindi la navigazione interna dell'applicazione. React non possiede un sistema di routing integrato perciò è stata installata la libreria react-router-dom (v6), specifica per le applicazioni web.

Per definire il routing, nel return del componente *App*, vengono utilizzati i componenti della libreria react-router-dom: *BrowserRouter*, *Routes* e *Route*.

```
function App() {
    const nav = [
        {url: "/starWarsCharacters-webApp", text: "Home", exact: true},
        {url: "/characters", text: "Characters", exact: true},
        {url: "/info", text: "Info", exact: true}
    ];

    return (
      <BrowserRouter>
          <TemplateApp navItems={nav} starWarsLogo={StarWarsLogo}
unimibLogo={UnimibLogo} urlUnimib="https://www.unimib.it/">
              <Routes>
                  <Route path="/starWarsCharacters-webApp" element={<Home />}/>
                  <Route path="/characters" element={<Characters />}/>
                  <Route path="/info" element={<Info />}/>
                  <Route path="/characters/:number" element={<CharacterDetails
/>}/>
                  <Route path="*" element={<NoMatch />} />
              </Routes>
          </TemplateApp>
      </BrowserRouter>
    );
}
```

Inoltre, nel return viene anche importato il componente *TemplateApp* a cui vengono passate come props: il logo dell'app, il logo dell'università, l'url dell'univeristà e la variabile *nav* ovvero un array di oggetti che rappresenta una lista di link con le proprietà url, text ed exact. All'interno di *TemplateApp*, queste props vengono poi passate e utilizzate dai componenti *NavbarApp* e *FooterApp* che in questo modo rimangono generici.

3. Componenti

3.1. TemplateApp

TemplateApp è un componente funzione stateless che renderizza il template base delle pagine.

TemplateApp è costituito dai componenti *NavbarApp* e *FooterApp* che renderizzano il menù ed il footer della web app. Tra di essi, è inserita la prop children che crea uno "spazio" tra il menù ed il footer che viene riempito con il contenuto del tag del componente view che viene renderizzato di volta in volta (*Home*, *Characters*, *Info*, *CharacterDetails* o *NoMatch*).

```
return (
    <div className={style.templateApp}>
        <NavbarApp navItems={navItems} starWarsLogo={starWarsLogo}/>
        <main>
            {children}
        </main>
        <FooterApp navItems={navItems} starWarsLogo={starWarsLogo}
unimibLogo={unimibLogo} urlUnimib={urlUnimib}/>
    </div>
);
```

3.2. NavbarApp

NavbarApp è un componente funzione stateless che renderizza il menù della web app.

NavbarApp riceve la prop *navItems* che consiste in un array di oggetti che rappresenta una lista di link con le proprietà url, text ed exact. L'array viene mappato nel componente *NavbarApp* con la seguente funzione anonima:

```
const itemList = navItems.map((item) => {
    return (
        <NavItem key={item.url} className={style.appNavbarLink}>
            <RouterLink className={style.appNavbarLinkA} exact={item.exact}
to={item.url}>
                {item.text}
            </RouterLink>
        </NavItem>
    );
});
```

Tale funzione trasforma tutti gli oggetti dell'array in voci del menù dell'applicazione, sfruttando anche i componenti di react-router-dom, che vengono salvati nella variabile *itemList*. Questa viene richiamata nel return del componente per renderizzare le voci del menù così ottenute.

```
return (
    <Navbar collapseOnSelect expand="lg" className={style.appNavbar}
variant="dark" >
        <Container>
            <Navbar.Brand>
```

```
                <RouterLink to="/starWarsCharacters-webApp">
                    <img src={starWarsLogo} className="d-inline-block align-top
app-logo" alt="Star Wars logo"/>
                </RouterLink>
            </Navbar.Brand>
            <Navbar.Toggle aria-controls="responsive-navbar-nav" />
            <Navbar.Collapse id="responsive-navbar-nav">
                <Nav className="me-auto">
                    {itemList}
                </Nav>
            </Navbar.Collapse>
        </Container>
    </Navbar>
);
```

3.3. FooterApp

FooterApp è un componente funzione stateless che renderizza il footer della web app.

In modo analogo al componente *NavbarApp*, anche *FooterApp* riceve la prop *navItems* che viene mappata con una funzione anonima al fine di ottenere le voci per la navigazione secondaria nel footer.

```
const itemList = navItems.map((item) => {
        return (
            <li key={item.url} className="nav-item mb-2">
                <NavLink exact={item.exact} to={item.url}>
                    {item.text}
                </NavLink>
            </li>
        );
    });

    return (
        <footer className={style.appFooter}>
            <Container>
                <Row className="align-items-center text-center mt-5 pt-5 mb-5
mb-md-0">
                    <Col md={4} className="mb-4 mb-md-0">
                        <RouterLink to="/starWarsCharacters-webApp">
                            <img src={starWarsLogo} className="d-inline-block
align-top app-logo" alt="Star Wars logo"/>
                        </RouterLink>
                    </Col>
                    <Col md={4} className="mb-4 mb-md-0">
                        <h5 className="mb-3">Explore Star Wars universe</h5>
                        <ul className={style.appFooterNav}>
                            {itemList}
                        </ul>
                    </Col>
                    <Col md={4}>
                        <a href={urlUnimib} target="blank">
                            <img src={unimibLogo} className="app-logo-unimib"
alt="Unimib logo"/>
                        </a>
                    </Col>
                </Row>
                <Row className={style.appFooterCopyright}>
                    <Col className="text-center">
```

```
                        <p className="mb-0">Developed by Daphne Giganti</p>
                        <p className="mt-0">Course of "Tecnologie e Applicazioni
dei Sistemi Distribuiti" 2021/2022, Università degli Studi Milano-Bicocca</p>
                    </Col>
                </Row>
            </Container>
        </footer>
    );
```

3.4. CardGridApp

CardGridApp è un componente funzione stateless che struttura il layout a griglia delle card dei personaggi dei film di Star Wars.

CardGridApp riceve come prop la lista dei personaggi da visualizzare in griglia (*charactersList*) e l'oggetto *col*, utilizzato per indicare il numero di card da visualizzare su ciascuna riga rispetto ai breakpoint di Bootstrap.

La griglia viene strutturata attraverso la seguente funzione anonima:

```
const charactersCardsCol = charactersList.map((character) => {
    return (
        <div key={character.id} className="col">
            <CardApp
                id={character.id}
                image={character.image}
                name={character.name}
                species={character.species}
            />
        </div>
    )
});
```

Tale funzione crea per ciascun oggetto di *charactersList* (ovvero per ciascun personaggio) un componente *CardApp* inserito in un div con classe col. Inoltre, per ciascun personaggio mappato, vengono passate a *CardApp* come props i valori delle proprietà id, images, name e species.

I componenti *CardApp* così creati vengono salvati nella variabile *charactersCardsCol*. Questa viene richiamata nel return di *CardGridApp* per renderizzare i componenti *CardApp* in una struttura a griglia, sfruttando Boostrap per assegnare le classi .row-cols-breakpoint-# a partire dai valori della prop *col*.

```
return (
    <div className={`row
                    row-cols-${col.xs}
                    row-cols-sm-${col.sm}
                    row-cols-md-${col.md}
                    row-cols-lg-${col.lg}
                    row-cols-xl-${col.xl}`} >
        {charactersCardsCol}
    </div>
);
```

3.5. CardApp

CardApp è un componente funzione stateless che renderizza le singole card dei personaggi dei film di Star Wars.

CardApp riceve come props *id*, *images*, *name* e *species* di ogni personaggio per cui viene renderizzata la card.

```
const {id, image, name, species} = props;

return (
    <NavLink to={`/characters/${id}`}>
        <Card className={style.appCard}>
            <CardImg className={style.appCardImg} top width="100%" src={image}
alt={name}
                     onError={(event) => starWarsDefaultImg(event)}/>
            <div className={style.appCardInfo}>
                <h5 className={style.appCardTitle}>{name}</h5>
                <p className="card-text text-capitalize">{species}</p>
            </div>
        </Card>
    </NavLink>
);
```

In particolare, in caso di errore nel caricamento dell'immagine del personaggio viene chiamata la funzione *starWarsDefaultImg*. Questa è definita all'interno del file *utility.js* e permette, in caso di errore nel caricamento di un'immagine, di sostituire il src dell'immagine con il logo statico dell'applicazione.

Inoltre, nel componente *CardApp* viene utilizzato il componente *NavLink* di react-router-dom. Ciò permette all'utente di cliccare sulla carta del personaggio per accedere alla sua scheda (renderizzata dalla view *CharacterDetails*).

3.6. TableApp

TableApp è un componente funzione stateless che struttura il layout a tabella dei personaggi dei film di Star Wars.

TableApp riceve come prop la lista dei personaggi da visualizzare in griglia (*charactersList*).

La tabella viene strutturata attraverso la seguente funzione anonima:

```
const charactersTr = charactersList.map((character) => {
    return (
        <tr key={character.id}>
            <td>{character.id}</td>
            <td>
                <NavLink to={`/characters/${character.id}`}>
                    <img className={style.imgTable} loading="lazy"
                         onError={(event) => starWarsDefaultImg(event)}
                         src={character.image}
                         alt={character.name} />
                </NavLink>
```

```
            </td>
            <td>
                <NavLink to={`/characters/${character.id}`}>
                    {character.name}
                </NavLink>
            </td>
            <td className="text-capitalize">{character.species}</td>
            <td>
                <NavLink to={`/characters/${character.id}`}>
                    <div className={style.appBtnTable}>➤</div>
                </NavLink>
            </td>
        </tr>
    );
});
```

Tale funzione crea per ciascun oggetto di *charactersList* (ovvero per ciascun personaggio) una nuova riga della tabella avente 5 colonne in cui vengono mostrati: l'id del personaggio, la sua immagine, il suo nome, la sua specie ed un pulsante che permette all'utente di visualizzare la scheda specifica del personaggio (renderizzata dalla view *CharacterDetails*), a cui è possibile accedere anche cliccando sull'immagine ed il nome di esso. In tutti questi casi, per accedere alla scheda del personaggio, viene utilizzato il componente *NavLink* di react-router-dom.

Come per le immagini di *CardApp*, anche per le immagini di *TableApp*, in caso di errore nel caricamento, viene chiamata la funzione *starWarsDefaultImg*.

4. View

4.1. Home

La view *Home* è un componente funzione stateless che renderizza la sezione Home della web app.

Home fornisce all'utente una breve descrizione della web app e mostra come preview 4 card dei personaggi nella visualizzazione a griglia (*CardGridApp*). Per poter mostrare solo le card di 4 personaggi, viene utilizzata la seguente funzione anonima:

```
const charactersListShort = CharactersListData.filter((character, idx) =>
idx<4);
```

Tale funzione filtra la lista completa dei personaggi (*CharactersListData*) e salva solo i primi 4 all'interno della variabile *charactersListShort*. Questa viene passata come prop al componente *CardGridApp* che quindi mostra in griglia solo le card dei 4 personaggi filtrati.

```
<Container>
    <CardGridApp charactersList={charactersListShort}
                 col={{xs:1, sm:2, md:4, lg:4, xl:4}}/>
</Container>
```

4.2. Characters

La view *Characters* è un componente funzione statefull che renderizza la sezione *Characters* della web app. In essa l'utente può visualizzare i personaggi dei film di Star Wars nel layout a griglia (*CardGridApp*) o tabella (*TableApp*) e filtrarli rispetto alla specie d'appartenenza e in base all'ordine crescente o decrescente degli id.

La visualizzazione griglia-tabella viene strutturata utilizzando lo stato *displayGrid*, definito attraverso l'utilizzo dello hook useState. L'idea alla base è che quando il suo valore è true allora viene visualizzato il layout a griglia (*CardGridApp*), quando è false viene visualizzato il layout a tabella (*TableApp*).

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

Per cambiare il valore dello stato, nel return del componente viene renderizzato il pulsante di switch. In particolare, quando l'utente clicca su grid lo stato *displayGrid* diventa true, mentre quando clicca su table diventa false.

```
<button className={clsx("option", { ["active-switch"]: displayGrid } ) }
onClick={() => setDisplayGrid(true)}>
    Grid
</button>
<button className={clsx("option", { ["active-switch"]: !displayGrid } ) }
onClick={() => setDisplayGrid(false)}>
    Table
</button>
```

Infine, nel return del componente viene utilizzato l'operatore ternario per effettuare il rendering condizionale, dove la condizione è il valore dello stato *displayGrid*. In particolare, se lo stato è true allora viene renderizzato il componente *CardGridApp* (layout a griglia), se invece è false viene renderizzato *TableApp* (layout a tabella).

```
{displayGrid ?
    <CardGridApp charactersList={filteredList}
                 col={{xs:1, sm:2, md:3, lg:3, xl:4}}/>
    :
    <TableApp charactersList={filteredList}/>
}
```

Il filtraggio dei personaggi, invece, viene gestito utilizzando lo stato *filteredList*. Questo gestisce la lista dei personaggi che vengono filtrati attraverso i 2 filtri (specie ed ordine) ed il suo valore iniziale è la lista completa dei personaggi (*CharactersListData*). *filteredList* viene passato come prop ai componenti *CardGridApp* e *TableApp* che quindi renderizzano la lista dei personaggi filtrati nel corrispettivo layout.

```
const [filteredList, setFilteredList] = useState(CharactersListData);
```

In particolare, il filtro specie utilizza anche lo stato *selectedSpecies*. Questo gestisce la specie selezionata dall'utente nel menù dropdown. Il suo valore iniziale è una stringa vuota che corrisponde al valore dell'opzione All del menù.

```
const [selectedSpecies, setSelectedSpecies] = useState("");
```

Nel return del componente viene renderizzato il menù dropdown con il relativo evento onChange.

```
<select
    id="species-input"
    className="form-select app-select"
    value={selectedSpecies}
    onChange={handleSpeciesChange}
>
    <option value="">All</option>
    <option value="human">Humans</option>
    <option value="droid">Droids</option>
    <option value="wookiee">Wookiees</option>
    <option value="gungan">Gungans</option>
</select>
```

Quando l'evento onChange si verifica, viene eseguita la funzione *handleSpeciesChange*. Questa aggiorna lo stato *selectedSpecies* con il value della option selezionata dall'utente nel menù dropdown.

```
function handleSpeciesChange(event) {
    setSelectedSpecies(event.target.value);
};
```

Nel componente viene anche definita la funzione *filterBySpecies* che consiste nel "vero filtro". Essa filtra la lista dei personaggi in base alla voce del menù dropdown selezionata dall'utente con i seguenti passaggi:

1. L'if iniziale evita di filtrare la lista dei personaggi (*filteredData*) quando *selectedSpecies* corrisponde ad una stringa vuota ovvero quando l'utente seleziona la voce All del menù dropdown. In questo caso, infatti, non è necessario eseguire una funzione filter e viene quindi restituita la lista completa dei personaggi (*filteredData*).
2. Se *selectedSpecies* non è una stringa vuota, ovvero se l'utente seleziona una specie dal menù dropdown, viene eseguita la funzione filter che restituisce solo i personaggi (oggetti) dell'array *filteredData* che hanno come valore della proprietà species lo stesso valore di *selectedSpecies*. La funzione restituisce quindi il nuovo array con i personaggi filtrati (*filteredCharacters*).

```
function filterBySpecies(filteredData) {
    if (!selectedSpecies) {
        return filteredData;
    }
    const filteredCharacters = filteredData.filter(
        (character) => character.species === selectedSpecies
    );
    return filteredCharacters;
};
```

Infine, per attivare l'esecuzione di *filterBySpecies* viene utilizzato l'hook useEffect. In particolare, la funzione di callback applica la funzione *filterBySpecies* a *CharactersListData* ed aggiorna lo stato *filteredList* con la lista dei personaggi filtrati

(*filteredData*). Inoltre, come dependency di useEffect viene passato lo stato *selectedSpecies* in modo tale che la funzione di callback venga eseguita ogni volta che esso cambia ovvero ogni volta che l'utente seleziona una nuova voce del menù dropdown.

```
useEffect(() => {
        let filteredData = filterBySpecies(CharactersListData);
        setFilteredList(filteredData);
    },
[selectedSpecies]);
```

Per il filtro ordine, invece, viene utilizzata la funzione *orderFunction* che inverte l'ordine dei personaggi anche quando questi sono stati filtrati per specie ed infatti la funzione reverse viene applicata a *filteredList*. Inoltre, *orderFunction* aggiorna lo stato *filteredList* con la nuova lista ordinata dei personaggi.

```
function orderFunction () {
    const charactersListOrdered = [...filteredList].reverse();
    setFilteredList(charactersListOrdered);
}
```

La funzione *orderFunction* viene eseguita ogni volta che l'utente clicca su un apposito pulsante, definito nel return del componente *Characters*.

```
<button className="btn app-btn app-order-btn" onClick={orderFunction}>
    ↓↑
</button>
```

4.3. CharacterDetails

La view *CharacterDetails* è un componente funzione statefull che renderizza le schede dei singoli personaggi dei film di Star Wars riportando delle informazioni su di essi. Alcune di queste vengono ricavate da due differenti API: SWAPI e starwars-api di Yoann Cribier su GitHub. Inoltre, all'interno della view sono presenti due pulsanti che permettono di passare alla scheda del personaggio precedente e successivo ed uno che permette di tornare alla view *Characters*.

Per poter visualizzare la scheda del personaggio corrente e per permettere il passaggio a quelle dei personaggi precedenti e successivi, l'id della pagina corrente (*noteNumber*) viene trasformato da stringa a numero intero con il seguente passaggio:

```
const id = parseInt(noteNumber);
```

Inoltre, per ottenere il personaggio corrente viene eseguita la funzione filter sulla lista completa dei personaggi (*CharcatersListData*) in modo da restituire il personaggio il cui id ha lo stesso valore dell'id della pagina corrente.

```
const currentCharacter = CharactersListData.filter((character) => character.id
=== id)[0];
```

In questo modo, all'interno del return, l'immagine, il nome e la specie del personaggio selezionato (*currentCharacter*) vengono recuperati direttamente da *CharcatersListData*.

```
<h2 className="app-card-title">{currentCharacter.name}</h2>
<div className="d-flex justify-content-center">
    <img className={style.imgDetail}
         onError={(event) => starWarsDefaultImg(event)}
         src={currentCharacter.image} alt={currentCharacter.name} loading="lazy"
/>
</div>
```

```
<p className="text-capitalize"><span
className={style.characterInfoSubtitle}>Species:</span>
{currentCharacter.species}</p>
```

Le altre informazioni, invece, vengono ottenute dalle API SWAPI e starwars-api di Yoann Cribier su GitHub. Per poter chiamare le due API vengono utilizzati gli stati *characterInfoSwapi* e *characterInfoGit*.

```
const [characterInfoSwapi, setCharacterInfoSwapi] = useState([]);
```

```
const [characterInfoGit, setCharacterInfoGit] = useState([]);
```

Inoltre, viene utilizzato l'hook useEffect con la modalità fetch. In particolare, quando i dati delle API sono pronti, i relativi stati vengono aggiornati con i dati recuperati. All'hook useEffect viene passata come dependency l'id della pagina corrente in modo tale che la fetch venga rieseguita solo quando esso cambia ovvero quando cambia il personaggio mostrato.

```
useEffect(() => {
    let isMounted = true;
    /* This fetch calls SWAPI API. */
    fetch(`https://swapi.dev/api/people/${id}/`)
        .then(res => res.json())
        .then(res => {
            if (isMounted)
                setCharacterInfoSwapi(res); /* When the data are ready, the
characterInfoSwapi state is updated with data. */
        })
        .catch((error) => console.log("Error"+error));

    /* This fetch calls GitHub API. */
    fetch(`https://akabab.github.io/starwars-api/api/id/${id}.json`)
        .then(res => res.json())
        .then(res => {
            if (isMounted)
                setCharacterInfoGit(res); /* When the data are ready, the
characterInfoGit state is updated with data. */
        })
        .catch((error) => console.log("Error"+error));
    return () => {
        isMounted = false;
    }
}, [id]);
```

In particolare, dall'API SWAPI vengono ricavate le informazioni sul personaggio riguardanti il genere, l'anno di nascita, l'altezza ed il peso. Dall'API starwars-api di Yoann Cribier su GitHub vengono invece ricavate quelle sul pianeta di origine, le affiliazioni, gli eventuali apprendisti ed il link alla pagina del personaggio su Wookieepedia. Tutte queste informazioni vengono renderizzate nel return attraverso il rendering condizionale in modo tale che vengano stampate solo se contengono delle informazioni valide.

```
<Col xs={12}>
    {characterInfoGit.homeworld &&
    <p className="text-capitalize"><span
className={style.characterInfoSubtitle}>Homeworld:</span>
{characterInfoGit.homeworld}</p>
    }
</Col>
<Col xs={12}>
    {characterInfoSwapi.gender &&
    <p className="text-capitalize"><span
className={style.characterInfoSubtitle}>Gender:</span>
{characterInfoSwapi.gender}</p>
    }
</Col>
```

Tra le informazioni recuperate dalle API, quelle sulle affiliazioni e gli apprendisti del personaggio corrente contengono spesso più di una opzione. Per questo esse vengono mappate con una funzione che permette di creare un elenco puntato.

```
<p className={style.characterInfoSubtitleAff}>Affiliations:</p>
{characterInfoGit.affiliations &&
    <ul className="mt-1">
        {characterInfoGit.affiliations.map((affiliationsItem) => {
            return <li key={affiliationsItem}>{affiliationsItem}</li>
            }
        )}
    </ul>
}
```

```
<p className={style.characterInfoSubtitleAff}>Apprentices:</p>
{characterInfoGit.apprentices &&
<ul className="mt-1">
    {characterInfoGit.apprentices.map((apprenticesItem) => {
        return <li key={apprenticesItem}>{apprenticesItem}</li>
        }
    )}
</ul>
}
```

All'interno della view vengono anche renderizzati due pulsanti che permettono di passare alla scheda del personaggio precedente e successivo ed uno che permette di tornare alla view *Characters*. In essi viene utilizzato il componente *NavLink* di react-router-dom.

```
<div className="d-flex justify-content-evenly">
    {id - 1 !== 0 &&
    <NavLink className="btn app-btn" to={`/characters/${id - 1}`}>⮜</NavLink>
    }
```

```
    {id + 1 <= Object.keys(CharactersListData).length &&
    <NavLink className="btn app-btn" to={`/characters/${id + 1}`}>➤</NavLink>
    }
</div>
```

```
<NavLink to={`/characters`}>
    <button type="button" className="btn app-btn">⮌  Back</button>
</NavLink>
```

4.4. Info

La view *Info* è un componente funzione stateless che renderizza la sezione Info della web app.

Info illustra brevemente all'utente la serie cinematografica dei film di Star Wars.

4.5. NoMatch

La view *NoMatch* è un componente funzione stateless che viene visualizzato quando l'utente scrive un URL diverso da quelli definiti negli altri componenti Routes in App. Esso, infatti, renderizza una view con un messaggio in cui si suggerisce all'utente di controllare che l'url inserito sia corretto e riprovare.